
Statistics and Machine Learning in Python

Release 0.8

Edouard Duchesnay, Tommy Löfstedt, Younes Feki

Nov 13, 2025

TABLE OF CONTENTS

1	Python ecosystem for data science	3
1.1	Introduction to Python language	3
1.2	Overview of Python ecosystem for data-science	4
1.3	Development Environment	5
1.4	Package & Environments Dependency Management	7
1.5	Development with Integrated Development Environment (IDE) and JupyterLab	10
2	Python language	13
2.1	Import libraries	13
2.2	Basic operations	14
2.3	Data types	14
2.4	Execution control statements	23
2.5	List comprehensions, iterators, etc.	24
2.6	Functions	28
2.7	Regular Expression	32
2.8	System programming	34
2.9	Scripts and argument parsing	42
2.10	Networking	43
2.11	Object Oriented Programming (OOP)	44
2.12	Style guide for Python programming	46
2.13	Documenting	46
2.14	Modules and packages	47
2.15	Unit testing	48
2.16	Exercises	51
3	Data Manipulation and Visualization	53
3.1	Numpy: Arrays and Matrices	53
3.2	Pandas: data manipulation	63
4	Numerical Methods in Python	79
4.1	Numerical Differentiation	79
4.2	Numerical Integration	84
4.3	Time Series	86
4.4	Optimization (Minimization) by Gradient Descent	105
5	Statistics	125
5.1	Univariate Statistics	125
5.2	Hands-On: Brain volumes study	169
5.3	Linear Mixed Models	180

5.4	Multivariate Statistics	200
5.5	Resampling and Monte Carlo Methods	212
6	Machine Learning Overview	229
6.1	Introduction	229
6.2	Overfitting and Regularization	231
7	Unsupervised Learning	239
7.1	Linear Dimensionality Reduction and Feature Extraction	239
7.2	Manifold learning: non-linear dimension reduction	249
7.3	Clustering	255
8	Supervised Learning	265
8.1	Linear Models for Regression	265
8.2	Linear Models for Classification	274
8.3	Non-Linear Kernel Methods and Support Vector Machines (SVM)	297
8.4	Non-Linear Ensemble Learning	301
9	Resampling Methods and Model Evaluation	307
9.1	Out-of-sample Validation for Model Selection and Evaluation	307
9.2	Random Permutations: sample the null distribution	315
9.3	Bootstrapping	318
9.4	Parallel Computation	320
9.5	Hands-On: Validation of Supervised Classification Pipelines	321
10	Deep Learning: Introduction	335
10.1	Backpropagation	335
10.2	Multilayer Perceptron (MLP)	349
11	Deep Learning for Computer Vision	367
11.1	Convolutional Neural Networks (CNNs)	367
11.2	Pretraining and Transfer Learning	396
12	Natural Language Processing	403
12.1	Bag-of-Words Models	403
13	Mathematical Details and Demonstrations	417
13.1	Negative Log-Likelihood (NLL) for Binary Classification with Sigmoid Activation	417
14	Indices and tables	423

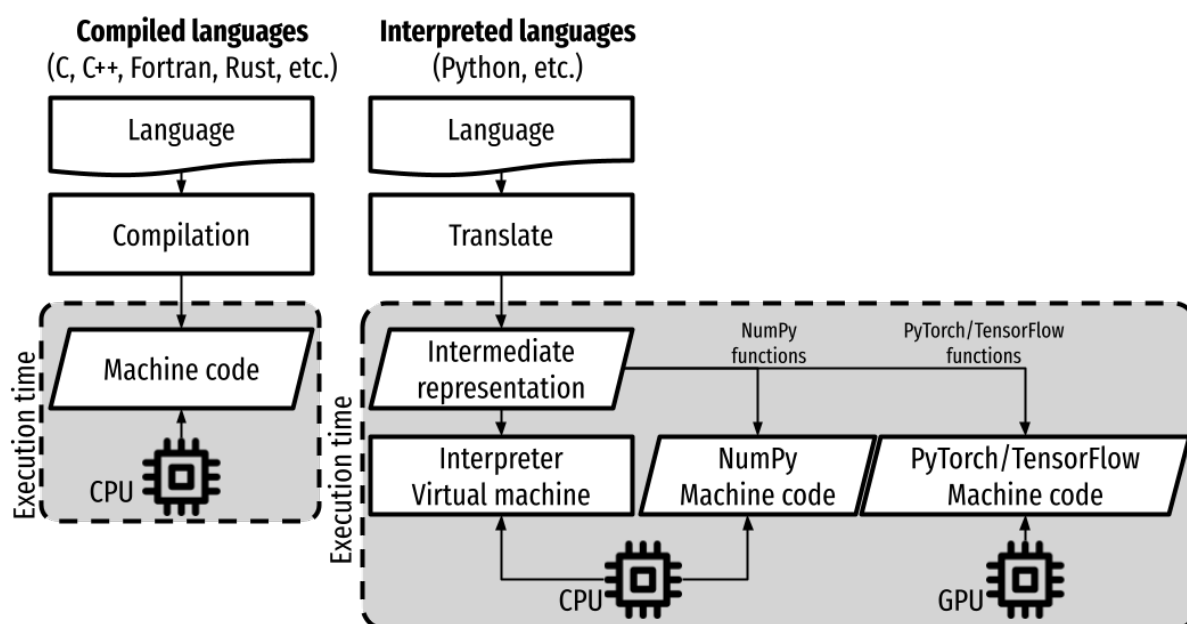
- [Github](#)
- [Latest pdf](#)
- [Official deposit for citation.](#)
- [Web page](#)

PYTHON ECOSYSTEM FOR DATA SCIENCE

1.1 Introduction to Python language

1.1.1 Python main features

- Python is popular [Google trends](#) (Python vs. R, Matlab, SPSS, Stata).
- Python is interpreted: source files .py are translated into an intermediate representation which is executed by the interpreter which is executed by the processor. Conversely, to interpreted languages, compiled languages, such as C or C++, rely on two steps: (i) source files are compiled into a binary program. (ii) binaries are executed by the CPU directly.
- Python integrates an automatic memory management mechanism: the **Garbage Collector (GC)**. (do not prevent from memory leak).
- Python is a dynamically-typed language (Java is statically typed).
- Efficient data manipulation is obtained using libraries (*NumPy*, *Scipy*, *Pytorch*) executed in compiled code.



1.1.2 Development process

Edit python file then execute

1. Write python file, file.py using any text editor:

```
a = 1
b = 2
print("Hello world")
```

2. Run with python interpreter. On the dos/unix command line execute whole file:

```
python file.py
```

Interactive mode

1. python interpreter:

```
python
```

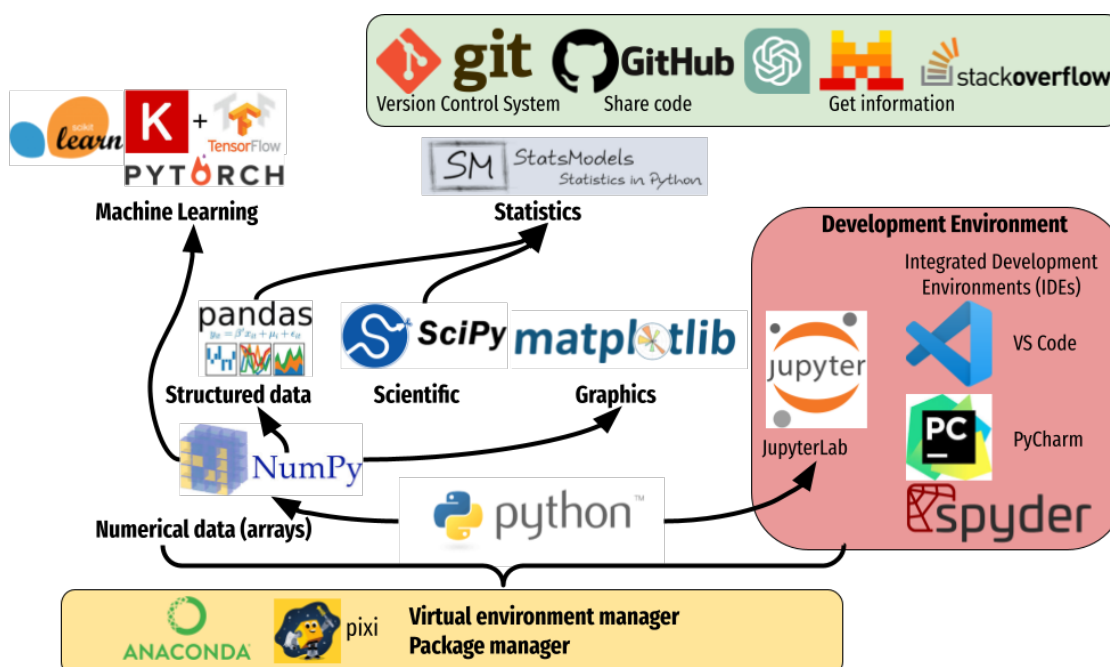
Quit with CTL-D

2. ipython: advanced interactive python interpreter:

```
ipython
```

Quit with CTL-D

1.2 Overview of Python ecosystem for data-science



NumPy: Basic numerical operation and matrix operation

```
import numpy as np
X = np.array([[1, 2], [3, 4]])
```

(continues on next page)

(continued from previous page)

```
v = np.array([1, 2])
np.dot(X, v)
X - X.mean(axis=0)
```

Scipy: General scientific libraries with advanced matrix operation and solver

```
import scipy
import scipy.linalg
scipy.linalg.svd(X, full_matrices=False)
```

Pandas: Manipulation of structured data (tables). Input/output excel files, etc.

```
import pandas as pd
data = pd.read_excel("datasets/iris.xls")
print(data.head())
Out[8]:
sepal_length  sepal_width  petal_length  petal_width  species
0             5.1         3.5          1.4          0.2  setosa
1             4.9         3.0          1.4          0.2  setosa
2             4.7         3.2          1.3          0.2  setosa
3             4.6         3.1          1.5          0.2  setosa
4             5.0         3.6          1.4          0.2  setosa
```

Matplotlib: visualization (low level primitives)

```
import numpy as np
import matplotlib.pyplot as plt
#%matplotlib qt
x = np.linspace(0, 10, 50)
sinus = np.sin(x)
plt.plot(x, sinus)
plt.show()
```

Seaborn: Data visualization (high level primitives for statistics)

See [Example gallery](#)

Statsmodel Advanced statistics (linear models, time series, etc.)

Scikit-learn Machine learning: non-deep learning models and toolbox to be combined with other learning models (Pytorch, etc.).

Pytorch for deep learning.

1.3 Development Environment

A typical Python development environment consists of several key components, which work together to facilitate coding, testing, and debugging. Here are the main components:

1. **Python Interpreter.** The core of any Python development environment is the Python interpreter (e.g., Python 3.x). It runs Python code and converts it into machine-readable form. You can download it from python.org.

2. Text Editor or Integrated Development Environment (IDE) or jupyter-notebook.
 - Text Editors: Lightweight editors like Sublime Text, Atom, or VS Code offer basic text editing with syntax highlighting and extensions for Python development.
 - IDEs: Full-featured IDEs like PyCharm, VS Code (with Python extensions), or Spyder offer advanced features like code completion, debugging, project management, version control, and testing integrations.
3. Package Manager & Dependency Management
 - pip: The default Python package manager, which allows you to install, upgrade, and manage external libraries and dependencies.
 - Conda: An alternative package and environment manager, often used in data science for managing dependencies and virtual environments.
 - [Pixi](#) is a fast software package manager built on top of the existing conda ecosystem.
 - Conda & Pixi: provide the Python Interpreter.
4. Virtual Environment Manager
 - Virtual environments allow you to create isolated environments for different projects, preventing conflicts between different project dependencies. Tools include:
 - [venv](#) (python module): Built-in module to create virtual environments.
 - [virtualenv](#): Another popular tool for creating isolated environments.
 - Conda & Pixi: Manages both packages and environments.
5. Version Control System
 - Git: Essential for source control, collaboration, and version management. Platforms like GitHub, GitLab, and Bitbucket integrate Git for remote repository management. IDEs often have built-in Git support or plugins that make using Git seamless.
6. Debugger
 - Python has a built-in debugger called pdb.
 - Most IDEs, like PyCharm or VS Code, offer graphical debugging tools with features like breakpoints, variable inspection, and step-through execution.
7. Testing Framework
 - Tools like unittest (built-in), pytest, or nose2 help automate testing and ensure code quality.
 - IDEs often integrate testing frameworks to run and debug tests efficiently.
8. Documentation Tools
 - Tools like Sphinx or pdoc help generate documentation from your code, making it easier for other developers (and your future self) to understand.
9. Containers (Optional)
 - Docker: Used to create isolated, reproducible development environments and ensure consistency between development and production environments.

1.4 Package & Environments Dependency Management

1.4.1 Pixi Package and Environment Manager

Pixi is a modern **package management** tool designed to enhance the experience of **managing Python environments** particularly for data science and machine learning workflows. It aims to improve upon the existing tools like Conda by offering faster and more efficient package management:

- [7 Reasons to Switch from Conda to Pixi.](#)
- [Transitioning from the conda or mamba to pixi](#)
- [Tutorial for python.](#)

Installation

Linux & macOS

```
curl -fsSL https://pixi.sh/install.sh | bash
```

Windows

```
powershell -ExecutionPolicy Bypass -c "irm -useb https://pixi.sh/install.ps1 | iex  
↪"
```

Uninstall

[Creating an Environment](#), then add python, and packages

```
pixi init myenv  
cd myenv  
pixi add python  
pixi add ipykernel  
pixi add scikit-learn pandas statsmodels seaborn openpyxl  
pixi add spyder spyder-kernels
```

Example with pystatsml, After downloading [pystatsml repository](#):

```
git clone https://github.com/duchesnay/pystatsml.git  
cd pystatsml
```

Install dependencies contained in *pixi.toml* file (within the project directory)

```
pixi install
```

Activate an environment (within the project directory)

```
pixi shell
```

What's in the environment?

```
pixi list
```

Deactivating an environment

```
exit
```

Install/uninstall a package

```
pixi add numpy
pixi remove numpy
```

1.4.2 Anaconda and Conda environments

Anaconda is a python distribution that ships most of python tools and libraries.

Installation

1. [Download anaconda](#)
2. Install it, on Linux

```
bash Anaconda3-2.4.1-Linux-x86_64.sh
```

3. Add anaconda path in your PATH variable (For Linux in your `.bashrc` file), example:

```
export PATH="${HOME}/anaconda3/bin:$PATH"
```

Conda environments

- A [Conda environments](#) contains a specific collection of conda packages that you have installed.
- Control packages environment for a specific purpose: collaborating with someone else, delivering an application to your client,
- Switch between environments

Creating an environment. Example, [environment_student.yml](#):

```
name: pystatsml
channels:
- conda-forge
dependencies:
- ipython
- scipy
- numpy
- pandas>=2.0.3
- jupyter
- matplotlib
- scikit-learn>=1.3.0
- seaborn
- statsmodels>=0.14.0
- torchvision
- skorch
```

Create the environment (go have a coffee):

```
conda env create -f pystatsml.yml
```


List of all environments. Activate/deactivate an environment:

```
conda env list
conda activate pystatsml
conda deactivate
```

Updating an environment (additional or better package, remove packages). Update the contents of your environment.yml file accordingly and then run the following command:

```
conda env update --file pystatsml.yml --prune
```

List all packages or search for a specific package in the current environment:

```
conda list
conda list numpy
```

Search for available versions of package in an environment:

```
conda search -f numpy
```

Install new package in an environment:

```
conda install numpy
```

Delete an environment:

```
conda remove -n pystatsml --all
```

Miniconda

Anaconda without the collection of (>700) packages. With Miniconda you download only the packages you want with the conda command: `conda install PACKAGENAME`

1. Download [Miniconda](#)
2. Install it, on Linux:

```
bash Miniconda3-latest-Linux-x86_64.sh
```

3. Add anaconda path in your PATH variable in your `.bashrc` file:

```
export PATH=${HOME}/miniconda3/bin:$PATH
```

4. Install required packages:

```
conda install -y scipy
conda install -y pandas
conda install -y matplotlib
conda install -y statsmodels
conda install -y scikit-learn
conda install -y spyder
conda install -y jupyter
```

1.4.3 Pip

pip alternative for packages management (update -U in user directory --user):

```
pip install -U --user seaborn
```

Example:

```
pip install -U --user nibabel
pip install -U --user nilearn
```

1.5 Development with Integrated Development Environment (IDE) and JupyterLab

Integrated Development Environment (IDE) are software development environment that provide:

- Source-code editor (auto-completion, etc.).
- Execution facilities (interactive, etc.).
- Debugger.

1.5.1 Visual Studio Code (VS Code)

Setup

- [Installation](#).
- Tuto for [Linux](#).
- Useful settings for python: [VS Code for python](#)
- Extensions for data-science in python: Python, Jupyter, Python Extension Pack, Python Pylance, Path Intellisense

Set Python environment: Open the Command Palette (Ctrl+Shift+P) search >Python: Select interpreter.

Execution, three possibilities:

1. Run Python file
2. Interactive execution in python interpreter, type: Shift/Enter
3. Interactive execution in Jupyter:
 - Install Jupyter Extension (cube icon / type jupyter / Install).
 - Optional, Shift/Enter will send selected text to interactive Jupyter notebook: in settings (gear wheel or CTL,: press control and comma keys), check box: Jupyter > Interactive Window Text Editor > Execute Selection

VS code with pixi <<https://pixi.sh/dev/integration/editor/vscode/>>`_

[Remote Development using SSH](#)

1. Setup ssh to hostname

2. Select Remote-SSH: Connect to Host... from the Command Palette (F1, Ctrl+Shift+P) and use the same `user@hostname` as in step 1
3. Remember hosts: (F1, Ctrl+Shift+P): Remote-SSH: Add New SSH Host or clicking on the Add New icon in the SSH Remote Explorer in the Activity Bar

1.5.2 Spyder

Spyder is a basic IDE dedicated to data-science.

- Syntax highlighting.
- Code introspection for code completion (use TAB).
- Support for multiple Python consoles (including IPython).
- Explore and edit variables from a GUI.
- Debugging.
- Navigate in code (go to function definition) CTL.

Shortcuts: - F9 run line/selection

1.5.3 JupyterLab (Jupyter Notebook)

JupyterLab allows data scientists to create and share document, ie, Jupyter Notebook. A Notebook is that is a document `.ipynb` including:

- Python code, text, figures (plots), equations, and other multimedia resources.
- The Notebook allows interactive execution of blocs of codes or text.
- Notebook is edited using a Web browsers and it is executed by (possibly remote) IPython kernel.

jupyter notebook

New/kernel

Advantages:

- Rapid and one shot data analysis
- Share all-in-one data analysis documents: including code, text and figures

Drawbacks ([source](#)):

- Difficult to maintain and keep in sync when collaboratively working on code.
- Difficult to operationalize your code when using Jupyter notebooks as they don't feature any built-in integration or tools for operationalizing your machine learning models.
- Difficult to scale: Jupyter notebooks are designed for single-node data science. If your data is too big to fit in your computer's memory, using Jupyter notebooks becomes significantly more difficult.

Source [Kevin Markham](#)

2.1 Import libraries

'generic import' of math module

```
import math  
math.sqrt(25)
```

```
5.0
```

import a function

```
from math import sqrt  
sqrt(25)    # no longer have to reference the module
```

```
5.0
```

import multiple functions at once

```
from math import sqrt, exp
```

import all functions in a module (strongly discouraged)

```
from os import *
```

define an alias

```
import nltk  
import numpy as np  
np.sqrt(9)
```

```
np.float64(3.0)
```

show all functions in math module

```
content = dir(math)
```

2.2 Basic operations

Numbers

```
10 + 4      # add (returns 14)
10 - 4      # subtract (returns 6)
10 * 4      # multiply (returns 40)
10 ** 4     # exponent (returns 10000)
10 / 4      # divide (returns 2 because both types are 'int')
10 / float(4) # divide (returns 2.5)
5 % 4       # modulo (returns 1) - also known as the remainder
10 / 4      # true division (returns 2.5)
10 // 4     # floor division (returns 2)
```

```
2
```

Boolean operations

comparisons (these return True)

```
5 > 3
5 >= 3
5 != 3
5 == 5
```

```
True
```

Boolean operations (these return True)

```
5 > 3 and 6 > 3
5 > 3 or 5 < 3
not False
False or not False and True    # evaluation order: not, and, or
```

```
True
```

2.3 Data types

Determine the type of an object

```
type(2)      # returns 'int'
type(2.0)    # returns 'float'
type('two')  # returns 'str'
type(True)   # returns 'bool'
type(None)   # returns 'NoneType'
```

Check if an object is of a given type

```
isinstance(2.0, int)      # returns False
isinstance(2.0, (int, float)) # returns True
```

```
True
```

Convert an object to a given type

```
float(2)
int(2.9)
str(2.9)
```

```
'2.9'
```

zero, None, and empty containers are converted to False

```
bool(0)
bool(None)
bool('')    # empty string
bool([])    # empty list
bool({})    # empty dictionary
```

```
False
```

Non-empty containers and non-zeros are converted to True

```
bool(2)
bool('two')
bool([2])
```

```
True
```

2.3.1 Lists

Different objects categorized along a certain ordered sequence, lists are ordered, iterable, mutable (adding or removing objects changes the list size), can contain multiple data types.

Creation

Empty list (two ways)

```
empty_list = []
empty_list = list()
```

List with values

```
simpsons = ['homer', 'marge', 'bart']
```

Examine a list

```
simpsons[0]    # print element 0 ('homer')
len(simpsons)  # returns the length (3)
```

```
3
```

Modify a list (does not return the list)

Append

```
simpsons.append('lisa')           # append element to end
simpsons.extend(['itchy', 'scratchy']) # append multiple elements to end
# insert element at index 0 (shifts everything right)
```

Insert

```
simpsons.insert(0, 'maggie')
# searches for first instance and removes it
```

Remove

```
simpsons.remove('bart')
simpsons.pop(0)           # removes element 0 and returns it
# removes element 0 (does not return it)
del simpsons[0]
simpsons[0] = 'krusty'    # replace element 0
```

Concatenate lists (slower than 'extend' method)

```
neighbors = simpsons + ['ned', 'rod', 'todd']
```

Replicate

```
rep = ["a"] * 2 + ["b"] * 3
```

Find elements in a list

```
'lisa' in simpsons
simpsons.count('lisa')    # counts the number of instances
simpsons.index('itchy')   # returns index of first instance
```

```
2
```

List slicing (selection) [start:end:stride]

```
weekdays = ['mon', 'tues', 'wed', 'thurs', 'fri']
weekdays[0]           # element 0
weekdays[0:3]          # elements 0, 1, 2
weekdays[:3]           # elements 0, 1, 2
weekdays[3:]           # elements 3, 4
weekdays[-1]           # last element (element 4)
weekdays[::2]          # every 2nd element (0, 2, 4)
```

```
['mon', 'wed', 'fri']
```

Reverse list


```
weekdays[::-1]      # backwards (4, 3, 2, 1, 0)
# alternative method for returning the list backwards
list(reversed(weekdays))
```

```
['fri', 'thurs', 'wed', 'tues', 'mon']
```

Sort list

Sort a list in place (modifies but does not return the list)

```
simpsons.sort()
simpsons.sort(reverse=True)    # sort in reverse
simpsons.sort(key=len)        # sort by a key
```

Return a sorted list (but does not modify the original list)

```
sorted(simpsons)
sorted(simpsons, reverse=True)
sorted(simpsons, key=len)
```

```
['lisa', 'itchy', 'krusty', 'scratchy']
```

2.3.2 Tuples

Like lists, but their size cannot change: ordered, iterable, immutable, can contain multiple data types

```
# create a tuple
digits = (0, 1, 'two')      # create a tuple directly
digits = tuple([0, 1, 'two']) # create a tuple from a list
# trailing comma is required to indicate it's a tuple
zero = (0,)

# examine a tuple
digits[2]                    # returns 'two'
len(digits)                  # returns 3
digits.count(0)              # counts the number of instances of that value (1)
digits.index(1)              # returns the index of the first instance of that value (1)

# elements of a tuple cannot be modified
# digits[2] = 2               # throws an error

# concatenate tuples
digits = digits + (3, 4)

# create a single tuple with elements repeated (also works with lists)
(3, 4) * 2                   # returns (3, 4, 3, 4)

# tuple unpacking
bart = ('male', 10, 'simpson') # create a tuple
```

2.3.3 Strings

A sequence of characters, they are iterable, immutable

```
# create a string
s = str(42)          # convert another data type into a string
s = 'I like you'

# examine a string
s[0]                 # returns 'I'
len(s)               # returns 10

# string slicing like lists
s[:6]                # returns 'I like'
s[7:]                # returns 'you'
s[-1]                # returns 'u'

# basic string methods (does not modify the original string)
s.lower()             # returns 'i like you'
s.upper()             # returns 'I LIKE YOU'
s.startswith('I')     # returns True
s.endswith('you')     # returns True
s.isdigit()           # returns False (True if every character is a digit)
s.find('like')         # returns index of first occurrence
s.find('hate')         # returns -1 since not found
s.replace('like', 'love') # replaces all instances of 'like' with 'love'

# split a string into a list of substrings separated by a delimiter
s.split(' ')          # returns ['I','like','you']
s.split()             # same thing
s2 = 'a, an, the'
s2.split(',')         # returns ['a',' an',' the']

# join a list of strings into one string using a delimiter
stooges = ['larry', 'curly', 'moe']
' '.join(stooges)     # returns 'larry curly moe'

# concatenate strings
s3 = 'The meaning of life is'
s4 = '42'
s3 + ' ' + s4          # returns 'The meaning of life is 42'
s3 + ' ' + str(42)     # same thing

# remove whitespace from start and end of a string
s5 = '  ham and cheese '
s5.strip()             # returns 'ham and cheese'
```

```
'ham and cheese'
```

Strings formatting

```
# string substitutions: all of these return 'raining cats and dogs'
'raining %s and %s' % ('cats', 'dogs')           # old way
'raining {} and {}'.format('cats', 'dogs')       # new way
'raining {arg1} and {arg2}'.format(arg1='cats', arg2='dogs') # named arguments

# String formatting
# See: https://realpython.com/python-formatted-output/
# Old method
print('6 %s' % 'bananas')
print('%d %s cost $%.1f' % (6, 'bananas', 3.14159))

# Format method positional arguments
print('{0} {1} cost ${2:.1f}'.format(6, 'bananas', 3.14159))
```

```
6 bananas
6 bananas cost $3.1
6 bananas cost $3.1
```

Strings encoding

Normal strings allow for escaped characters. The default strings use unicode string (u string)

```
print('first line\nsecond line') # or
print(u'first line\nsecond line')
print('first line\nsecond line' == u'first line\nsecond line')
```

```
first line
second line
first line
second line
True
```

Raw strings treat backslashes as literal characters

```
print(r'first line\nfirst line')
print('first line\nsecond line' == r'first line\nsecond line')
```

```
first line\nfirst line
False
```

Sequence of bytes are not strings, should be decoded before some operations

```
s = b'first line\nsecond line'
print(s)
print(s.decode('utf-8').split())
```

```
b'first line\nsecond line'
['first', 'line', 'second', 'line']
```

2.3.4 Dictionaries

Dictionary is the must-known data structure. Dictionaries are structures which can contain multiple data types, and is ordered with key-value pairs: for each (unique) key, the dictionary outputs one value. Keys can be strings, numbers, or tuples, while the corresponding values can be any Python object. Dictionaries are: unordered, iterable, mutable

Creation

```
# Empty dictionary (two ways)
empty_dict = {}
empty_dict = dict()

simpsons_roles_dict = {'Homer': 'father', 'Marge': 'mother',
                       'Bart': 'son', 'Lisa': 'daughter', 'Maggie': 'daughter'}

simpsons_roles_dict = dict(Homer='father', Marge='mother',
                           Bart='son', Lisa='daughter', Maggie='daughter')

simpsons_roles_dict = dict([('Homer', 'father'), ('Marge', 'mother'),
                             ('Bart', 'son'), ('Lisa', 'daughter'), ('Maggie',
                             ↪ 'daughter')])

print(simpsons_roles_dict)
```

```
{'Homer': 'father', 'Marge': 'mother', 'Bart': 'son', 'Lisa': 'daughter', 'Maggie'
↪ ': 'daughter'}
```

Access

```
# examine a dictionary
simpsons_roles_dict['Homer']    # 'father'
len(simpsons_roles_dict)       # 5
simpsons_roles_dict.keys()      # list: ['Homer', 'Marge', ...]
simpsons_roles_dict.values()    # list: ['father', 'mother', ...]
simpsons_roles_dict.items()     # list of tuples: [('Homer', 'father') ...]
'Homer' in simpsons_roles_dict  # returns True
'John' in simpsons_roles_dict   # returns False (only checks keys)

# accessing values more safely with 'get'
simpsons_roles_dict['Homer']     # returns 'father'
simpsons_roles_dict.get('Homer') # same thing

try:
    simpsons_roles_dict['John']   # throws an error
except KeyError as e:
    print("Error", e)

simpsons_roles_dict.get('John')  # None
# returns 'not found' (the default)
simpsons_roles_dict.get('John', 'not found')
```

```
Error 'John'

'not found'
```

Modify a dictionary (does not return the dictionary)

```
simpsons_roles_dict['Snowball'] = 'dog'           # add a new entry
simpsons_roles_dict['Snowball'] = 'cat'           # add a new entry
simpsons_roles_dict['Snoop'] = 'dog'              # edit an existing entry
del simpsons_roles_dict['Snowball']               # delete an entry

simpsons_roles_dict.pop('Snoop') # removes and returns ('dog')
simpsons_roles_dict.update(
    {'Mona': 'grandma', 'Abraham': 'grandpa'}) # add multiple entries
print(simpsons_roles_dict)
```

```
{'Homer': 'father', 'Marge': 'mother', 'Bart': 'son', 'Lisa': 'daughter', 'Maggie
→': 'daughter', 'Mona': 'grandma', 'Abraham': 'grandpa'}
```

Intersecting two dictionaries

```
simpsons_ages_dict = {'Homer': 45, 'Marge': 43,
                      'Bart': 11, 'Lisa': 10, 'Maggie': 1}

print(simpsons_roles_dict.keys() & simpsons_ages_dict.keys())

inter = simpsons_roles_dict.keys() & simpsons_ages_dict.keys()

l = list()

for n in inter:
    l.append([n, simpsons_ages_dict[n], simpsons_roles_dict[n]])

[[n, simpsons_ages_dict[n], simpsons_roles_dict[n]] for n in inter]
```

```
{'Homer', 'Marge', 'Lisa', 'Maggie', 'Bart'}

[['Homer', 45, 'father'], ['Marge', 43, 'mother'], ['Lisa', 10, 'daughter'], [
→ 'Maggie', 1, 'daughter'], ['Bart', 11, 'son']]
```

Iterating both key and values

```
[[key, val] for key, val in simpsons_ages_dict.items()]
```

```
[['Homer', 45], ['Marge', 43], ['Bart', 11], ['Lisa', 10], ['Maggie', 1]]
```

String substitution using a dictionary: syntax %(key)format, where format is the formatting character e.g. s for string.

```
print('Homer is the %(Homer)s of the family' % simpsons_roles_dict)
```

```
Homer is the father of the family
```

2.3.5 Sets

Like dictionaries, but with unique keys only (no corresponding values). They are: unordered, iterable, mutable, can contain multiple data types made up of unique elements (strings, numbers, or tuples)

Creation

```
# create an empty set
empty_set = set()

# create a set
languages = {'python', 'r', 'java'}          # create a set directly
snakes = set(['cobra', 'viper', 'python'])    # create a set from a list
```

Examine a set

```
len(languages)          # 3
'python' in languages    # True
```

```
True
```

Set operations

```
languages & snakes        # intersection: {'python'}
languages | snakes        # union: {'cobra', 'r', 'java', 'viper', 'python'}
languages - snakes        # set difference: {'r', 'java'}
snakes - languages        # set difference: {'cobra', 'viper'}

# modify a set (does not return the set)
languages.add('sql')       # add a new element
# try to add an existing element (ignored, no error)
languages.add('r')
languages.remove('java')   # remove an element

try:
    languages.remove('c')   # remove a non-existing element: throws an error
except KeyError as e:
    print("Error", e)

# removes an element if present, but ignored otherwise
languages.discard('c')
languages.pop()            # removes and returns an arbitrary element
languages.clear()          # removes all elements
languages.update('go', 'spark') # add multiple elements (list or set)

# get a sorted list of unique elements from a list
sorted(set([9, 0, 2, 1, 0])) # returns [0, 1, 2, 9]
```

Error 'c'

[0, 1, 2, 9]

2.4 Execution control statements

2.4.1 Conditional statements

if statement

```
x = 3
if x > 0:
    print('positive')
```

positive

if/else statement

```
if x > 0:
    print('positive')
else:
    print('zero or negative')
```

positive

Single-line if/else statement, known as a ‘ternary operator’

```
sign = 'positive' if x > 0 else 'zero or negative'
print(sign)
```

positive

if/elif/else statement

```
if x > 0:
    print('positive')
elif x == 0:
    print('zero')
else:
    print('negative')
```

positive

2.4.2 Loops

Loops are a set of instructions which repeat until termination conditions are met. This can include iterating through all values in an object, go through a range of values, etc

```
# range returns a list of integers
# returns [0, 1, 2]: includes first value but excludes second value
range(0, 3)
range(3)          # same thing: starting at zero is the default
range(0, 5, 2)    # returns [0, 2, 4]: third argument specifies the 'stride'
```

```
range(0, 5, 2)
```

Iterate on list values

```
fruits = ['Apple', 'Banana', 'cherry']
for fruit in fruits:
    print(fruit.upper())
```

```
APPLE
BANANA
CHERRY
```

Iterate with index

```
for i in range(len(fruits)):
    print(fruits[i].lower())
```

```
apple
banana
cherry
```

Iterate with index and values: enumerate

```
for i, val in enumerate(fruits):
    print(i, val.upper())

# Use range when iterating over a large sequence to avoid actually
# creating the integer list in memory
v = 0
for i in range(10 ** 6):
    v += 1
```

```
0 APPLE
1 BANANA
2 CHERRY
```

2.5 List comprehensions, iterators, etc.

2.5.1 List comprehensions

List comprehensions provides an elegant syntax for the most common processing pattern:

1. iterate over a list,
2. apply some operation

3. store the result in a new list

Classical iteration over a list

```
nums = [1, 2, 3, 4, 5]
cubes = []
for num in nums:
    cubes.append(num ** 3)
```

Equivalent list comprehension

```
cubes = [num**3 for num in nums]    # [1, 8, 27, 64, 125]
```

Classical iteration over a list with **if condition**: create a list of cubes of even numbers

```
cubes_of_even = []
for num in nums:
    if num % 2 == 0:
        cubes_of_even.append(num**3)
```

Equivalent list comprehension with **if condition** syntax: [expression for variable in iterable if condition]

```
cubes_of_even = [num**3 for num in nums if num % 2 == 0]    # [8, 64]
```

Classical iteration over a list with **if else condition**: for loop to cube even numbers and square odd numbers

```
cubes_and_squares = []
for num in nums:
    if num % 2 == 0:
        cubes_and_squares.append(num**3)
    else:
        cubes_and_squares.append(num**2)
```

Equivalent list comprehension (using a ternary expression) for loop to cube even numbers and square odd numbers syntax: [true_condition if condition else false_condition for variable in iterable]

```
cubes_and_squares = [num**3 if num % 2 == 0 else num**2 for num in nums]
print(cubes_and_squares)
```

```
[1, 8, 9, 64, 25]
```

Nested loops: flatten a 2d-matrix

```
matrix = [[1, 2], [3, 4]]
items = []
for row in matrix:
    for item in row:
        items.append(item)
```

Equivalent list comprehension with Nested loops

```
items = [item for row in matrix
         for item in row]

print(items)
```

```
[1, 2, 3, 4]
```

2.5.2 Set comprehension

```
fruits = ['apple', 'banana', 'cherry']
unique_lengths = {len(fruit) for fruit in fruits}
print(unique_lengths)
```

```
{5, 6}
```

2.5.3 Dictionary comprehension

Create a dictionary from a list

```
fruit_lengths = {fruit: len(fruit) for fruit in fruits}
print(fruit_lengths)
```

```
{'apple': 5, 'banana': 6, 'cherry': 6}
```

Iterate over keys and values. Increase age of each subject:

```
simpsons_ages_ = {key: val + 1 for key, val in simpsons_ages_dict.items()}
print(simpsons_ages_)
```

```
{'Homer': 46, 'Marge': 44, 'Bart': 12, 'Lisa': 11, 'Maggie': 2}
```

Combine two dictionaries sharing key. Example, a function that joins two dictionaries (intersecting keys) into a dictionary of lists

```
simpsons_info_dict = {name: [simpsons_roles_dict[name], simpsons_ages_dict[name]]
                     for name in simpsons_roles_dict.keys() &
                     simpsons_ages_dict.keys()}
print(simpsons_info_dict)
```

```
{'Homer': ['father', 45], 'Marge': ['mother', 43], 'Lisa': ['daughter', 10],
 → 'Maggie': ['daughter', 1], 'Bart': ['son', 11]}
```

2.5.4 Iterators itertools package

```
import itertools
```

Example: Cartesian product

```
print([[x, y] for x, y in itertools.product(['a', 'b', 'c'], [1, 2])])
```

```
[['a', 1], ['a', 2], ['b', 1], ['b', 2], ['c', 1], ['c', 2]]
```

2.5.5 Example, use loop, dictionary and set to count words in a sentence

```
quote = """Tick-tow
our incomes are like our shoes; if too small they gall and pinch us
but if too large they cause us to stumble and to trip
"""

words = quote.split()
len(words)

count = {word: 0 for word in set(words)}

for word in words:
    count[word] += 1
    # count[word] = count[word] + 1

print(count)

import numpy as np
freq_veq = np.array(list(count.values())) / len(words)
```

```
{'small': 1, 'like': 1, 'stumble': 1, 'trip': 1, 'shoes;': 1, 'cause': 1, 'and': 2,
'pinch': 1, 'to': 2, 'large': 1, 'us': 2, 'but': 1, 'our': 2, 'if': 2, 'too': 2,
'they': 2, 'Tick-tow': 1, 'gall': 1, 'incomes': 1, 'are': 1}
```

2.5.6 Exceptions handling

```
dct = dict(a=[1, 2], b=[4, 5])

key = 'c'
try:
    dct[key]
except:
    print("Key %s is missing. Add it with empty value" % key)
    dct['c'] = []

print(dct)
```

```
Key c is missing. Add it with empty value
{'a': [1, 2], 'b': [4, 5], 'c': []}
```

2.6 Functions

Functions are sets of instructions launched when called upon, they can have multiple input values and a return value

Function with no arguments and no return values

```
def print_text():  
    print('this is text')  
  
# call the function  
print_text()
```

```
this is text
```

Function with one argument and no return values

```
def print_this(x):  
    print(x)  
  
# call the function  
print_this(3)      # prints 3  
n = print_this(3)  # prints 3, but doesn't assign 3 to n  
# because the function has no return statement  
print(n)
```

```
3  
3  
None
```

Dynamic typing

Important remarque: **Python is a dynamically typed language**, meaning that the Python interpreter does type checking at runtime (as opposed to compiled language that are statically typed). As a consequence, the function behavior, decided, at execution time, will be different and specific to parameters type. Python function are polymorphic.

```
def add(a, b):  
    return a + b  
  
print(add(2, 3), add("deux", "trois"), add(["deux", "trois"], [2, 3]))
```

```
5 deuxtros ['deux', 'trois', 2, 3]
```

Default arguments

```
def power_this(x, power=2):  
    return x ** power
```

(continues on next page)

(continued from previous page)

```
print(power_this(2), power_this(2, 3))
```

```
4 8
```

Docstring to describe the effect of a function IDE, ipython (type: ?power_this) to provide function documentation.

```
def power_this(x, power=2):
    """Return the power of a number.

    Args:
        x (float): the number
        power (int, optional): the power. Defaults to 2.
    """
    return x ** power
```

Return several values as tuple

```
def min_max(nums):
    return min(nums), max(nums)

# return values can be assigned to a single variable as a tuple
min_max_num = min_max([1, 2, 3])      # min_max_num = (1, 3)

# return values can be assigned into multiple variables using tuple unpacking
min_num, max_num = min_max([1, 2, 3])  # min_num = 1, max_num = 3
```

Arbitrary number of Arguments

Packing and Unpacking Arguments in Python

*args packs many positional arguments e.g., `add(1, 2, 3)` as a tuple, arguments can be manipulated as a tuple, ie `args[0]`, etc.

```
def add(*args):
    print(args)
    s = 0
    for x in args:
        s += x
    return s

print(add(2, 3) + add(1, 2, 3))
```

```
(2, 3)
(1, 2, 3)
11
```

Pass arbitrary number of arguments to another function. re-pack arguments while passing them using `*args`

```
def dummy(*args):
    # do something
    return add(*args)

print(dummy(2, 3) + dummy(1, 2, 3))
```

```
(2, 3)
(1, 2, 3)
11
```

***kwargs* packs many keywords arguments e.g., `add(x=1, y=2, z=3)` as a dictionary:

```
def add(**kwargs):
    s = 0
    for key, val in kwargs.items():
        s += val
    return s

add(x=2, y=3) + add(x=1, y=2, z=3)

# - `*args` packs many positional arguments e.g., `add(1, 2, 3)` as a tuple:
```

```
11
```

2.6.1 Reference and copy

References are used to access objects in memory, here lists. A single object may have multiple references. Modifying the content of the one reference will change the content of all other references.

Modify a a reference of a list

```
num = [1, 2, 3]
same_num = num    # create a second reference to the same list
same_num[0] = 0    # modifies both 'num' and 'same_num'
print(num, same_num)
```

```
[0, 2, 3] [0, 2, 3]
```

Copies are references to different objects. Modifying the content of the one reference, will not affect the others.

Modify a copy of a list

```
new_num = num.copy()
new_num = num[:]
new_num = list(num)
new_num[0] = -1    # modifies 'new_num' but not 'num'
print(num, new_num)
```

```
[0, 2, 3] [-1, 2, 3]
```

Examine objects

```
id(num) == id(same_num) # returns True
id(num) == id(new_num) # returns False
num is same_num         # returns True
num is new_num          # returns False
num == same_num         # returns True
num == new_num          # returns True (their contents are equivalent)
```

```
False
```

Functions' arguments are references to objects. Thus functions can modify their arguments with possible side effect.

```
def change(x, index, newval):
    x[index] = newval

l = [0, 1, 2]
change(x=l, index=1, newval=33)
print(l)
```

```
[0, 33, 2]
```

2.6.2 Example: function, and dictionary comprehension

Example of a function `join_dict_to_table(dict1, dict2)` joining two dictionaries (intersecting keys) into a table, i.e., a list of tuples, where the first column is the key, the second and third columns are the values of the dictionaries.

```
simpsons_ages_dict = {'Homer': 45, 'Marge': 43, 'Bart': 11, 'Lisa': 10}
simpsons_roles_dict = {'Homer': 'father', 'Marge': 'mother', 'Bart': 'son',
                       'Maggie': 'daughter'}

def join_dict_to_table(dict1, dict2):
    table = [[key] + [dict1[key], dict2[key]]
              for key in dict1.keys() & dict2.keys()]
    return table

print("Roles:", simpsons_roles_dict)
print("Ages:", simpsons_ages_dict)
print("Join:", join_dict_to_table(simpsons_roles_dict,
                                  simpsons_ages_dict))
```

```
Roles: {'Homer': 'father', 'Marge': 'mother', 'Bart': 'son', 'Maggie': 'daughter'}
Ages: {'Homer': 45, 'Marge': 43, 'Bart': 11, 'Lisa': 10}
Join: [['Bart', 'son', 11], ['Homer', 'father', 45], ['Marge', 'mother', 43]]
```

2.7 Regular Expression

Regular Expression (RE, or RegEx) allow to search and patterns in strings. See [this page](#) for the syntax of the RE patterns.

```
import re
```

Usual patterns

- . period symbol matches any single character (except newline 'n').
- pattern``+`` plus symbol matches one or more occurrences of the pattern.
- [] square brackets specifies a set of characters you wish to match
- [abc] matches a, b or c
- [a-c] matches a to z
- [0-9] matches 0 to 9
- [a-zA-Z0-9]+ matches words, at least one alphanumeric character (digits and alphabets)
- [\w]+ matches words, at least one alphanumeric character including underscore.
- \s Matches where a string contains any whitespace character, equivalent to [\t\nr\v].
- [^s] Caret ^ symbol (the start of a square-bracket) inverts the pattern selection .

```
# regex = re.compile("^.(firstname:.+)(lastname:.+)(mod-.+)")
# regex = re.compile("(firstname:.+)(lastname:.+)(mod-.+)")
```

Compile (re.compile(string)) regular expression with a pattern that captures the pattern `firstname:<subject_id>_lastname:<session_id>`. Note that we use raw string `r'string'` so ```` is not interpreted as the start of an escape sequence.

```
pattern = re.compile(r'firstname:[\w]+_lastname:[\w]+')
```

Match (re.match(string)) to be used in test, loop, etc. Determine if the RE matches **at the beginning** of the string.

```
yes_ = True if pattern.match("firstname:John_lastname:Doe") else False
no_ = True if pattern.match("blahbla_firstname:John_lastname:Doe") else False
no2_ = True if pattern.match("OUPS-John_lastname:Doe") else False
print(yes_, no_, no2_)
```

```
True False False
```

Match (re.search(string)) to be used in test, loop, etc. Determine if the RE matches **at any location** in the string.

```
yes_ = True if pattern.search("firstname:John_lastname:Doe") else False
yes2_ = True if pattern.search(
    "blahbla_firstname:John_lastname:Doe") else False
no_ = True if pattern.search("OUPS-John_lastname:Doe") else False
print(yes_, yes2_, no_)
```



```
True True False
```

Find (`re.findall(string)`) all substrings where the RE matches, and returns them as a list.

```
# Find the whole pattern within the string
pattern = re.compile(r'firstname:[\w]+_lastname:[\w]+')
print(pattern.findall("firstname:John_lastname:Doe blah blah"))

# Find words
print(re.compile("[a-zA-Z0-9]+").findall("firstname:John_lastname:Doe"))

# Find words with including underscore
print(re.compile(r'[\w]+').findall("firstname:John_lastname:Doe"))
```

```
['firstname:John_lastname:Doe']
['firstname', 'John', 'lastname', 'Doe']
['firstname', 'John_lastname', 'Doe']
```

Extract specific parts of the RE: use parenthesis (part of pattern to be matched) Extract John and Doe, such as John is suffixed with `firstname:` and Doe is suffixed with `lastname:`

```
pattern = re.compile("firstname:([\w]+)_lastname:([\w]+)")
print(pattern.findall("firstname:John_lastname:Doe \
    firstname:Bart_lastname:Simpson"))
```

```
/home/ed203246/git/pystatsml/python_lang/python_lang.py:1031: SyntaxWarning: _
↳invalid escape sequence '\w'
    pattern = re.compile("firstname:([\w]+)_lastname:([\w]+)")
[('John', 'Doe'), ('Bart', 'Simpson')]
```

Split (`re.split(string)`) splits the string where there is a match and returns a list of strings where the splits have occurred. Example, match any non alphanumeric character (digits and alphabets) `[^a-zA-Z0-9]` to split the string.

```
print(re.compile("[^a-zA-Z0-9]").split("firstname:John_lastname:Doe"))
```

```
['firstname', 'John', 'lastname', 'Doe']
```

Substitute (`re.sub(pattern, replace, string)`) returns a string where matched occurrences are replaced with the content of `replace` variable.

```
print(re.sub('\s', "_", "Sentence with white      space"))
print(re.sub('\s+', "_", "Sentence with white      space"))
```

```
/home/ed203246/git/pystatsml/python_lang/python_lang.py:1048: SyntaxWarning: _
↳invalid escape sequence '\s'
    print(re.sub('\s', "_", "Sentence with white      space"))
/home/ed203246/git/pystatsml/python_lang/python_lang.py:1049: SyntaxWarning: _
↳invalid escape sequence '\s'
    print(re.sub('\s+', "_", "Sentence with white      space"))
```

(continues on next page)

(continued from previous page)

```
Sentence_with_white_____space  
Sentence_with_white_space
```

Remove all non-alphanumeric characters and space in a string

```
re.sub('[^0-9a-zA-Z\s]+', '', 'H^&ell`. ,|o W]{+orld')
```

```
/home/ed203246/git/pystatsml/python_lang/python_lang.py:1054: SyntaxWarning: _  
→invalid escape sequence '\s'  
  re.sub('[^0-9a-zA-Z\s]+', '', 'H^&ell`. ,|o W]{+orld')  
  
'Hello World'
```

2.8 System programming

2.8.1 Operating system interfaces (os)

```
import os
```

Get/set current working directory

```
# Get the current working directory  
cwd = os.getcwd()  
print(cwd)  
  
# Set the current working directory  
os.chdir(cwd)
```

```
/home/ed203246/git/pystatsml/python_lang
```

Temporary directory

```
import tempfile  
tmpdir = tempfile.gettempdir()  
print(tmpdir)
```

```
/tmp
```

Join paths

```
mytmpdir = os.path.join(tmpdir, "foobar")
```

Create a directory

```
os.makedirs(os.path.join(tmpdir, "foobar", "plop", "toto"), exist_ok=True)  
  
# list containing the names of the entries in the directory given by path.  
os.listdir(mytmpdir)
```

```
['myfile.txt', 'plop']
```

2.8.2 File input/output

```
filename = os.path.join(mytmpdir, "myfile.txt")
print(filename)
lines = ["Dans python tout est bon", "Enfin, presque"]
```

```
/tmp/foobar/myfile.txt
```

Write line by line

```
fd = open(filename, "w")
fd.write(lines[0] + "\n")
fd.write(lines[1] + "\n")
fd.close()
```

Context manager to automatically close your file

```
with open(filename, 'w') as f:
    for line in lines:
        f.write(line + '\n')
```

Read read one line at a time (entire file does not have to fit into memory)

```
f = open(filename, "r")
f.readline()    # one string per line (including newlines)
f.readline()    # next line
f.close()

# read the whole file at once, return a list of lines
f = open(filename, 'r')
f.readlines()   # one list, each line is one string
f.close()

# use list comprehension to duplicate readlines without reading entire file at
↳ once
f = open(filename, 'r')
[line for line in f]
f.close()

# use a context manager to automatically close your file
with open(filename, 'r') as f:
    lines = [line for line in f]
```

2.8.3 Explore, list directories

Walk through directories and subdirectories `os.walk(dir)`

```
WD = os.path.join(tmpdir, "foobar")

for dirpath, dirnames, filenames in os.walk(WD):
    print(dirpath, dirnames, filenames)
```

```
/tmp/foobar ['plop'] ['myfile.txt']
/tmp/foobar/plop ['toto'] []
/tmp/foobar/plop/toto [] []
```

Search for a file using a wildcard `glob.glob(dir)`

```
import glob
filenames = glob.glob(os.path.join(tmpdir, "*", "*.txt"))
print(filenames)
```

```
['/tmp/foobar/myfile.txt', '/tmp/plop2/myfile.txt']
```

Manipulating file names, `basename` and `extension`

```
def split_filename_inparts(filename):
    dirname_ = os.path.dirname(filename)
    filename_noext_, ext_ = os.path.splitext(filename)
    basename_ = os.path.basename(filename_noext_)
    return dirname_, basename_, ext_

print(filenames[0], "=>", split_filename_inparts(filenames[0]))
```

```
/tmp/foobar/myfile.txt => ('/tmp/foobar', 'myfile', '.txt')
```

File operations: (recursive) copy, move, test if exists: `shutil` package

```
import shutil
```

Copy

```
src = os.path.join(tmpdir, "foobar", "myfile.txt")
dst = os.path.join(tmpdir, "foobar", "plop", "myfile.txt")
shutil.copy(src, dst)
print("copy %s to %s" % (src, dst))
```

```
copy /tmp/foobar/myfile.txt to /tmp/foobar/plop/myfile.txt
```

Test if file exists ?

```
print("File %s exists ?" % dst, os.path.exists(dst))
```

```
File /tmp/foobar/plop/myfile.txt exists ? True
```

Recursive copy, deletion and move

```

src = os.path.join(tmpdir, "foobar", "plop")
dst = os.path.join(tmpdir, "plop2")

try:
    print("Copy tree %s under %s" % (src, dst))
    # Note that by default (dirs_exist_ok=True), meaning that copy will fail
    # if destination exists.
    shutil.copytree(src, dst, dirs_exist_ok=True)

    print("Delete tree %s" % dst)
    shutil.rmtree(dst)

    print("Move tree %s under %s" % (src, dst))
    shutil.move(src, dst)
except (FileExistsError, FileNotFoundError) as e:
    pass

```

```

Copy tree /tmp/foobar/plop under /tmp/plop2
Delete tree /tmp/plop2
Move tree /tmp/foobar/plop under /tmp/plop2

```

2.8.4 Command execution with subprocess

For more advanced use cases, the underlying Popen interface can be used directly.

```
import subprocess
```

```
subprocess.run([command, args*])
```

- Run the command described by args.
- Wait for command to complete
- return a CompletedProcess instance.
- Does not capture stdout or stderr by default. To do so, pass PIPE for the stdout and/or stderr arguments.

```

p = subprocess.run(["ls", "-l"])
print(p.returncode)

```

```
0
```

Run through the shell

```
subprocess.run("ls -l", shell=True)
```

```
CompletedProcess(args='ls -l', returncode=0)
```

Capture output

```
out = subprocess.run(
    ["ls", "-a", "/"], stdout=subprocess.PIPE, stderr=subprocess.STDOUT)
# out.stdout is a sequence of bytes that should be decoded into a utf-8 string
print(out.stdout.decode('utf-8').split("\n")[:5])
```

```
['.', '..', 'bin', 'bin.usr-is-merged', 'boot']
```

2.8.5 Multiprocessing and multithreading

Difference between multiprocessing and multithreading is essential to perform efficient parallel processing on multi-cores computers.

Multiprocessing

A process is a program instance that has been loaded into memory and managed by the operating system. Process = address space + execution context (thread of control)

- Process address space is made of (memory) segments for (i) code, (ii) data (static/global), (iii) heap (dynamic memory allocation), and the execution stack (functions' execution context).
- Execution context consists of (i) data registers, (ii) Stack Pointer (SP), (iii) Program Counter (PC), and (iv) working Registers.

OS Scheduling of processes: context switching (ie. save/load Execution context)

Pros/cons

- Context switching expensive.
- (potentially) complex data sharing (not necessary true).
- Cooperating processes - no need for memory protection (separate address spaces).
- Relevant for parallel computation with memory allocation.

Multithreading

- Threads share the same address space (Data registers): access to code, heap and (global) data.
- Separate execution stack, PC and Working Registers.

Pros/cons

- **Faster context switching** only SP, PC and Working Registers.
- Can exploit fine-grain concurrency
- Simple data sharing through the shared address space.
- **But most of concurrent memory operations are serialized (blocked) by the global interpreter lock (GIL).** The GIL prevents two threads writing to the same memory at the same time.
- Relevant for GUI, I/O (Network, disk) concurrent operation

In Python

- As long the GIL exists favor multiprocessing over multithreading
- Multithreading rely on threading module.
- Multiprocessing rely on multiprocessing module.

Example: Random forest

Random forest are the obtained by Majority vote of decision tree on estimated on bootstrapped samples.

Toy dataset

```
import time
import numpy as np
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import balanced_accuracy_score

# Toy dataset
X, y = make_classification(n_features=1000, n_samples=5000, n_informative=20,
                          random_state=1, n_clusters_per_class=3)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.8,
                                                    random_state=42)
```

Random forest algorithm: (i) In parallel, fit decision trees on bootstrapped data samples. Make predictions. (ii) Majority vote on predictions

1. In parallel, fit decision trees on bootstrapped data sample. Make predictions.

```
def boot_decision_tree(X_train, X_test, y_train, predictions_list=None):
    N = X_train.shape[0]
    boot_idx = np.random.choice(np.arange(N), size=N, replace=True)
    clf = DecisionTreeClassifier(random_state=0)
    clf.fit(X_train[boot_idx], y_train[boot_idx])
    y_pred = clf.predict(X_test)
    if predictions_list is not None:
        predictions_list.append(y_pred)
    return y_pred
```

Independent runs of decision tree, see variability of predictions

```
for i in range(5):
    y_test_boot = boot_decision_tree(X_train, X_test, y_train)
    print("%.2f" % balanced_accuracy_score(y_test, y_test_boot))
```

```
0.64
0.63
0.66
0.62
0.65
```

2. Majority vote on predictions

```
def vote(predictions):
    maj = np.apply_along_axis(
        lambda x: np.argmax(np.bincount(x)),
        axis=1,
        arr=predictions
    )
    return maj
```

Sequential execution

Sequentially fit decision tree on bootstrapped samples, then apply majority vote

```
nboot = 2
start = time.time()
y_test_boot = np.dstack([boot_decision_tree(X_train, X_test, y_train)
                        for i in range(nboot)]).squeeze()
y_test_vote = vote(y_test_boot)
print("Balanced Accuracy: %.2f" % balanced_accuracy_score(y_test, y_test_vote))
print("Sequential execution, elapsed time:", time.time() - start)
```

```
Balanced Accuracy: 0.63
Sequential execution, elapsed time: 1.3636796474456787
```

Multithreading

Concurrent (parallel) execution of the function with two threads.

```
from threading import Thread

predictions_list = list()
thread1 = Thread(target=boot_decision_tree,
                 args=(X_train, X_test, y_train, predictions_list))
thread2 = Thread(target=boot_decision_tree,
                 args=(X_train, X_test, y_train, predictions_list))

# Will execute both in parallel
start = time.time()
thread1.start()
thread2.start()

# Joins threads back to the parent process
thread1.join()
thread2.join()

# Vote on concatenated predictions
y_test_boot = np.dstack(predictions_list).squeeze()
y_test_vote = vote(y_test_boot)
print("Balanced Accuracy: %.2f" % balanced_accuracy_score(y_test, y_test_vote))
print("Concurrent execution with threads, elapsed time:", time.time() - start)
```


Balanced Accuracy: 0.64

Concurrent execution with threads, elapsed time: 0.6563949584960938

Multiprocessing

Concurrent (parallel) execution of the function with processes (jobs) executed in different address (memory) space. [Process-based parallelism](#)

`Process()` for parallel execution and `Manager()` for data sharing

Sharing data between process with Managers Therefore, sharing data requires specific mechanism using `Manager`. Managers provide a way to create data which can be shared between different processes, including sharing over a network between processes running on different machines. A manager object controls a server process which manages shared objects.

```
from multiprocessing import Process, Manager

predictions_list = Manager().list()
p1 = Process(target=boot_decision_tree,
             args=(X_train, X_test, y_train, predictions_list))
p2 = Process(target=boot_decision_tree,
             args=(X_train, X_test, y_train, predictions_list))

# Will execute both in parallel
start = time.time()
p1.start()
p2.start()

# Joins processes back to the parent process
p1.join()
p2.join()

# Vote on concatenated predictions
y_test_boot = np.dstack(predictions_list).squeeze()
y_test_vote = vote(y_test_boot)
print("Balanced Accuracy: %.2f" % balanced_accuracy_score(y_test, y_test_vote))
print("Concurrent execution with processes, elapsed time:", time.time() - start)
```

Balanced Accuracy: 0.64

Concurrent execution with processes, elapsed time: 0.6514365673065186

`Pool()` of **workers (processes or Jobs)** for concurrent (parallel) execution of multiples tasks. `Pool` can be used when N independent tasks need to be executed in parallel, when there are more tasks than cores on the computer.

1. Initialize a `Pool()`, `map()`, `apply_async()`, of P workers (`Process`, or `Jobs`), where $P <$ number of cores in the computer. Use `cpu_count` to get the number of logical cores in the current system, See: [Number of CPUs and Cores in Python](#).
2. Map N tasks to the P workers, here we use the function `Pool.apply_async()` that runs the jobs asynchronously. Asynchronous means that calling `pool.apply_async` does not block the execution of the caller that carry on, i.e., it returns immediately with a `AsyncResult` object for the task.

that the caller (than runs the sub-processes) is not blocked by the to the process pool does not block, allowing the caller that issued the task to carry on. # 3. Wait for all jobs to complete `pool.join()` 4. Collect the results

```
from multiprocessing import Pool, cpu_count
# Numbers of logical cores in the current system.
# Rule of thumb: Divide by 2 to get nb of physical cores
njobs = int(cpu_count() / 2)
start = time.time()
ntasks = 12

pool = Pool(njobs)
# Run multiple tasks each with multiple arguments
async_results = [pool.apply_async(boot_decision_tree,
                                args=(X_train, X_test, y_train))
                 for i in range(ntasks)]

# Close the process pool & wait for all jobs to complete
pool.close()
pool.join()

# Collect the results
y_test_boot = np.dstack([ar.get() for ar in async_results]).squeeze()

# Vote on concatenated predictions
y_test_vote = vote(y_test_boot)
print("Balanced Accuracy: %.2f" % balanced_accuracy_score(y_test, y_test_vote))
print("Concurrent execution with processes, elapsed time:", time.time() - start)
```

```
Balanced Accuracy: 0.64
Concurrent execution with processes, elapsed time: 1.8547492027282715
```

2.9 Scripts and argument parsing

Example, the word count script

```
import os
import os.path
import argparse
import re
import pandas as pd

if __name__ == "__main__":
    # parse command line options
    output = "word_count.csv"
    parser = argparse.ArgumentParser()
    parser.add_argument('-i', '--input',
                        help='list of input files.',
                        nargs='+', type=str)
```

(continues on next page)

(continued from previous page)

```

parser.add_argument('-o', '--output',
                    help='output csv file (default %s)' % output,
                    type=str, default=output)
options = parser.parse_args()

if options.input is None :
    parser.print_help()
    raise SystemExit("Error: input files are missing")
else:
    filenames = [f for f in options.input if os.path.isfile(f)]

# Match words
regex = re.compile("[a-zA-Z]+")

count = dict()
for filename in filenames:
    fd = open(filename, "r")
    for line in fd:
        for word in regex.findall(line.lower()):
            if not word in count:
                count[word] = 1
            else:
                count[word] += 1

fd = open(options.output, "w")

# Pandas
df = pd.DataFrame([[k, count[k]] for k in count], columns=["word", "count"])
df.to_csv(options.output, index=False)

```

2.10 Networking

```
# TODO
```

2.10.1 FTP

FTP with ftplib

```

import ftplib

ftp = ftplib.FTP("ftp.cea.fr")
ftp.login()
ftp.cwd('/pub/unati/people/educhesnay/pystatml')
ftp.retrlines('LIST')

fd = open(os.path.join(tmpdir, "README.md"), "wb")
ftp.retrbinary('RETR README.md', fd.write)
fd.close()
ftp.quit()

```

```
-rwxrwxr-x    1 ftp      ftp          3019 Oct 16  2019 README.md
-rwxrwxr-x    1 ftp      ftp       10672252 Dec 18  2020_
↪StatisticsMachineLearningPython.pdf
-rwxrwxr-x    1 ftp      ftp       9676120 Nov 12  2020_
↪StatisticsMachineLearningPythonDraft.pdf
-rwxrwxr-x    1 ftp      ftp       9798485 Jul  08  2020_
↪StatisticsMachineLearningPythonDraft_202007.pdf

'221 Goodbye.'
```

FTP file download with urllib

```
import urllib
ftp_url = 'ftp://ftp.cea.fr/pub/unati/people/educhesnay/pystatml/README.md'
urllib.request.urlretrieve(ftp_url, os.path.join(tmpdir, "README2.md"))
```

```
(' /tmp/README2.md', <email.message.Message object at 0x7a05a10ea490>)
```

2.10.2 HTTP

```
# TODO
```

2.10.3 Sockets

```
# TODO
```

2.10.4 xmlrpc

```
# TODO
```

2.11 Object Oriented Programming (OOP)

Sources

- http://python-textbok.readthedocs.org/en/latest/Object_Oriented_Programming.html

Principles

- **Encapsulate** data (attributes) and code (methods) into objects.
- **Class** = template or blueprint that can be used to create objects.
- An **object** is a specific instance of a class.
- **Inheritance**: OOP allows classes to inherit commonly used state and behavior from other classes. Reduce code duplication
- **Polymorphism**: (usually obtained through polymorphism) calling code is agnostic as to whether an object belongs to a parent class or one of its descendants (abstraction, modularity). The same method called on 2 objects of 2 different classes will behave differently.

```

class Shape2D:
    def area(self):
        raise NotImplementedError()

# __init__ is a special method called the constructor

# Inheritance + Encapsulation
class Square(Shape2D):
    def __init__(self, width):
        self.width = width

    def area(self):
        return self.width ** 2

class Disk(Shape2D):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return math.pi * self.radius ** 2

```

Object creation

```
square = Square(2)
```

Call a method of the object

```
square.area()
```

```
4
```

More sophisticated use

```

shapes = [Square(2), Disk(3)]

# Polymorphism
print([s.area() for s in shapes])

s = Shape2D()
try:
    s.area()
except NotImplementedError as e:
    print("NotImplementedError", e)

```

```

[4, 28.274333882308138]
NotImplementedError

```

2.12 Style guide for Python programming

See [PEP 8](#)

- Spaces (four) are the preferred indentation method.
- Two blank lines for top level function or classes definition.
- One blank line to indicate logical sections.
- Never use: `from lib import *`
- Bad: `Capitalized_Words_With_Underscores`
- Function and Variable Names: `lower_case_with_underscores`
- Class Names: `CapitalizedWords` (aka: CamelCase)

2.13 Documenting

See [Documenting Python](#) Documenting = comments + docstrings (Python documentation string)

- [Docstrings](#) are use as documentation for the class, module, and packages. See it as “living documentation”.
- Comments are used to explain non-obvious portions of the code. “Dead documentation”.

Docstrings for functions (same for classes and methods):

```
def my_function(a, b=2):
    """
    This function ...

    Parameters
    -----
    a : float
        First operand.
    b : float, optional
        Second operand. The default is 2.

    Returns
    -----
    Sum of operands.

    Example
    -----
    >>> my_function(3)
    5
    """
    # Add a with b (this is a comment)
    return a + b

print(help(my_function))
```

Help on function my_function in module __main__:

```
my_function(a, b=2)
    This function ...
```

Parameters

a : float

First operand.

b : float, optional

Second operand. The default is 2.

Returns

Sum of operands.

Example

```
>>> my_function(3)
5
```

None

Docstrings for scripts:

At the begining of a script add a pream:

```
"""
Created on Thu Nov 14 12:08:41 CET 2019

@author: firstname.lastname@email.com

Some description
"""
```

2.14 Modules and packages

Python [packages](#) and [modules](#) structure python code into modular “libraries” to be shared.

2.14.1 Package

Packages are a way of structuring Python’s module namespace by using “dotted module names”. A package is a directory (here, `stat_pkg`) containing a `__init__.py` file.

Example, package

```
stat_pkg/
├── __init__.py
└── datasets_mod.py
```

The `__init__.py` can be empty. Or it can be used to define the package API, i.e., the modules (*.py files) that are exported and those that remain internal.

Example, file `stat_pkg/__init__.py`

```
# 1) import function for modules in the packages
from .module import make_regression

# 2) Make them visible in the package
__all__ = ["make_regression"]
```

2.14.2 Module

A module is a python file. Example, `stat_pkg/datasets_mod.py`

```
import numpy as np
def make_regression(n_samples=10, n_features=2, add_intercept=False):
    ...
    return X, y, coef
```

Usage

```
import stat_pkg as pkg

X, y, coef = pkg.make_regression()
print(X.shape)
```

```
(10, 2)
```

2.14.3 The search path

With a directive like `import stat_pkg`, Python will search for

- a module, file named `stat_pkg.py` or,
- a package, directory named `stat_pkg` containing a `stat_pkg/__init__.py` file.

Python will search in a list of directories given by the variable `sys.path`. This variable is initialized from these locations:

- The directory containing the input script (or the current directory when no file is specified).
- ``PYTHONPATH`` (a list of directory names, with the same syntax as the shell variable `PATH`).

In our case, to be able to import `stat_pkg`, the parent directory of `stat_pkg` must be in `sys.path`. You can modify `PYTHONPATH` by any method, or access it via `sys` package, example:

```
import sys
sys.path.append("/home/ed203246/git/pystatsml/python_lang")
```

2.15 Unit testing

When developing a library (e.g., a python package) that is bound to evolve and being corrected, we want to ensure that: (i) The code correctly implements some expected functionalities; (ii) the modifications and additions don't break those functionalities;

Unit testing is a framework to assess those two points. See sources:

- [Unit testing reference doc](#)
- [Getting Started With Testing in Python](#)

2.15.1 unittest: test your code

1) Write unit tests (test cases)

In a directory usually called `tests` create a [test case](#), i.e., a python file `test_datasets_mod.py` (general syntax is `test_<my module>.py`) that will execute some functionalities of the module and test if the output are as expected. `test_datasets_mod.py` file contains specific directives:

- `import unittest,`
- `class TestDatasets(unittest.TestCase),` the test case class. The general syntax is `class Test<MyModule>(unittest.TestCase)`
- `def test_make_regression(self),` test a function of an element of the module. The general syntax is `test_<my function>(self)`
- `self.assertTrue(np.allclose(X.shape, (10, 4))),` test a specific functionality. The general syntax is `self.assert<True|Equal|...>(<some boolean expression>)`
- `unittest.main(),` where tests should be executed.

Example:

```
import unittest
import numpy as np
from stat_pkg import make_regression

class TestDatasets(unittest.TestCase):

    def test_make_regression(self):
        X, y, coefs = make_regression(n_samples=10, n_features=3,
                                     add_intercept=True)
        self.assertTrue(np.allclose(X.shape, (10, 4)))
        self.assertTrue(np.allclose(y.shape, (10, )))
        self.assertTrue(np.allclose(coefs.shape, (4, )))

if __name__ == '__main__':
    unittest.main()
```

2) Run the tests (test runner)

The [test runner](#) orchestrates the execution of tests and provides the outcome to the user. Many [test runners](#) are available.

`unittest` is the first unit test framework, it comes with Python standard library. It employs an object-oriented approach, grouping tests into classes known as test cases, each containing distinct methods representing individual tests.

Unittest generally requires that tests are organized as importable modules, [see details](#). Here, we do not introduce this complexity: we directly execute a test file that isn't importable as a module.

```
python tests/test_datasets_mod.py
```

Unittest test discovery: (`-m unittest discover`) within (`-s`) tests directory, with verbose (`-v`) outputs.

```
python -m unittest discover -v -s tests
```

2.15.2 Doctest: add unit tests in docstring

Doctest is an inbuilt test framework that comes bundled with Python by default. The doctest module searches for code fragments that resemble interactive Python sessions and runs those sessions to confirm they operate as shown. It promotes **Test-driven (TDD) methodology**.

1) Add doc test in the docstrings, see `python stat_pkg/supervised_models.py`:

```
class LinearRegression:
    """Ordinary least squares Linear Regression.
    ...
    Examples
    -----
    >>> import numpy as np
    >>> from stat_pkg import LinearRegression
    >>> X = np.array([[1, 1], [1, 2], [2, 2], [2, 3]])
    >>> # y = 1 * x_0 + 2 * x_1 + 3
    >>> y = np.dot(X, np.array([1, 2])) + 3
    >>> reg = LinearRegression().fit(X, y)
    >>> reg.coef_
    array([3., 1., 2.0])
    >>> reg.predict(np.array([[3, 5]]))
    array([16.])
    """
    def __init__(self, fit_intercept=True):
        self.fit_intercept = fit_intercept
    ...
```

2) Add the call to doctest module at the end of the python file:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

3) Run doc tests:

```
python stat_pkg/supervised_models.py
```

Test failed with the output:

```
*****
File ".../supervised_models.py", line 36, in __main__.LinearRegression
Failed example:
    reg.coef_
```

(continues on next page)

(continued from previous page)

```

Expected:
    array([3., 1., 2.0])
Got:
    array([3., 1., 2.])
*****
1 items had failures:
  1 of  7 in __main__.LinearRegression
***Test Failed*** 1 failures.

```

2.16 Exercises

2.16.1 Exercise 1: functions

Create a function that acts as a simple calculator taking three parameters: the two operand and the operation in “+”, “-”, and “*”. As default use “+”. If the operation is misspecified, return a error message Ex: `calc(4,5,"*")` returns 20 Ex: `calc(3,5)` returns 8 Ex: `calc(1, 2, "something")` returns error message

2.16.2 Exercise 2: functions + list + loop

Given a list of numbers, return a list where all adjacent duplicate elements have been reduced to a single element. Ex: `[1, 2, 2, 3, 2]` returns `[1, 2, 3, 2]`. You may create a new list or modify the passed in list.

Remove all duplicate values (adjacent or not) Ex: `[1, 2, 2, 3, 2]` returns `[1, 2, 3]`

2.16.3 Exercise 3: File I/O

1. Copy/paste the BSD 4 clause license (https://en.wikipedia.org/wiki/BSD_licenses) into a text file. Read, the file and count the occurrences of each word within the file. Store the words' occurrence number in a dictionary.

2. Write an executable python command `count_words.py` that parse a list of input files provided after `--input` parameter. The dictionary of occurrence is save in a csv file provides by `--output`. with default value `word_count.csv`. Use: - open - regular expression - argparse (<https://docs.python.org/3/howto/argparse.html>)

2.16.4 Exercise 4: OOP

1. Create a class `Employee` with 2 attributes provided in the constructor: `name`, `years_of_service`. With one method `salary` with is obtained by $1500 + 100 * \text{years\_of\_service}$.
2. Create a subclass `Manager` which redefine `salary` method $2500 + 120 * \text{years\_of\_service}$.
3. Create a small dictionary-nosed database where the key is the employee's name. Populate the database with: `samples = Employee('lucy', 3), Employee('john', 1), Manager('julie', 10), Manager('paul', 3)`
4. Return a table of made name, salary rows, i.e. a list of list `[[name, salary]]`
5. Compute the average salary

Total running time of the script: (0 minutes 8.467 seconds)

DATA MANIPULATION AND VISUALIZATION

3.1 Numpy: Arrays and Matrices

NumPy is an extension to the Python programming language, adding support for large, multi-dimensional (numerical) arrays and matrices, along with a large library of high-level mathematical functions to operate on these arrays.

NumPy functions are executed by **compiled in C or Fortran libraries**, providing the performance of compiled languages.

Sources: [Kevin Markham](#)

Computation time:

```
import numpy as np
import time

start_time = time.time()
l = [v for v in range(10 ** 8)]
s = 0
for v in l: s += v
print("Python code, time elapsed: %.2fs" % (time.time() - start_time))

start_time = time.time()
arr = np.arange(10 ** 8)
arr.sum()
print("Numpy code, time elapsed: %.2fs" % (time.time() - start_time))
```

```
Python code, time elapsed: 7.72s
Numpy code, time elapsed: 0.56s
```

3.1.1 Create arrays

Create ndarrays from lists. note: every element must be the same type (will be converted if possible)

```
data1 = [1, 2, 3, 4, 5]           # list
arr = np.array(data1)             # 1d array
data = [range(1, 5), range(5, 9)] # list of lists
arr = np.array(data)              # 2d array
```

(continues on next page)

(continued from previous page)

```
print(arr)
arr.tolist()           # convert array back to list
```

```
[[1 2 3 4]
 [5 6 7 8]]

[[1, 2, 3, 4], [5, 6, 7, 8]]
```

Create special arrays

```
np.zeros(10)           # [0, 0, ..., 0]
np.zeros((3, 6))       # 3 x 6 array of zeros
np.ones(10)
np.linspace(0, 1, 5)    # 0 to 1 (inclusive) with 5 points
np.logspace(0, 3, 4)    # 10^0 to 10^3 (inclusive) with 4 points
np.arange(10)           # [0, 1 ..., 9]
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Examining arrays

```
print("Shape of the array: ",
      arr.shape)

print("Type of the array: ",
      arr.dtype)

print("Number of items in the array: ",
      arr.size)

print("Memory size of one array item in bytes: ",
      arr.itemsize)

# memory size of numpy array in bytes
print("Memory size of numpy array in bytes: %i, and in bits: %i" %
      (arr.size * arr.itemsize, arr.size * arr.itemsize * 8 ))
```

```
Shape of the array: (2, 4)
Type of the array: int64
Number of items in the array: 8
Memory size of one array item in bytes: 8
Memory size of numpy array in bytes: 64, and in bits: 512
```

3.1.2 Selection

```
arr[1, 2] # Get third item of the second line
```

```
np.int64(7)
```

Slicing

Syntax: start:stop:step with start (default 0) stop (default last) step (default 1)

- : is equivalent to 0:last:1; ie, take all elements, from 0 to the end with step = 1.
- :k is equivalent to 0:k:1; ie, take all elements, from 0 to k with step = 1.
- k: is equivalent to k:end:1; ie, take all elements, from k to the end with step = 1.
- ::-1 is equivalent to 0:end:-1; ie, take all elements, from k to the end in reverse order, with step = -1.

```
arr[0, :] # Get first line
arr[:, 2] # Get third column
arr[:, :2]    # columns strictly before index 2 (2 first columns)
arr[:, 2:]    # columns after index 2 included
arr2 = arr[:, 1:4] # columns between index 1 (included) and 4 (excluded)
print(arr2)

# Slicing returns a view (not a copy)
# Modification

arr2[0, 0] = 33
print(arr2)
print(arr)
```

```
[[2 3 4]
 [6 7 8]]
[[33 3 4]
 [ 6 7 8]]
[[ 1 33 3 4]
 [ 5 6 7 8]]
```

Reverse order of row 0

```
print(arr[0, ::-1])
```

```
[ 4 3 33 1]
```

Fancy indexing: Integer or boolean array indexing

Fancy indexing returns a copy not a view.

Integer array indexing

```
arr2 = arr[:, [1, 2, 3]] # return a copy
print(arr2)
arr2[0, 0] = 44
print(arr2)
print(arr)
```

```
[[33  3  4]
 [ 6  7  8]]
[[44  3  4]
 [ 6  7  8]]
[[ 1 33  3  4]
 [ 5  6  7  8]]
```

Boolean arrays indexing

```
arr2 = arr[arr > 5] # return a copy

print(arr2)
arr2[0] = 44
print(arr2)
print(arr)
```

```
[33  6  7  8]
[44  6  7  8]
[[ 1 33  3  4]
 [ 5  6  7  8]]
```

However, In the context of lvalue indexing (left hand side value of an assignment) Fancy authorizes the modification of the original array

```
arr[arr > 5] = 0
print(arr)
```

```
[[1 0 3 4]
 [5 0 0 0]]
```

Array indexing return copy or view?

General rules:

- Slicing always returns a view.
- Fancy indexing (boolean mask, integers) returns copy
- lvalue indexing i.e. the indices are placed in the left hand side value of an assignment, provides a view.

3.1.3 Array manipulation

Reshaping

```
arr = np.arange(10, dtype=float).reshape((2, 5))
print(arr.shape)
print(arr.reshape(5, 2))
```

```
(2, 5)
[[0.  1.]
 [2.  3.]
```

(continues on next page)

(continued from previous page)

```
[4. 5.]
[6. 7.]
[8. 9.]]
```

Add an axis

```
a = np.array([0, 1])
print(a)
a_col = a[:, np.newaxis]
print(a_col)
#or
a_col = a[:, None]
```

```
[0 1]
[[0]
 [1]]
```

Transpose

```
print(a_col.T)
```

```
[[0 1]]
```

Flatten: always returns a flat copy of the original array

```
arr_flat = arr.flatten()
arr_flat[0] = 33
print(arr_flat)
print(arr)
```

```
[33.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
[[0.  1.  2.  3.  4.]
 [5.  6.  7.  8.  9.]]
```

Ravel: returns a view of the original array whenever possible.

```
arr_flat = arr.ravel()
arr_flat[0] = 33
print(arr_flat)
print(arr)
```

```
[33.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
[[33.  1.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]]
```

Stack arrays [NumPy Joining Array](#)

```
a = np.array([0, 1])
b = np.array([2, 3])
```

Horizontal stacking

```
np.hstack([a, b])
```

```
array([0, 1, 2, 3])
```

Vertical stacking

```
np.vstack([a, b])
```

```
array([[0, 1],  
       [2, 3]])
```

Default Vertical

```
np.stack([a, b])
```

```
array([[0, 1],  
       [2, 3]])
```

3.1.4 Advanced Numpy: reshaping/flattening and selection

Numpy internals: By default Numpy use C convention, ie, Row-major language: The matrix is stored by rows. In C, the last index changes most rapidly as one moves through the array as stored in memory.

For 2D arrays, sequential move in the memory will:

- **iterate over rows (axis 0)**
 - iterate over columns (axis 1)

For 3D arrays, sequential move in the memory will:

- **iterate over plans (axis 0)**
 - **iterate over rows (axis 1)**
 - * iterate over columns (axis 2)

```
x = np.arange(2 * 3 * 4)  
print(x)
```

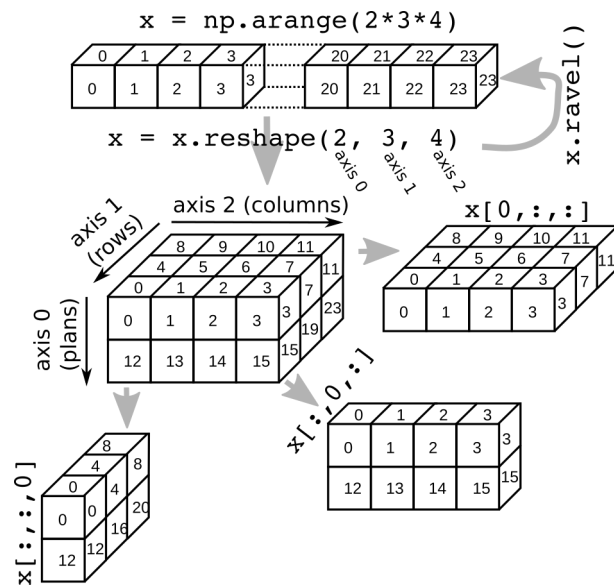
```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
```

Reshape into 3D (axis 0, axis 1, axis 2)

```
x = x.reshape(2, 3, 4)  
print(x)
```

```
[[[ 0  1  2  3]  
  [ 4  5  6  7]  
  [ 8  9 10 11]]]
```

(continues on next page)



(continued from previous page)

```
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

Selection get first plan

```
print(x[0, :, :])
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

Selection get first rows

```
print(x[:, 0, :])
```

```
[[ 0  1  2  3]
 [12 13 14 15]]
```

Selection get first columns

```
print(x[:, :, 0])
```

```
[[ 0  4  8]
 [12 16 20]]
```

Ravel

```
print(x.ravel())
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
```

3.1.5 Vectorized operations

```

nums = np.arange(5)
nums * 10                # multiply each element by 10
nums = np.sqrt(nums)     # square root of each element
np.ceil(nums)            # also floor, rint (round to nearest int)
np.isnan(nums)           # checks for NaN
nums + np.arange(5)      # add element-wise
np.maximum(nums, np.array([1, -2, 3, -4, 5])) # compare element-wise

# Compute Euclidean distance between 2 vectors
vec1 = np.random.randn(10)
vec2 = np.random.randn(10)
dist = np.sqrt(np.sum((vec1 - vec2) ** 2))

# math and stats
rnd = np.random.randn(4, 2) # random normals in 4x2 array
rnd.mean()
rnd.std()
rnd.argmax()              # index of minimum element
rnd.sum()
rnd.sum(axis=0)           # sum of columns
rnd.sum(axis=1)           # sum of rows

# methods for boolean arrays
(rnd > 0).sum()           # counts number of positive values
(rnd > 0).any()           # checks if any value is True
(rnd > 0).all()           # checks if all values are True

# random numbers
np.random.seed(12234)     # Set the seed
np.random.rand(2, 3)      # 2 x 3 matrix in [0, 1]
np.random.randn(10)       # random normals (mean 0, sd 1)
np.random.randint(0, 2, 10) # 10 randomly picked 0 or 1

```

```
array([0, 0, 0, 1, 1, 0, 1, 1, 1, 1])
```

3.1.6 Broadcasting

Sources: <https://docs.scipy.org/doc/numpy-1.13.0/user/basics.broadcasting.html> Implicit conversion to allow operations on arrays of different sizes. - The smaller array is stretched or “broadcasted” across the larger array so that they have compatible shapes. - Fast vectorized operation in C instead of Python. - No needless copies.

Rules

Starting with the trailing axis and working backward, Numpy compares arrays dimensions.

- If two dimensions are equal then continues
- If one of the operand has dimension 1 stretches it to match the largest one
- When one of the shapes runs out of dimensions (because it has less dimensions than

the other shape), Numpy will use 1 in the comparison process until the other shape's dimensions run out as well.

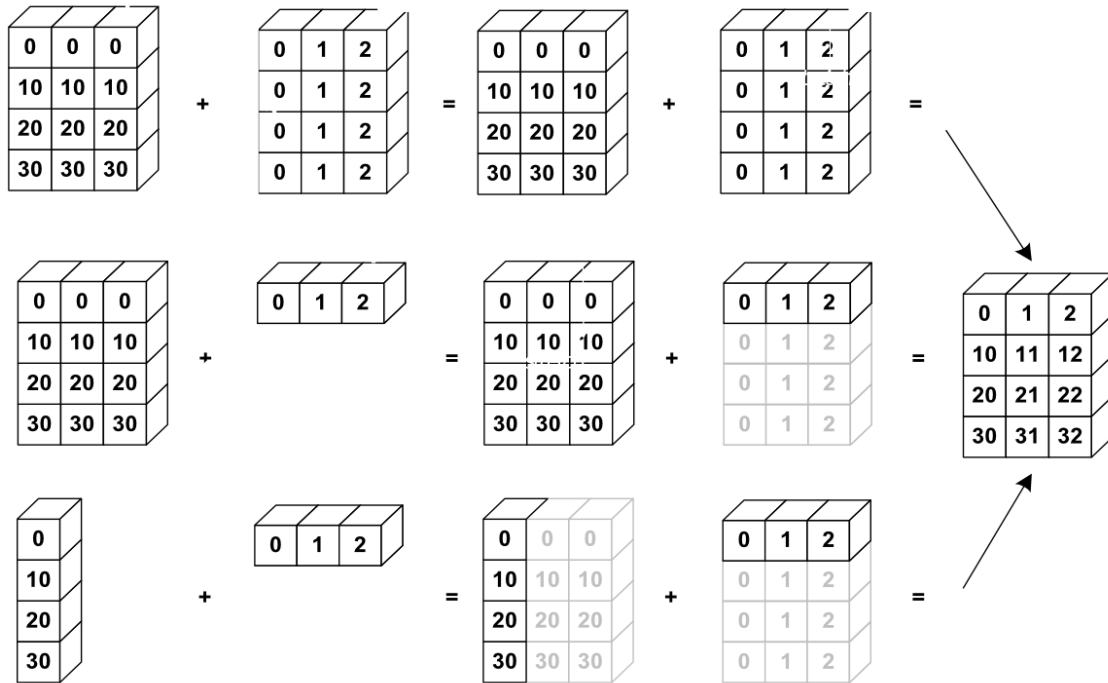


Fig. 1: Source: <http://www.scipy-lectures.org>

```
a = np.array([[ 0,  0,  0],
              [10, 10, 10],
              [20, 20, 20],
              [30, 30, 30]])

b = np.array([0, 1, 2])

print(a + b)
```

```
[[ 0  1  2]
 [10 11 12]
 [20 21 22]
 [30 31 32]]
```

Center data column-wise

```
a - a.mean(axis=0)
```

```
array([[ -15.,  -15.,  -15.],
       [  -5.,   -5.,   -5.]])
```

(continues on next page)

(continued from previous page)

```
[ 5.,  5.,  5.],  
[15., 15., 15.]])
```

Scale (center, normalise) data column-wise

```
(a - a.mean(axis=0)) / a.std(axis=0)
```

```
array([[ -1.34164079,  -1.34164079,  -1.34164079],  
       [-0.4472136 , -0.4472136 , -0.4472136 ],  
       [ 0.4472136 ,  0.4472136 ,  0.4472136 ],  
       [ 1.34164079,  1.34164079,  1.34164079]])
```

Examples

Shapes of operands A, B and result:

```
A      (2d array):  5 x 4  
B      (1d array):    1  
Result (2d array):  5 x 4
```

```
A      (2d array):  5 x 4  
B      (1d array):    4  
Result (2d array):  5 x 4
```

```
A      (3d array): 15 x 3 x 5  
B      (3d array): 15 x 1 x 5  
Result (3d array): 15 x 3 x 5
```

```
A      (3d array): 15 x 3 x 5  
B      (2d array):   3 x 5  
Result (3d array): 15 x 3 x 5
```

```
A      (3d array): 15 x 3 x 5  
B      (2d array):   3 x 1  
Result (3d array): 15 x 3 x 5
```

3.1.7 Exercises

Given the array:

```
X = np.random.randn(4, 2) # random normals in 4x2 array
```

- For each column find the row index of the minimum value.
- Write a function `standardize(X)` that return an array whose columns are centered and scaled (by std-dev).

Total running time of the script: (0 minutes 8.314 seconds)

3.2 Pandas: data manipulation

It is often said that 80% of data analysis is spent on the cleaning and small, but important, aspect of data manipulation and cleaning with Pandas.

Sources:

- Kevin Markham: <https://github.com/justmarkham>
- Pandas doc: <http://pandas.pydata.org/pandas-docs/stable/index.html>

Data structures

- **Series** is a one-dimensional labeled array capable of holding any data type (integers, strings, floating point numbers, Python objects, etc.). The axis labels are collectively referred to as the index. The basic method to create a Series is to call `pd.Series([1,3,5,np.nan,6,8])`
- **DataFrame** is a 2-dimensional labeled data structure with columns of potentially different types. You can think of it like a spreadsheet or SQL table, or a dict of Series objects. It stems from the *R* `data.frame()` object.

```
import pandas as pd
import numpy as np
```

3.2.1 The pandas library at a glance

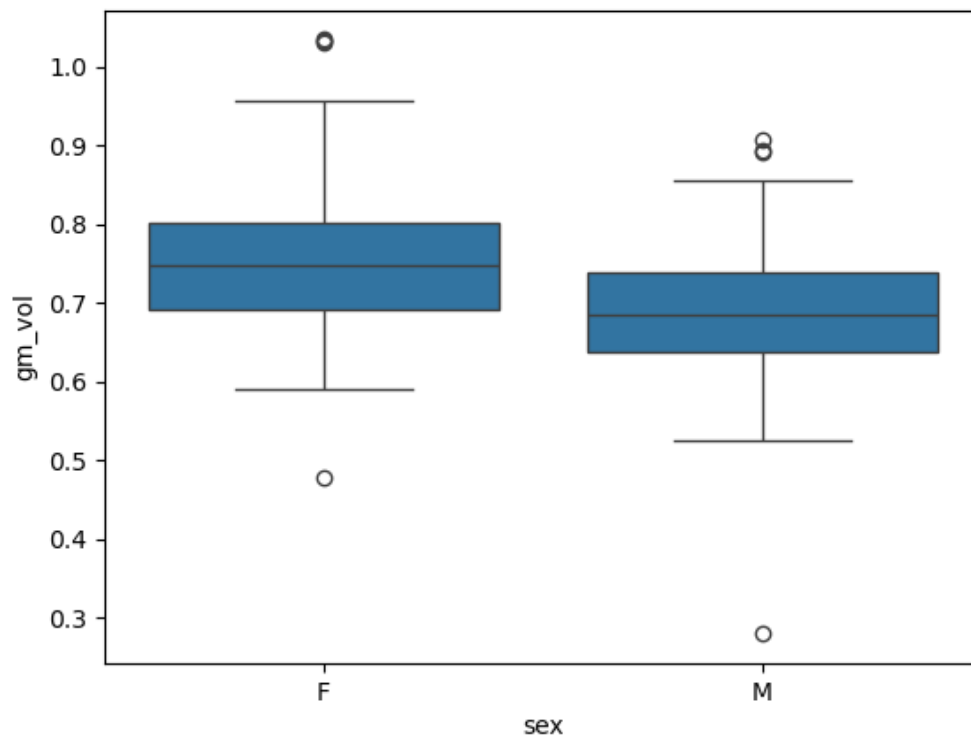
```
data = pd.read_excel("../datasets/brain_volumes/brain_volumes.xlsx")
data.columns

import seaborn as sns
import matplotlib.pyplot as plt

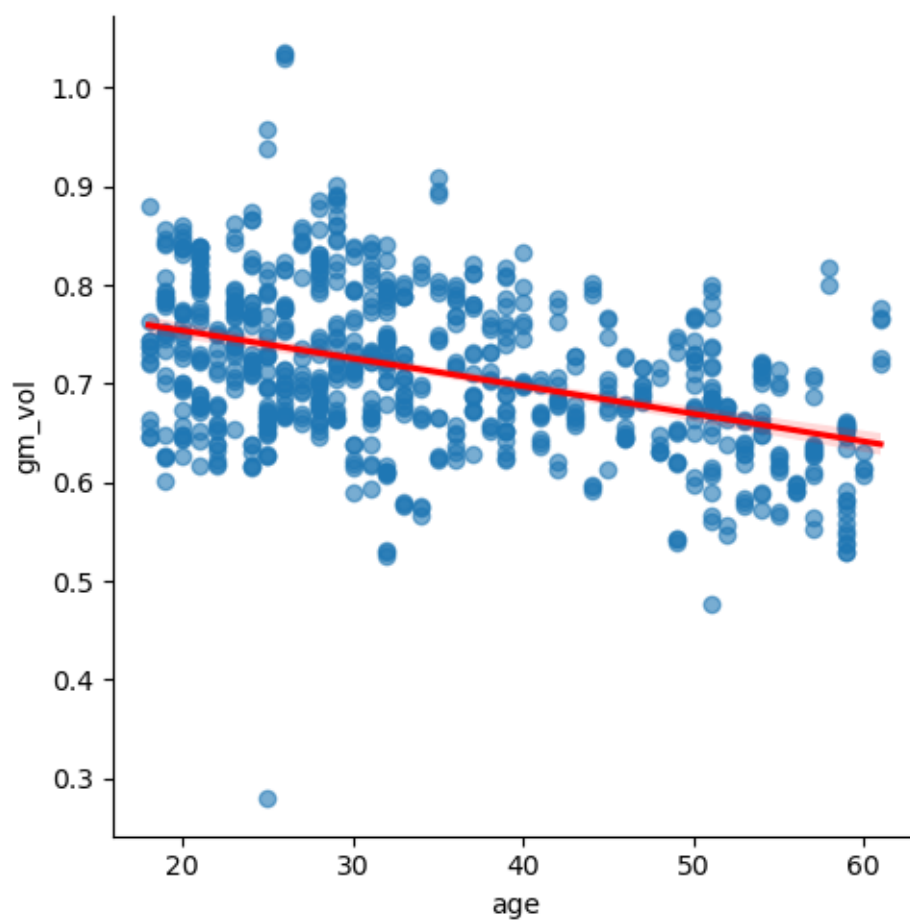
# Boxplot of gm_vol by sex
sns.boxplot(x="sex", y="gm_vol", data=data)

# Scatterplot with regression line
sns.lmplot(x="age", y="gm_vol", data=data,
           scatter_kws={"alpha":0.6},    # transparency for points
           line_kws={"color":"red"})    # regression line color

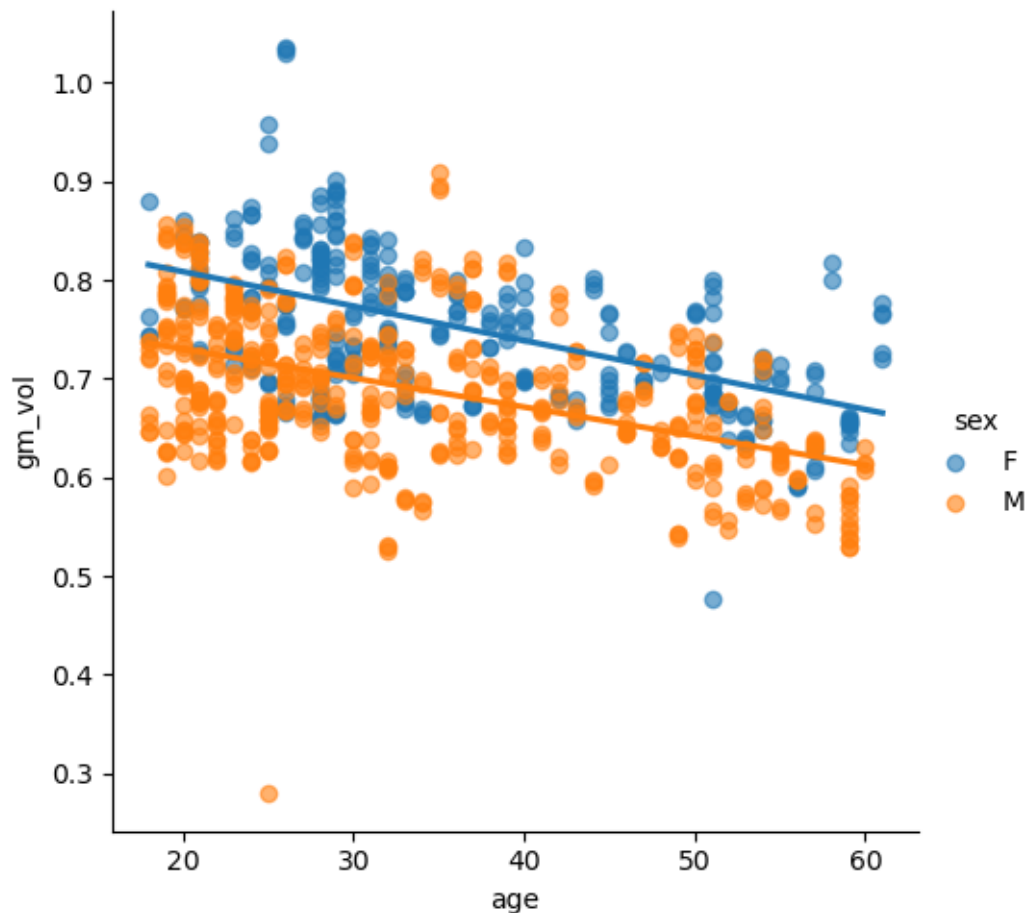
sns.lmplot(
    x="age", y="gm_vol",
    hue="sex",          # couleurs par sexe
    data=data,
    scatter_kws={"alpha": 0.6}, # transparence des points
    ci=None           # supprime l'intervalle de confiance (optionnel)
)
```



•



•



```
<seaborn.axisgrid.FacetGrid object at 0x71840437b390>
```

3.2.2 Create DataFrame

```
columns = ['name', 'age', 'gender', 'job']

user1 = pd.DataFrame([['alice', 19, "F", "student"],
                      ['john', 26, "M", "student"]],
                     columns=columns)

user2 = pd.DataFrame([['eric', 22, "M", "student"],
                      ['paul', 58, "F", "manager"]],
                     columns=columns)

user3 = pd.DataFrame(dict(name=['peter', 'julie'],
                          age=[33, 44], gender=['M', 'F'],
                          job=['engineer', 'scientist']))

print(user3)
```

name	age	gender	job
peter	33	M	engineer
julie	44	F	scientist

(continues on next page)

(continued from previous page)

```
0 peter 33 M engineer
1 julie 44 F scientist
```

3.2.3 Combining DataFrames

Concatenate DataFrame

Concatenate columns (axis = 1).

```
height = pd.DataFrame(dict(height=[1.65, 1.8]))
print(user1, "\n", height)

print(pd.concat([user1, height], axis=1))
```

```
   name  age gender  job
0  alice   19     F student
1   john   26     M student
   height
0    1.65
1    1.80
   name  age gender  job  height
0  alice   19     F student    1.65
1   john   26     M student    1.80
```

Concatenate rows (default: axis = 0)

```
users = pd.concat([user1, user2, user3])
print(users)
```

```
   name  age gender  job
0  alice   19     F student
1   john   26     M student
0   eric   22     M student
1   paul   58     F manager
0  peter   33     M engineer
1  julie   44     F scientist
```

Join DataFrame

```
user4 = pd.DataFrame(dict(name=['alice', 'john', 'eric', 'julie'],
                           height=[165, 180, 175, 171]))
print(user4)
```

```
   name  height
0  alice    165
1   john    180
2   eric    175
3  julie    171
```

Use intersection of keys from both frames

```
merge_inter = pd.merge(users, user4)
print(merge_inter)
```

	name	age	gender	job	height
0	alice	19	F	student	165
1	john	26	M	student	180
2	eric	22	M	student	175
3	julie	44	F	scientist	171

Use union of keys from both frames

```
users = pd.merge(users, user4, on="name", how='outer')
print(users)
```

	name	age	gender	job	height
0	alice	19	F	student	165.0
1	eric	22	M	student	175.0
2	john	26	M	student	180.0
3	julie	44	F	scientist	171.0
4	paul	58	F	manager	NaN
5	peter	33	M	engineer	NaN

Reshaping by pivoting

“Unpivots” a DataFrame from wide format to long (stacked) format,

```
staked = pd.melt(users, id_vars="name", var_name="variable", value_name="value")
print(staked)
```

	name	variable	value
0	alice	age	19
1	eric	age	22
2	john	age	26
3	julie	age	44
4	paul	age	58
5	peter	age	33
6	alice	gender	F
7	eric	gender	M
8	john	gender	M
9	julie	gender	F
10	paul	gender	F
11	peter	gender	M
12	alice	job	student
13	eric	job	student
14	john	job	student
15	julie	job	scientist
16	paul	job	manager
17	peter	job	engineer
18	alice	height	165.0
19	eric	height	175.0

(continues on next page)

(continued from previous page)

```

20  john  height    180.0
21  julie height    171.0
22  paul  height      NaN
23  peter height      NaN

```

“pivots” a DataFrame from long (stacked) format to wide format,

```

wide = staked.pivot(index='name', columns='variable', values='value')
print(wide)

```

```

variable age gender height    job
name
alice     19      F  165.0  student
eric     22      M  175.0  student
john     26      M  180.0  student
julie    44      F  171.0  scientist
paul     58      F   NaN   manager
peter    33      M   NaN   engineer

```

3.2.4 Summarizing

```

users                # print the first 30 and last 30 rows
type(users)          # DataFrame
users.head()          # print the first 5 rows
users.tail()          # print the last 5 rows

```

Meta-information

```

users.columns         # Column names
users.index           # Row name"
users.shape           # number of rows and columns
users.dtypes          # data types of each column
users.values          # underlying numpy array

```

```

array([[ 'alice', 19, 'F', 'student', 165.0],
       [ 'eric', 22, 'M', 'student', 175.0],
       [ 'john', 26, 'M', 'student', 180.0],
       [ 'julie', 44, 'F', 'scientist', 171.0],
       [ 'paul', 58, 'F', 'manager', nan],
       [ 'peter', 33, 'M', 'engineer', nan]], dtype=object)

```

3.2.5 Columns selection

```

print(users.columns)

users['gender']        # select one column
type(users['gender']) # Series
users.gender           # select one column using the DataFrame

```

(continues on next page)

(continued from previous page)

```
# select multiple columns
users[['age', 'gender']]      # select two columns
my_cols = ['age', 'gender']  # or, create a list...
users[my_cols]               # ...and use that list to select columns
type(users[my_cols])         # DataFrame
```

```
Index(['name', 'age', 'gender', 'job', 'height'], dtype='object')
```

iloc is strictly integer position based

```
users.iloc[:, 2] # select third column
```

```
0    F
1    M
2    M
3    F
4    F
5    M
Name: gender, dtype: object
```

3.2.6 Rows selection (basic)

iloc is strictly integer position based

```
df = users.copy()
df.iloc[0]      # first row
df.iloc[0, :]   # first row
df.iloc[[0, 1], :] # Two first row

df.iloc[0, 0]   # first item of first row
df.iloc[0, 0] = 55
```

loc supports mixed integer and label based access.

```
df.loc[0]      # first row
df.loc[0, :]   # first row
df.loc[0, "age"] # age item of first row
df.loc[0, "age"] = 55
```

Selection and index

Select females into a new DataFrame

```
df = users[users.gender == "F"]
print(df)
```

```
   name  age gender    job  height
0  alice   19     F  student  165.0
```

(continues on next page)

(continued from previous page)

3	julie	44	F	scientist	171.0
4	paul	58	F	manager	NaN

Reset index, useful when index is meaningless

```
df = df.reset_index(drop=True) # Watch the index
print(df)
```

	name	age	gender	job	height
0	alice	19	F	student	165.0
1	julie	44	F	scientist	171.0
2	paul	58	F	manager	NaN

3.2.7 Rows iteration

```
df = users[:2].copy()
```

iterrows(): slow, get series, **read-only**

- Returns (index, Series) pairs.
- Slow because *iterrows* boxes the data into a Series.
- Retrieve fields with column name
- **Don't modify something you are iterating over.** Depending on the data types, the iterator returns a copy and not a view, and writing to it will have no effect.

```
for idx, row in df.iterrows():
    print(row["name"], row["age"])
```

```
alice 19
eric 22
```

itertuples(): fast, get namedtuples, **read-only**

- Returns namedtuples of the values and which is generally faster than *iterrows*.
- Fast, because *itertuples* does not box the data into a Series.
- Retrieve fields with integer index starting from 0.
- Names will be renamed to positional names if they are invalid Python identifier

```
for tup in df.itertuples():
    print(tup[1], tup[2])
```

```
alice 19
eric 22
```

iter using *loc[i, ...]*: read and **write**

```
for i in range(df.shape[0]):
    df.loc[i, "age"] *= 10 # df is modified
```

3.2.8 Rows selection (filtering)

simple logical filtering on numerical values

```
users[users.age < 20]          # only show users with age < 20
young_bool = users.age < 20   # or, create a Series of booleans...
young = users[young_bool]     # ...and use that Series to filter rows
users[users.age < 20].job     # select one column from the filtered results
print(young)
```

	name	age	gender	job	height
0	alice	19	F	student	165.0

simple logical filtering on categorical values

```
users[users.job == 'student']
users[users.job.isin(['student', 'engineer'])]
users[users['job'].str.contains("stu|scient")]
```

Advanced logical filtering

```
users[users.age < 20][['age', 'job']]          # select multiple columns
users[(users.age > 20) & (users.gender == 'M')] # use multiple conditions
```

3.2.9 Sorting

```
df = users.copy()

df.age.sort_values()          # only works for a Series
df.sort_values(by='age')      # sort rows by a specific column
df.sort_values(by='age', ascending=False) # use descending order instead
df.sort_values(by=['job', 'age']) # sort by multiple columns
df.sort_values(by=['job', 'age'], inplace=True) # modify df

print(df)
```

	name	age	gender	job	height
5	peter	33	M	engineer	NaN
4	paul	58	F	manager	NaN
3	julie	44	F	scientist	171.0
0	alice	19	F	student	165.0
1	eric	22	M	student	175.0
2	john	26	M	student	180.0

3.2.10 Descriptive statistics

Summarize all numeric columns

```
print(df.describe())
```

	age	height
count	6.000000	4.000000
mean	33.666667	172.750000
std	14.895189	6.344289
min	19.000000	165.000000
25%	23.000000	169.500000
50%	29.500000	173.000000
75%	41.250000	176.250000
max	58.000000	180.000000

Summarize all columns

```
print(df.describe(include='all'))
print(df.describe(include=['object'])) # limit to one (or more) types
```

	name	age	gender	job	height
count	6	6.000000	6	6	4.000000
unique	6	NaN	2	4	NaN
top	peter	NaN	M	student	NaN
freq	1	NaN	3	3	NaN
mean	NaN	33.666667	NaN	NaN	172.750000
std	NaN	14.895189	NaN	NaN	6.344289
min	NaN	19.000000	NaN	NaN	165.000000
25%	NaN	23.000000	NaN	NaN	169.500000
50%	NaN	29.500000	NaN	NaN	173.000000
75%	NaN	41.250000	NaN	NaN	176.250000
max	NaN	58.000000	NaN	NaN	180.000000

	name	gender	job
count	6	6	6
unique	6	2	4
top	peter	M	student
freq	1	3	3

Categorical columns: count and proportions of values

```
df['job'].value_counts()
df['job'].value_counts(normalize=True).round(2)
```

```
job
student    0.50
engineer   0.17
manager    0.17
scientist   0.17
Name: proportion, dtype: float64
```

Categorical columns: length of strings


```
df['job'].str.len()
```

```
5    8
4    7
3    9
0    7
1    7
2    7
Name: job, dtype: int64
```

Statistics per group (groupby)

```
print(df.groupby("job")["age"].mean())
# print(df.groupby("job").describe(include='all'))
```

```
job
engineer    33.000000
manager     58.000000
scientist   44.000000
student     22.333333
Name: age, dtype: float64
```

Groupby in a loop

```
for grp, data in df.groupby("job"):
    print(grp, data)
```

```
engineer    name  age  gender    job  height
5  peter    33      M  engineer    NaN
manager    name  age  gender    job  height
4  paul    58      F  manager    NaN
scientist   name  age  gender    job  height
3  julie   44      F  scientist  171.0
student    name  age  gender    job  height
0  alice   19      F  student   165.0
1  eric    22      M  student   175.0
2  john    26      M  student   180.0
```

3.2.11 Quality check

Remove duplicate data

```
df = users.copy()

# Create a duplicate: Append the first at the end
df.loc[len(df.index)] = users.iloc[0]

print(df.duplicated())          # Series of booleans
# (True if a row is identical to a previous row)
df.duplicated().sum()           # count of duplicates
```

(continues on next page)

(continued from previous page)

```
df[df.duplicated()]          # only show duplicates
df.age.duplicated()          # check a single column for duplicates
df.duplicated(['age', 'gender']).sum() # specify columns for finding duplicates
df = df.drop_duplicates()    # drop duplicate rows
```

```
0    False
1    False
2    False
3    False
4    False
5    False
6     True
dtype: bool
```

Missing data

```
# Missing values are often just excluded
df = users.copy()

df.describe(include='all')

# find missing values in a Series
df.height.isnull()          # True if NaN, False otherwise
df.height.notnull()         # False if NaN, True otherwise
df[df.height.notnull()]     # only show rows where age is not NaN
df.height.isnull().sum()    # count the missing values

# find missing values in a DataFrame
df.isnull()                 # DataFrame of booleans
df.isnull().sum()           # calculate the sum of each column
```

```
name      0
age       0
gender    0
job       0
height    2
dtype: int64
```

Strategy 1: drop missing values

```
df.dropna()                 # drop a row if ANY values are missing
df.dropna(how='all')        # drop a row only if ALL values are missing
```

Strategy 2: fill in missing values

```
df.height.mean()
df = users.copy()
df.loc[df.height.isnull(), "height"] = df["height"].mean()
```

(continues on next page)

(continued from previous page)

```
print(df)
```

	name	age	gender	job	height
0	alice	19	F	student	165.00
1	eric	22	M	student	175.00
2	john	26	M	student	180.00
3	julie	44	F	scientist	171.00
4	paul	58	F	manager	172.75
5	peter	33	M	engineer	172.75

3.2.12 Operation: multiplication

Multiplication of dataframe and other, element-wise

```
df = users.dropna()
df.insert(0, 'random', np.arange(df.shape[0]))
print(df)
df[["age", "height"]].multiply(df["random"], axis="index")
```

	random	name	age	gender	job	height
0	0	alice	19	F	student	165.0
1	1	eric	22	M	student	175.0
2	2	john	26	M	student	180.0
3	3	julie	44	F	scientist	171.0

3.2.13 Renaming

Rename columns

```
df = users.copy()
df.rename(columns={'name': 'NAME'})
```

Rename values

```
df.job = df.job.map({'student': 'etudiant', 'manager': 'manager',
                    'engineer': 'ingenieur', 'scientist': 'scientific'})
```

3.2.14 Dealing with outliers

```
size = pd.Series(np.random.normal(loc=175, size=20, scale=10))
# Corrupt the first 3 measures
size[:3] += 500
```

Based on parametric statistics: use the mean

Assume random variable follows the normal distribution Exclude data outside 3 standard-deviations: - Probability that a sample lies within 1 sd: 68.27% - Probability that a sample lies within 3 sd: 99.73% ($68.27 + 2 * 15.73$)

```
size_outlr_mean = size.copy()
size_outlr_mean[((size - size.mean()).abs() > 3 * size.std())] = size.mean()
print(size_outlr_mean.mean())
```

```
250.98518980168814
```

Based on non-parametric statistics: use the median

Median absolute deviation (MAD) is based on the median, is a robust non-parametric statistics.

```
mad = 1.4826 * np.median(np.abs(size - size.median()))
size_outlr_mad = size.copy()

size_outlr_mad[((size - size.median()).abs() > 3 * mad)] = size.median()
print(size_outlr_mad.mean(), size_outlr_mad.median())
```

```
175.9778997443072 177.4650456862703
```

3.2.15 File I/O

csv

```
import tempfile, os.path

tmpdir = tempfile.gettempdir()
csv_filename = os.path.join(tmpdir, "users.csv")
users.to_csv(csv_filename, index=False)
other = pd.read_csv(csv_filename)
```

Read csv from url

```
url = 'https://github.com/duchesnay/pystatsml/raw/master/datasets/salary_table.csv'
↳
salary = pd.read_csv(url)
```

Excel

Package *openpyxl* is required. To install type:

```
conda install -c conda-forge openpyxl
```

```
xls_filename = os.path.join(tmpdir, "users.xlsx")

# Write
users.to_excel(xls_filename, sheet_name='users', index=False)

# Read
pd.read_excel(xls_filename, sheet_name='users')
```

(continues on next page)

(continued from previous page)

```
# Multiple sheets
with pd.ExcelWriter(xls_filename) as writer:
    users.to_excel(writer, sheet_name='users', index=False)
    df.to_excel(writer, sheet_name='salary', index=False)

pd.read_excel(xls_filename, sheet_name='users')
pd.read_excel(xls_filename, sheet_name='salary')
```

SQL (SQLite)

```
import pandas as pd
import sqlite3

db_filename = os.path.join(tmpdir, "users.db")
```

Connect

```
conn = sqlite3.connect(db_filename)
```

Creating tables with pandas

```
url = 'https://github.com/duchesnay/pystatsml/raw/master/datasets/salary_table.csv'
salary = pd.read_csv(url)

salary.to_sql("salary", conn, if_exists="replace")
```

46

Push modifications

```
cur = conn.cursor()
values = (100, 14000, 5, 'Bachelor', 'N')
cur.execute("insert into salary values (?, ?, ?, ?, ?)", values)
conn.commit()
```

Reading results into a pandas DataFrame

```
salary_sql = pd.read_sql_query("select * from salary;", conn)
print(salary_sql.head())

pd.read_sql_query("select * from salary;", conn).tail()
pd.read_sql_query('select * from salary where salary>25000;', conn)
pd.read_sql_query('select * from salary where experience=16;', conn)
pd.read_sql_query('select * from salary where education="Master";', conn)
```

	index	salary	experience	education	management
0	0	13876	1	Bachelor	Y
1	1	11608	1	Ph.D	N

(continues on next page)

(continued from previous page)

2	2	18701	1	Ph.D	Y
3	3	11283	1	Master	N
4	4	11767	1	Ph.D	N

3.2.16 Exercises

Data Frame

1. Read the iris dataset at '<https://github.com/duchesnay/pystatsml/raw/master/datasets/iris.csv>'
2. Print column names
3. Get numerical columns
4. For each species compute the mean of numerical columns and store it in a stats table like:

	species	sepal_length	sepal_width	petal_length	petal_width
0	setosa	5.006	3.428	1.462	0.246
1	versicolor	5.936	2.770	4.260	1.326
2	virginica	6.588	2.974	5.552	2.026

Missing data

Add some missing data to the previous table users:

```
df = users.copy()
df.loc[[0, 2], "age"] = None
df.loc[[1, 3], "gender"] = None
```

1. Write a function `fillmissing_with_mean(df)` that fill all missing value of numerical column with the mean of the current columns.
2. Save the original users and "imputed" frame in a single excel file "users.xlsx" with 2 sheets: original, imputed.

Total running time of the script: (0 minutes 14.141 seconds)

NUMERICAL METHODS IN PYTHON

4.1 Numerical Differentiation

```
import numpy as np

# Plot
import matplotlib.pyplot as plt
import seaborn as sns
import pystatsml.plot_utils

# Plot parameters
plt.style.use('seaborn-v0_8-whitegrid')
fig_w, fig_h = plt.rcParams.get('figure.figsize')
plt.rcParams['figure.figsize'] = (fig_w, fig_h * .5)
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
%matplotlib inline
```

Sources:

- [Patrick Walls course](#) of Dept of Mathematics, University of British Columbia.
- [Wikipedia](#)

The derivative of a function at is the limit

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

For a fixed step size h , the previous formula provides the slope of the function using the forward difference approximation of the derivative. Equivalently, the slope could be estimated using backward approximation with positions $x - h$ and x .

The most efficient numerical derivative use the central difference formula with step size is the average of the forward and backward approximation (known as symmetric difference quotient):

$$f'(a) \approx \frac{1}{2} \left(\frac{f(a+h) - f(a)}{h} + \frac{f(a) - f(a-h)}{h} \right) = \frac{f(a+h) - f(a-h)}{2h}$$

Chose the step size depends of two issues

1. Numerical precision: if h is chosen too small, the subtraction will yield a large rounding error due to cancellation will produce a value of zero. For basic central differences, the optimal (Sauer, Timothy (2012). Numerical Analysis. Pearson. p.248.) step is the cube-root of machine epsilon ($2.2 * 10^{-16}$ for double precision), i.e.: $h \approx 10^{-5}$:

```
eps = np.finfo(np.float64).eps
print("Machine epsilon: {:e}, Min step size: {:e}".format(eps, np.cbrt(eps)))
```

```
Machine epsilon: 2.220446e-16, Min step size: 6.055454e-06
```

2. The error of the central difference approximation is upper bounded by a function in $\mathcal{O}(h^2)$. I.e., large step size $h = 10^{-2}$ leads to large error of 10^{-4} . Small step size e.g., $h = 10^{-4}$ provide accurate slope estimation in 10^{-8} .

Those two points argue for a step size $h \in [10^{-3}, 10^{-6}]$

Example: Numerical differentiation of the function:

$$f(x) = \frac{7x^3 - 5x + 1}{2x^4 + x^2 + 1}, x \in [-5, 5]$$

Numerical differentiation with Numpy `gradient` given values y and x (or spacing dx) of a function.

```
range_ = [-5, 5]
dx = 1e-3
n = int((range_[1] - range_[0]) / dx)
x = np.linspace(range_[0], range_[1], n)
f = lambda x: (7 * x ** 3 - 5 * x + 1) / (2 * x ** 4 + x ** 2 + 1)

y = f(x) # values
dydx = np.gradient(y, dx) # values
```

Symbolic differentiation with `sympy` to compute true derivative f'

Installation:

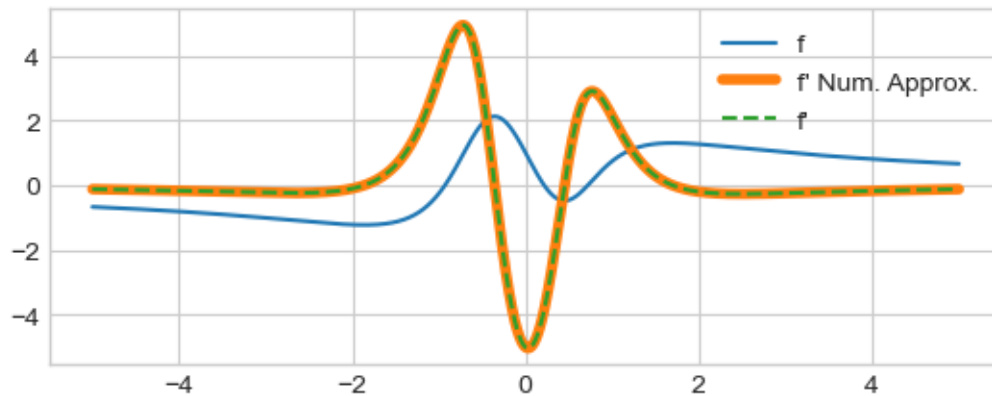
```
conda install conda-forge::sympy
```

```
import sympy as sp
from sympy import lambdify
x_s = sp.symbols('x', real=True) # defining the variables

f_sym = (7 * x_s ** 3 - 5 * x_s + 1) / (2 * x_s ** 4 + x_s ** 2 + 1)
dfdx_sym = sp.simplify(sp.diff(f_sym))
print("f =", f_sym)
print("f' =", dfdx_sym)
dfdx_sym = lambdify(x_s, dfdx_sym, "numpy")
```

```
f = (7*x**3 - 5*x + 1)/(2*x**4 + x**2 + 1)
f' = (-14*x**6 + 37*x**4 - 8*x**3 + 26*x**2 - 2*x - 5)/(4*x**8 + 4*x**6 + 5*x**4 + 2*x**2 + 1)
```

```
plt.plot(x, y, label="f")
plt.plot(x[1:-1], dydx[1:-1], lw=4, label="f' Num. Approx.")
plt.plot(x, dfdx_sym(x), "--", label="f'")
plt.legend()
plt.show()
```

Numerical differentiation with `numdifftools`

The `numdifftools` numerical differentiation problems in one or more variables.

Installation:

```
conda install conda-forge::numdifftools
```

`numdifftools.Derivative` computes the derivatives of order 1 through 10 on any scalar function. It takes a function `f` as argument and return a function `dfdx` that compute the derivatives at `x` values. Example of first and second order derivative of $f(x) = x^2$, $f'(x) = 2x$, $f''(x) = 2$:

```
import numdifftools as nd

# Example f(x) = x ** 2

# First order derivative: dfdx = 2 x
print("dfdx = 2 x:", nd.Derivative(lambda x: x ** 2)([1, 2, 3]))

# Second order derivative df^2dx^2 = 2 (Cte)
print("df2dx2 = 2:", nd.Derivative(lambda x: x ** 2, n=2)([1, 2, 3]))
```

```
dfdx = 2 x: [2. 4. 6.]
df2dx2 = 2: [2. 2. 2.]
```

Example with $f = x^3 - 27x - 1$. We have $f' = 3x^2 - 27$, with roots $(-3, 3)$, and $f'' = 6x$, with root 0.

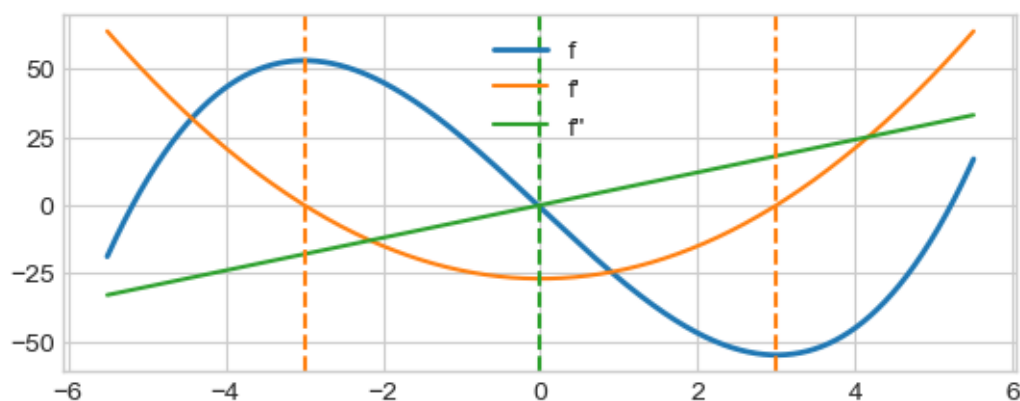
```
range_ = [-5.5, 5.5]
dx = 1e-3
n = int((range_[1] - range_[0]) / dx)
x = np.linspace(range_[0], range_[1], n)
f = lambda x: 1 * (x ** 3) - 27 * x - 1

# First derivative (! callable function, not values)
dfdx = nd.Derivative(f)

# Second derivative (! callable function, not values)
df2dx2 = nd.Derivative(f, n=2)
```

Second-order derivative, “the rate of change of the rate of change” corresponds to the **curvature** or **concavity** of the function.

```
x = np.linspace(range_[0], range_[1], n)
plt.plot(x, f(x), color=colors[0], lw=2, label="f")
plt.plot(x, dfdx(x), color=colors[1], label="f'")
plt.plot(x, df2dx2(x), color=colors[2], label="f''")
plt.axvline(x=-3, ls='--', color=colors[1])
plt.axvline(x= 3, ls='--', color=colors[1])
plt.axvline(x= 0, ls='--', color=colors[2])
plt.legend()
plt.show()
```



- $f'' < 0, x < 0$, f is concave down.
- $f'' > 0, x > 0$, f is concave up.
- $f'' = 0, x = 0$, is an inflection point.

4.1.1 Multivariate functions

$f(\mathbf{x})$ is function of a vector $\mathbf{x}$ of several p variables $\mathbf{x} = [x_1, \dots, x_p]^T$.

Example: $f(x, y) = x^2 + y^2$

```
f = lambda x: x[0]**2 + x[1]**2

import matplotlib.pyplot as plt
from matplotlib import cm

# Make data.
x = np.arange(-5, 5, 0.25)
y = np.arange(-5, 5, 0.25)
xx, yy = np.meshgrid(x, y)

zz = f([xx, yy])

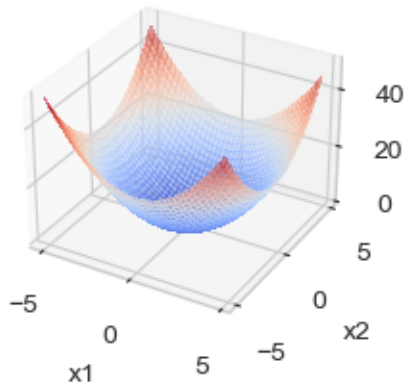
# Plot
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
```

(continues on next page)

(continued from previous page)

```
# Plot the surface.
surf = ax.plot_surface(xx, yy, zz, cmap=cm.coolwarm,
                      linewidth=0, antialiased=False, alpha=0.5, zorder=10)
ax.set_xlabel('x1')
ax.set_ylabel('x2')
```

```
Text(0.5, 0.5, 'x2')
```



The **Gradient** at a given point $\mathbf{x}$ is the vector of partial derivative of f at gives the direction of **fastest increase**.

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \partial f / \partial x_1 \\ \vdots \\ \partial f / \partial x_p \end{bmatrix},$$

```
f_grad = nd.Gradient(f)
print(f_grad([0, 0]))
print(f_grad([1, 1]))
print(f_grad([-1, 2]))
```

```
[0. 0.]
[2. 2.]
[-2. 4.]
```

The **Hessian** matrix contains the second-order partial derivatives of f . It describes the **local curvature** of a function of many variables. It is noted:

$$f''(\mathbf{x}_k) = \nabla^2 f(\mathbf{x}_k) = \mathbf{H}_{f(\mathbf{x}_k)} = \begin{bmatrix} \frac{\partial^2 f}{\partial^2 x_1} & \cdots & \frac{\partial^2 f}{\partial x_p \partial x_1} \\ \vdots & & \vdots \\ \frac{\partial^2 f}{\partial x_1 \partial x_p} & \cdots & \frac{\partial^2 f}{\partial^2 x_p} \end{bmatrix},$$

```
H = nd.Hessian(f)([0, 0])
print(H)
```

```
[[2. 0.]
 [0. 2.]]
```

4.2 Numerical Integration

- Principles Patrick Walls course.
- Library: Scipy integrate package.

```
import numpy as np

# Plot
import matplotlib.pyplot as plt
import seaborn as sns
#import pystatsml.plot_utils

# Plot parameters
plt.style.use('seaborn-v0_8-whitegrid')
fig_w, fig_h = plt.rcParams.get('figure.figsize')
plt.rcParams['figure.figsize'] = (fig_w, fig_h * .5)
%matplotlib inline
```

Methods for integrating functions given fixed samples: $[(x_1, f(x_1)), \dots, (x_i, f(x_i)), \dots, (x_N, f(x_N))]$.

Riemann sums of rectangles to approximate the area.

$$\sum_{i=1}^N f(x_i^*)(x_i - x_{i-1}) \quad , \quad x_i^* \in [x_{i-1}, x_i]$$

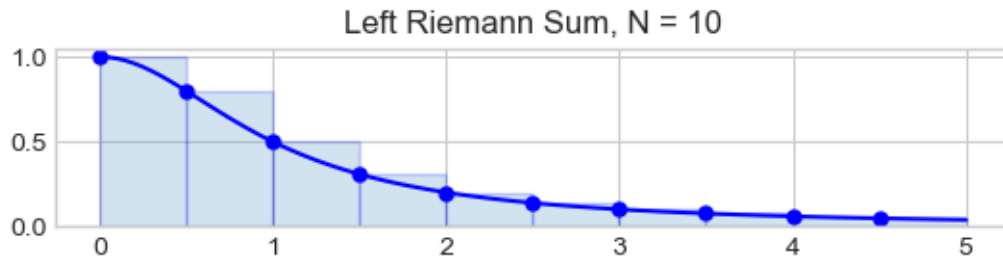
The error is in $\mathcal{O}(\frac{1}{N})$

```
f = lambda x : 1 / (1 + x ** 2)
a, b, N = 0, 5, 10
dx = (b - a) / N

x = np.linspace(a, b, N+1)
y = f(x)

x_ = np.linspace(a,b, 10*N+1) # 10 * N points to plot the function smoothly
plt.plot(x_, f(x_), 'b')
x_left = x[:-1] # Left endpoints
y_left = y[:-1]
plt.plot(x_left,y_left,'b.',markersize=10)
plt.bar(x_left,y_left,width=dx, alpha=0.2,align='edge',edgecolor='b')
plt.title('Left Riemann Sum, N = {}'.format(N))
```

```
Text(0.5, 1.0, 'Left Riemann Sum, N = 10')
```



Compute Riemann sums with 100 points

```
a, b, N = 0, 5, 50
dx = (b - a) / N
x = np.linspace(a, b, N+1)

y = f(x)
print("Integral:", np.sum(f(x[:-1]) * np.diff(x)))
```

Integral: 1.4214653634808756

Trapezoid Rule sum the trapezoids connecting the points. The error is in $\mathcal{O}(\frac{1}{N^2})$. Use `scipy.integrate.trapezoid` function:

```
from scipy import integrate
integrate.trapezoid(f(x[:-1]), dx=dx)
```

np.float64(1.369466163161004)

Simpson's rule uses a quadratic polynomial on each subinterval of a partition to approximate the function and to compute the definite integral. The error is in $\mathcal{O}(\frac{1}{N^4})$. Use `scipy.integrate.simpson` function:

```
from scipy import integrate
integrate.simpson(f(x[:-1]), dx=dx)
```

np.float64(1.3694791829077122)

Gauss-Legendre Quadrature approximate the integral of a function as a weighted sum of Legendre polynomials.

Methods for Integrating functions given function object $f()$ that could be evaluated for any value x in a range $[a, b]$.

Use `scipy.integrate.quad` function. The first argument to `quad` is a “callable” Python object (i.e., a function, method, or class instance). Notice the use of a lambda- function in this case as the argument. The next two arguments are the limits of integration.

```
import scipy.integrate as integrate
integrate.quad(f, a=a, b=b)
```

(1.3734007669450166, 7.167069904541812e-09)

The return values are the estimated of the integral and the estimate of the absolute integration error.

4.3 Time Series

Tools:

- [Pandas](#)
- [Pandas user guide](#)
- [Time Series analysis \(TSA\) from statsmodels](#)

References:

- [Basic](#)
- [Detailed](#)
- [PennState Time Series course](#)

```
%matplotlib inline

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Plot parameters
plt.style.use('seaborn-v0_8-whitegrid')
fig_w, fig_h = plt.rcParams.get('figure.figsize')
plt.rcParams['figure.figsize'] = (fig_w, fig_h * .5)
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
```

Time series with Pandas

```
idx = pd.date_range("2018-01-01", periods=5, freq="YS")
ts = pd.Series(range(len(idx)), index=idx)
print(ts)
```

```
2018-01-01    0
2019-01-01    1
2020-01-01    2
2021-01-01    3
2022-01-01    4
Freq: YS-JAN, dtype: int64
```

4.3.1 Decomposition Methods: Periodic Patterns (Trend/Seasonal) and Autocorrelation

Stationarity

A TS is said to be stationary if its statistical properties such as mean, variance remain constant over time.

- constant mean
- constant variance
- an autocovariance that does not depend on time.

what is making a TS non-stationary. There are 2 major reasons behind non-stationary of a TS:

1. Trend - varying mean over time. For eg, in this case we saw that on average, the number of passengers was growing over time.
2. Seasonality - variations at specific time-frames. eg people might have a tendency to buy cars in a particular month because of pay increment or festivals.

Time series analysis of Google trends

Get Google Trends data of keywords such as 'diet' and 'gym' and see how they vary over time while learning about trends and seasonality in time series data.

In the Facebook Live code along session on the 4th of January, we checked out Google trends data of keywords 'diet', 'gym' and 'finance' to see how they vary over time. We asked ourselves if there could be more searches for these terms in January when we're all trying to turn over a new leaf?

In this tutorial, you'll go through the code that we put together during the session step by step. You're not going to do much mathematics but you are going to do the following:

- Read data
- Recode data
- Exploratory Data Analysis

Read data

```
try:
    url = "https://github.com/datacamp/datacamp_facebook_live_ny_resolution/raw/
↪master/data/multiTimeline.csv"
    df = pd.read_csv(url, skiprows=2)
except:
    df = pd.read_csv("../datasets/multiTimeline.csv", skiprows=2)

print(df.head())

# Rename columns
df.columns = ['month', 'diet', 'gym', 'finance']

# Describe
print(df.describe())
```

	Month	diet: (Worldwide)	gym: (Worldwide)	finance: (Worldwide)
0	2004-01	100	31	48
1	2004-02	75	26	49
2	2004-03	67	24	47
3	2004-04	70	22	48
4	2004-05	72	22	43

(continues on next page)

(continued from previous page)

	diet	gym	finance
count	168.000000	168.000000	168.000000
mean	49.642857	34.690476	47.148810
std	8.033080	8.134316	4.972547
min	34.000000	22.000000	38.000000
25%	44.000000	28.000000	44.000000
50%	48.500000	32.500000	46.000000
75%	53.000000	41.000000	50.000000
max	100.000000	58.000000	73.000000

Recode data

Next, you'll turn the 'month' column into a DateTime data type and make it the index of the DataFrame.

Note that you do this because you saw in the result of the `.info()` method that the 'Month' column was actually an of data type object. Now, that generic data type encapsulates everything from strings to integers, etc. That's not exactly what you want when you want to be looking at time series data. That's why you'll use `.to_datetime()` to convert the 'month' column in your DataFrame to a DateTime.

Be careful! Make sure to include the in place argument when you're setting the index of the DataFrame `df` so that you actually alter the original index and set it to the 'month' column.

```
df.month = pd.to_datetime(df.month)
df.set_index('month', inplace=True)

df = df[["diet", "gym"]]

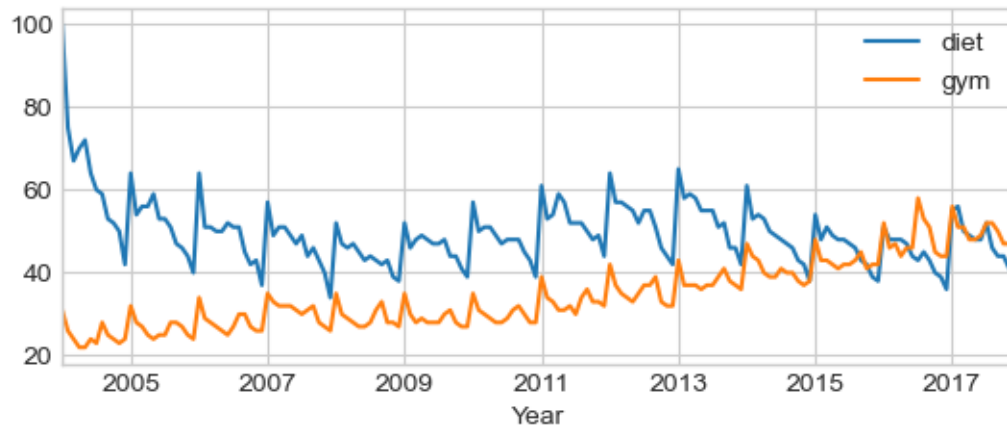
print(df.head())
```

	diet	gym
month		
2004-01-01	100	31
2004-02-01	75	26
2004-03-01	67	24
2004-04-01	70	22
2004-05-01	72	22

Exploratory data analysis

You can use a built-in pandas visualization method `.plot()` to plot your data as 3 line plots on a single figure (one for each column, namely, 'diet', 'gym', and 'finance').

```
df.plot()
plt.xlabel('Year');
```

Note that this data is relative. As you can read on Google trends:

Numbers represent search interest relative to the highest point on the chart for the given region and time. A value of 100 is the peak popularity for the term. A value of 50 means that the term is half as popular. Likewise a score of 0 means the term was less than 1% as popular as the peak.

Trends : Resampling, Rolling average, (Smoothing, Windowing)

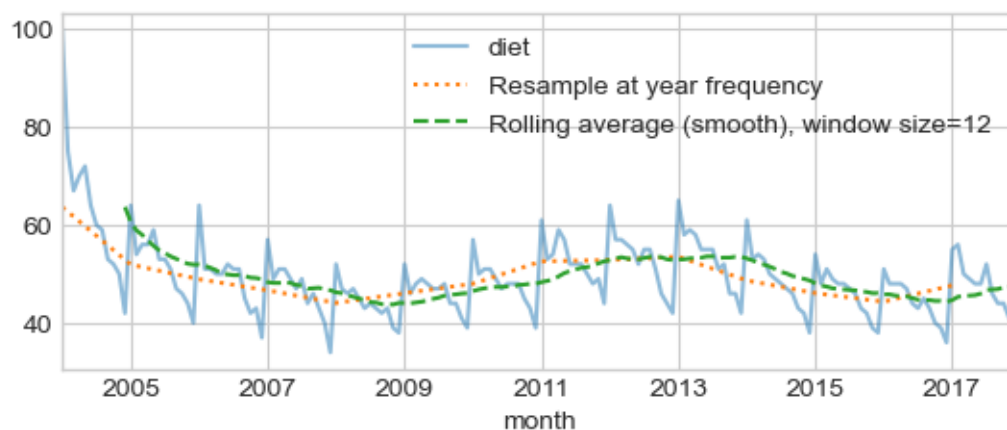
Identify trends or remove seasonality

1. Subsampling at year frequency
2. Rolling average (Smoothing, Windowing), for each time point, take the average of the points on either side of it. Note that the number of points is specified by a window size.

```
diet = df['diet']

diet_resamp_yr = diet.resample('YE').mean()
diet_roll_yr = diet.rolling(12).mean()

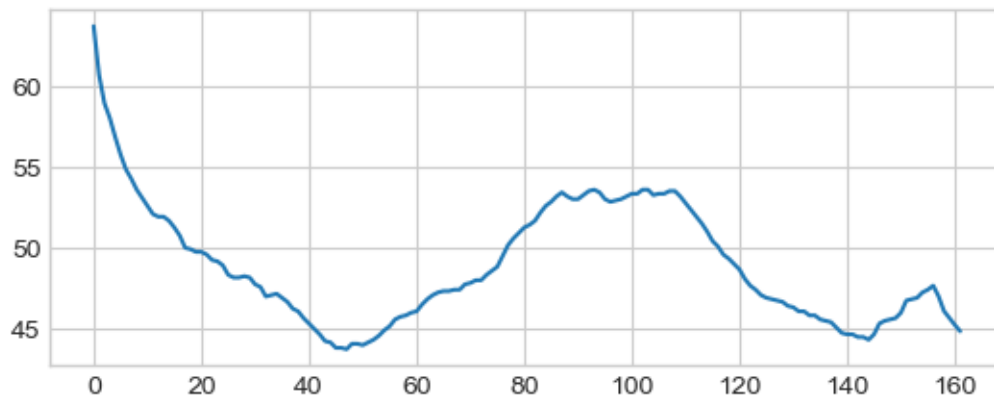
ax = diet.plot(alpha=0.5, style='-') # store axis (ax) for latter plots
diet_resamp_yr.plot(style=':', label='Resample at year frequency', ax=ax)
diet_roll_yr.plot(style='--', label='Rolling average (smooth), window size=12',
                  ax=ax)
_ = ax.legend()
```



Rolling average (smoothing) with Numpy

```
x = np.asarray(df[['diet']])
win = 12
win_half = int(win / 2)

diet_smooth = np.array([x[(idx-win_half):(idx+win_half)].mean()
                        for idx in np.arange(win_half, len(x))])
_ = plt.plot(diet_smooth)
```

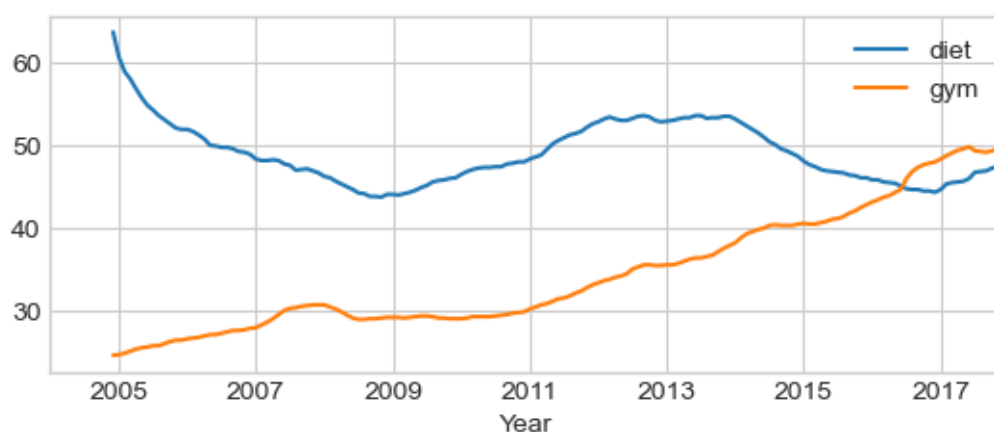


Trends: Plot Diet and Gym using rolling average

Build a new DataFrame which is the concatenation diet and gym smoothed data

```
df_trend = pd.concat([df['diet'].rolling(12).mean(), df['gym'].rolling(12).
                    ↪mean()], axis=1)
df_trend.plot()
plt.xlabel('Year')
```

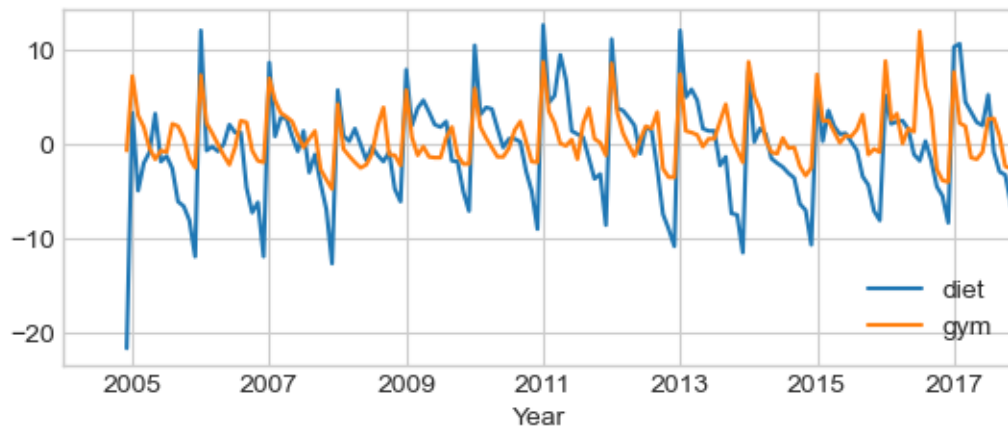
```
Text(0.5, 0, 'Year')
```



Seasonality by detrending (remove average)

```
df_dtrend = df[["diet", "gym"]] - df_trend
df_dtrend.plot()
plt.xlabel('Year')
```

```
Text(0.5, 0, 'Year')
```



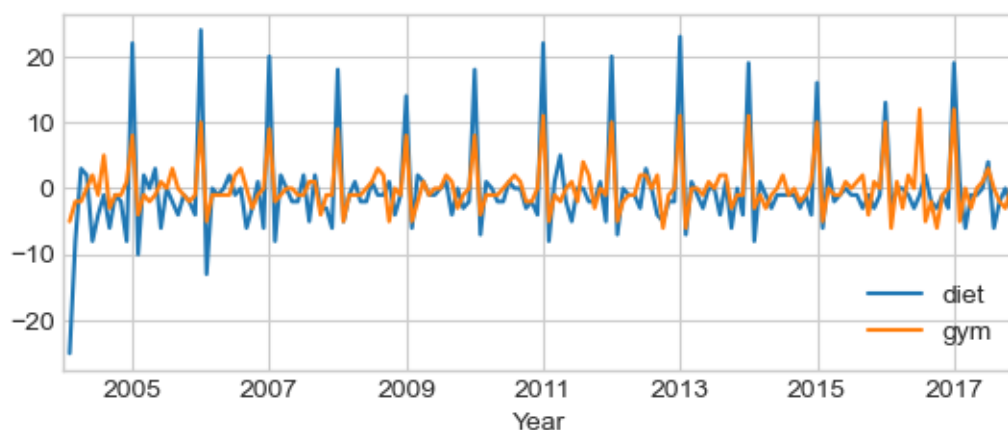
Seasonality by First-order Differencing

First-order approximation using diff method which compute original - shifted data:

```
# exclude first term for some implementation details
assert np.all((diet.diff() == diet - diet.shift())[1:])

df.diff().plot()
plt.xlabel('Year')
```

```
Text(0.5, 0, 'Year')
```



Periodicity and Autocorrelation

Correlation matrix

```
print(df.corr())
```

```

      diet      gym
diet  1.000000 -0.100764
gym  -0.100764  1.000000

```

‘diet’ and ‘gym’ are negatively correlated! Remember that you have a seasonal and a trend component. The correlation is actually capturing both of those. Decomposing into separate components provides a better insight of the data:

Trends components that are negatively correlated:

```
df_trend.corr()
```

Seasonal components (Detrended or First-order Differencing) are positively correlated

```
print(df_dtrend.corr())
print(df.diff().corr())
```

```

      diet      gym
diet  1.000000  0.600208
gym   0.600208  1.000000

      diet      gym
diet  1.000000  0.758707
gym   0.758707  1.000000

```

Seasonal_decompose function of `statsmodels`. “The results are obtained by first estimating the trend by applying a using moving averages or a convolution filter to the data. The trend is then removed from the series and the average of this de-trended series for each period is the returned seasonal component.”

We use additive (linear) model, i.e., $TS = \text{Level} + \text{Trend} + \text{Seasonality} + \text{Noise}$

- Level: The average value in the series.
- Trend: The increasing or decreasing value in the series.
- Seasonality: The repeating short-term cycle in the series.
- Noise: The random variation in the series.

```

from statsmodels.tsa.seasonal import seasonal_decompose

x = df.gym.astype(float) # force float
decomposition = seasonal_decompose(x)
trend = decomposition.trend
seasonal = decomposition.seasonal
residual = decomposition.resid

```

(continues on next page)

(continued from previous page)

```
fig, axis = plt.subplots(4, 1, figsize=(fig_w, fig_h))

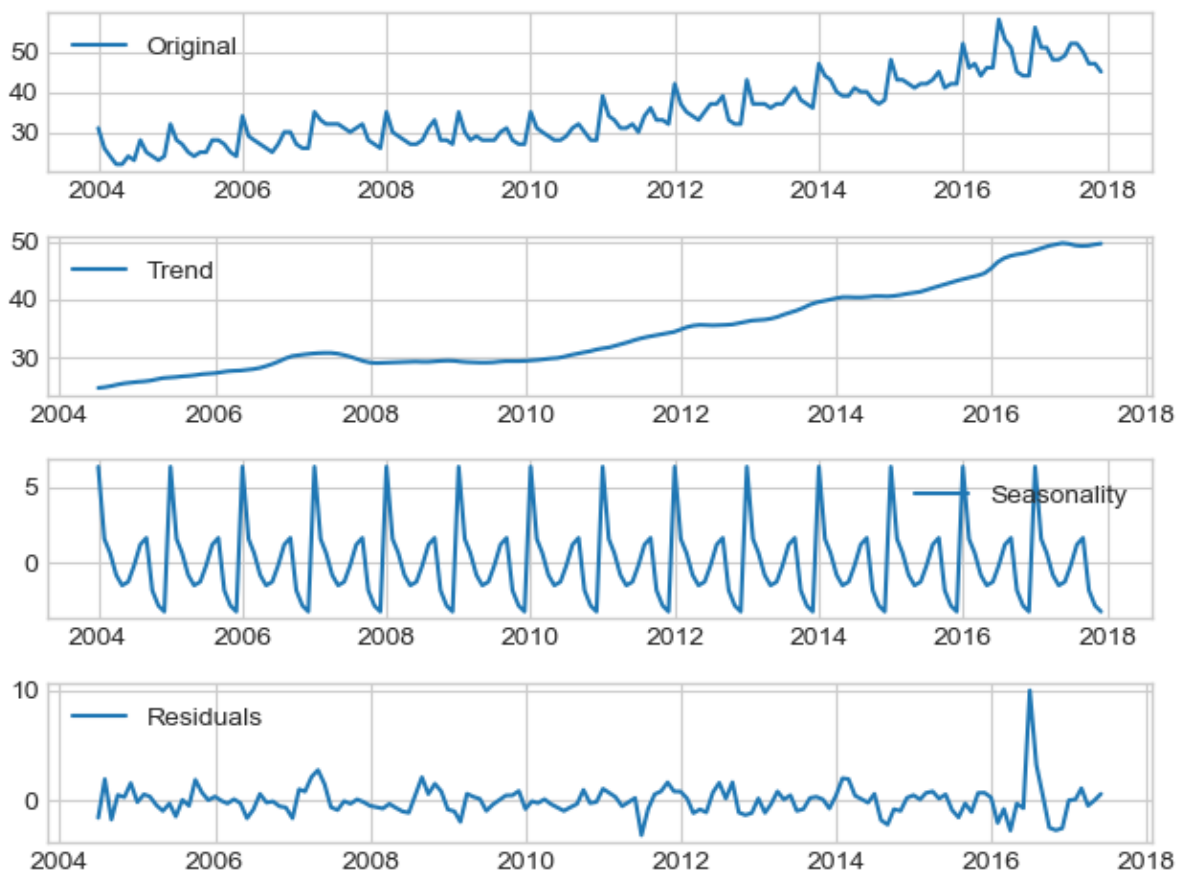
axis[0].plot(x, label='Original')
axis[0].legend(loc='best')

axis[1].plot(trend, label='Trend')
axis[1].legend(loc='best')

axis[2].plot(seasonal, label='Seasonality')
axis[2].legend(loc='best')

axis[3].plot(residual, label='Residuals')
axis[3].legend(loc='best')

plt.tight_layout()
```



Autocorrelation function (ACF)

A time series is periodic if it repeats itself at equally spaced intervals, say, every 12 months. Autocorrelation Function (ACF): It is a measure of the correlation between the TS with a lagged version of itself. For instance at lag 5, ACF would compare series at time instant t with series at instant $t - h$.

- The autocorrelation measures the linear relationship between an observation and its previous observations at different lags (h).

- Represents the overall correlation structure of the time series.
- Used to identify the order of a moving average (MA) process.

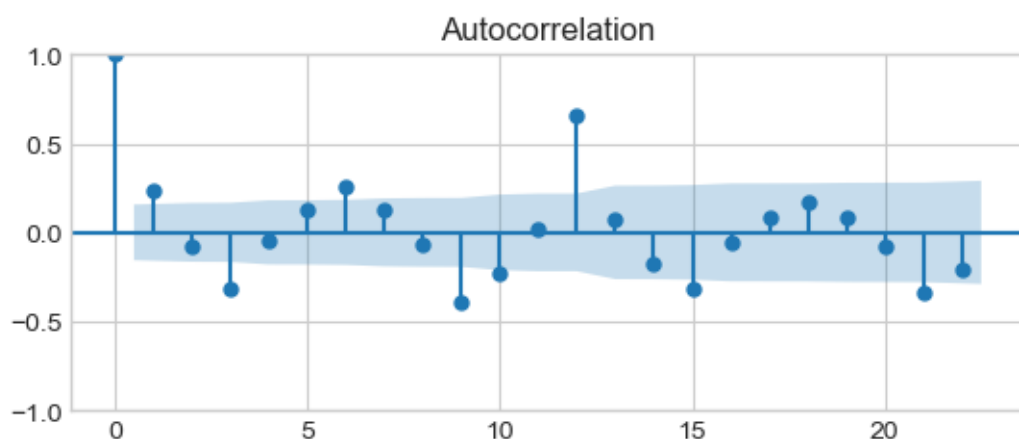
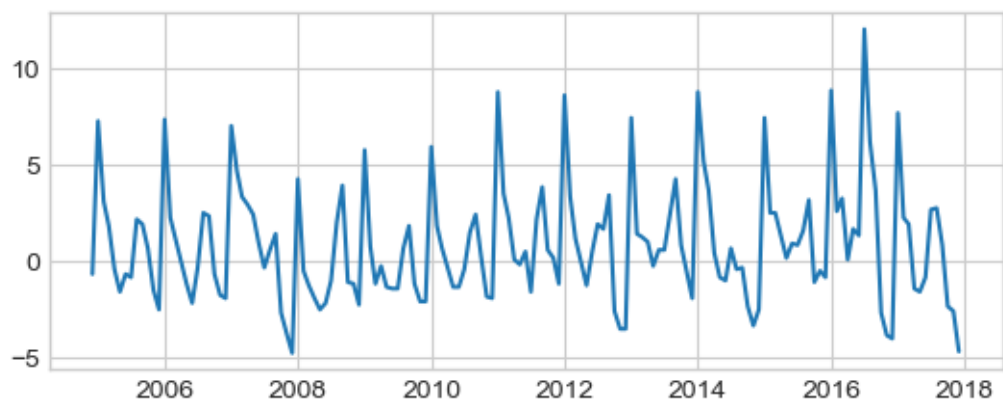
```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
# from statsmodels.tsa.stattools import acf, pacf

# We could have considered the first order differences to capture the seasonality
# x = df["gym"].astype(float).diff().dropna()

# But we use the detrended signal
x = df_dtrend.gym.dropna()

plt.plot(x)
plt.show()

plot_acf(x)
plt.show()
```

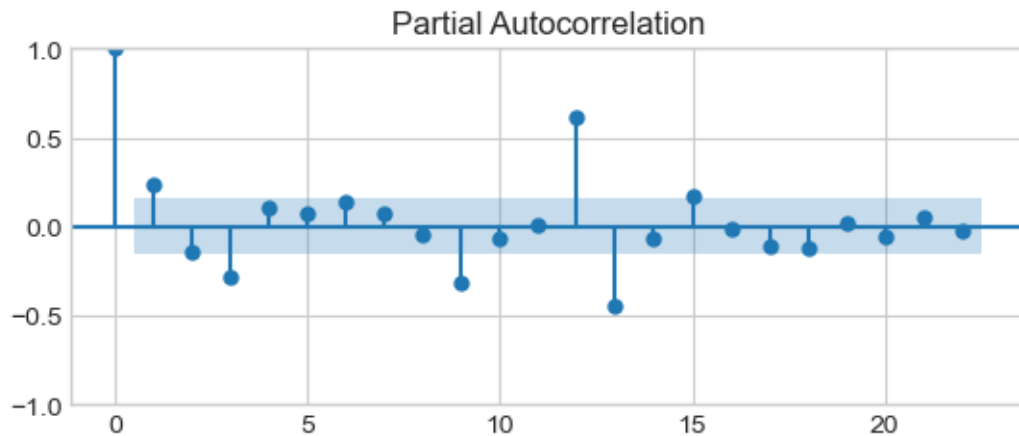


Partial autocorrelation function (PACF)

- Partial autocorrelation measures the direct linear relationship between an observation and its previous observations at a specific offset, excluding contributions from intermediate offsets.
- Highlights direct relationships between observations at specific lags.

- Used to identify the order of an autoregressive (AR) process. The partial autocorrelation of an AR(p) process equals zero at lags larger than p, so the appropriate maximum lag p is the one after which the partial autocorrelations are all zero.

```
plot_pacf(x)
plt.show()
```



PACF peaks every 12 months, i.e., the signal is correlated with itself shifted by 12 months. Its, then slowly decrease is due to the trend.

4.3.2 Time series forecasting using Autoregressive AR(p) models

Sources:

- [Simple modeling with AutoReg](#)

The autoregressive orders. In general, we can define an AR(p) model with p autoregressive terms as follows:

$$x_t = \sum_i^p a_i x_{t-i} + \varepsilon_t$$

```
from sklearn.metrics import root_mean_squared_error as rmse
from statsmodels.tsa.api import AutoReg

# We set the frequency for the time series to "MS" (month-start) to avoid
↳ warnings when using AutoReg.
x = df_dtrend.gym.dropna().asfreq("MS")
ar1 = AutoReg(x, lags=1).fit()
print(ar1.summary())
```

AutoReg Model Results			
=====			
Dep. Variable:	gym	No. Observations:	157
Model:	AutoReg(1)	Log Likelihood	-387.902
Method:	Conditional MLE	S.D. of innovations	2.908
Date:	Thu, 03 Apr 2025	AIC	781.803
Time:	13:00:10	BIC	790.953

(continues on next page)

(continued from previous page)

Sample:	01-01-2005		HQIC	785.519		
	- 12-01-2017					
=====						
	coef	std err	z	P> z	[0.025	0.975]

const	0.6416	0.243	2.641	0.008	0.165	1.118
gym.L1	0.2448	0.078	3.119	0.002	0.091	0.399
Roots						
=====						
	Real	Imaginary	Modulus	Frequency		

AR.1	4.0853	+0.0000j	4.0853	0.0000		

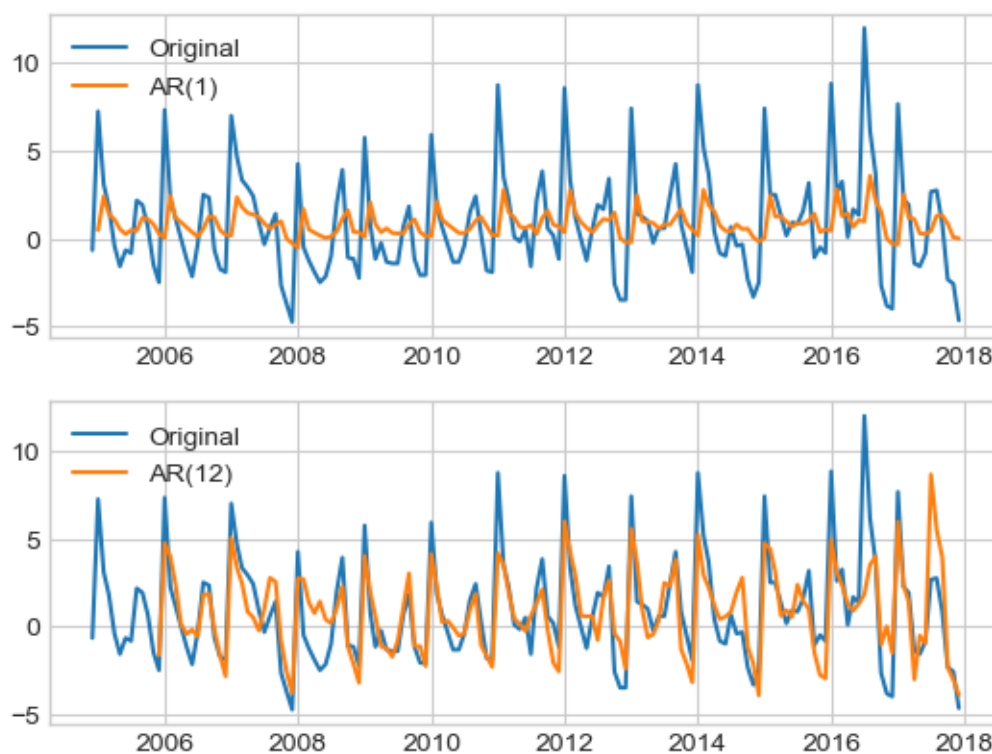
Partial autocorrelation function (PACF) peaks at $p = 12$, try AR(12):

```
ar12 = AutoReg(x, lags=12).fit()

fig, axis = plt.subplots(2, 1, figsize=(fig_w, fig_h))

axis[0].plot(x, label='Original')
axis[0].plot(ar1.predict(), label='AR(1)')
axis[0].legend(loc='best')

axis[1].plot(x, label='Original')
axis[1].plot(ar12.predict(), label='AR(12)')
_ = axis[1].legend(loc='best')
```

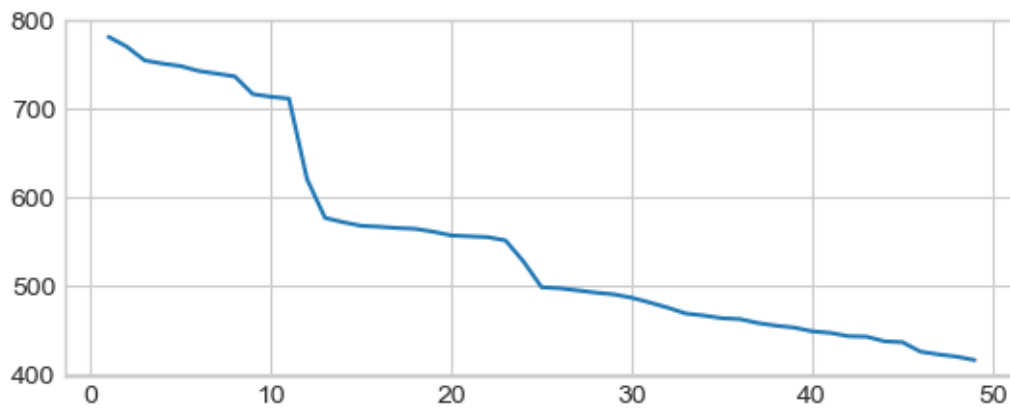



```
mae = lambda y_true, y_pred : (y_true - y_pred).dropna().abs().mean()
print("MAE: AR(1) %.3f" % mae(x, ar1.predict()),
      "AR(12) %.3f" % mae(x, ar12.predict()))
```

```
MAE: AR(1) 2.093 AR(12) 1.375
```

Automatic model selection using Akaike information criterion (AIC). AIC drops at $p = 12$.

```
aics = [AutoReg(x, lags=p).fit().aic for p in range(1, 50)]
_ = plt.plot(range(1, len(aics)+1), aics)
```



4.3.3 Discrete Fourier Transform (DFT)

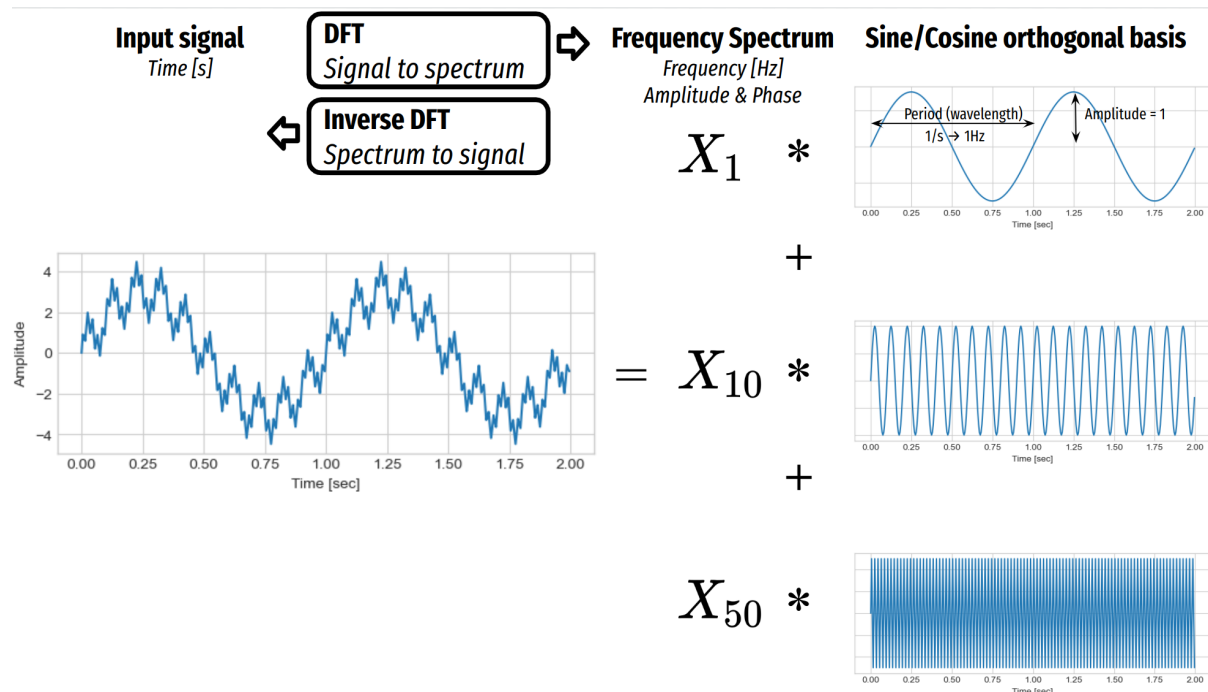


Fig. 1: Discrete Fourier Transform

Fourier Analysis

- Fourier analysis is a mathematical method used to decompose functions or signals into

their constituent frequencies, known as sine and cosine components, that form an orthogonal basis.

- It transforms a time-domain signal into a frequency-domain representation, making it useful for analyzing periodic or non-periodic signals.
- This technique is widely applied in fields like signal processing, image analysis, and solving differential equations.

Discrete Fourier Transform (DFT)

- The Discrete Fourier Transform (DFT) is a specific form of Fourier analysis applied to discrete signals, transforming a finite sequence of equally spaced samples into a frequency-domain representation.
- It breaks down a discrete signal into a sum of sine and cosine waves, each with specific amplitudes and frequencies.
- The DFT is widely used in digital signal processing for tasks like filtering and spectral analysis.

The Basics of Waves

A cosine wave can be represented by the following equation:

$$y(t) = A \cos(2\pi ft + \phi)$$

- ϕ is the phase of the signal.
- $T = 1/f$ is the period of the wave,
- f is the frequency of the wave
- A is the amplitude of the signal

To generate sample we need to define the sampling rate which is the number of sample per second.

```
def sine(A=1, f=10, phase=0, duration=1.0, sr=100.0):
    # sampling interval
    ts = 1.0 / sr
    t = np.arange(0, duration, ts)
    return t, A * np.sin(2 * np.pi * f * t + phase)

def cosine(A=1, f=10, phase=0, duration=1.0, sr=100.0):
    # sampling interval
    ts = 1.0 / sr
    t = np.arange(0, duration, ts)
    return t, A * np.cos(2 * np.pi * f * t + phase)

sr = 200

t, sine_1hz = cosine(A=3, f=1, sr=sr, duration=2)
#t, sine_1hz = sine(A=2, f=10, sr=sr, duration=2)
t, sine_10hz = cosine(A=1, f=10, sr=sr, duration=2)
t, sine_50hz = cosine(A=.5, f=50, sr=sr, duration=2)
#t, sine_20hz = sine(A=.5, f=10, sr=sr, duration=2)
```

(continues on next page)

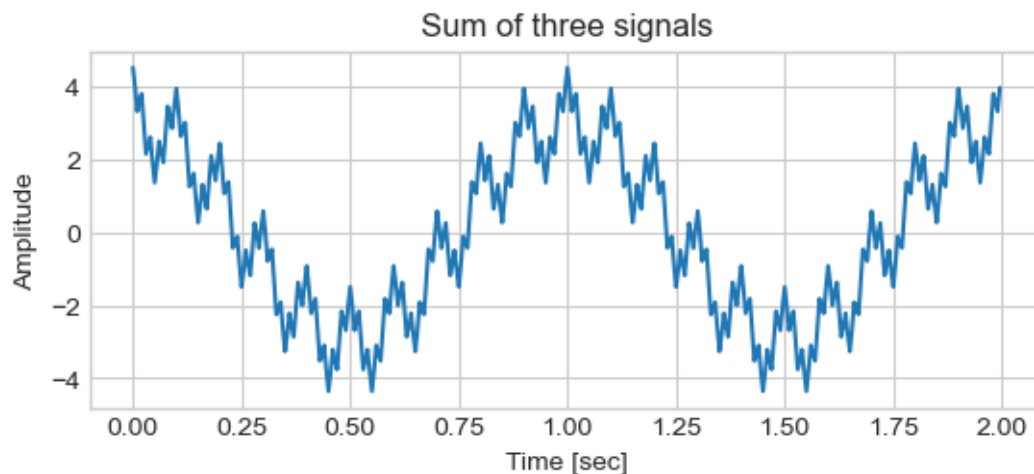
(continued from previous page)

```

y = sine_1hz + sine_10hz + sine_50hz

# Plot the signal
plt.plot(t, y, c=colors[0])
plt.xlabel('Time [sec]')
plt.ylabel('Amplitude')
plt.title('Sum of three signals')
plt.show()

```



Discrete Cosine Transform DCT

- A discrete cosine transform (DCT) expresses a finite sequence of data points in terms of a sum of cosine functions oscillating at different frequencies.
- A DCT is a Fourier-related transform similar to the discrete Fourier transform (DFT), but using only real numbers.
- See also [Discrete Sine Transform DST](#)

There are several definitions of the DCT, see [DCT with Scipy](#) for details. For educational purpose, we use a simplified modified version:

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left(\frac{2\pi kn}{N}\right),$$

where

- N is the number of samples
- n is the current sample
- k is the current frequency, where $k \in [0, N - 1]$
- x_n is the sine value at sample n
- X_k are the **(frequency terms or DCT)** which include information of amplitude. It is called the **spectrum of the signal**.

Relation between

- f_s : Sampling rate or Frequency of Sampling, where

- T : Duration

$$f_s = \frac{N}{T}$$

Generate Signal, as an addition of three cosines at different frequencies: 1 Hz, 10 Hz, and 50 Hz:

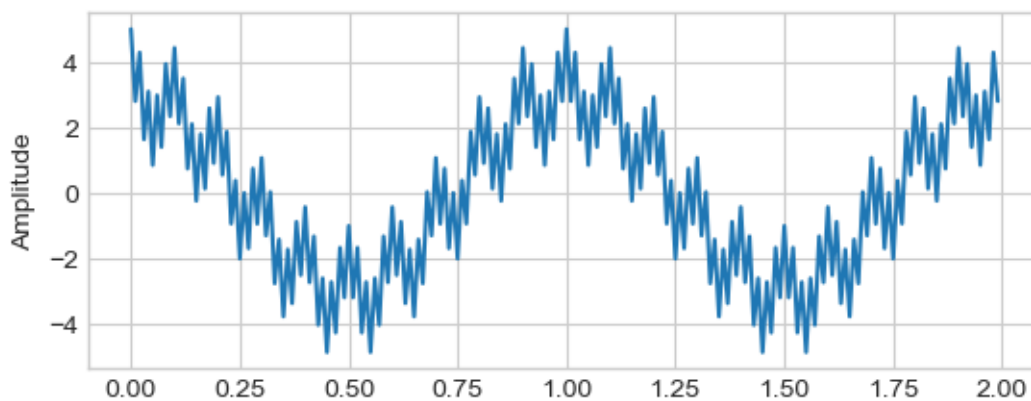
```
T = 2. # duration
fs = 100 # Sampling rate/frequency: number of samples per second
ts = 1.0 / fs # sampling interval
t = np.arange(0, T, ts) # time axis
N = len(t)

# Generate Signal

x = 0
x += 3.0 * np.cos(2 * np.pi * 1.00 * t)
x += 1.0 * np.cos(2 * np.pi * 10.0 * t)
x += 1.0 * np.cos(2 * np.pi * 50.0 * t)

# Plot

plt.figure(figsize = (8, 6))
plt.plot(t, x)
plt.ylabel('Amplitude')
plt.show()
```



Cosines Basis

```
N = len(x)
n = np.arange(N)
k = n.reshape((N, 1))
cosines = np.cos(2 * np.pi * k * n / N)

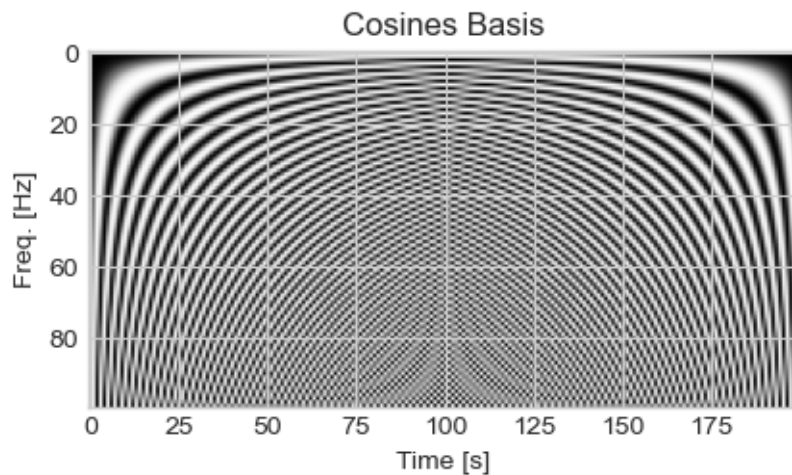
# Plot

plt.imshow(cosines[:100, :])
plt.xlabel('Time [s]')
plt.ylabel('Freq. [Hz]')
```

(continues on next page)

(continued from previous page)

```
plt.title('Cosines Basis')
plt.show()
```



Decompose signal on cosine basis (dot product), i.e., DCT without signal normalization

```
X = np.dot(cosines, x)

# Frequencies = N / T
freqs = np.arange(N) / T

# Examine Spectrum, look for frequencies below N / 2

res = pd.DataFrame(dict(freq=freqs, val=X))
res = res[: (N // 2 + 1)]
res = res.iloc[np.where(res.val > 0.01)]
print(res)
```

```
   freq  val
2    1.0  300.0
20   10.0  100.0
100  50.0  200.0
```

Discrete Fourier Transform (DFT)

- The Fourier Transform (FT) decompose any signal into a sum of simple sine and cosine waves that we can easily measure the frequency, amplitude and phase.
- FT can be applied to continuous or discrete waves, in this chapter, we will only talk about the **Discrete Fourier Transform (DFT)**.”

$$X_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N} = \sum_{n=0}^{N-1} x_n [\cos(2\pi kn/N) - i \sin(2\pi kn/N)],$$

Where

- X_k are the **(frequency terms or DFT)** which include information of both amplitude and phase. It is called the **spectrum of the signal**.

If the input signal is a real-valued sequence the negative frequency terms are just the complex conjugates of the corresponding positive-frequency terms, and the negative-frequency terms are therefore redundant. The DFT spectrum will be symmetric. Therefore, usually we only plot the DFT corresponding to the positive frequencies and divide by $N/2$ to get the amplitude corresponding to the time domain signal.

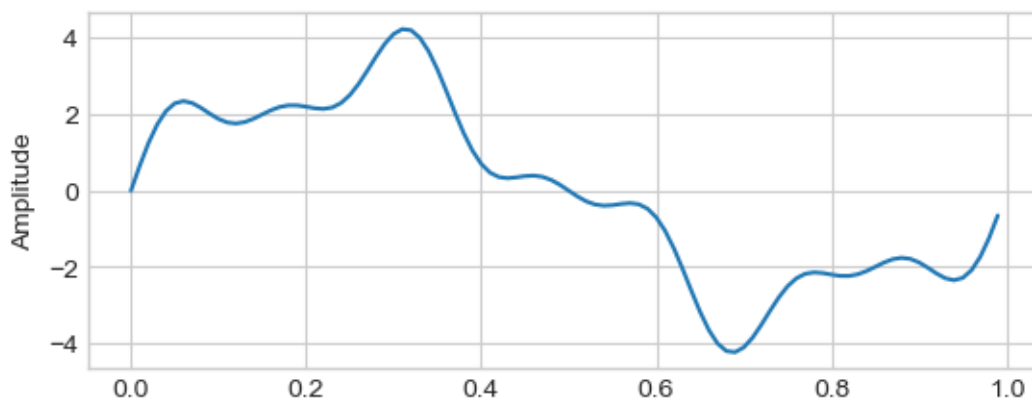
The amplitude and phase of the signal can be calculated as:

TODO FFT

```
fs = 100 # sampling rate/frequency: number of samples per second
ts = 1.0 / fs # sampling interval
T = 1 # duration
t = np.arange(0, T, ts)

x = 3 * np.sin(2 * np.pi * 1 * t)
x += np.sin(2 * np.pi * 4 * t)
x += 0.5 * np.sin(2 * np.pi * 7 * t)

plt.figure(figsize=(8, 6))
plt.plot(t, x)
plt.ylabel('Amplitude')
plt.show()
```



```
from numpy.fft import fft, ifft

X = fft(x)

# Frequencies

N = len(X) # number of frequencies = number of samples
T = N / fs # duration
freqs = np.arange(N) / T # Frequencies = N / T

# Examine Spectrum, look for frequencies below N / 2

res = pd.DataFrame(dict(freq=freqs, val=abs(X)))
res = res[: (N // 2 + 1)]
res = res.iloc[np.where(res.val > 0.01)]
```

(continues on next page)

(continued from previous page)

```

print(res)

def plot_fft(X, freqs, t, xlim):

    plt.figure()
    plt.subplot(121)

    plt.stem(freqs, np.abs(X), 'b', \
             markerfmt=" ", basefmt="-b")
    plt.xlabel('Freq (Hz)')
    plt.ylabel('FFT Amplitude |X(freq)|')
    plt.xlim(0, xlim)

    plt.subplot(122)
    plt.plot(t, ifft(X), 'r')
    plt.xlabel('Time (s)')
    plt.ylabel('Amplitude')
    plt.tight_layout()
    plt.show()

plot_fft(X, freqs, t, xlim=10)

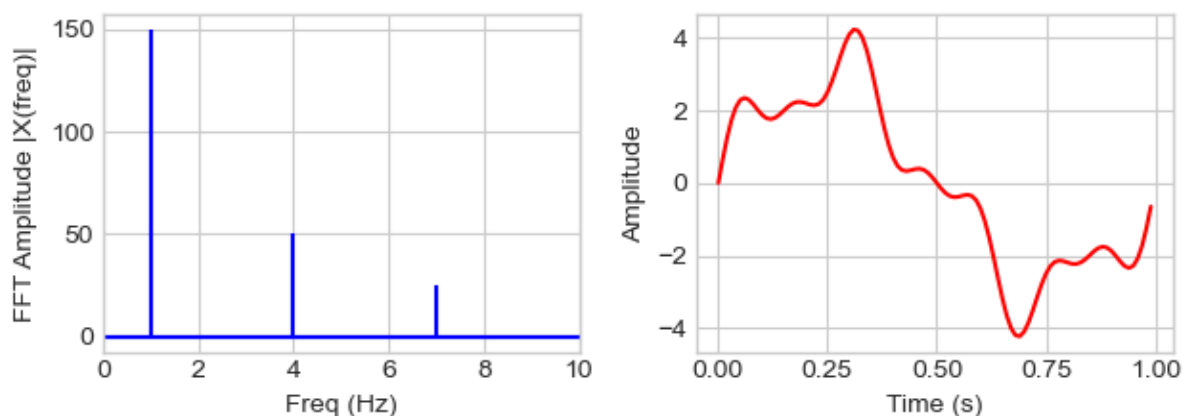
```

	freq	val
1	1.0	150.0
4	4.0	50.0
7	7.0	25.0

```

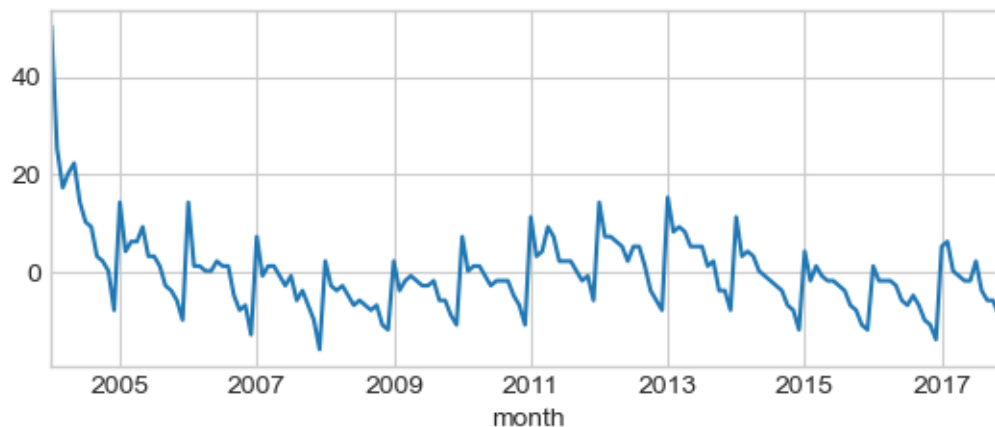
/home/ed203246/git/pystatsml/.pixi/envs/default/lib/python3.13/site-packages/
↳matplotlib/cbook.py:1719: ComplexWarning: Casting complex values to real_
↳discards the imaginary part
    return math.isfinite(val)
/home/ed203246/git/pystatsml/.pixi/envs/default/lib/python3.13/site-packages/
↳matplotlib/cbook.py:1355: ComplexWarning: Casting complex values to real_
↳discards the imaginary part
    return np.asarray(x, float)

```



```
x = df['diet']
x -= x.mean()
x.plot()

fs = 12 # sampling frequency 12 sample / year
```



```
X = fft(x)

# Frequencies

N = len(X) # number of frequencies = number of samples
T = N / fs # duration
freqs = np.arange(N) / T # Frequencies = N / T

# Examine Spectrum, look for frequencies below N / 2

Xn = abs(X)
print(pd.Series(Xn, index=freqs).describe())

res = pd.DataFrame(dict(freq_year=freqs, freq_month=12 / freqs, val=Xn))
res = res[: (N // 2 + 1)]
res = res.iloc[np.where(res.val > 200)]
print(res)

# plot_fft(X, freqs, t, xlim=15)
```

```
count    1.680000e+02
mean      6.938104e+01
std       7.745030e+01
min       5.115908e-13
25%       3.132051e+01
50%       4.524089e+01
75%       6.551233e+01
max       4.429661e+02
dtype: float64
      freq_year  freq_month      val
```

(continues on next page)

(continued from previous page)

2	0.142857	84.0	442.966103
14	1.000000	12.0	422.372698
28	2.000000	6.0	271.070102
42	3.000000	4.0	215.675682
56	4.000000	3.0	248.131014
70	5.000000	2.4	216.030794
84	6.000000	2.0	240.000000

```
/tmp/ipykernel_216680/2580075766.py:14: RuntimeWarning: divide by zero_
↪ encountered in divide
res = pd.DataFrame(dict(freq_year=freqs, freq_month=12 / freqs, val=Xn))
```

4.4 Optimization (Minimization) by Gradient Descent

4.4.1 Gradient Descent

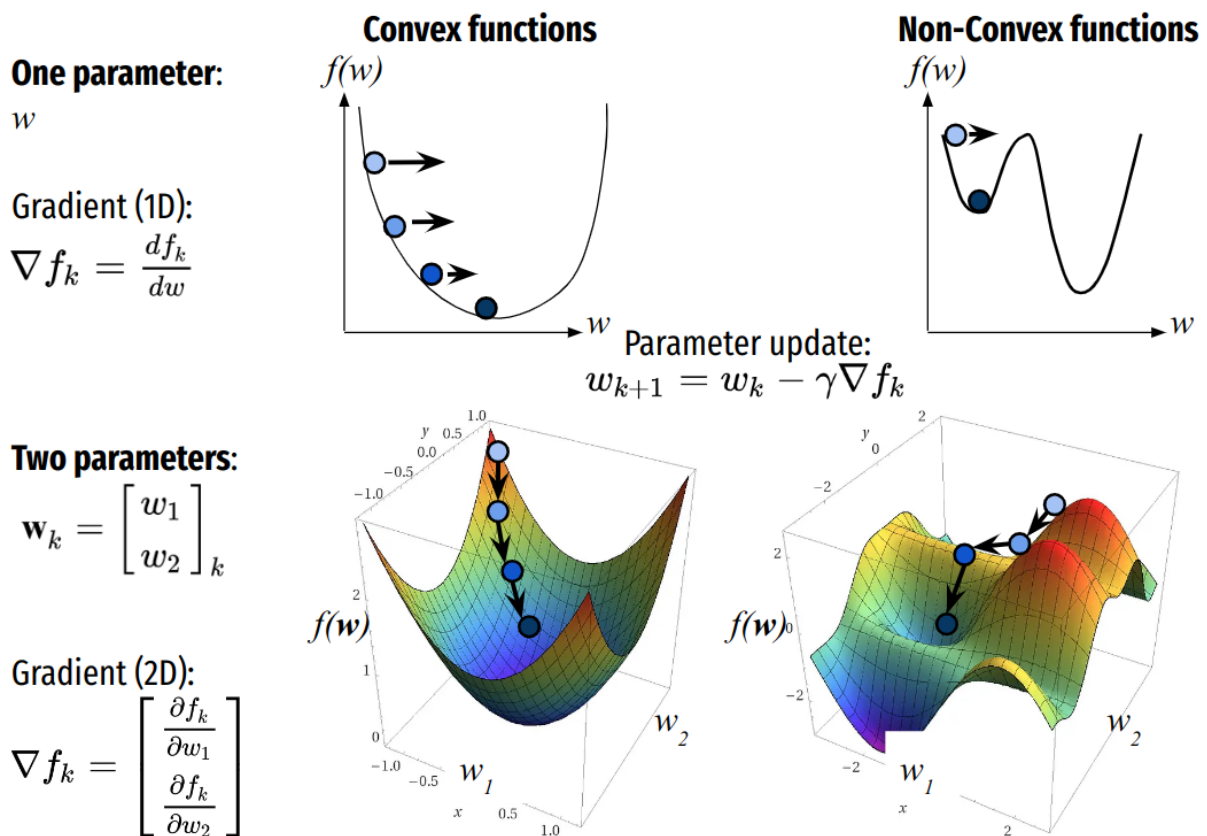


Fig. 2: Optimization (minimization) by gradient descent

Gradient descent is an optimization algorithm to minimize a **cost or loss function** f given its parameter w . The algorithm **iteratively moves in the direction of steepest descent** as defined by the **opposite direction the gradient**. In machine learning, we use gradient descent to update the parameters of our model. **Parameters** refer to **coefficients** in **Linear Regression** and **weights** in **neural networks**.

Local minimization of f at point x_k make use of first-order **Taylor expansion** local estimation of

f (in one dimension):

$$f(w_k + t) \approx f(w_k) + f'(w_k)t$$

Therefore, to minimize $f(w_k + t)$ we just have to move in the opposite direction of the derivative $f'(w_k)$:

$$w_{k+1} = w_k - \gamma f'(w_k)$$

With a **learning rate** γ that determines the step size at each iteration while moving toward a minimum of a cost function.

In multidimensional problems $\mathbf{w}_k \in \mathbb{R}^p$, where:

$$\mathbf{w}_k = \begin{bmatrix} w_1 \\ \vdots \\ w_p \end{bmatrix}_k,$$

the derivative $f'(\mathbf{w}_k)$ is the gradient (direction) of f at $\mathbf{w}_k$:

$$\nabla f(\mathbf{w}_k) = \begin{bmatrix} \partial f / \partial w_1 \\ \vdots \\ \partial f / \partial w_p \end{bmatrix}_k,$$

Leading to the minimization scheme:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \gamma \nabla f(\mathbf{w}_k)$$

Choosing the Step Size

With large learning rate γ we can cover more ground each step, but we **risk overshooting the lowest point** since the slope of the hill is constantly changing.

With a very small learning rate**, we can confidently move in the direction of the negative gradient since we are recalculating it so frequently. A small learning rate is more precise, but calculating the gradient is time-consuming, leading too slow convergence

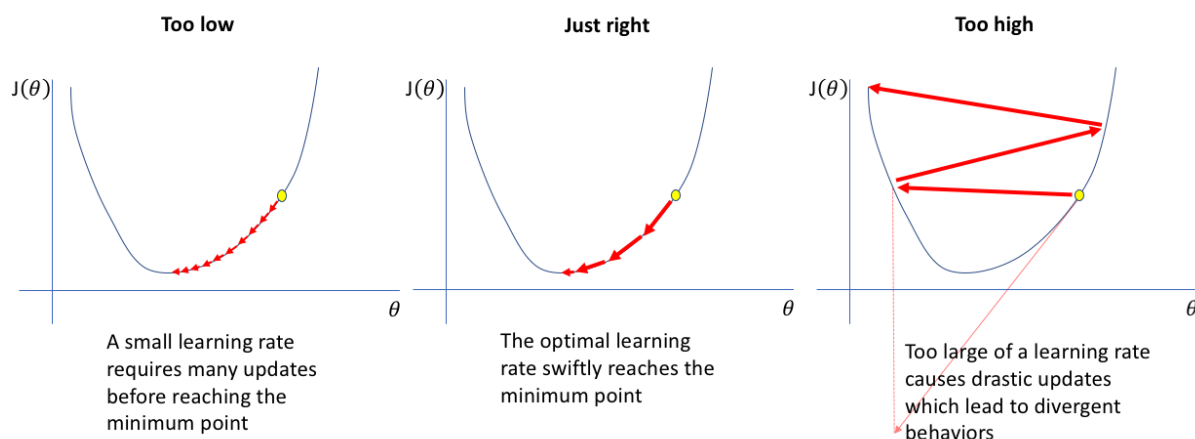


Fig. 3: jeremyjordan

Line search can be used (or more sophisticated **Backtracking line search**) to find value of γ such that $f(\mathbf{w}_k - \gamma \nabla f(\mathbf{w}_k))$ is minimum. However such simple method ignore possible change of the curvature.

- Benefit of gradient decent: simplicity and versatility, almost any function with a gradient can be minimized.
- Limitations:
 - Local minima (local optimization) for non-convex problem.
 - Convergence speed: With fast changing curvature (gradient direction) the estimation of gradient will rapidly become wrong moving away from x_k suggesting small step-size. This also suggest the integration of change of the gradient direction in the calculus of the step size.

Libraries

```
import numpy as np
import pandas as pd
from scipy.optimize import minimize
import numdifftools as nd

# Plot
import matplotlib.pyplot as plt
from matplotlib import cm # color map
import seaborn as sns

# Plot parameters
plt.style.use('seaborn-v0_8-whitegrid')
fig_w, fig_h = plt.rcParams.get('figure.figsize')
plt.rcParams['figure.figsize'] = (fig_w, fig_h * 1.)
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
#%matplotlib inline
```

Gradient Descent Algorithm

```
def gradient_descent(fun, x0, args=(), method="first-order", jac=None,
                    hess=None, tol=1.5e-08,
                    options=dict(learning_rate=0.01,
                                maxiter=1000,
                                intermediate_res=False)):
    """ Gradient Descent minimization.
    To make API compatible with scipy.optimize.minimize, it takes a
    required fun parameters that is not used and an optional jac
    that is used to compute the gradient.

    Parameters
    -----
    fun : callable
        The objective function to be minimized.
    x0 : ndarray, shape (n_features,)
        Initial guess.
    args : tuple, optional
        Extra arguments passed to the objective function and its derivatives
        (fun, jac and hess functions)
```

(continues on next page)

(continued from previous page)

```

method : string, optional
    the solver, by default "first-order" if the basic first-order gradient
    descent.
jac : callable, optional
    Method for computing the gradient vector, the Jacobian.,
    by default None.
    jac(x, *args) -> array_like, shape (n_features,)
hess : callable, optional
    Method for computing the Hessian matrix, by default None
    hess(x, *args) -> ndarray, (n_features, n_features)
tol : float, optional
    Tolerance for termination. Default = 1.5e-08,
    sqrt(np.finfo(np.float64).eps)
options : dict, optional
    A dictionary of solver options., by default dict(learning_rate=0.01,
    maxiter=1000, intermediate_res=False)

```

Returns

```

ndarray, shape (n_features,): the solution, intermediate_res dict

```

"""

```

# Initialize parameters

```

```

weights_k = x0.copy()

```

```

# Termination criteria

```

```

k, eps = 0, np.inf

```

```

# Dict to store intermediate results

```

```

intermediate_res = dict(eps=[], weights=[])

```

```

# Perform gradient descent

```

```

while eps > tol and k < options["maxiter"]:

```

```

    for k in range(options["maxiter"]):

```

```

        weights_prev = weights_k.copy()

```

```

        weights_grad = jac(weights_k, *args)

```

```

        # Update the parameters

```

```

        if method == "first-order" and "learning_rate" in options:

```

```

            weights_k -= options["learning_rate"] * weights_grad

```

```

        if method == "Newton" and hess is not None:

```

```

            H = hess(weights_k, *args)

```

```

            Hinv = np.linalg.inv(H)

```

```

            weights_k -= options["learning_rate"] * np.dot(Hinv, weights_grad)

```

```

        # Update termination criteria

```

```

        k, eps = k + 1, np.sum((weights_k - weights_prev) ** 2)

```

```

    if options["intermediate_res"]:

```

```

        intermediate_res["eps"].append(eps)

```

```

        intermediate_res["weights"].append(weights_prev)

```

(continues on next page)

(continued from previous page)

```
return weights_k, intermediate_res
```

Gradient Descent with Exact Gradient

Minimize:

$$f(\mathbf{w}) = f(x, y) = x^2 + y^2 + xy$$

$$\nabla f(\mathbf{w}) = \begin{bmatrix} \partial f / \partial x \\ \partial f / \partial y \end{bmatrix} = \begin{bmatrix} 2x + 1 \\ 2y + 1 \end{bmatrix},$$

```
def f(x):
    x = np.asarray(x)
    x, y = (x[0], x[1]) if x.ndim == 1 else (x[:, 0], x[:, 1])
    return x ** 2 + y ** 2 + 1 * x * y

print("f:", f([1, 2], [3, 4]))
print("f:", f([1, 2]), f([3, 4]))

def f_grad(x):
    x = np.asarray(x, dtype=float)
    x, y = (x[0], x[1]) if x.ndim == 1 else (x[:, 0], x[:, 1])
    return np.array([2 * x + 1, 2 * y + 1])

print("Grad f:", f_grad([1, 1]))
print("Grad f:", f_grad([1, 2]))
print("Grad f:", f_grad([5, 5]))
```

```
f: [ 7 37]
f: 7 37
Grad f: [3. 3.]
Grad f: [3. 5.]
Grad f: [11. 11.]
```

```
x0 = np.array([30., 40.])
lr = 0.1
weights_sol, intermediate_res = \
    gradient_descent(fun=f, x0=x0, jac=f_grad,
                    options=dict(learning_rate=lr,
                                maxiter=100,
                                intermediate_res=True))

res_ = pd.DataFrame(intermediate_res)
print(res_.head(5))
print(res_.tail(5))
print("Solution: ", weights_sol)
```

```

        eps                weights
0  102.820000              [30.0, 40.0]
1   65.804800              [23.9, 31.9]
2   42.115072    [19.02, 25.419999999999998]
3   26.953646    [15.116, 20.235999999999997]
4   17.250333    [11.992799999999999, 16.0888]

        eps                weights
47  7.989809e-08    [-0.499149784089306, -0.49887102477432443]
48  5.113478e-08    [-0.4993198272714448, -0.4990968198194595]
49  3.272626e-08    [-0.49945586181715584, -0.4992774558555676]
50  2.094480e-08    [-0.4995646894537247, -0.4994219646844541]
51  1.340467e-08    [-0.49965175156297975, -0.4995375717475633]
Solution: [-0.4997214 -0.49963006]

```

Plot solution functions

```

def plot_surface(x_range, y_range, f, surf=True, wireframe=False):
    x, y = np.linspace(x_range[0], x_range[1], 100), np.linspace(y_range[0], y_
→range[1], 100)
    #x, y = np.arange(-5, 5, 0.25), np.arange(-5, 5, 0.25)
    xx, yy = np.meshgrid(x, y)
    zz = f(np.column_stack((xx.ravel(), yy.ravel()))).reshape(xx.shape)

    # Figure
    fig, ax = plt.subplots(subplot_kw={"projection": "3d"})

    # Plot the surface.
    if surf:
        surf = ax.plot_surface(xx, yy, zz, cmap=cm.coolwarm,
                               linewidth=0, antialiased=True, alpha=0.5, zorder=10)
    if wireframe:
        ax.plot_wireframe(xx, yy, zz, rstride=2, cstride=2, color='gray', alpha=0.
→5, lw=1)

    ax.set_xlabel('x'); ax.set_ylabel('y')
    return ax, (xx, yy, zz)

def crop(x, x_range, y_range):
    mask = (x[:, 0] >= x_range[0]) & (x[:, 0] <= x_range[1]) & \
           (x[:, 1] >= y_range[0]) & (x[:, 1] <= y_range[1])
    return x[mask]
    #return x[np.all((x >= min) & (x <= max), axis=1)]

def plot_path(x, y, z, color, label, ax):
    sc = ax.plot3D(x, y, z, c=color)
    sc = ax.scatter3D(x, y, z, c=color, s=30, label=label)
    return ax

```

Solutions'path for different learning rates

```

x_range = y_range = [-5., 5.]

# Plot function surface
ax, _ = plot_surface(x_range=x_range, y_range=y_range, f=f)

# Plot solution paths
x0 = np.array([3., 4.])
lr = 0.01
weights_sol, intermediate_res = \
    gradient_descent(fun=f, x0=x0, jac=f_grad,
                    options=dict(learning_rate=lr,
                                maxiter=10,
                                intermediate_res=True))

sols = crop(np.array(intermediate_res["weights"]),
            x_range, y_range)
plot_path(sols[:, 0], sols[:, 1], f(sols), colors[0],
          'lr:%.02f' % lr, ax)

lr = 0.1
weights_sol, intermediate_res = \
    gradient_descent(fun=f, x0=x0, jac=f_grad,
                    options=dict(learning_rate=lr,
                                maxiter=10,
                                intermediate_res=True))

sols = crop(np.array(intermediate_res["weights"]),
            x_range, y_range)
plot_path(sols[:, 0], sols[:, 1], f(sols), colors[1],
          'lr:%.02f' % lr, ax)

lr = 0.9
weights_sol, intermediate_res = \
    gradient_descent(fun=f, x0=x0, jac=f_grad,
                    options=dict(learning_rate=lr,
                                maxiter=10,
                                intermediate_res=True))

sols = crop(np.array(intermediate_res["weights"]),
            x_range, y_range)
plot_path(sols[:, 0], sols[:, 1], f(sols), colors[2],
          'lr:%.02f' % lr, ax)

lr = 1.
weights_sol, intermediate_res = \
    gradient_descent(fun=f, x0=x0, jac=f_grad,
                    options=dict(learning_rate=lr,
                                maxiter=10,
                                intermediate_res=True))

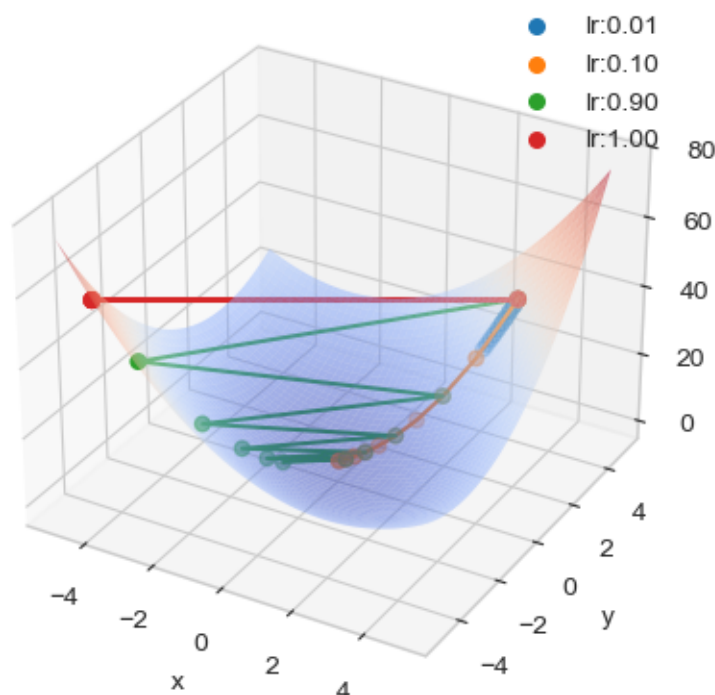
```

(continues on next page)

(continued from previous page)

```
sols = crop(np.array(intermediate_res["weights"]),
             x_range, y_range)
plot_path(sols[:, 0], sols[:, 1], f(sols), colors[3],
          'lr:%.02f' % lr, ax)

plt.legend()
plt.show()
```



Gradient Descent Numerical Approximation of the Gradient

```
# Numerical approximation
f_grad = nd.Gradient(f)
print(f_grad([1, 1]))
print(f_grad([1, 2]))
print(f_grad([5, 5]))

lr = 0.1
weights_sol, intermediate_res = \
    gradient_descent(fun=f, x0=x0, jac=f_grad,
                    options=dict(learning_rate=lr,
                                maxiter=10,
                                intermediate_res=True))

res_ = pd.DataFrame(intermediate_res)
print(res_.head(5))
print(res_.tail(5))
print("Solution: ", weights_sol)
```



```

[3. 3.]
[4. 5.]
[15. 15.]
      eps                               weights
0  2.210000                               [3.0, 4.0]
1  1.084500                               [2.0, 2.90000000000000203]
2  0.532701  [1.3099999999999983, 2.12000000000000156]
3  0.262073  [0.8359999999999975, 1.5650000000000128]
4  0.129266  [0.5122999999999966, 1.1684000000000108]
      eps                               weights
5  0.064029  [0.29299999999999615, 0.8834900000000089]
6  0.031932  [0.14605099999999605, 0.6774920000000075]
7  0.016099  [0.04909159999999599, 0.5273885000000065]
8  0.008254  [-0.01346557000000384, 0.4170016400000056]
9  0.004341  [-0.05247262000000364, 0.33494786900000495]
Solution: [-0.07547288  0.27320556]

```

First-order Gradient Descent to Minimize Linear Regression

Least squares problem solved by linear regression

Given a linear model where the output is a weighted sum of the inputs, the model can be expressed as:

$$y_i = \sum_p w_p x_{ip}$$

where:

- y_i is the predicted output for the i -th sample,
- w_p are the weights (parameters) of the model,
- x_{ip} is the p -th feature of the i -th sample.

The objective in least squares minimization is to minimize the following cost function $J(\mathbf{w})$:

$$J(\mathbf{w}) = \frac{1}{2} \sum_i \left(y_i - \sum_p w_p x_{ip} \right)^2,$$

where $\mathbf{w}$ is the vector of weights $\mathbf{w} = [w_1, w_2, \dots, w_p]^T$.

Gradient vector (Jacobian) of the least squares problem solved by linear regression

Gradient of the cost function $\nabla J(\mathbf{w})$:

$$\nabla J(\mathbf{w}) = \frac{\partial J(\mathbf{w})}{\partial w_p} \sum_i \left(\sum_q w_q x_{iq} - y_i \right) x_{ip}.$$

Note that the gradient is also called the **Jacobian** which is the vector of first-order partial derivatives of a scalar-valued function of several variables.

```
def lse(weights, X, y):
    """
    Least Squared Error function.

    Parameters
    -----
    weights: coefficients of the linear model, (n_features) numpy array
    X: input variables, (n_samples x n_features) numpy array
    y: target variable, (n_samples,) numpy array

    Returns
    -----
    Least Squared Error, scalar
    """

    y_pred = np.dot(X, weights)
    err = y_pred - y
    return np.sum(err ** 2)


def gradient_lse_lr(weights, X, y):
    """Gradient of Least Squared Error cost function of linear regression.

    Parameters
    -----
    weights: coefficients of the linear model, (n_features) numpy array
    X: input variables, (n_samples x n_features) numpy array
    y: target variable, (n_samples,) numpy array

    Returns
    -----
    Gradient array, shape (n_features,)
    """

    y_pred = np.dot(X, weights)
    err = y_pred - y
    grad = np.dot(err, X)
    return grad
```

```
import numpy as np
n_sample, n_features = 100, 2
X = np.random.randn(n_sample, n_features)
weights = np.array((3, 2))
y = np.dot(X, weights)

lr = 0.01
weights_sol, intermediate_res = \
    gradient_descent(fun=lse, x0=np.zeros(weights.shape), args=(X, y),
                    jac=gradient_lse_lr,
                    options=dict(learning_rate=lr,
                                maxiter=15,
```

(continues on next page)

(continued from previous page)

```

intermediate_res=True))

import pandas as pd
print(pd.DataFrame(intermediate_res))
print("Solution: ", weights_sol)

```

```

      eps                               weights
0  1.295793e+01                        [0.0, 0.0]
1  1.286344e-04      [3.0004562311061, 1.988767621340993]
2  2.094142e-08  [2.9998942753289706, 2.0000953997679636]
3  5.573591e-12  [3.0000015352872462, 1.9999982569697312]
Solution: [2.99999997 2.00000003]

```

4.4.2 Second-order Newton's method in optimization

Newton's method integrates the change of the curvature (ie, change of gradient direction) in the minimization process. Since gradient direction is the change of f , i.e., the first order derivative, thus the change of gradient is second order derivative of f . See [Visually Explained: Newton's Method in Optimization](#)

For univariate functions. Like gradient descent Newton's method try to locally minimize $f(w_k + t)$ given a current position w_k . However, while gradient descent use first order local estimation of f , Newton's method increases this approximation using second-order **Taylor expansion** of f around an iterate w_k :

$$f(w_k + t) \approx f(w_k) + f'(w_k)t + \frac{1}{2}f''(w_k)t^2$$

Cancelling the derivative of this expression: $\frac{d}{dt}(f(w_k) + f'(w_k)t + \frac{1}{2}f''(w_k)t^2) = 0$, provides $f'(w_k) + f''(w_k)t = 0$, and thus $t = -\frac{f'(w_k)}{f''(w_k)}$. The learning rate is $\gamma = \frac{1}{f''(w_k)}$, and optimization scheme becomes:

$$w_{k+1} = w_k - \frac{1}{f''(w_k)}f'(w_k).$$

In multidimensional problems $\mathbf{w}_k \in \mathbb{R}^p$, $[f''(\mathbf{w}_k)]^{-1}$ is the inverse of the $(p \times p)$ **Hessian matrix** containing the second-order partial derivatives of f . It is noted:

$$f''(\mathbf{w}_k) = \nabla^2 f(\mathbf{w}_k) = \mathbf{H}_{f(\mathbf{w}_k)}$$

The optimization scheme becomes:

$$\mathbf{w}_{k+1} = \mathbf{w}_k - \gamma [\mathbf{H}_{f(\mathbf{w}_k)}]^{-1} \nabla f(\mathbf{w}_k).$$

We can introduce a small step size $0 \leq \gamma < 1$ instead of $\gamma = 1$.

- Benefit of Newton's method: Convergence speed considering the change of the curvature of f to adapt the learning rate and direction.
- Problems:
 - Local minima (local optimization) for non-convex problem.
 - In large dimension, computing the Newton direction $-[f''(w_k)]^{-1}f'(w_k)$ can be an expensive operation.

Second-order Newton's Method to Minimize Linear Regression

**** Hessian Matrix of the Least Squares Problem solved by Linear Regression****

The Hessian matrix H of the least squares problem is a square matrix of second-order partial derivatives of the cost function with respect to the model parameters. It is given by:

$$H = \nabla^2 J(\mathbf{w})$$

For the least squares cost function, $J(\mathbf{w})$, the Hessian is calculated as follows:

The Hessian H is the matrix of second derivatives of $J(\mathbf{w})$ with respect to w_p and w_q . H is a measure of the curvature of J : The eigenvectors of H point in the directions of the major and minor axes. The eigenvalues measure the steepness of J along the corresponding eigendirection. Thus, each eigenvalue of H is also a measure of the covariance or spread of the inputs along the corresponding eigendirection.

$$H_{pq} = \frac{\partial^2 J(\mathbf{w})}{\partial w_p \partial w_q}$$

Given the form of the gradient, the second derivative with respect to w_p and w_q simplifies to:

$$H_{pq} = \sum_i x_{ip} x_{iq}$$

This can be written more compactly in matrix form as:

$$H = \mathbf{X}^T \mathbf{X}$$

where X is the matrix of input features (each row corresponds to a sample, and each column corresponds to a feature) with $X_{ip} = x_{ip}$.

In this case the Hessian turns out to be the same as the covariance matrix of the inputs. Thus, each eigenvalue of H is also a measure of the covariance or spread of the inputs along the corresponding eigendirection.

```
def hessian_lse_lr(weights, X, y):
    """Hessian of Least Squared Error cost function of linear regression.
    To make API compatible with scipy.optimize.minimize, it takes a required_
    ↪ weights parameters
    that is not used.

    Parameters
    -----
    weights: coefficients of the linear model, (n_features) numpy array
    It is not used, you can safely give None.
    X: input variables, (n_samples x n_features) numpy array
    y: target variable, (n_samples,) numpy array

    Returns
    -----
    Hessian array, shape (n_features, n_features)
    """
    return np.dot(X.T, X)
```

(continues on next page)

(continued from previous page)

```

weights_sol, intermediate_res = \
    gradient_descent(fun=lse, x0=np.zeros(weights.shape), args=(X, y),
                    jac=gradient_lse_lr, hess=hessian_lse_lr,
                    options=dict(learning_rate=0.01,
                                maxiter=15,
                                intermediate_res=True))

print(pd.DataFrame(intermediate_res))
print("Solution: ", weights_sol)

```

```

      eps                               weights
0  1.295793e+01                             [0.0, 0.0]
1  1.286344e-04  [3.0004562311061, 1.988767621340993]
2  2.094142e-08  [2.9998942753289706, 2.0000953997679636]
3  5.573591e-12  [3.0000015352872462, 1.9999982569697312]
Solution:  [2.99999997 2.00000003]

```

4.4.3 Quasi-Newton Methods

Quasi-Newton Methods are an alternative of Newton Methods when Hessian is unavailable or is too expensive to compute at every iteration.

The most popular quasi-Newton method is the Broyden–Fletcher–Goldfarb–Shanno algorithm BFGS

Example: Minimizes linear regression

Note, that we provide the function to be minimized (Mean Squared Error) but the expression of the gradient which is estimated numerically.

```

from scipy.optimize import minimize

result = minimize(fun=lse, x0=[0, 0], args=(X, y), method='BFGS')
b0, b1 = result.x

print("Solution: {:.e} x + {:.e}".format(b1, b0))

```

```
Solution: 2.000000e+00 x + 3.000000e+00
```

4.4.4 Gradient Descent Variants: Data Sampling Strategies

There are three variants of gradient descent, which differ on the use of the dataset made of n samples of input data $\mathbf{x}_i$'s, and possibly their corresponding targets y_i 's.

Batch gradient descent

Batch gradient descent, known also as Vanilla gradient descent, computes the gradient of the cost function with respect to the parameters θ for the **entire training dataset** :

- Choose an initial vector of parameters $\mathbf{w}_0$ and learning rate γ .

- Repeat until an approximate minimum is obtained:

$$- \mathbf{w}_{k+1} = \mathbf{w}_k - \gamma \sum_{i=1}^n \nabla f(\mathbf{w}_k, \mathbf{x}_i, y_i)$$

Advantages:

- Batch Gradient Descent is suited for convex or relatively smooth error manifolds. Since it directly towards an optimum solution.

Limitations:

- Fast convergence toward “bad” local minimum (on non-convex functions)
- As we need to calculate the gradients for the whole dataset is intractable for datasets that don’t fit in memory and doesn’t allow us to update our model online.

Stochastic gradient descent

Stochastic gradient descent (SGD) in contrast performs a parameter update for each training example $x^{(i)}$ and $y^{(i)}$. A complete passes through the training dataset is called an **epoch**. The number of epochs is a hyperparameter to be determined observing the convergence.

- Choose an initial vector of parameters $\mathbf{w}_0$ and learning rate γ .
- Repeat epochs until an approximate minimum is obtained:
 - Randomly shuffle examples in the training set.
 - For $i \in 1, \dots, n$

$$* \mathbf{w}_{k+1} = \mathbf{w}_k - \gamma \nabla f(\mathbf{w}_k, \mathbf{x}_i, y_i)$$

Advantages:

- Often provide better local minimum. Minimization will not be smooth but rather slightly erratic and jumpy. But this ‘random walk,’ of SGD’s fluctuation, enables it to jump from a basin to another, with possibly deeper, local minimum.
- Online learning

Limitations:

- Large fluctuation that ultimately complicates convergence to the exact minimum, as SGD will keep overshooting. However, when we slowly decrease the learning rate, SGD shows the same convergence behaviour as batch gradient descent, almost certainly converging to a local or the global minimum for non-convex and convex optimization respectively.
- Slow down computation by not taking advantage of vectorized numerical libraries.

Mini-batch gradient descent

Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch (subset of) training samples:

- Divide the training set in subsets of size m .
- Choose an initial vector of parameters $\mathbf{w}_0$ and learning rate γ .
- Repeat epochs until an approximate minimum is obtained:
 - Randomly pick a mini-batch.
 - For each mini-batch b

$$* \mathbf{w}_{k+1} = \mathbf{w}_k - \gamma \sum_{i=b}^{b+m} \nabla f(\mathbf{w}_k, \mathbf{x}_i, y_i)$$

Advantages:

- Reduces the variance of the parameter updates, which can lead to more stable convergence.
- Make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient very efficient. Common mini-batch sizes range between 50 and 256, but can vary for different applications.

Mini-batch gradient descent is typically the algorithm of choice when training a neural network.

4.4.5 Momentum update

Momentum and Adaptive Learning Rate Optimizers

SGD has trouble navigating ravines (areas where the surface curves much more steeply in one dimension than in another), which are common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress, along the bottom towards the local optimum as in the image below.

Source

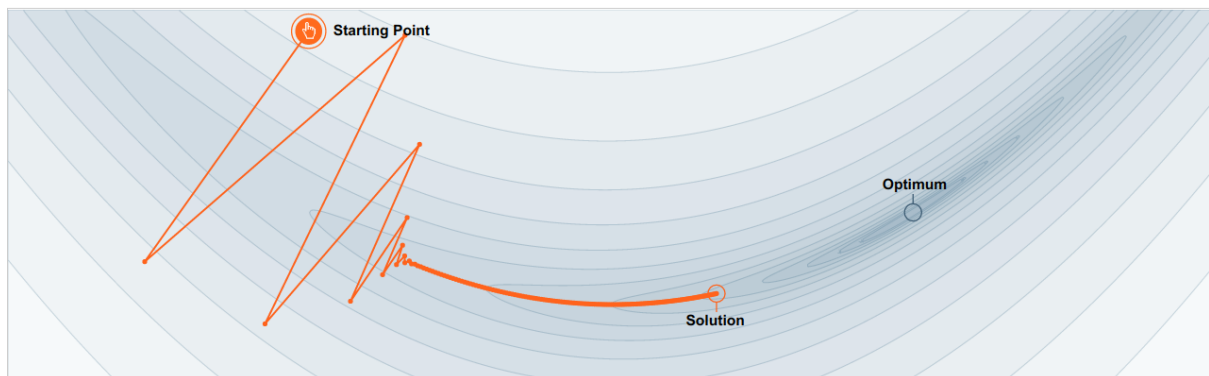


Fig. 4: No momentum: oscillations toward local largest gradient

No momentum: moving toward local largest gradient create oscillations.

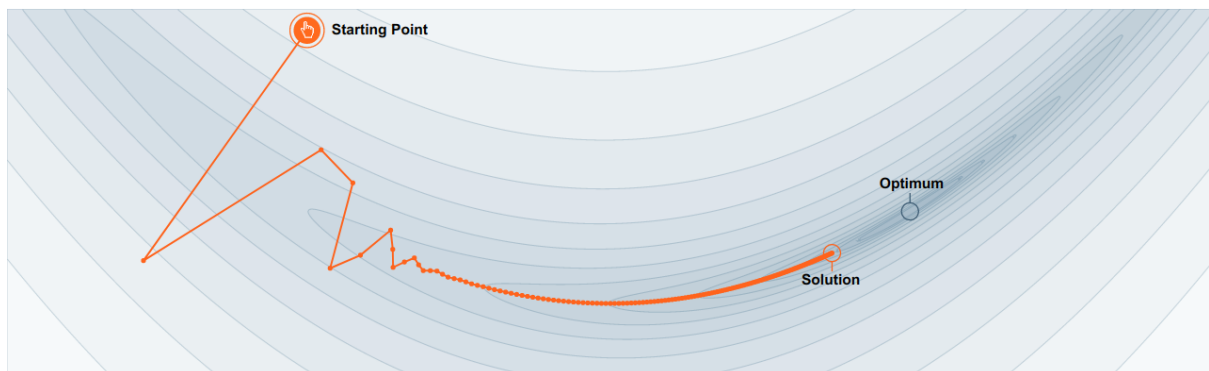


Fig. 5: With momentum: accumulate velocity to avoid oscillations

With momentum: accumulate velocity to avoid oscillations.

Momentum is a method that helps to accelerate SGD in the relevant direction and dampens oscillations as can be seen in image above. It does this by adding a fraction γ of the update

vector of the past time step to the current update vector.

$$\mathbf{v}_{k+1} = \beta \mathbf{v}_k + \nabla J(\mathbf{w}_k)$$
$$\mathbf{w}_{k+1} = \mathbf{w}_k - \gamma \nabla J(\mathbf{w}_{k+1})$$

```
v = 0
while True:
    dw = gradient(J, w)
    vx = beta * v + dw
    w -= learning_rate * v
```

Note: The momentum term **math:**`beta` is usually set to 0.9 or a similar value.

Essentially, when using momentum, we push a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way, until it reaches its terminal velocity if there is air resistance, i.e. $\beta < 1$.

The same thing happens to our parameter updates: The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, we gain faster convergence and reduced oscillation.

AdaGrad: Adaptive Learning Rates

- Added element-wise scaling of the gradient based on the historical sum of squares in each dimension.
- “Per-parameter learning rates” or “adaptive learning rates”

```
grad_squared = 0
while True:
    dw = gradient(J, w)
    grad_squared += dw * dw
    w -= learning_rate * dw / (np.sqrt(grad_squared) + 1e-7)
```

- Progress along “steep” directions is damped.
- Progress along “flat” directions is accelerated.
- Problem: step size over long time => Decays to zero.

RMSPProp: “Leaky AdaGrad”

```
grad_squared = 0
while True:
    dw = gradient(J, w)
    grad_squared += decay_rate * grad_squared + (1 - decay_rate) * dw * dw
    w -= learning_rate * dw / (np.sqrt(grad_squared) + 1e-7)
```

- decay_rate = 1: gradient descent
- decay_rate = 0: AdaGrad

Nesterov accelerated gradient

However, a ball that rolls down a hill, blindly following the slope, is highly **unsatisfactory**. We'd like to have a smarter ball, a ball that has a **notion of where it is going** so that it **knows to slow down before the hill slopes up again**. Nesterov accelerated gradient (NAG) is a way to give **our momentum term this kind of prescience**. We know that we will use our momentum term γv_{t-1} to move the parameters θ .

Computing $\theta - \gamma v_{t-1}$ thus gives us **an approximation of the next position of the parameters** (the gradient is missing for the full update), a rough idea where our parameters are going to be. We can now effectively look ahead by calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters:

$$\begin{aligned}\mathbf{v}_t &= \gamma \mathbf{v}_{t-1} + \eta \nabla_{\mathbf{w}} J(\mathbf{w} - \gamma v_{t-1}) \\ \mathbf{w} &= \mathbf{w} - v_t\end{aligned}$$

Again, we set the momentum term γ to a value of around 0.9. While **Momentum first computes the current gradient and then takes a big jump in the direction of the updated accumulated gradient**, NAG **first makes a big jump in the direction of the previous accumulated gradient, measures the gradient and then makes a correction, which results in the complete NAG update**. This anticipatory update **prevents us from going too fast** and results in **increased responsiveness**, which has significantly **increased the performance of RNNs** on a number of tasks

Adam

Adaptive Moment Estimation (Adam) is a method that computes **adaptive learning rates** for each parameter. In addition to storing an **exponentially decaying average of past squared gradients** v_t , Adam also keeps an **exponentially decaying average of past gradients** m_t , **similar to momentum**. Whereas momentum can be seen as a ball running down a slope, Adam behaves like a **heavy ball with friction**, which thus prefers **flat minima in the error surface**. We compute the decaying averages of past and past squared gradients $\mathbf{m}_t$ and $\mathbf{v}_t$ respectively as follows:

$$\begin{aligned}\mathbf{m}_t &= \beta_1 \mathbf{m}_{t-1} + (1 - \beta_1) \nabla_{\mathbf{w}} J(\mathbf{w}) \\ \mathbf{v}_t &= \beta_2 \mathbf{v}_{t-1} + (1 - \beta_2) \nabla_{\mathbf{w}} J(\mathbf{w})^2\end{aligned}$$

$\mathbf{m}_t$ and $\mathbf{v}_t$ are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
    dx = gradient(J, x)
    # Momentum:
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    # AdaGrad/RMSProp
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7)
```

As $\mathbf{m}_t$ and $\mathbf{v}_t$ are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1). They counteract these biases by computing bias-corrected

first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

They then use these to update the parameters (Adam update rule):

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

- $\hat{m}_t$ Accumulate gradient: velocity.
- $\hat{v}_t$ Element-wise scaling of the gradient based on the historical sum of squares in each dimension.
- Choose Adam as default optimizer
- Default values of 0.9 for β_1 , 0.999 for β_2 , and 10^{-7} for ϵ .
- learning rate in a range between $1e - 3$ and $5e - 4$

4.4.6 Conclusion

Sources:

- LeCun Y.A., Bottou L., Orr G.B., Müller K.R. (2012) [Efficient BackProp](#). In: Montavon G., Orr G.B., Müller K.R. (eds) *Neural Networks: Tricks of the Trade*. Lecture Notes in Computer Science, vol 7700. Springer, Berlin, Heidelberg
- Introduction to [Gradient Descent Algorithm](#) (along with variants) in *Machine Learning: Gradient Descent with Momentum, ADAGRAD and ADAM*.

Summary:

- Choosing a proper learning rate can be difficult. A learning rate that is too small leads to painfully slow convergence, while a learning rate that is too large can hinder convergence and cause the loss function to fluctuate around the minimum or even to diverge.
- Learning rate schedules try to adjust the learning rate during training by e.g. annealing, i.e. reducing the learning rate according to a pre-defined schedule or when the change in objective between epochs falls below a threshold. These schedules and thresholds, however, have to be defined in advance and are thus unable to adapt to a dataset's characteristics.
- Additionally, the same learning rate applies to all parameter updates. If our data is sparse and our features have very different frequencies, we might not want to update all of them to the same extent, but perform a larger update for rarely occurring features.
- Another key challenge of minimizing highly non-convex error functions common for neural networks is avoiding getting trapped in their numerous suboptimal local minima. These [saddle points](#) are usually surrounded by a plateau of the same error, which makes it notoriously hard for SGD to escape, as the gradient is close to zero in all dimensions.

Recommendation:

- Shuffle the examples (SGD)
- Center the input variables by subtracting the mean

- Normalize the input variable to a standard deviation of 1
- Initializing the weight
- Adaptive learning rates (momentum), using separate learning rate for each weight

STATISTICS

5.1 Univariate Statistics

Basics univariate statistics are required to explore dataset:

- Discover associations between a variable of interest and potential predictors. It is strongly recommended to start with simple univariate methods before moving to complex multivariate predictors.
- Assess the prediction performances of machine learning predictors.
- Most of the univariate statistics are based on the linear model which is one of the main model in machine learning.

5.1.1 Libraries

Statistics

- Descriptive statistics and distributions: [Numpy](#)
- Distributions and tests: [scipy.stats](#)
- Advanced statistics (linear models, tests, time series): [Statsmodels](#), see also [Statsmodels API](#):
 - `statsmodels.api`: Imported using `import statsmodels.api as sm`.
 - `statsmodels.formula.api`: A convenience interface for specifying models using formula strings and DataFrames. Canonically imported using `import statsmodels.formula.api as smf`
 - `statsmodels.tsa.api`: Time-series models and methods. Canonically imported using `import statsmodels.tsa.api as tsa`.

```
# Manipulate data
import numpy as np
import pandas as pd

# Statistics
import scipy.stats
import statsmodels.api as sm
#import statsmodels.stats.api as sms
import statsmodels.formula.api as smf
from statsmodels.stats.stattools import jarque_bera
```

Plots

```
import matplotlib.pyplot as plt
import seaborn as sns
import pystatsml.plot_utils

# Plot parameters
plt.style.use('seaborn-v0_8-whitegrid')
fig_w, fig_h = plt.rcParams.get('figure.figsize')
plt.rcParams['figure.figsize'] = (fig_w, fig_h * .5)
```

Datasets

Salary

```
try:
    salary = pd.read_csv("../datasets/salary_table.csv")
except:
    url = 'https://github.com/duchesnay/pystatsml/raw/master/datasets/salary_
↪table.csv'
    salary = pd.read_csv(url)
```

Iris

```
# Load iris dataset
iris = sm.datasets.get_rdataset("iris").data
iris.columns = [s.replace('.', '') for s in iris.columns]
iris.columns
```

```
Index(['SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth', 'Species'],
↪dtype='object')
```

5.1.2 Descriptive Statistics

Mean

Properties of the expected value operator $E(\cdot)$ of a random variable X

$$\begin{aligned}E(X + c) &= E(X) + c \\E(X + Y) &= E(X) + E(Y) \\E(aX) &= aE(X)\end{aligned}$$

The estimator $\bar{x}$ on a sample of size n : $x = x_1, \dots, x_n$ is given by

$$\bar{x} = \frac{1}{n} \sum_i x_i$$

$\bar{x}$ is itself a random variable with properties:

- $E(\bar{x}) = \bar{x}$,
- $\text{Var}(\bar{x}) = \frac{\text{Var}(X)}{n}$.

Variance

$$\text{Var}(X) = E((X - E(X))^2) = E(X^2) - (E(X))^2$$

The estimator is

$$\sigma_x^2 = \frac{1}{n-1} \sum_i (x_i - \bar{x})^2$$

Note here the subtracted 1 degree of freedom (df) in the divisor. In standard statistical practice, $df = 1$ provides an unbiased estimator of the variance of a hypothetical infinite population. With $df = 0$ it instead provides a maximum likelihood estimate of the variance for normally distributed variables.

Standard deviation

$$\text{Std}(X) = \sqrt{\text{Var}(X)}$$

The estimator is simply $\sigma_x = \sqrt{\sigma_x^2}$.

Covariance

$$\text{Cov}(X, Y) = E((X - E(X))(Y - E(Y))) = E(XY) - E(X)E(Y).$$

Properties:

$$\text{Cov}(X, X) = \text{Var}(X)$$

$$\text{Cov}(X, Y) = \text{Cov}(Y, X)$$

$$\text{Cov}(cX, Y) = c \text{Cov}(X, Y)$$

$$\text{Cov}(X + c, Y) = \text{Cov}(X, Y)$$

The estimator with $df = 1$ is

$$\sigma_{xy} = \frac{1}{n-1} \sum_i (x_i - \bar{x})(y_i - \bar{y}).$$

Correlation

$$\text{Cor}(X, Y) = \frac{\text{Cov}(X, Y)}{\text{Std}(X) \text{Std}(Y)}$$

The estimator is

$$\rho_{xy} = \frac{\sigma_{xy}}{\sigma_x \sigma_y}.$$

Standard Error (SE)

The standard error (SE) is the standard deviation (of the sampling distribution) of a statistic. It is most commonly considered for the estimator of the mean ($\bar{x}$) whose estimator $\sigma_{\bar{x}}$ is:

$$\sigma_{\bar{x}} = \frac{\sigma_x}{\sqrt{n}}$$

Descriptive statistics with Numpy

- Generate 2 random samples: $x \sim N(1.78, 0.1)$ and $y \sim N(1.66, 0.1)$, both of size 10.
- Compute $\bar{x}, \sigma_x, \sigma_{xy}$ (xbar, xvar, xycov) using only the np.sum() operation.

Explore the np. module to find out which Numpy functions performs the same computations and compare them (using assert) with your previous results.

Caution! By default np.var() used the biased estimator (with ddof=0). Set ddof=1 to use unbiased estimator.

```
n = 10
np.random.seed(seed=42) # make the example reproducible
x = np.random.normal(loc=1.78, scale=.1, size=n)
y = np.random.normal(loc=1.66, scale=.1, size=n)

xbar = np.mean(x)
assert xbar == np.sum(x) / x.shape[0]

xvar = np.var(x, ddof=1)
assert xvar == np.sum((x - xbar) ** 2) / (n - 1)
```

Covariance

```
xycov = np.cov(x, y)
print(xycov)

ybar = np.sum(y) / n
assert np.allclose(xycov[0, 1], np.sum((x - xbar) * (y - ybar)) / (n - 1))
assert np.allclose(xycov[0, 0], xvar)
assert np.allclose(xycov[1, 1], np.var(y, ddof=1))
```

```
[[ 0.00522741 -0.00060351]
 [-0.00060351  0.00570515]]
```

Descriptives Statistics on Iris Dataset

With Pandas

Columns' means

```
iris[['SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth']].mean()
```

```
SepalLength    5.843333
SepalWidth     3.057333
PetalLength    3.758000
PetalWidth     1.199333
dtype: float64
```

Columns' std-dev. Pandas normalizes by N-1 by default.

```
iris[['SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth']].std()
```



```
SepalLength    0.828066
SepalWidth     0.435866
PetalLength    1.765298
PetalWidth     0.762238
dtype: float64
```

With Numpy

```
X = iris[['SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth']].values
X.mean(axis=0)
```

```
array([5.84333333, 3.05733333, 3.758      , 1.19933333])
```

Columns' std-dev. Numpy normalizes by N by default. Set ddof=1 to normalize by N-1 to get the unbiased estimator.

```
X.std(axis=0, ddof=1)
```

```
array([0.82806613, 0.43586628, 1.76529823, 0.76223767])
```

5.1.3 Probability Distributions

- Probabilities of occurrence of possible outcomes
- Description of a random phenomenon in terms of its sample space

Terminology:

- Random variable, RV: X : takes values from a sample space.
- **Probability Density Function PDF**: $P(X) \in [0, 1]$ for $X \in \mathbb{R}$ or **Probability mass function PMF** if X is a discrete RV.
- **Cumulative Distribution Function CDF** $P(X \leq x)$.
- **Percent Point Function** or **Quantile Function** (inverse of CDF), i.e., values of x such $P(X \leq x) = a$ given quantile.

Histogram as probability density function estimator

`numpy.histogram` can be used to probability density function at the each histogram bin, setting `density=True` parameter. Warning, `histogram doesn't sum to 1`. Histogram as PDF estimator should be multiplied by `dx`'s to sum to 1.

```
x = np.random.normal(size=50000)
hist, bins = np.histogram(x, bins=50, density=True)
dx = np.diff(bins)
print("Sum(Hist)=", np.sum(hist), "Sum(Hist * dx)=", np.sum(hist * dx))

pdf = scipy.stats.norm.pdf(bins) # True probability density function

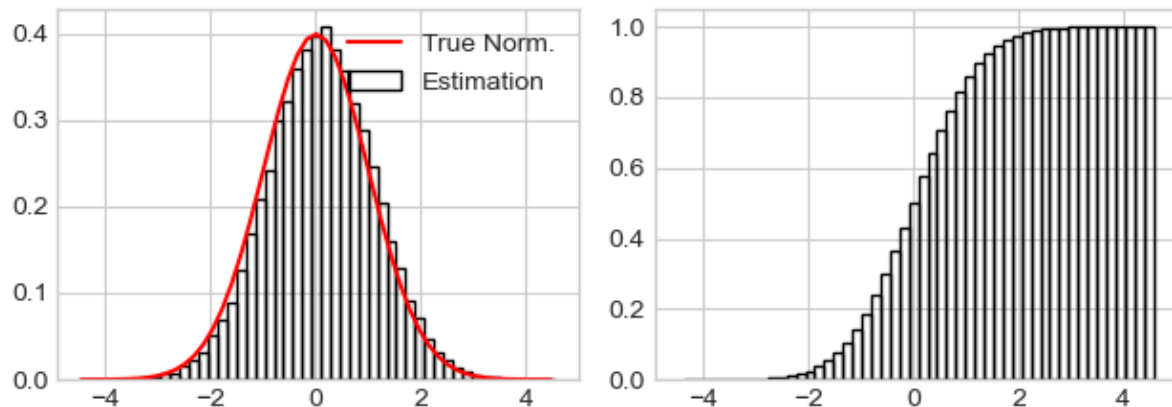
fig, (ax1, ax2) = plt.subplots(nrows=1, ncols=2, sharex=True)
ax1.bar(bins[1:], hist, width=dx, fill=False, label="Estimation")
```

(continues on next page)

(continued from previous page)

```
ax1.plot(bins, pdf, 'r-', label="True Norm.")
ax1.legend()
ax2.bar(bins[1:], (hist * dx).cumsum(), fill=False, width=dx)
fig.tight_layout()
```

```
Sum(Hist)= 5.589909828006291 Sum(Hist * dx)= 1.0
```



Kernel Density Estimation (KDE)

TODO

Normal distribution

The normal distribution, noted $\mathcal{N}(\mu, \sigma)$ with parameters: μ mean (location) and $\sigma > 0$ std-dev. Estimators: $\bar{x}$ and σ_x .

The normal distribution, noted $\mathcal{N}$, is useful because of the central limit theorem (CLT) which states that: given certain conditions, the arithmetic mean of a sufficiently large number of iterates of independent random variables, each with a well-defined expected value and well-defined variance, will be approximately normally distributed, regardless of the underlying distribution.

Documentation:

- [numpy.random.normal](#)
- [scipy.stats.norm](#)

Random number generator using Numpy

```
# using numpy:
x = np.random.normal(loc=10, scale=10, size=(3, 2))
```

Distribution using Scipy

- Random number generator $X \sim \mathcal{N}(\mu, \sigma^2)$: `norm.rvs(loc=mean, scale=sd, size=n)`
- **Probability Density Function (PDF)**: $P(X) \in [0,1]$ for $X \in \mathbb{R}$: `norm.pdf(values, loc=mean, scale=sd)`
- **Cumulative Distribution Function (CDF)** $P(X \leq x)$: `norm.cdf(x, loc=mean, scale=sd)`

- Percent Point Function (inverse of CDF), i.e., values of x such $P(X < x) = a$ given percentiles : `norm.ppf(q, loc=0, scale=1)`

```
mean, sd, n = 0, 1, 10000
```

```
# Random number generator
```

```
x_rv = scipy.stats.norm.rvs(loc=mean, scale=sd, size=n)
```

```
x_range = np.linspace(mean-3*sd, mean+3*sd, 100)
```

```
# PDF: P(values)
```

```
pdf_x_range = scipy.stats.norm.pdf(x_range, loc=mean, scale=sd)
```

```
# CDF: P(X < values)
```

```
cdf_x_range = scipy.stats.norm.cdf(x_range, loc=mean, scale=sd)
```

```
# PPF: Values such P(X < values) = percentiles
```

```
percentiles_of_cdf = [0.025, 0.10, 0.25, 0.5, 0.75, 0.90, 0.975]
```

```
x_for_percentiles_of_cdf = scipy.stats.norm.ppf(percentiles_of_cdf,
                                                  loc=mean, scale=sd)
```

```
# Percentiles of CDF = CDF(x values for CDF percentiles)
```

```
assert np.allclose(percentiles_of_cdf,
                    scipy.stats.norm.cdf(x_for_percentiles_of_cdf))
```

```
# Values for percentiles of CDF
```

```
["P(X<{:0.2f})={:0.1%}".format(val, ppf) for
```

```
    ppf, val in zip(percentiles_of_cdf,
                    scipy.stats.norm.ppf(percentiles_of_cdf,
                                          loc=mean, scale=sd))]
```

```
['P(X<-1.96)=2.5%',
 'P(X<-1.28)=10.0%',
 'P(X<-0.67)=25.0%',
 'P(X<0.00)=50.0%',
 'P(X<0.67)=75.0%',
 'P(X<1.28)=90.0%',
 'P(X<1.96)=97.5%']
```

Plot histogram, true distribution (PDF), and area of CDF

```
_ = plt.hist(x_rv, density=True, bins=100, fill=False,
             label="Histogram (Estimator)")
```

```
_ = plt.plot(x_range, pdf_x_range, 'r-', label="PDF: P(X)")
```

```
# PPF: Values such P(X < values) = 2.5%
```

```
percentile_of_cdf = 0.025
```

```
x_for_percentile_of_cdf = scipy.stats.norm.ppf(percentile_of_cdf,
                                                  loc=mean, scale=sd)
```

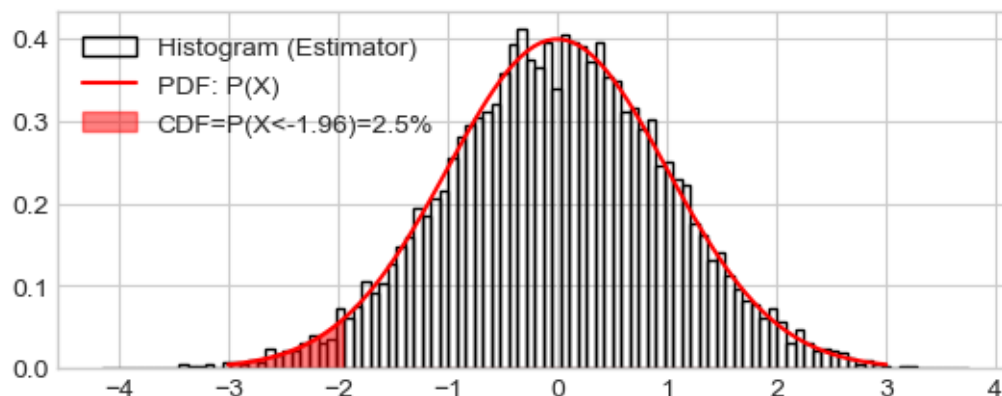
```
x_range = np.linspace(mean-3*sd, mean+3*sd, 10000)
```

```
pdf_x_range = scipy.stats.norm.pdf(x_range, loc=mean, scale=sd)
```

(continues on next page)

(continued from previous page)

```
pdf_x_range[x_range > x_for_percentile_of_cdf] = 0
_ = plt.fill_between(x=x_range, y1=np.zeros(len(pdf_x_range)), y2=pdf_x_range,
                    alpha=.5,
                    label="CDF=P(X<{:0.02f})={:0.01%}".format(x_for_percentile_of_
↪cdf,
                                                            percentile_of_cdf),
                    color='r')
_ = plt.legend()
```



The Chi-Square Distribution

The chi-square or χ_n^2 distribution with n degrees of freedom (df) is the distribution of a sum of the squares of n independent standard normal random variables $\mathcal{N}(0, 1)$. Let $X \sim \mathcal{N}(\mu, \sigma^2)$, then, $Z = (X - \mu)/\sigma \sim \mathcal{N}(0, 1)$, then:

- The squared standard $Z^2 \sim \chi_1^2$ (one df).
- **The distribution of sum of squares** of n normal random variables: $\sum_i^n Z_i^2 \sim \chi_n^2$

The sum of two χ^2 RV with p and q df is a χ^2 RV with $p + q$ df. This is useful when summing/subtracting sum of squares.

The χ^2 -distribution is used to model **errors** measured as **sum of squares** or the distribution of the sample **variance**.

The chi-squared distribution is a special case of the gamma distribution, with gamma parameters $a = df/2$, $loc = 0$ and $scale = 2$.

Documentation: - [numpy.random.chisquare](#) - [scipy.stats.chi2](#)

```
df, mean, sd, n = 30, 0, 1, 10000
x_rv = scipy.stats.chi2.rvs(df=df, loc=mean, scale=sd, size=n)
x_range = np.linspace(mean-3*sd, mean+3*sd, 100)
prob_x_range = scipy.stats.chi2.pdf(x_range, df=df, loc=mean, scale=sd)
cdf_x_range = scipy.stats.chi2.cdf(x_range, df=df, loc=mean, scale=sd)
```

The Fisher's F-Distribution

The F -distribution, $F_{n,p}$, with n and p degrees of freedom is the ratio of two independent χ^2 variables. Let $X \sim \chi_n^2$ and $Y \sim \chi_p^2$ then:

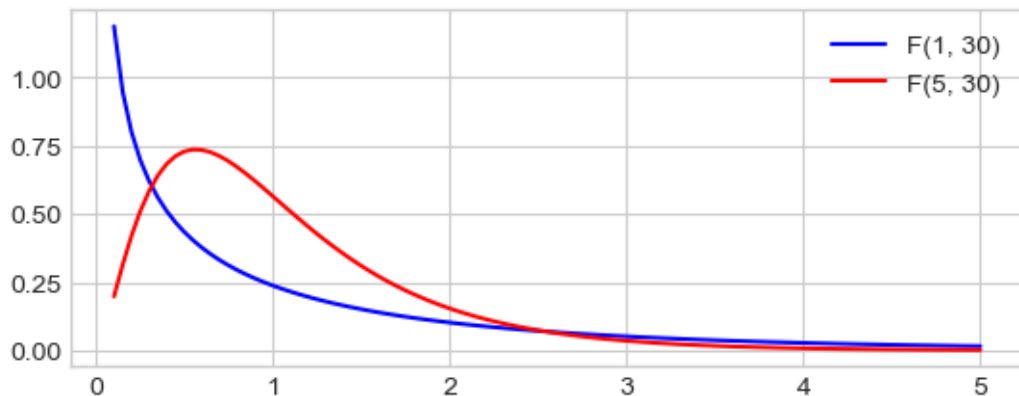
$$F_{n,p} = \frac{X/n}{Y/p}$$

The F -distribution plays a central role in hypothesis testing answering the question: **Are two variances equals?, is the ratio or two errors significantly large ?.**

Documentation: - [scipy.stats.f](#)

```
dfn, dfd, mean, sd, n = 30, 5, 0, 1, 10000
x_rv = scipy.stats.f.rvs(dfn=dfn, dfd=dfd, loc=mean, scale=sd, size=n)
x_range = np.linspace(mean-3*sd, mean+3*sd, 100)
prob_x_range = scipy.stats.f.pdf(x_range, dfn=dfn, dfd=dfd, loc=mean, scale=sd)
cdf_x_range = scipy.stats.f.cdf(x_range, dfn=dfn, dfd=dfd, loc=mean, scale=sd)

fvalues = np.linspace(.1, 5, 100)
# pdf(x, df1, df2): Probability density function at x of F.
plt.plot(fvalues, scipy.stats.f.pdf(fvalues, 1, 30), 'b-', label="F(1, 30)")
plt.plot(fvalues, scipy.stats.f.pdf(fvalues, 5, 30), 'r-', label="F(5, 30)")
_ = plt.legend()
```



The Student's t -Distribution

Let $M \sim \mathcal{N}(0, 1)$ and $V \sim \chi_n^2$. The t -distribution, T_n , with n degrees of freedom is the ratio:

$$T_n = \frac{M}{\sqrt{V/n}}$$

The distribution of the difference between an estimated parameter and its true (or assumed) value divided by the standard deviation of the estimated parameter (standard error) follow a t -distribution.

Documentation: [scipy.stats.t](#)

```
df, mean, sd, n = 30, 0, 1, 10000
x_rv = scipy.stats.t.rvs(df=df, loc=mean, scale=sd, size=n)
x_range = np.linspace(mean-5*sd, mean+5*sd, 100)
```

(continues on next page)

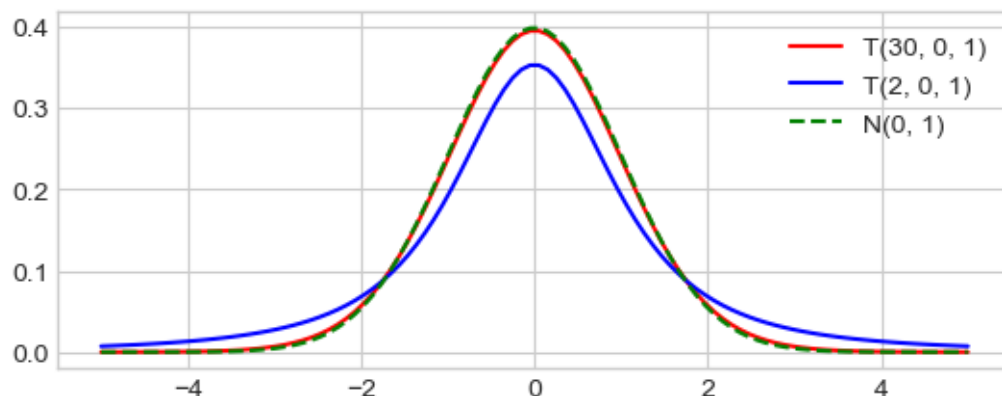
(continued from previous page)

```

prob_x_range = scipy.stats.t.pdf(x_range, df=df, loc=mean, scale=sd)
cdf_x_range = scipy.stats.t.cdf(x_range, df=df, loc=mean, scale=sd)

plt.plot(x_range, scipy.stats.t.pdf(x_range, 30), 'r-', label="T(30, 0, 1)")
plt.plot(x_range, scipy.stats.t.pdf(x_range, 2), 'b-', label="T(2, 0, 1)")
plt.plot(x_range, scipy.stats.norm.pdf(x_range, loc=mean, scale=sd), 'g--',
         label="N(0, 1)")
_ = plt.legend()

```



5.1.4 Central Limit Theorem (CLT)

See [3Blue1Brown](#): But what is the Central Limit Theorem?.

Let $\{X_1, \dots, X_i, \dots, X_n\}$ be a sequence of independent and identically distributed (i.i.d.) ($\sim X$) random variables (RV) with parameters:

- $E[X_i] = \mu_X$,
- $Var[X_i] = \sigma_X^2 < \infty$ (finite variance).

Distribution of the Sum of Samples

Let $S_n = \sum_{i=1}^n X_i$ the sum of those RV. Then, the sum of RV converge in distribution to a normal distribution:

$$S_n = \sum_{i=1}^n X_i \rightarrow \mathcal{N}(n\mu_X, \sqrt{n}\sigma_X)$$

Note that the centered and scaled sum converge in distribution to a normal distribution of parameters 0, 1: $\frac{\sum_{i=1}^n X_i - n\mu_X}{\sqrt{n}\sigma_X} \rightarrow \mathcal{N}(0, 1)$

The Distribution of the Sample Mean

Central Limit Theorem also apply for the sample mean: Let i.i.d. samples X_i from from *almost any* distribution of parameters μ_X, σ_X . Then the sample mean $\bar{X}$, for samples of size 30 or more, is approximately normally distributed:

$$\bar{X} = \frac{\sum_{i=1}^n X_i}{n} \rightarrow \mathcal{N}(\mu_X, \frac{\sigma_X}{\sqrt{n}})$$

Simple but useful demonstrations:

$$E[\bar{X}_n] = E\left[\frac{1}{n} \sum_i^n X_i\right] = \frac{1}{n} \sum_i^n E[X_i],$$

since X_i are i.i.d., i.e., $E[X_i] = \mu_X \forall i$ then

$$= \frac{1}{n} n \mu_X = \mu_X$$

$$\text{Var}[\bar{X}_n] = \text{Var}\left[\frac{1}{n} \sum_i^n X_i\right] = \left(\frac{1}{n}\right)^2 \sum_X^2 \text{Var}[X_i],$$

since X_i are i.i.d., i.e., $\text{Var}[X_i] = \sigma_X^2 \forall i$ then

$$= \left(\frac{1}{n}\right)^2 n \sigma_X^2 = \sigma_X^2 / n$$

and

$$\text{Sd}[\bar{X}_n] = \sigma_X / \sqrt{n}$$

Note that the **standard deviation of the sample mean is the standard deviation of the parent RV scaled by $\sqrt{n}$** . The larger the sample size, the better the approximation.

The Central Limit Theorem is illustrated for several common population distributions in [The Sampling Distribution of the Sample Mean](#).

Examples

Distribution of the sum of samples from the uniform distribution

See [Scipy Uniform Distribution](#)

```
n_sample = 1000
n_repeat = 10000

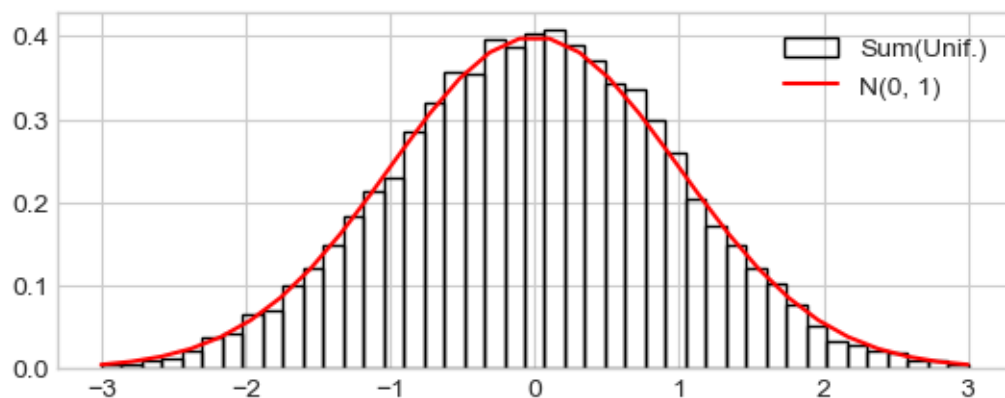
# Distribution parameters, true mean and standard deviation
a, b = 0, 1
mu_unif, sd_unif = 1 / 2 * (b - a), np.sqrt(1 / 12 * (b - a) ** 2)

# Xn's
xn_s = np.array([scipy.stats.uniform.rvs(size=n_sample).sum()
                 for i in range(n_repeat)])

# Xn's centered and scaled
xn_s_cs = (xn_s - n_sample * mu_unif) / (np.sqrt(n_sample) * sd_unif)

h_ = plt.hist(xn_s_cs, range=(-3, 3), density=True, bins=43, fill=False,
              label="Sum(Unif.)")

# Normal distribution
x_range = np.linspace(-3, 3, 30)
prob_x_range = scipy.stats.norm.pdf(x_range, loc=0, scale=1)
plt.plot(x_range, prob_x_range, 'r-', label="N(0, 1)")
_ = plt.legend()
```



Distribution of the sum of samples from the the exponential distribution

See [Scipy Exponential Distribution](#)

```
n_sample = 1000
n_repeat = 10000

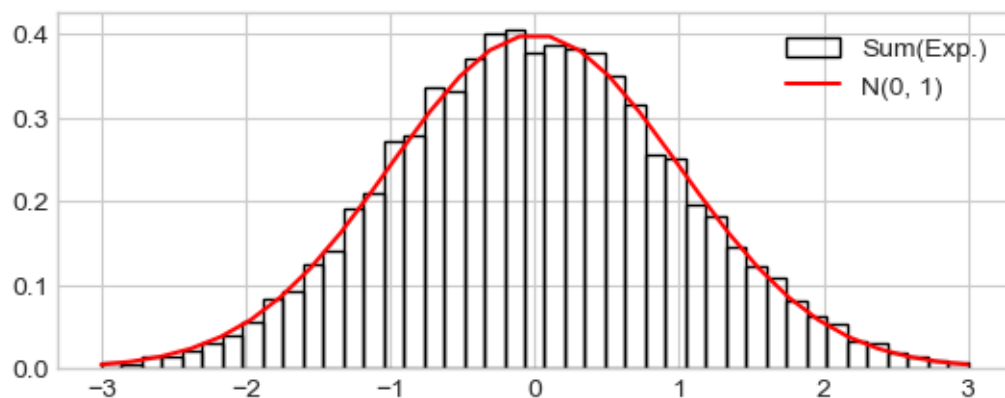
# Distribution parameters, true mean and standard deviation
lambda_ = 1
mu_exp, sd_exp = 1 / lambda_, np.sqrt(1 / (lambda_ ** 2))

# Xn's
xn_s = np.array([scipy.stats.expon.rvs(size=n_sample).sum()
                 for i in range(n_repeat)])

# Xn's centered and scaled
xn_s_cs = (xn_s - n_sample * mu_exp) / (np.sqrt(n_sample) * sd_exp)

h_ = plt.hist(xn_s_cs, range=(-3, 3), density=True, bins=43, fill=False,
              label="Sum(Exp.)")

# Normal distribution
x_range = np.linspace(-3, 3, 30)
prob_x_range = scipy.stats.norm.pdf(x_range, loc=0, scale=1)
plt.plot(x_range, prob_x_range, 'r-', label="N(0, 1)")
_ = plt.legend()
```



Distribution of the mean from the Binomial distribution

Binomial distribution with scipy:

```
n_sample = 1000
n_repeat = 10000

# Binomial distribution parameters
n, p = n_sample, 0.5

# Distribution parameters, true mean and standard deviation
mu, sd = n * p, np.sqrt(n * p * (1 - p))

# Xbar's
xbar_s = np.array([scipy.stats.binom.rvs(n=n, p=p, size=n_sample).mean()
                   for i in range(n_repeat)])

print("True stat.: mu={:.01f}, sd={:.03f}".format(mu, sd / np.sqrt(n_sample)))
print("Est. stat.: mu={:.01f}, sd={:.03f}".format(xbar_s.mean(), xbar_s.std()))
```

```
True stat.: mu=500.0, sd=0.500
Est. stat.: mu=500.0, sd=0.504
```

5.1.5 Statistical inference and Decision Making using Hypothesis Testing

Inferential statistics involves the use of a sample (1) to estimate some characteristic in a large population; and (2) to test a research hypothesis about a given population.

Typology of tests

Tests should be adapted to the type of variable:

1. For **categorical variables** tests use **count** of categories, **proportions**, or **frequencies**.
Examples:
 - Test a proportion: 200 heads have been found over 300 flips, is this coin biased toward head or could it be observed by chance?
 - 1,000 voters are questioned about their vote. 55% said they had voted for candidate A and 45% for candidate B. Is this a significant difference?
2. For **numerical variables** tests use **means**, **standard-deviations** or **medians**. Examples:
 - Test the effect of some condition (treatment or some action):
 - We observed an increase of monthly revenue of 100 stores after marketing campaign. Could this increase be attributed to chance or to the marketing campaign?
 - Arterial hypertension of 50 patients has been reduced by some medication. Is it pure randomness?
 - Test the association between two variables:
 - Height and sex: In a sample of 25 individuals (15 females, 10 males), is female height is different from male height?

- Age and arterial hypertension: In a sample of 25 individuals is age height correlated with arterial hypertension ?

Tests can be grouped in two categories:

1. **Parametric tests** assume that the data follow some distribution, and can be summarized by a parameters: mean and standard-deviation for quantitative variables; count proportion or frequencies for categorical variables.
2. **Non-Parametric tests**. Non-parametric tests are not based on a model of the data and therefore do not make the assumption that they follow a certain distribution. Non-Parametric tests are generally based on ranking of values or medians.

General Testing Procedure

1. Model the data (for parametric tests).

E.g., the height of males and females can be represented by their means, i.e., assuming two normal distributions. Then fit the model to the data, i.e., estimate the model parameters (frequency, mean, correlation, regression coefficient). E.g., compute the means of females and males height.

2. Calculate a decision statistic (for all tests)

- Formulate the null hypothesis H_0 , i.e., what would be situation under pure chance? E.g., if sex has no effect on individuals' height males and females means height will be equals.
- Derive a test statistic on the data capturing deviation from null hypothesis taking account the number of samples. For parametric statistics, the test statistic is derived from model parameters, e.g., the differences of means of males and females height, taking account the number of samples.

3. Inference

Assess the deviation of the test statistic from its expected value under H_0 (pure chance). Two possibilities:

P-value based on null hypothesis:

What is the probability that the computed test statistic $\bar{X}$ would be observed under pure chance? I.e., What is the probability that the test statistics under H_0 would be more extreme, i.e., “larger” or “smaller” than $\bar{X}$?

- Calculate the distribution of test statistic X under H_0 .
- Compute the probability (P-value) to obtain a larger test statistic by chance (under the null hypothesis).

For a symmetric distribution the two sided p-value P is defined as:

- $P(\bar{X} \leq X|H_0) = P/2$, or
- $P(X \geq \bar{X}) = P/2$

$\bar{X}$ is declared to be **significantly different to the null hypothesis** if the p-value is less than a **significance level** α generally considered as 5%.

Confidence interval (CI)

CI is a range of values x_1, x_2 that is likely (given a **confidence level**, e.g., 95%) to contain the true value of the statistic $\bar{X}$. Outside this range the value is considered to be unlikely. Note that the confidence level is $1 - \alpha$, the significance level. See [Interpreting Confidence Intervals](#)

The 95% CI (Confidence Interval) is the range of values x_1, x_2 such $P(x_1 < \bar{X} < x_2) = 95\%$.

For a symmetric distribution the two sided 95% (= 1 - 5%) confidence interval, is defined as:

- x_1 such $P(\bar{X} \leq x_1) = 2.5\% = 5\%/2$
- x_2 such $P(x_2 \leq \bar{X}) = 2.5\%$

Terminology

- Margin of error = $\bar{X} - x_1$ (for symmetric distribution).
- Confidence Interval = $[x_1, x_2]$.
- Confidence level is 1 - significance level

Categorical variable: the Binomial Test

Simplified example (small sample) of the [the binomial test](#): Three voters are questioned about their vote. Two voted for candidate A and one for B. How likely this difference Is this a significant difference,

1. Model the data: Let x the number of vote for A. It follows a [Binomial distribution](#). Compute the model parameters: $N = 3$, and $\hat{p} = 2/3$ (the frequency of number of vote A over the number of voters).

2. Compute a test statistic measuring the deviation of the number of vote for A ($x = 2$) over three voters from the expected values under the null hypothesis where x would be 1.5. Similarly, we could consider the deviation of the observed proportion $\hat{p} = 2/3$ from $\pi_0 = 50\%$.

3. To make inference, we have to compute the probability to obtain more than two votes for A by chance. We need the distribution of x under H_0 ($P(x|H_0)$) to sum all the probabilities where x is larger or equal to 2, i.e., $P(x \geq 2|H_0)$. With such small sample size ($n = 3$) this distribution is obtained by enumerating all configurations that produce a given number of heads x :

1	2	3	count vote for A
			0
H			1
	H		1
		H	1
H	H		2
H		H	2
	H	H	2
H	H	H	3

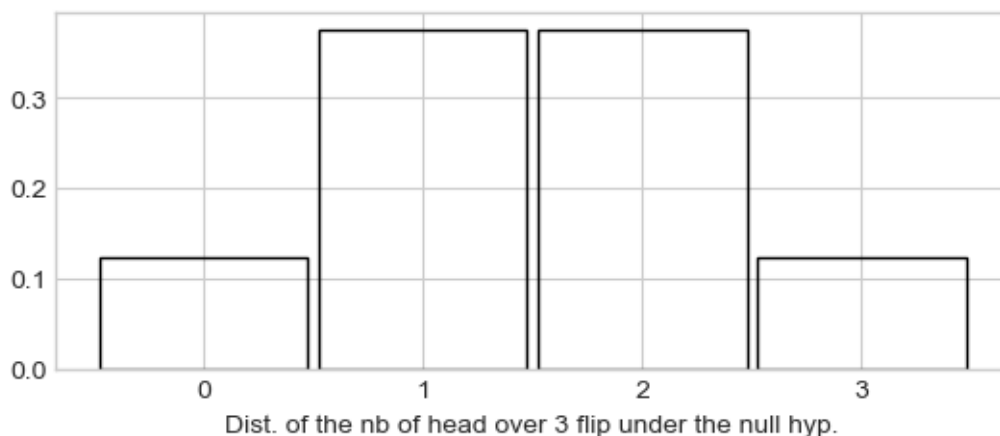
Eight possibles configurations, probabilities of different values for x are:

- $P(x = 0) = 1/8$

- $P(x = 1) = 3/8$
- $P(x = 2) = 3/8$
- $P(x = 3) = 1/8$

Plot of the distribution of the number of x (A vote over 3 voters) under the null hypothesis:

```
plt.bar([0, 1, 2, 3], [1/8, 3/8, 3/8, 1/8], width=.95, fill=False)
_ = plt.xticks([0, 1, 2, 3], [0, 1, 2, 3])
_ = plt.xlabel("Dist. of the nb of head over 3 flip under the null hyp.")
```



Finally, we compute the probability (P-value) to observe a value larger or equal that $x = 2$ (or $P = 2/3$) under the null hypothesis? This probability is the p -value:

$$P(x \geq 2|H_0) = P(x = 2|H_0) + P(x = 3|H_0) = 3/8 + 1/8 = 1/2$$

P-value = 0.5, meaning that there is 50% of chance to get $x = 2$ or larger by chance.

large sample example: 100 voters are questioned about their vote. 60 declared they voted for candidate A and 40 for candidate B. Is this a significant difference?

1. Model the data: Let x the number of vote for A. x follows a binomial distribution. Compute model parameters: $n = 100$, $\hat{p} = 60/100$. Where $\hat{p}$ is the observed proportion of A.

2. Compute a test statistic that measure the deviation of $x = 60$ (vote for A) from the expected value: $n\pi_0 = 50$ under the null hypothesis, i.e., where $\pi_0 = 50\%$. The distribution of the number of vote for A (x) follow the **Binomial distribution** of parameters $N = 100$, $P = 0.5$ approximated with normal distribution when n is large enough.

For large sample, the most usual (and easiest) approximation is through the standard normal distribution, in which a z-test is performed of the test statistic Z , given by:

$$Z = \frac{x - n\pi_0}{\sqrt{np_0(1 - \pi_0)}}$$

one may rearrange and write the z-test above as deviation of $\hat{p}$ from $\pi_0 = 50\%$

$$Z = \frac{\hat{p} - \pi_0}{\sqrt{\pi_0(1 - \pi_0)}} \sqrt{n}$$

Note that the statistic is the product of two parts:

- The **effect size**: $\frac{\hat{p} - \pi_0}{\sqrt{\pi_0(1 - \pi_0)}}$ that measure a standardized deviation.

- The squared root of the sample size $\sqrt{n}$.

Large statistic is obtained with large deviation with large sample size.

5. Inference

Compute the p-value using [Scipy](#) to compute the CDF of the binomial distribution:

```
n, k, pi0 = 100, 60, 0.5
pval_greater = 1 - scipy.stats.binom.cdf(k, n, pi0)
pval_greater = scipy.stats.binom.sf(k, n, pi0)

# Two sided pval = 2 * pval_greater
pval = pval_greater * 2
print("P-value (Two sided): P(X<={:0d} or X>={:0d}|H0)={:.4f}".format(n-k, k,
↪pval))
```

P-value (Two sided): P(X<=40 or X>=60|H0)=0.0352

[Scipy](#) normal distribution

```
n = 100
z = (0.6 - 0.5) / (0.5 * (1 - 0.5)) * np.sqrt(n)
#z = (60 - n * 0.5 + 1/2) / (n * 0.5 * (1 - 0.5)) * np.sqrt(n)

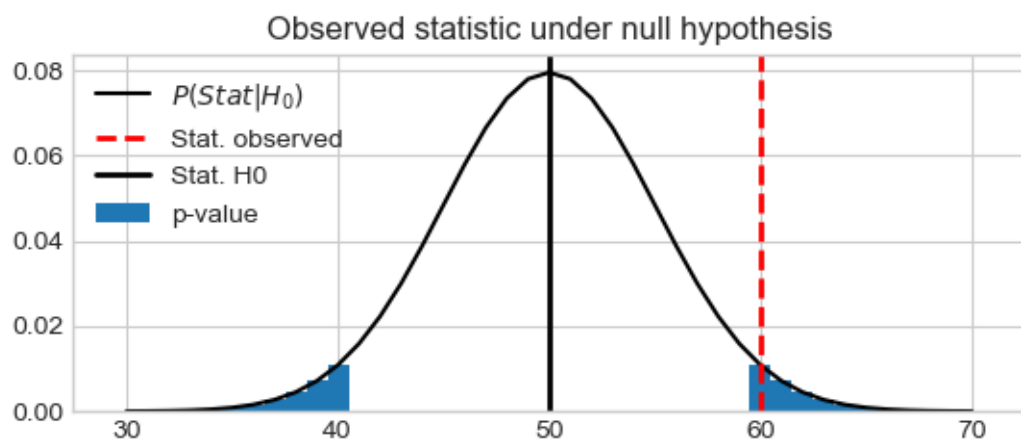
scipy.stats.norm.sf(z, loc=0) * 2
```

np.float64(6.334248366623996e-05)

Plot of the binomial distribution and the probability to observe more than 60 vote for A by chance:

```
stat_vals = np.linspace(30, 70, 41)
stat_probs = scipy.stats.binom.pmf(stat_vals, n, pi0) # H0: 0.5
stat_obs = k

pystatsml.plot_utils.plot_pvalue_under_h0(stat_vals, stat_probs,
                                          stat_obs=60, stat_h0=50,
                                          thresh_low=40, thresh_high=60)
```



Simply use `Scipy binomial test` that the probability of success is p .

```
test = scipy.stats.binomtest(k=k, n=n, p=pi0, alternative='two-sided')
ci_low, ci_high = test.proportion_ci(confidence_level=0.95, method='exact')

print("Estimate: {:.2f}, p-val: {:e}, CI: [{:.5f}, {:.5f}]" .\
      format(test.statistic, test.pvalue, ci_low, ci_high))
```

```
Estimate: 0.60, p-val: 5.688793e-02, CI: [0.49721, 0.69671]
```

Quantitative variable: One Sample T-test

The `one sample t-test` is used to determine whether a sample comes from a population with a specific mean. For example you want to test if the average height of a population is 1.75 *m*.

This test is used when we have two measurements for each individual at two different times or under two conditions: for each individual, we calculate the difference between the two conditions and test the positivity (increase) or negativity (decrease) of the mean of the differences.

Example: is the arterial hypertension of 50 patients measured before and after some medication has been reduced by the treatment?

Example: `Monthly revenue figures of for 100 stores before and after a marketing campaigns`. We compute the difference ($x_i = x_i^{\text{after}} - x_i^{\text{before}}$) for each store i . If the average difference $\bar{x} = 1/n \sum_i x_i$ is significantly positive (resp. negative), then the marketing campaigns will be considered as efficient (resp. detrimental).

```
df = pd.read_csv("../datasets/Monthly Revenue (in thousands).csv")
print(df.head())
df = df.pivot(index='store_id', columns='time', values='revenue')
# Keep only the 30 first samples
df = df[:30]
df.after -= 3 # force to smaller effect size
print(df.head())
x = df.after - df.before
```

	store_id	time	revenue
0	1	before	54.96714
1	2	before	48.61736
2	3	before	56.47689
3	4	before	65.23030
4	5	before	47.65847
	time	after	before
	store_id		
1		49.89029	54.96714
2		48.51413	48.61736
3		56.76331	56.47689
4		63.21891	65.23030
5		48.85204	47.65847

1. Model the data (parametric test)

We model the observation as the sample mean $\bar{x}$ plus some error ε_i , i.e., $x_i = \bar{x} + \varepsilon_i$. The ε_i are

called the **residuals**.

Assumptions:

- The x_i 's are not expected to follow a normal distribution. But the ε_i should be approximately normally distributed.
- The ε_i must be independent and identically distributed *i.i.d.*.

Indeed according to the central limit theorem, if the sampling of the parent population x is independent then the sample mean $\bar{x}$ will be approximately normal.

Fit: estimate the model parameters, the mean $\bar{x} = 1/n \sum_i x_i = 2.26$ (thousands of dollars) and standard deviation s . Warning, when computing the std or the variance, set `ddof=1`. The default value, `ddof=0`, leads to the biased estimator of the variance.

```
xbar = np.mean(x)      # sample mean
mu0 = 0                # mean under H0
s = np.std(x, ddof=1)  # sample standard deviation
n = len(x)             # sample size
df = n - 1
```

2. Compute a test statistic

- Formulate hypothesis:
 - Null hypothesis: $H_0 : \bar{x} = 0$, i.e., the marketing campaign had no effect on sales.
 - Alternative hypothesis: $H_0 : \bar{x} \neq 0$, i.e., the marketing campaign had positive ($\bar{x} > 0$) or negative effect ($\bar{x} < 0$) on sales.

Note that is is a **two-sided test** of effects on both ways.

- Compute a test statistic that measure the deviation of $\bar{x} = 2.26$ from the expected value under the null hypothesis (no effect of the campaign) which is: $\mu_0 = 0$. The test statistic T , is given by:

$$T = \frac{\bar{x} - \mu_0}{s} \sqrt{n}$$

Note that the statistic is the product of two parts:

- The **effect size**: $\frac{\bar{x} - \mu_0}{s}$ that measure a standardized deviation. It a “signal to noise ratio” of what is explained by the model divided by the error.
- The squared root of the sample size.

Under the null hypothesis the distribution of T follow the [Student t-distribution](#) of parameters `df=n-1`. Note that according the central limit theorem, if the observations are independent, then T will be approximately normal $\mathcal{N}(0, 1)$.

```
tval = (xbar - mu0) / s * np.sqrt(n)
```

3. Inference

P-value (null hypothesis) is computed using [Scipy to compute the CDF](#) of the student distribution.

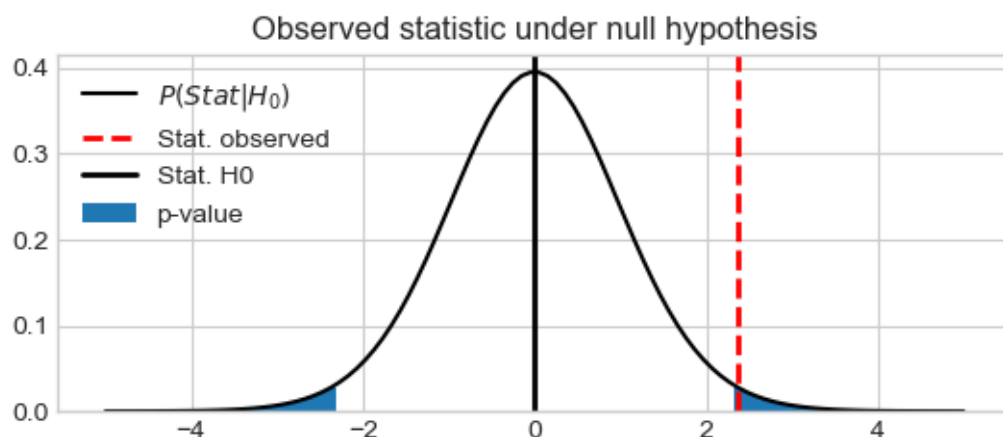
```
pval_greater = 1 - scipy.stats.t.cdf(tval, df)
pval_greater = scipy.stats.t.sf(tval, df)
# T distribution is symmetric
# => pval_lower = pval_greater
# => two-sided = pval_greater * 2
pval = 2 * pval_greater
```

Plot observed T value under null hypothesis

- the distribution of t-statistic under the null hypothesis $P_T(X|H_0)$,
- the two sided p-value, CDF of $P_T(X \leq -T) + P_T(X \geq T|H_0)$, and
- the observed T value

```
stat_vals = np.linspace(-5, 5, 100)
stat_probs = scipy.stats.t.pdf(x=stat_vals, df=df, loc=0) # H0 => loc=0

pystatsml.plot_utils.plot_pvalue_under_h0(stat_vals, stat_probs,
                                           stat_obs=tval, stat_h0=0,
                                           thresh_low=-tval, thresh_high=tval)
```



Confidence interval (alternative hypothesis) of the observed estimate $\bar{x}$ is given by:

$$\bar{x} \pm t_{\alpha/2} \frac{s}{\sqrt{n}}$$

Where $t_{\alpha/2}$ is the statistic critical value, obtained by the the CMF of the t-distribution with $n - 1$ degrees of freedom.

Use the Percent Point Function [PPF](#) or quantile function of the to compute the critical value .

See [Confidence Interval for a population mean, t distribution](#).

```
# Critical value for t at alpha / 2:
t_alpha2 = -scipy.stats.t.ppf(q=0.05/2, df=df, loc=0)

ci_low = xbar - t_alpha2 * s / np.sqrt(n)
ci_high = xbar + t_alpha2 * s / np.sqrt(n)

print("Estimate: {:.2f}, t-val: {:.2f}, p-val: {:e}, df: {}, CI: [{:.5f}, {:.5f}]")
```

(continues on next page)

(continued from previous page)

```
↪ ".
    format(xbar, tval, pval, df, ci_low, ci_high))
```

```
Estimate: 2.26, t-val: 2.36, p-val: 2.500022e-02, df: 29, CI: [0.30515, 4.22272]
```

Simply use `Scipy` one sample t-test

```
ttest = scipy.stats.ttest_1samp(x, 0, alternative='two-sided')
ci_low, ci_high = ttest.confidence_interval()

print("Estimate: {:.2f}, t-val: {:.2f}, p-val: {:e}, df: {}, CI: [{:.5f}, {:.5f}]
↪ ".
    format(ttest._estimate, ttest.statistic, ttest.pvalue, ttest.df, ci_low, ci_
↪ high))
```

```
Estimate: 2.26, t-val: 2.36, p-val: 2.500022e-02, df: 29, CI: [0.30515, 4.22272]
```

Bootstrapping for Confidence Intervals

5.1.6 Statistical Tests of Pairwise Associations

Univariate statistical analysis: explore association between pairs of variables.

- In statistics, a **categorical variable** or **factor** is a variable that can take on one of a limited, and usually fixed, number of possible values, thus assigning each individual to a particular group or “category”. The levels are the possible values of the variable. Number of levels = 2: binomial; Number of levels > 2: multinomial. There is no intrinsic ordering to the categories. For example, gender is a categorical variable having two categories (male and female) and there is no intrinsic ordering to the categories. For example, Sex (Female, Male), Hair color (blonde, brown, etc.).
- An **ordinal variable** is a categorical variable with a clear ordering of the levels. For example: drinks per day (none, small, medium and high).
- A **continuous** or **quantitative variable** $x \in \mathbb{R}$ is one that can take any value in a range of possible values, possibly infinite. E.g.: salary, experience in years, weight.

What statistical test should I use?

Pearson Correlation: Test Association Between Two Quantitative Variables

Test the correlation coefficient of two quantitative variables. The test calculates a Pearson correlation coefficient and the p -value for testing non-correlation.

Let x and y two quantitative variables, where n samples were observed. The linear correlation coefficient is defined as :

$$r = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^n (x_i - \bar{x})^2} \sqrt{\sum_{i=1}^n (y_i - \bar{y})^2}}.$$

Under H_0 , the test statistic $t = \sqrt{n-2} \frac{r}{\sqrt{1-r^2}}$ follow Student distribution with $n-2$ degrees of freedom.

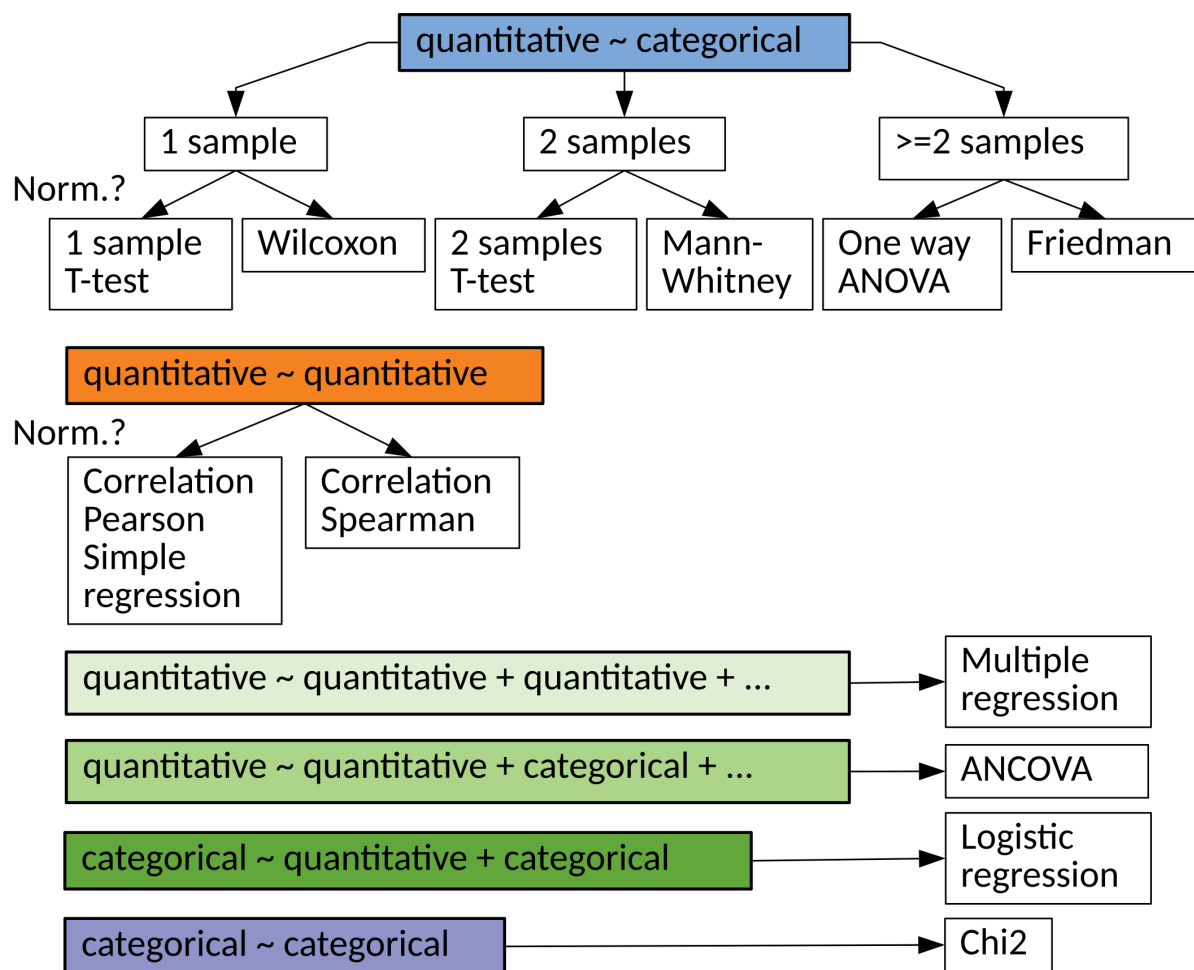


Fig. 1: Statistical tests

```

n = 50
x = np.random.normal(size=n)
y = 2 * x + np.random.normal(size=n)

# Compute with scipy
cor, pval = scipy.stats.pearsonr(x, y)
print(cor, pval)

```

```

0.8838265556020785 1.8786054559764614e-17

```

Two sample (Student) T-test: Compare Two Means

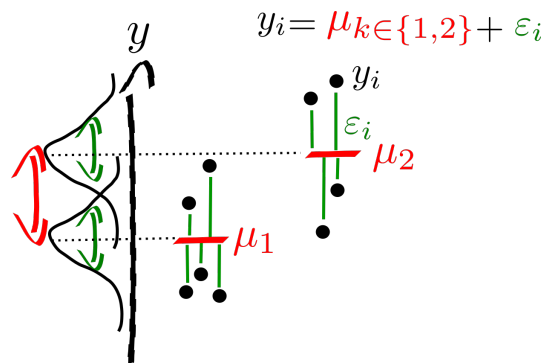


Fig. 2: Two-sample model

The two-sample t -test (Snedecor and Cochran, 1989) is used to determine if two population means are equal. There are several variations on this test. If data are paired (e.g. 2 measures, before and after treatment for each individual) use the one-sample t -test of the difference. The variances of the two samples may be assumed to be equal (a.k.a. homoscedasticity) or unequal (a.k.a. heteroscedasticity).

1. Model the data

Assumptions:

- Independence of **residuals** (ε_i). This assumptions **must** be satisfied.
- Normality of residuals. Approximately normally distributed can be accepted.
- Homoscedasticity use T-test, Heteroscedasticity use Welch t-test.

Assume that the two random variables are normally distributed: $y_1 \sim \mathcal{N}(\mu_1, \sigma_1), y_2 \sim \mathcal{N}(\mu_2, \sigma_2)$.

Fit: estimate the model parameters, means and variances: $\bar{y}_1, s_{y_1}^2, \bar{y}_2, s_{y_2}^2$.

2. t-test

The general principle is

$$\begin{aligned}
 t &= \frac{\text{difference of means}}{\text{standard dev of error}} = \frac{\bar{y}_1 - \bar{y}_2}{s_{\bar{y}_1 - \bar{y}_2}} \\
 &= \frac{\bar{y}_1 - \bar{y}_2}{\sqrt{\sum \varepsilon^2}} \sqrt{n-2}
 \end{aligned}$$

Since y_1 and y_2 are independent:

$$s_{\bar{y}_1 - \bar{y}_2}^2 = s_{y_1}^2 + s_{y_2}^2 = \frac{s_{y_1}^2}{n_1} + \frac{s_{y_2}^2}{n_2}, \text{ thus}$$

$$s_{\bar{y}_1 - \bar{y}_2} = \sqrt{\frac{s_{y_1}^2}{n_1} + \frac{s_{y_2}^2}{n_2}}$$

Equal or unequal sample sizes, unequal variances (Welch's t -test)

Welch's t -test defines the t statistic as

$$t = \frac{\bar{y}_1 - \bar{y}_2}{\sqrt{\frac{s_{y_1}^2}{n_1} + \frac{s_{y_2}^2}{n_2}}}.$$

To compute the p -value one needs the degrees of freedom associated with this variance estimate. It is approximated using the Welch–Satterthwaite equation:

$$\nu \approx \frac{\left(\frac{s_{y_1}^2}{n_1} + \frac{s_{y_2}^2}{n_2}\right)^2}{\frac{s_{y_1}^4}{n_1^2(n_1-1)} + \frac{s_{y_2}^4}{n_2^2(n_2-1)}}.$$

Equal or unequal sample sizes, equal variances

If we assume equal variance (ie, $s_{y_1}^2 = s_{y_2}^2 = s^2$), where s^2 is an estimator of the common variance of the two samples:

$$s^2 = \frac{s_{y_1}^2(n_1 - 1) + s_{y_2}^2(n_2 - 1)}{n_1 + n_2 - 2}$$

$$= \frac{\sum_i^{n_1} (y_{1i} - \bar{y}_1)^2 + \sum_j^{n_2} (y_{2j} - \bar{y}_2)^2}{(n_1 - 1) + (n_2 - 1)}$$

then

$$s_{\bar{y}_1 - \bar{y}_2} = \sqrt{\frac{s^2}{n_1} + \frac{s^2}{n_2}} = s \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}$$

Therefore, the t statistic, that is used to test whether the means are different is:

$$t = \frac{\bar{y}_1 - \bar{y}_2}{s \cdot \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}},$$

Equal sample sizes, equal variances

If we simplify the problem assuming equal samples of size $n_1 = n_2 = n/2$ we get

$$t = \frac{\bar{y}_1 - \bar{y}_2}{s} \cdot \sqrt{n}$$

$$\approx \frac{\text{difference of means}}{\text{standard deviation of the noise}} \cdot \sqrt{n} \approx \text{effect size} \cdot \sqrt{n}$$

Example

Given the following two samples, test whether their means are equal using the **standard t-test, assuming equal variance**.

```

height = np.array([1.83, 1.83, 1.73, 1.82, 1.83, 1.73, 1.99, 1.85, 1.68, 1.87,
                  1.66, 1.71, 1.73, 1.64, 1.70, 1.60, 1.79, 1.73, 1.62, 1.77])

grp = np.array(["M"] * 10 + ["F"] * 10)

# Compute with scipy
ttest = scipy.stats.ttest_ind(height[grp == "M"], height[grp == "F"], equal_
    ↪var=True)

print("Estimate: {:.2f}, t-val: {:.2f}, p-val: {:e}, df: {}".\
      format(ttest._estimate, ttest.statistic, ttest.pvalue, ttest.df))

```

```
Estimate: 0.12, t-val: 3.55, p-val: 2.282089e-03, df: 18.0
```

ANOVA F-test: Quantitative as a function of Categorical Factor with Three Levels or More

Analysis of variance (ANOVA) provides a statistical test of whether or not the means of several (k) groups are equal, and therefore generalizes the t -test to more than two groups. ANOVAs are useful for comparing (testing) three or more means (groups or variables) for statistical significance. It is conceptually similar to multiple two-sample t -tests, but is less conservative.

Here we will consider one-way ANOVA with one independent variable, ie one-way ANOVA, [see](#):

- Test if any group is on average superior, or inferior, to the others versus the null hypothesis that all four strategies yield the same mean response
- Detect any of several possible differences.
- The advantage of the ANOVA F -test is that we do not need to pre-specify which strategies are to be compared, and we do not need to adjust for making multiple comparisons.
- The disadvantage of the ANOVA F -test is that if we reject the null hypothesis, we do not know which strategies can be said to be significantly different from the others.

1. Model the data

Assumptions

- The samples are randomly selected in an independent manner from the k populations.
- All k populations have distributions that are approximately normal. Check by plotting groups distribution.
- The k population variances are equal. Check by plotting groups distribution.

The question is: Is there a difference in Petal Width in species from iris dataset? Let y_1, y_2 and y_3 be Petal Width in three species.

Here we assume (see assumptions) that the three populations were sampled from three random variables that are normally distributed. I.e., $Y_1 \sim N(\mu_1, \sigma_1)$, $Y_2 \sim N(\mu_2, \sigma_2)$ and $Y_3 \sim N(\mu_3, \sigma_3)$.

2. Fit: estimate the model parameters

Estimate means and variances: $\bar{y}_i, \sigma_i, \forall i \in \{1, 2, 3\}$.

3. :math:`F`-test

The formula for the one-way ANOVA F-test statistic is

$$F = \frac{\text{Explained variance}}{\text{Unexplained variance}} = \frac{\text{Between-group variability}}{\text{Within-group variability}} = \frac{s_B^2}{s_W^2}.$$

The “explained variance”, or “between-group variability” is

$$s_B^2 = \sum_i n_i (\bar{y}_i - \bar{y})^2 / (K - 1),$$

where $\bar{y}_i$ denotes the sample mean in the i th group, n_i is the number of observations in the i th group, $\bar{y}$ denotes the overall mean of the data, and K denotes the number of groups.

The “unexplained variance”, or “within-group variability” is

$$s_W^2 = \sum_{ij} (y_{ij} - \bar{y}_i)^2 / (N - K),$$

where y_{ij} is the j th observation in the i th out of K groups and N is the overall sample size. This F -statistic follows the F -distribution with $K - 1$ and $N - K$ degrees of freedom under the null hypothesis. The statistic will be large if the between-group variability is large relative to the within-group variability, which is unlikely to happen if the population means of the groups all have the same value.

Note that when there are only two groups for the one-way ANOVA F-test, $F = t^2$ where t is the Student’s t statistic.

Example with the Iris Dataset:

```
# Group means
means = iris.groupby("Species").mean().reset_index()
print(means)

# Group Stds (equal variances ?)
stds = iris.groupby("Species").std(ddof=1).reset_index()
print(stds)

# Plot groups
ax = sns.violinplot(x="Species", y="SepalLength", data=iris)
ax = sns.swarmplot(x="Species", y="SepalLength", data=iris,
                  color="white")
ax = sns.swarmplot(x="Species", y="SepalLength", color="black",
                  data=means, size=10)

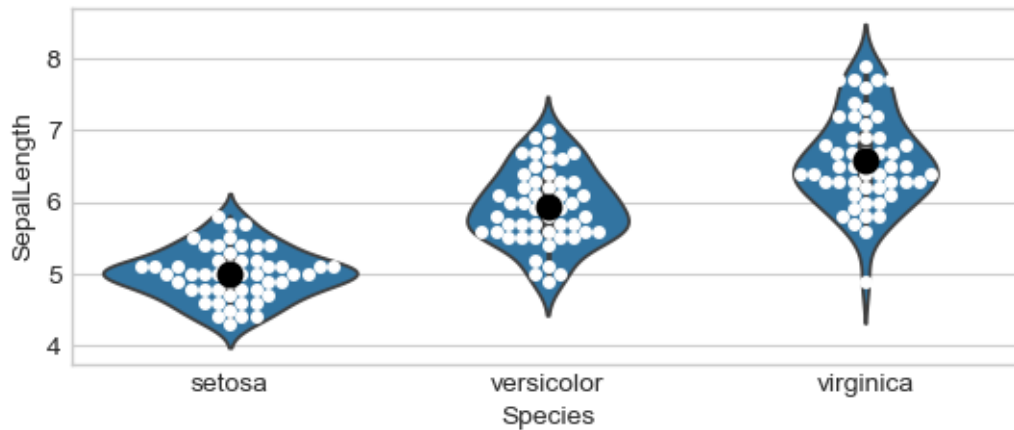
# ANOVA
lm = smf.ols('SepalLength ~ Species', data=iris).fit()
sm.stats.anova_lm(lm, typ=2) # Type 2 ANOVA DataFrame
```

	Species	SepalLength	SepalWidth	PetalLength	PetalWidth
0	setosa	5.006	3.428	1.462	0.246
1	versicolor	5.936	2.770	4.260	1.326

(continues on next page)

(continued from previous page)

2	virginica	6.588	2.974	5.552	2.026
	Species	SepalLength	SepalWidth	PetalLength	PetalWidth
0	setosa	0.352490	0.379064	0.173664	0.105386
1	versicolor	0.516171	0.313798	0.469911	0.197753
2	virginica	0.635880	0.322497	0.551895	0.274650



Chi-square, χ^2 : Categorical v.s. Categorical Factors

Computes the chi-square, χ^2 , statistic and p -value for the hypothesis test of independence of frequencies in the observed contingency table (cross-table). The observed frequencies are tested against an expected contingency table obtained by computing expected frequencies based on the marginal sums under the assumption of independence.

Example

20 participants: 10 exposed to some chemical product and 10 non exposed (exposed = 1 or 0). Among the 20 participants 10 had cancer 10 not (cancer = 1 or 0). χ^2 tests the association between those two variables.

```
# Dataset:
# 15 samples:
# 10 first exposed
exposed = np.array([1] * 10 + [0] * 10)
# 8 first with cancer, 10 without, the last two with.
cancer = np.array([1] * 8 + [0] * 10 + [1] * 2)

crosstab = pd.crosstab(exposed, cancer, rownames=['exposed'],
                       colnames=['cancer'])
print("Observed table:")
print("-----")
print(crosstab)

chi2, pval, dof, expected = scipy.stats.chi2_contingency(crosstab)
print("Statistics:")
print("-----")
print("Chi2 = %f, pval = %f" % (chi2, pval))
```

(continues on next page)

(continued from previous page)

```
print("Expected table:")
print("-----")
print(expected)
```

Observed table:

```
-----
cancer  0  1
exposed
0        8  2
1        2  8
```

Statistics:

```
-----
Chi2 = 5.000000, pval = 0.025347
```

Expected table:

```
-----
[[5. 5.]
 [5. 5.]]
```

Computing expected cross-table

```
# Compute expected cross-table based on proportion
exposed_marg = crosstab.sum(axis=0)
exposed_freq = exposed_marg / exposed_marg.sum()

cancer_marg = crosstab.sum(axis=1)
cancer_freq = cancer_marg / cancer_marg.sum()

print('Exposed frequency? Yes: %.2f' % exposed_freq[0],
      'No: %.2f' % exposed_freq[1])
print('Cancer frequency? Yes: %.2f' % cancer_freq[0],
      'No: %.2f' % cancer_freq[1])

print('Expected frequencies:')
print(np.outer(exposed_freq, cancer_freq))

print('Expected cross-table (frequencies * N): ')
print(np.outer(exposed_freq, cancer_freq) * len(exposed))
```

```
Exposed frequency? Yes: 0.50 No: 0.50
Cancer frequency? Yes: 0.50 No: 0.50
Expected frequencies:
[[0.25 0.25]
 [0.25 0.25]]
Expected cross-table (frequencies * N):
[[5. 5.]
 [5. 5.]]
```


Non-parametric Tests of Pairwise Associations

Spearman Rank-Order Correlation (Quantitative vs Quantitative)

The Spearman correlation is a non-parametric measure of the monotonicity of the relationship between two datasets.

When to use it? Observe the data distribution: - presence of **outliers** - the distribution of the residuals is not Gaussian.

Like other correlation coefficients, this one varies between -1 and +1 with 0 implying no correlation. Correlations of -1 or +1 imply an exact monotonic relationship. Positive correlations imply that as x increases, so does y . Negative correlations imply that as x increases, y decreases.

```
np.random.seed(3)

# Age uniform distribution between 20 and 40
age = np.random.uniform(20, 60, 40)

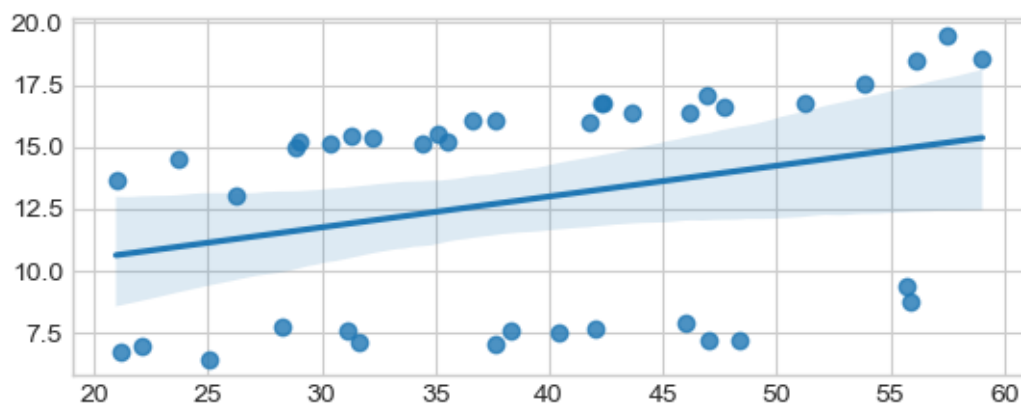
# Systolic blood pressure, 2 groups:
# - 15 subjects at 0.05 * age + 6
# - 25 subjects at 0.15 * age + 10
sbp = np.concatenate((0.05 * age[:15] + 6, 0.15 * age[15:] + 10)) + \
    .5 * np.random.normal(size=40)

sns.regplot(x=age, y=sbp)

# Non-Parametric Spearman
cor, pval = scipy.stats.spearmanr(age, sbp)
print("Non-Parametric Spearman cor test, cor: %.4f, pval: %.4f" % (cor, pval))

# "Parametric Pearson cor test
cor, pval = scipy.stats.pearsonr(age, sbp)
print("Parametric Pearson cor test: cor: %.4f, pval: %.4f" % (cor, pval))
```

```
Non-Parametric Spearman cor test, cor: 0.5122, pval: 0.0007
Parametric Pearson cor test: cor: 0.3085, pval: 0.0528
```



Wilcoxon Signed-Rank Test (Quantitative vs Cte)

Wikipedia: The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test used when comparing two related samples, matched samples, or repeated measurements on a single sample to assess whether their population mean ranks differ (i.e. it is a paired difference test). It is equivalent to one-sample test of the difference of paired samples.

It can be used as an alternative to the paired Student's t -test, t -test for matched pairs, or the t -test for dependent samples when the population cannot be assumed to be normally distributed.

When to use it? Observe the data distribution: - presence of outliers - the distribution of the residuals is not Gaussian

It has a lower sensitivity compared to t -test. May be problematic to use when the sample size is small.

Null hypothesis H_0 : difference between the pairs follows a symmetric distribution around zero.

```
n = 20
# Buisness Volume time 0
bv0 = np.random.normal(loc=3, scale=.1, size=n)
# Buisness Volume time 1
bv1 = bv0 + 0.1 + np.random.normal(loc=0, scale=.1, size=n)

# create an outlier
bv1[0] -= 10

# Paired t-test
print(scipy.stats.ttest_rel(bv0, bv1))

# Wilcoxon
print(scipy.stats.wilcoxon(bv0, bv1))
```

```
TtestResult(statistic=np.float64(0.7766377807752968), pvalue=np.float64(0.
↪44693401731548044), df=np.int64(19))
WilcoxonResult(statistic=np.float64(23.0), pvalue=np.float64(0.
↪001209259033203125))
```

Mann-Whitney U test (Quantitative vs Categorical Factor with Two Levels)

In statistics, the Mann-Whitney U test (also called the Mann-Whitney-Wilcoxon, Wilcoxon rank-sum test or Wilcoxon-Mann-Whitney test) is a nonparametric test of the null hypothesis that two samples come from the same population against an alternative hypothesis, especially that a particular population tends to have larger values than the other.

It can be applied on unknown distributions contrary to e.g. a t -test that has to be applied only on normal distributions, and it is nearly as efficient as the t -test on normal distributions.

```
n = 20
# Buisness Volume group 0
bv0 = np.random.normal(loc=1, scale=.1, size=n)

# Buisness Volume group 1
```

(continues on next page)

(continued from previous page)

```

bv1 = np.random.normal(loc=1.2, scale=.1, size=n)

# create an outlier
bv1[0] -= 10

# Two-samples t-test
print(scipy.stats.ttest_ind(bv0, bv1))

# Wilcoxon
print(scipy.stats.mannwhitneyu(bv0, bv1))

```

```

TtestResult(statistic=np.float64(0.6104564820307219), pvalue=np.float64(0.
↪5451934484051324), df=np.float64(38.0))
MannwhitneyuResult(statistic=np.float64(41.0), pvalue=np.float64(1.
↪8074477738835562e-05))

```

5.1.7 Linear Model

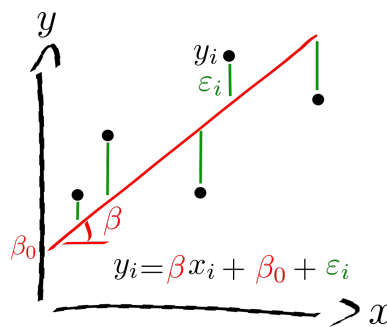


Fig. 3: Linear model

Given n random samples $(y_i, x_{1i}, \dots, x_{pi})$, $i = 1, \dots, n$, the linear regression models the relation between the observations y_i and the independent variables x_i^p is formulated as

$$y_i = \beta_0 + \beta_1 x_{1i} + \dots + \beta_p x_{pi} + \varepsilon_i \quad i = 1, \dots, n$$

- The β 's are the model parameters, ie, the regression coefficients.
- β_0 is the intercept or the bias.
- ε_i are the **residuals**.
- **An independent variable (IV)**. It is a variable that stands alone and isn't changed by the other variables you are trying to measure. For example, someone's age might be an independent variable. Other factors (such as what they eat, how much they go to school, how much television they watch) aren't going to change a person's age. In fact, when you are looking for some kind of relationship between variables you are trying to see if the independent variable causes some kind of change in the other variables, or dependent variables. In Machine Learning, these variables are also called the **predictors**.
- A **dependent variable**. It is something that depends on other factors. For example, a test score could be a dependent variable because it could change depending on several factors such as how much you studied, how much sleep you got the night before you took the

test, or even how hungry you were when you took it. Usually when you are looking for a relationship between two things you are trying to find out what makes the dependent variable change the way it does. In Machine Learning this variable is called a **target variable**.

Assumptions

1. Independence of residuals (ϵ_i). This assumptions **must** be satisfied
2. Normality of residuals (ϵ_i). Approximately normally distributed can be accepted.

Regression diagnostics: testing the assumptions of linear regression

Simple Regression: Test Association Between Two Quantitative Variables

Using the dataset “salary”, explore the association between the dependant variable (e.g. Salary) and the independent variable (e.g.: Experience is quantitative), considering only non-managers.

```
df = salary[salary.management == 'N']
```

1. Model the data

Model the data on some **hypothesis** e.g.: salary is a linear function of the experience.

$$\text{salary}_i = \beta_0 + \beta \text{ experience}_i + \epsilon_i,$$

more generally

$$y_i = \beta_0 + \beta x_i + \epsilon_i$$

This can be rewritten in the matrix form using the design matrix made of values of independent variable and the intercept:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 1 & x_1 \\ 1 & x_2 \\ 1 & x_3 \\ 1 & x_4 \\ 1 & x_5 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \end{bmatrix}$$

- β : the slope or coefficient or parameter of the model,
- β_0 : the **intercept** or **bias** is the second parameter of the model,
- ϵ_i : is the i th error, or residual with $\epsilon \sim \mathcal{N}(0, \sigma^2)$.

The simple regression is equivalent to the Pearson correlation.

2. Fit: estimate the model parameters

The goal is to estimate β , β_0 and σ^2 .

Minimizes the **mean squared error (MSE)** or the **Sum squared error (SSE)**. The so-called **Ordinary Least Squares (OLS)** finds β, β_0 that minimizes the $SSE = \sum_i \epsilon_i^2$

$$SSE = \sum_i (y_i - \beta x_i - \beta_0)^2$$

Recall from calculus that an extreme point can be found by computing where the derivative is zero, i.e. to find the intercept, we perform the steps:

$$\begin{aligned}\frac{\partial SSE}{\partial \beta_0} &= \sum_i (y_i - \beta x_i - \beta_0) = 0 \\ \sum_i y_i &= \beta \sum_i x_i + n \beta_0 \\ n \bar{y} &= n \beta \bar{x} + n \beta_0 \\ \beta_0 &= \bar{y} - \beta \bar{x}\end{aligned}$$

To find the regression coefficient, we perform the steps:

$$\frac{\partial SSE}{\partial \beta} = \sum_i x_i (y_i - \beta x_i - \beta_0) = 0$$

Plug in β_0 :

$$\begin{aligned}\sum_i x_i (y_i - \beta x_i - \bar{y} + \beta \bar{x}) &= 0 \\ \sum_i x_i y_i - \bar{y} \sum_i x_i &= \beta \sum_i (x_i - \bar{x})\end{aligned}$$

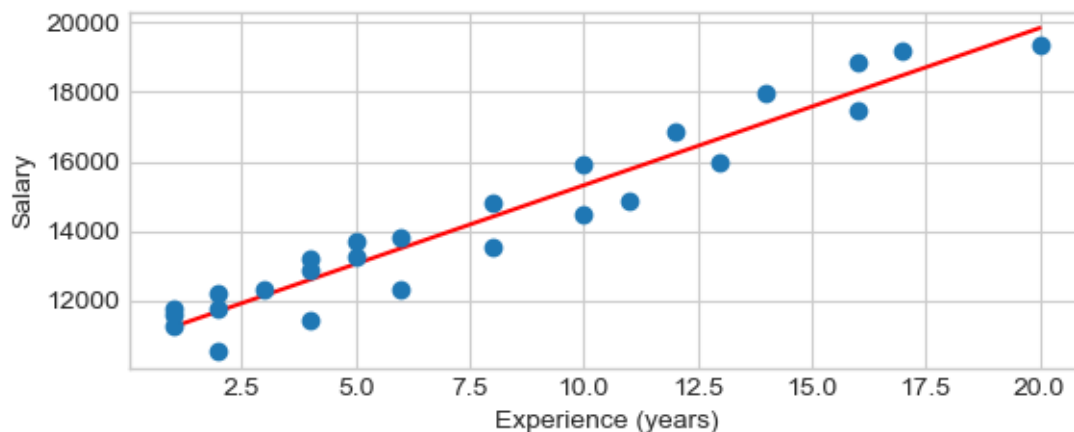
Divide both sides by n :

$$\begin{aligned}\frac{1}{n} \sum_i x_i y_i - \bar{y} \bar{x} &= \frac{1}{n} \beta \sum_i (x_i - \bar{x}) \\ \beta &= \frac{\frac{1}{n} \sum_i x_i y_i - \bar{y} \bar{x}}{\frac{1}{n} \sum_i (x_i - \bar{x})} = \frac{Cov(x, y)}{Var(x)}.\end{aligned}$$

```
y, x = df.salary, df.experience
beta, beta0, r_value, p_value, std_err = scipy.stats.linregress(x,y)
print("y = %f x + %f, r: %f, r-squared: %f,\nnp-value: %f, std_err: %f"
      % (beta, beta0, r_value, r_value**2, p_value, std_err))

print("Regression line with the scatterplot")
yhat = beta * x + beta0 # regression line
plt.plot(x, yhat, 'r-', x, y, 'o')
plt.xlabel('Experience (years)')
plt.ylabel('Salary')
plt.show()
```

```
y = 452.658228 x + 10785.911392, r: 0.965370, r-squared: 0.931939,
p-value: 0.000000, std_err: 24.970021
Regression line with the scatterplot
```



Multiple Regression

Theory

Multiple Linear Regression is the most basic supervised learning algorithm. Given: a set of training data $\{x_1, \dots, x_N\}$ with corresponding targets $\{y_1, \dots, y_N\}$.

In linear regression, we assume that the model that generates the data involves only a linear combination of the input variables, i.e.

$$y_i = \beta_0 + \beta_1 x_{i1} + \dots + \beta_P x_{iP} + \varepsilon_i,$$

or, simplified

$$y_i = \beta_0 + \sum_{j=1}^{P-1} \beta_j x_i^j + \varepsilon_i.$$

Extending each sample with an intercept, $x_i := [1, x_i] \in R^{P+1}$ allows us to use a more general notation based on linear algebra and write it as a simple dot product:

$$y_i = \mathbf{x}_i^T \boldsymbol{\beta} + \varepsilon_i,$$

where $\boldsymbol{\beta} \in R^{P+1}$ is a vector of weights that define the $P + 1$ parameters of the model. From now we have P regressors + the intercept.

This can be re-written using the matrix notation:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & \dots & x_{1P} \\ 1 & x_{21} & \dots & x_{2P} \\ 1 & x_{31} & \dots & x_{3P} \\ 1 & x_{41} & \dots & x_{4P} \\ 1 & x_{51} & \dots & x_{5P} \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_P \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \end{bmatrix}$$

Or more generally,

$$\mathbf{y} = \mathbf{X}\boldsymbol{\beta} + \boldsymbol{\varepsilon},$$

where $\mathbf{X} = [\mathbf{x}_0^\top, \dots, \mathbf{x}_N^\top]$ be the $(N \times P + 1)$ **design matrix** of N samples of P input features with one column of one and let be $\mathbf{y} = [y_1, \dots, y_N]$ be a vector of the N targets.

Minimize the Mean Squared Error MSE loss:

$$MSE(\beta) = \frac{1}{N} \sum_{i=1}^N (y_i - \mathbf{x}_i^\top \beta)^2$$

Or equivalently using the matrix notation:

$$MSE(\beta) = \frac{1}{N} \|y - \mathbf{X}\beta\|_2^2.$$

The β that minimizes the MSE can be found by:

$$\begin{aligned} \nabla_{\beta} \left(\frac{1}{N} \|y - X\beta\|_2^2 \right) &= 0 \\ \frac{1}{N} \nabla_{\beta} (y - X\beta)^T (y - X\beta) &= 0 \\ \frac{1}{N} \nabla_{\beta} (y^T y - 2\beta^T X^T y + \beta^T X^T X \beta) &= 0 \\ -2X^T y + 2X^T X \beta &= 0 \\ X^T X \beta &= X^T y \\ \beta &= (X^T X)^{-1} X^T y, \end{aligned}$$

where $(X^T X)^{-1} X^T$ is a pseudo inverse of X .

Simulated dataset where:

$$\begin{bmatrix} y_1 \\ \vdots \\ y_{50} \end{bmatrix} = \begin{bmatrix} 1 & x_{1,1} & x_{1,2} & x_{1,3} \\ \vdots & \vdots & \vdots & \vdots \\ 1 & x_{50,1} & x_{50,2} & x_{50,3} \end{bmatrix} \begin{bmatrix} 10 \\ 1 \\ 0.5 \\ 0.1 \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \vdots \\ \epsilon_{50} \end{bmatrix}$$

```
from scipy import linalg
np.random.seed(seed=42) # make the example reproducible

# Dataset
N, P = 50, 4
X = np.random.normal(size= N * P).reshape((N, P))
## Our model needs an intercept so we add a column of 1s:
X[:, 0] = 1
print(X[:5, :])

betastar = np.array([10, 1., .5, 0.1])
e = np.random.normal(size=N)
y = np.dot(X, betastar) + e
```

```
[[ 1.          -0.1382643  0.64768854  1.52302986]
 [ 1.          -0.23413696  1.57921282  0.76743473]
 [ 1.           0.54256004 -0.46341769 -0.46572975]
 [ 1.         -1.91328024 -1.72491783 -0.56228753]
 [ 1.           0.31424733 -0.90802408 -1.4123037 ]]
```

Fit with ``numpy``

Estimate the parameters

```
Xpinv = linalg.pinv(X)
betahat = np.dot(Xpinv, y)
print("Estimated beta:\n", betahat)
```

```
Estimated beta:
[10.14742501  0.57938106  0.51654653  0.17862194]
```

Linear Model with Statsmodels

Statmodels examples

Multiple Regression Using Numpy Array

Interface with statsmodels without formulae (sm)

```
## Fit and summary:
model = sm.OLS(y, X).fit()
print(model.summary())

# prediction of new values
ypred = model.predict(X)

# residuals + prediction == true values
assert np.all(ypred + model.resid == y)
```

OLS Regression Results						
=====						
Dep. Variable:	y	R-squared:	0.363			
Model:	OLS	Adj. R-squared:	0.322			
Method:	Least Squares	F-statistic:	8.748			
Date:	Wed, 14 May 2025	Prob (F-statistic):	0.000106			
Time:	23:26:21	Log-Likelihood:	-71.271			
No. Observations:	50	AIC:	150.5			
Df Residuals:	46	BIC:	158.2			
Df Model:	3					
Covariance Type:	nonrobust					
=====						
	coef	std err	t	P> t	[0.025	0.975]

const	10.1474	0.150	67.520	0.000	9.845	10.450
x1	0.5794	0.160	3.623	0.001	0.258	0.901
x2	0.5165	0.151	3.425	0.001	0.213	0.820
x3	0.1786	0.144	1.240	0.221	-0.111	0.469
=====						
Omnibus:	2.493	Durbin-Watson:	2.369			
Prob(Omnibus):	0.288	Jarque-Bera (JB):	1.544			
Skew:	0.330	Prob(JB):	0.462			
Kurtosis:	3.554	Cond. No.	1.27			
=====						

(continues on next page)

(continued from previous page)

Notes:

[1] Standard Errors assume that the covariance matrix of the errors **is** correctly_↪specified.

Multiple Regression Pandas using formulae (smf)

Use R language syntax for data.frame. For an additive model:

$$y_i = \beta^0 + x_i^1\beta^1 + x_i^2\beta^2 + \epsilon_i \equiv y \sim x1 + x2.$$

```
df = pd.DataFrame(np.column_stack([X, y]),
                  columns=['inter', 'x1', 'x2', 'x3', 'y'])
print(df.columns, df.shape)
# Build a model excluding the intercept, it is implicit
model = smf.ols("y~x1 + x2 + x3", df).fit()
print(model.summary())
```

Index(['inter', 'x1', 'x2', 'x3', 'y'], dtype='object') (50, 5)

OLS Regression Results

Dep. Variable:	y	R-squared:	0.363
Model:	OLS	Adj. R-squared:	0.322
Method:	Least Squares	F-statistic:	8.748
Date:	Wed, 14 May 2025	Prob (F-statistic):	0.000106
Time:	23:26:21	Log-Likelihood:	-71.271
No. Observations:	50	AIC:	150.5
Df Residuals:	46	BIC:	158.2
Df Model:	3		
Covariance Type:	nonrobust		
=====			
	coef	std err	t
			P> t
			[0.025
			0.975]
Intercept	10.1474	0.150	67.520
x1	0.5794	0.160	3.623
x2	0.5165	0.151	3.425
x3	0.1786	0.144	1.240
=====			
Omnibus:	2.493	Durbin-Watson:	2.369
Prob(Omnibus):	0.288	Jarque-Bera (JB):	1.544
Skew:	0.330	Prob(JB):	0.462
Kurtosis:	3.554	Cond. No.	1.27
=====			

Notes:

[1] Standard Errors assume that the covariance matrix of the errors **is** correctly_↪specified.

Multiple Regression Mixing Covariates and Factors (ANCOVA)

Analysis of covariance (ANCOVA) is a linear model that blends ANOVA and linear regression. ANCOVA evaluates whether population means of a dependent variable (DV) are equal across levels of a categorical independent variable (IV) often called a treatment, while statistically controlling for the effects of other quantitative or continuous variables that are not of primary interest, known as covariates (CV).

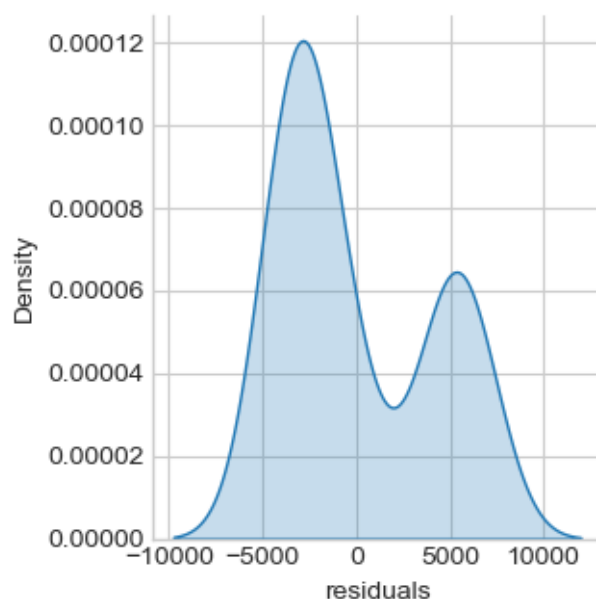
```
df = salary.copy()

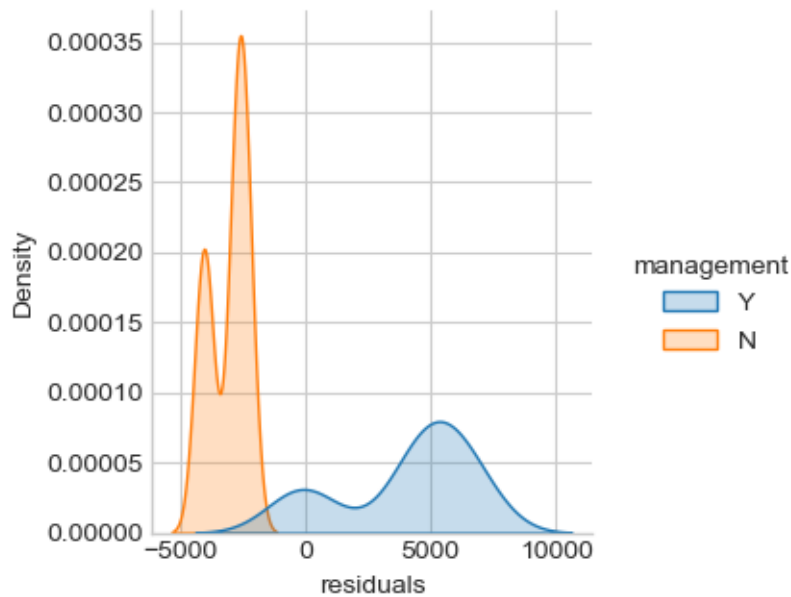
lm = smf.ols('salary ~ experience', df).fit()
df["residuals"] = lm.resid

print("Jarque-Bera normality test p-value %.5f" % \
      sm.stats.jarque_bera(lm.resid)[1])

ax = sns.displot(df, x='residuals', kind="kde", fill=True, aspect=1, height=fig_
    h*0.7)
ax = sns.displot(df, x='residuals', kind="kde", hue='management', fill=True,
    aspect=1, height=fig_h*0.7)
```

Jarque-Bera normality test p-value 0.04374





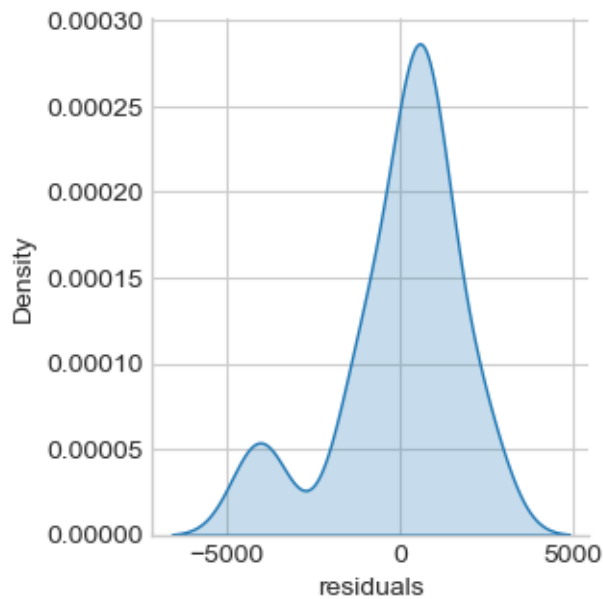
Normality assumption of the residuals can be rejected ($p\text{-value} < 0.05$). There is an effect of the “management” factor, take it into account.

One-way AN(C)OVA

- ANOVA: one categorical independent variable, i.e. one factor.
- ANCOVA: ANOVA with some covariates.

```
oneway = smf.ols('salary ~ management + experience', df).fit()
df["residuals"] = oneway.resid
sns.displot(df, x='residuals', kind="kde", fill=True,
            aspect=1, height=fig_h*0.7)
print(sm.stats.anova_lm(oneway, typ=2))
print("Jarque-Bera normality test p-value %.3f" % \
      sm.stats.jarque_bera(oneway.resid)[1])
```

	sum_sq	df	F	PR(>F)
management	5.755739e+08	1.0	183.593466	4.054116e-17
experience	3.334992e+08	1.0	106.377768	3.349662e-13
Residual	1.348070e+08	43.0	NaN	NaN
Jarque-Bera normality test p-value	0.004			



Distribution of residuals is still not normal but closer to normality. Both management and experience are significantly associated with salary.

Two-way AN(C)OVA

Ancova with two categorical independent variables, i.e. two factors.

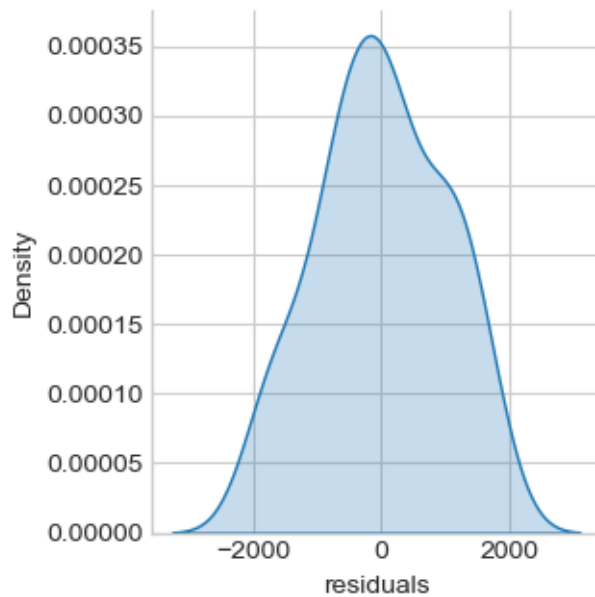
```
twoway = smf.ols('salary ~ education + management + experience', df).fit()

df["residuals"] = twoway.resid
sns.displot(df, x='residuals', kind="kde", fill=True,
            aspect=1, height=fig_h*0.7)
print(sm.stats.anova_lm(twoway, typ=2))

print("Jarque-Bera normality test p-value %.3f" % \
      sm.stats.jarque_bera(twoway.resid)[1])
```

	sum_sq	df	F	PR(>F)
education	9.152624e+07	2.0	43.351589	7.672450e-11
management	5.075724e+08	1.0	480.825394	2.901444e-24
experience	3.380979e+08	1.0	320.281524	5.546313e-21
Residual	4.328072e+07	41.0	NaN	NaN

Jarque-Bera normality test p-value 0.506



Normality assumption cannot be rejected. Assume it. Education, management and experience are significantly associated with salary.

Comparing Two Nested Models

oneway is nested within twoway. Comparing two nested models tells us if the additional predictors (i.e. education) of the full model significantly decrease the residuals. Such comparison can be done using an F -test on residuals:

```
print(twoway.compare_f_test(oneway)) # return F, pval, df
```

```
(np.float64(43.35158945918104), np.float64(7.6724495704955e-11), np.float64(2.0))
```

twoway is significantly better than one way

Factor Coding

Statsmodels contrasts. By default Pandas use “dummy coding”. Explore:

```
print(twoway.model.data.param_names)
print(twoway.model.data.exog[:10, :])
```

```
['Intercept', 'education[T.Master]', 'education[T.Ph.D]', 'management[T.Y]',
 → 'experience']
[[1. 0. 0. 1. 1.]
 [1. 0. 1. 0. 1.]
 [1. 0. 1. 1. 1.]
 [1. 1. 0. 0. 1.]
 [1. 0. 1. 0. 1.]
 [1. 1. 0. 1. 2.]
 [1. 1. 0. 0. 2.]
 [1. 0. 0. 0. 2.]
```

(continues on next page)

(continued from previous page)

```
[1. 0. 1. 0. 2.]
[1. 1. 0. 0. 3.]]
```

Contrasts and Post-hoc Tests

```
# t-test of the specific contribution of experience:
ttest_exp = twoway.t_test([0, 0, 0, 0, 1])
ttest_exp.pvalue, ttest_exp.tvalue
print(ttest_exp)

# Alternatively, you can specify the hypothesis tests using a string
twoway.t_test('experience')

# Post-hoc is salary of Master different salary of Ph.D?
# ie. t-test salary of Master = salary of Ph.D.
print(twoway.t_test('education[T.Master] = education[T.Ph.D]'))
```

Test for Constraints						
	coef	std err	t	P> t	[0.025	0.975]
c0	546.1840	30.519	17.896	0.000	484.549	607.819
Test for Constraints						
	coef	std err	t	P> t	[0.025	0.975]
c0	147.8249	387.659	0.381	0.705	-635.069	930.719

5.1.8 Multiple Comparisons

```
# Dataset

from pystatsml import datasets

n_features, n_informative = 1000, 100
Y, grp = datasets.make_twosamples(n_samples=100,
                                  n_features=n_features, n_informative=n_
↳ informative,
                                  group_scale=.5, noise_scale=1, shared_scale=0,
                                  random_state=42)

# Stats
import scipy.stats as stats
#import matplotlib.pyplot as plt
tvals, pvals = np.full(n_features, np.nan), np.full(n_features, np.nan)
for j in range(n_features):
```

(continues on next page)

(continued from previous page)

```

    tvals[j], pvals[j] = stats.ttest_ind(Y[grp==0, j], Y[grp==1, j],
                                         equal_var=True)

# Plot
fig, axis = plt.subplots(3, 1, figsize=(9, 9))#, sharex='col')

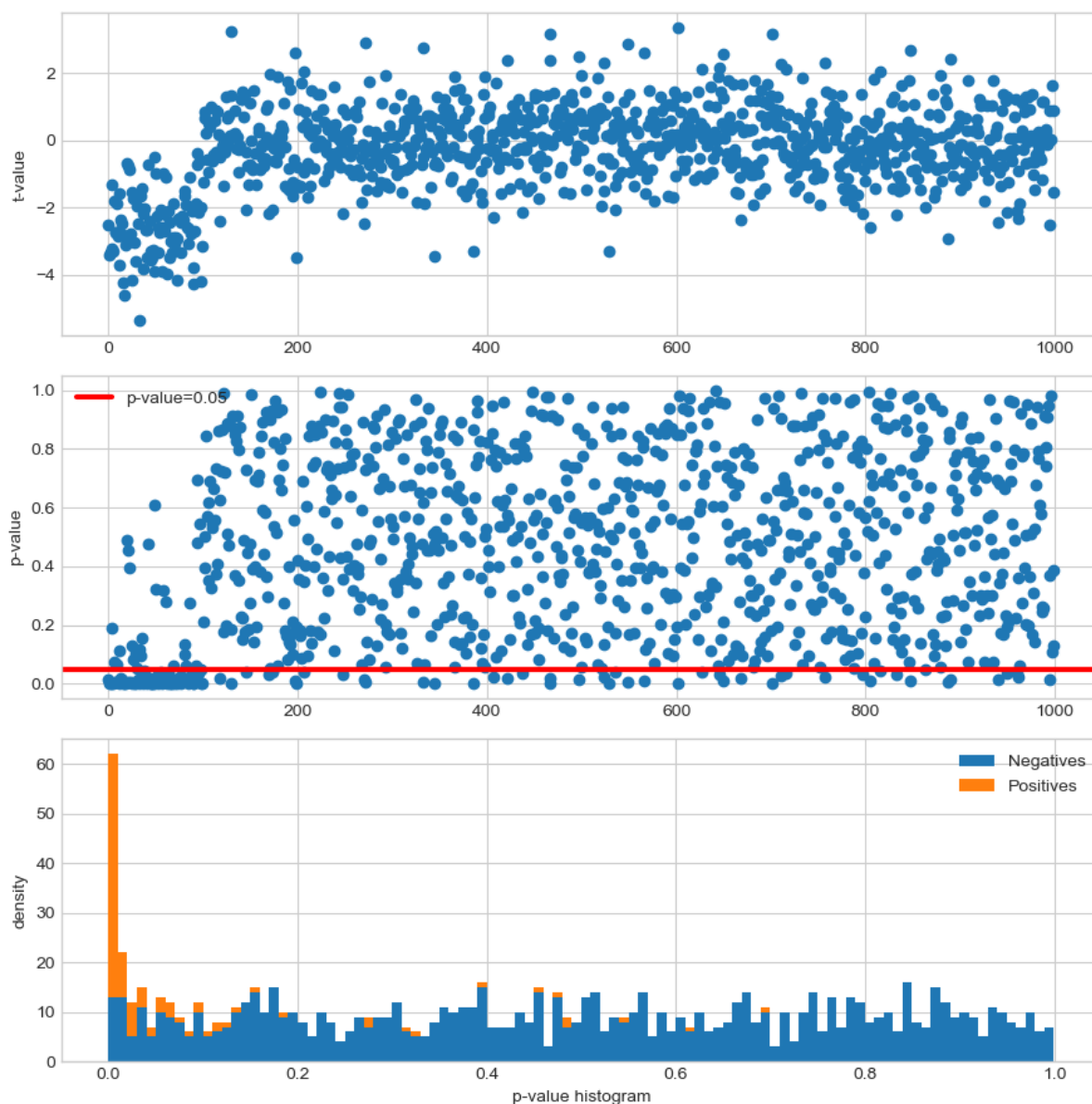
axis[0].plot(range(n_features), tvals, 'o')
axis[0].set_ylabel("t-value")

axis[1].plot(range(n_features), pvals, 'o')
axis[1].axhline(y=0.05, color='red', linewidth=3, label="p-value=0.05")
#axis[1].axhline(y=0.05, label="toto", color='red')
axis[1].set_ylabel("p-value")
axis[1].legend()

axis[2].hist([pvals[n_informative:], pvals[:n_informative]],
             stacked=True, bins=100, label=["Negatives", "Positives"])
axis[2].set_xlabel("p-value histogram")
axis[2].set_ylabel("density")
axis[2].legend()

plt.tight_layout()

```



Note that under the null hypothesis the distribution of the p -values is uniform.

Statistical measures:

- **True Positive (TP)** equivalent to a hit. The test correctly concludes the presence of an effect.
- **True Negative (TN)**. The test correctly concludes the absence of an effect.
- **False Positive (FP)** equivalent to a false alarm, **Type I error**. The test improperly concludes the presence of an effect. Thresholding at $p\text{-value} < 0.05$ leads to 47 FP.
- **False Negative (FN)** equivalent to a miss, **Type II error**. The test improperly concludes the absence of an effect.

```
P, N = n_info, n_features - n_info # Positives, Negatives
TP = np.sum(pvals[:n_info] < 0.05) # True Positives
FP = np.sum(pvals[n_info:] < 0.05) # False Positives
print("No correction, FP: %i (expected: %.2f), TP: %i" % (FP, N * 0.05, TP))
```


Bonferroni Correction for Multiple Comparisons

The Bonferroni correction is based on the idea that if an experimenter is testing P hypotheses, then one way of maintaining the **Family-wise error rate** **FWER** is to test each individual hypothesis at a statistical significance level of $1/P$ times the desired maximum overall level.

So, if the desired significance level for the whole family of tests is α (usually 0.05), then the Bonferroni correction would test each individual hypothesis at a significance level of α/P . For example, if a trial is testing $P = 8$ hypotheses with a desired $\alpha = 0.05$, then the Bonferroni correction would test each individual hypothesis at $\alpha = 0.05/8 = 0.00625$.

```
import statsmodels.sandbox.stats.multicomp as multicomp

_, pvals_fwer, _, _ = multicomp.multipletests(pvals, alpha=0.05,
                                              method='bonferroni')

TP = np.sum(pvals_fwer[:n_info] < 0.05) # True Positives
FP = np.sum(pvals_fwer[n_info:] < 0.05) # False Positives
print("FWER correction, FP: %i, TP: %i" % (FP, TP))
```

The False Discovery Rate (FDR) Correction for Multiple Comparisons

FDR-controlling procedures are designed to control the expected proportion of rejected null hypotheses that were incorrect rejections (“false discoveries”). FDR-controlling procedures provide less stringent control of Type I errors compared to the familywise error rate (FWER) controlling procedures (such as the Bonferroni correction), which control the probability of at least one Type I error. Thus, FDR-controlling procedures have greater power, at the cost of increased rates of Type I errors.

```
_, pvals_fdr, _, _ = multicomp.multipletests(pvals, alpha=0.05,
                                              method='fdr_bh')

TP = np.sum(pvals_fdr[:n_info] < 0.05) # True Positives
FP = np.sum(pvals_fdr[n_info:] < 0.05) # False Positives

print("FDR correction, FP: %i, TP: %i" % (FP, TP))
```

5.2 Hands-On: Brain volumes study

The study provides the brain volumes of grey matter (gm), white matter (wm) and cerebrospinal fluid) (csf) of 808 anatomical MRI scans.

```
import os
import tempfile
import urllib.request

import pandas as pd

# Stat
import statsmodels.formula.api as smfmla
import statsmodels.api as sm
import scipy.stats
```

(continues on next page)

(continued from previous page)

```
# Plot
import matplotlib.pyplot as plt
import seaborn as sns

# Plot parameters
plt.style.use('seaborn-v0_8-whitegrid')
fig_w, fig_h = plt.rcParams.get('figure.figsize')
plt.rcParams['figure.figsize'] = (fig_w, fig_h * .5)
```

5.2.1 Manipulate data

Set the working directory within a directory called “brainvol”

Create 2 subdirectories: *data* that will contain downloaded data and *reports* for results of the analysis.

```
WD = os.path.join(tempfile.gettempdir(), "brainvol")
os.makedirs(WD, exist_ok=True)
#os.chdir(WD)

# use cookiecutter file organization
# https://drivendata.github.io/cookiecutter-data-science/
os.makedirs(os.path.join(WD, "data"), exist_ok=True)
#os.makedirs("reports", exist_ok=True)
```

Fetch data

- Demographic data *demo.csv* (columns: *participant_id*, *site*, *group*, *age*, *sex*) and tissue volume data: *group* is Control or Patient. *site* is the recruiting site.
- Gray matter volume *gm.csv* (columns: *participant_id*, *session*, *gm_vol*)
- White matter volume *wm.csv* (columns: *participant_id*, *session*, *wm_vol*)
- Cerebrospinal Fluid *csf.csv* (columns: *participant_id*, *session*, *csf_vol*)

```
base_url = 'https://github.com/duchesnay/pystatsml/raw/master/datasets/brain_
↪volumes/%s'
data = dict()
for file in ["demo.csv", "gm.csv", "wm.csv", "csf.csv"]:
    urllib.request.urlretrieve(base_url % file, os.path.join(WD, "data", file))

# Read all CSV in one line
# dicts = {k: pd.read_csv(os.path.join(WD, "data", "%s.csv" % k))
#           for k in ["demo", "gm", "wm", "csf"]}

demo = pd.read_csv(os.path.join(WD, "data", "demo.csv"))
gm = pd.read_csv(os.path.join(WD, "data", "gm.csv"))
wm = pd.read_csv(os.path.join(WD, "data", "wm.csv"))
csf = pd.read_csv(os.path.join(WD, "data", "csf.csv"))
```

(continues on next page)

(continued from previous page)

```
print("tables can be merge using shared columns")
print(gm.head())
```

```
tables can be merge using shared columns
  participant_id session    gm_vol
0    sub-S1-0002  ses-01  0.672506
1    sub-S1-0002  ses-02  0.678772
2    sub-S1-0002  ses-03  0.665592
3    sub-S1-0004  ses-01  0.890714
4    sub-S1-0004  ses-02  0.881127
```

Merge tables according to *participant_id*

```
brain_vol = pd.merge(pd.merge(pd.merge(demo, gm), wm), csf)
assert brain_vol.shape == (808, 9)
```

Drop rows with missing values

```
brain_vol = brain_vol.dropna()
assert brain_vol.shape == (766, 9)
```

Compute Total Intra-cranial volume $tiv\_vol = gm\_vol + csf\_vol + wm\_vol$.

```
brain_vol["tiv_vol"] = brain_vol["gm_vol"] + \
    brain_vol["wm_vol"] + brain_vol["csf_vol"]
```

Compute tissue fractions $gm\_f = gm\_vol / tiv\_vol$, $wm\_f = wm\_vol / tiv\_vol$.

```
brain_vol["gm_f"] = brain_vol["gm_vol"] / brain_vol["tiv_vol"]
brain_vol["wm_f"] = brain_vol["wm_vol"] / brain_vol["tiv_vol"]
```

Save in a excel file *brain_vol.xlsx*

```
brain_vol.to_excel(os.path.join(WD, "data", "brain_vol.xlsx"),
                  sheet_name='data', index=False)
```

5.2.2 Descriptive Statistics

Load excel file *brain_vol.xlsx*

```
brain_vol = pd.read_excel(os.path.join(WD, "data", "brain_vol.xlsx"),
                          sheet_name='data')
# Round float at 2 decimals when printing
pd.options.display.float_format = '{:,.2f}'.format
```

Descriptive statistics Most of participants have several MRI sessions (column *session*) Select on rows from session one “ses-01”

```
brain_vol1 = brain_vol[brain_vol.session == "ses-01"]
# Check that there are no duplicates
assert len(brain_vol1.participant_id.unique()) == len(brain_vol1.participant_id)
```

Global descriptives statistics of numerical variables

```
desc_glob_num = brain_vol1.describe()
print(desc_glob_num)
```

	age	gm_vol	wm_vol	csf_vol	tiv_vol	gm_f	wm_f
count	244.00	244.00	244.00	244.00	244.00	244.00	244.00
mean	34.54	0.71	0.44	0.31	1.46	0.49	0.30
std	12.09	0.08	0.07	0.08	0.17	0.04	0.03
min	18.00	0.48	0.05	0.12	0.83	0.37	0.06
25%	25.00	0.66	0.40	0.25	1.34	0.46	0.28
50%	31.00	0.70	0.43	0.30	1.45	0.49	0.30
75%	44.00	0.77	0.48	0.37	1.57	0.52	0.31
max	61.00	1.03	0.62	0.63	2.06	0.60	0.36

Global Descriptive statistics of categorical variable

```
desc_glob_cat = brain_vol1[["site", "group", "sex"]].describe(include='all')
print(desc_glob_cat)

print("Get count by level")
desc_glob_cat = pd.DataFrame({col: brain_vol1[col].value_counts().to_dict()
                              for col in ["site", "group", "sex"]})
print(desc_glob_cat)
```

	site	group	sex
count	244	244	244
unique	7	2	2
top	S7	Patient	M
freq	65	157	155

Get count by level

	site	group	sex
S7	65.00	NaN	NaN
S5	62.00	NaN	NaN
S8	59.00	NaN	NaN
S3	29.00	NaN	NaN
S4	15.00	NaN	NaN
S1	13.00	NaN	NaN
S6	1.00	NaN	NaN
Patient	NaN	157.00	NaN
Control	NaN	87.00	NaN
M	NaN	NaN	155.00
F	NaN	NaN	89.00

Remove the single participant from site 6

```
brain_vol = brain_vol[brain_vol.site != "S6"]
brain_vol1 = brain_vol[brain_vol.session == "ses-01"]
desc_glob_cat = pd.DataFrame({col: brain_vol1[col].value_counts().to_dict()
                              for col in ["site", "group", "sex"]})
print(desc_glob_cat)
```

	site	group	sex
S7	65.00	NaN	NaN
S5	62.00	NaN	NaN
S8	59.00	NaN	NaN
S3	29.00	NaN	NaN
S4	15.00	NaN	NaN
S1	13.00	NaN	NaN
Patient	NaN	157.00	NaN
Control	NaN	86.00	NaN
M	NaN	NaN	155.00
F	NaN	NaN	88.00

Descriptives statistics of numerical variables per clinical status

```
desc_group_num = brain_vol1[["group", 'gm_vol']].groupby("group").describe()
print(desc_group_num)
```

	gm_vol							
	count	mean	std	min	25%	50%	75%	max
group								
Control	86.00	0.72	0.09	0.48	0.66	0.71	0.78	1.03
Patient	157.00	0.70	0.08	0.53	0.65	0.70	0.76	0.90

5.2.3 Statistics

Objectives:

1. Site effect of gray matter atrophy
2. Test the association between the age and gray matter atrophy in the control and patient population independently.
3. Test for differences of atrophy between the patients and the controls
4. Test for interaction between age and clinical status, ie: is the brain atrophy process in patient population faster than in the control population.
5. The effect of the medication in the patient population.

1 Site effect on Grey Matter atrophy

The model is Oneway Anova $gm\_f \sim site$ The ANOVA test has important assumptions that must be satisfied in order for the associated p-value to be valid.

- The samples are independent.
- Each sample is from a normally distributed population.
- The population standard deviations of the groups are all equal. This property is known as homoscedasticity.

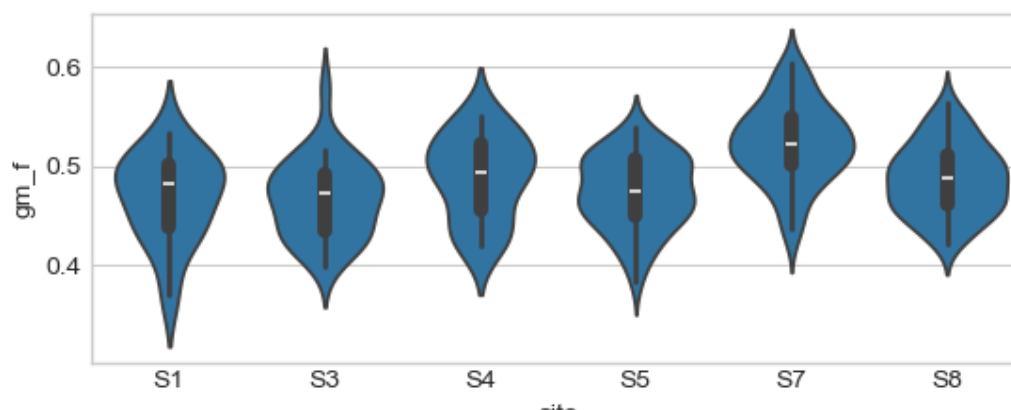
Plot

```
sns.violinplot(x="site", y="gm_f", data=brain_vol1)
# sns.violinplot(x="site", y="wm_f", data=brain_vol1)
```

(continues on next page)

(continued from previous page)

```
brain_vol1.groupby('site')['age'].describe()
```



Stats with scipy

```
fstat, pval = scipy.stats.f_oneway(*[brain_vol1.gm_f[brain_vol1.site == s]
                                     for s in brain_vol1.site.unique()])
print("Oneway Anova gm_f ~ site F=%.2f, p-value=%E" % (fstat, pval))
```

```
Oneway Anova gm_f ~ site F=14.82, p-value=1.188136E-12
```

Stats with statsmodels

```
anova = smfmla.ols("gm_f ~ site", data=brain_vol1).fit()
# print(anova.summary())
print("Site explains %.2f%% of the grey matter fraction variance" %
      (anova.rsquared * 100))

print(sm.stats.anova_lm(anova, typ=2))
```

```
Site explains 23.82% of the grey matter fraction variance
```

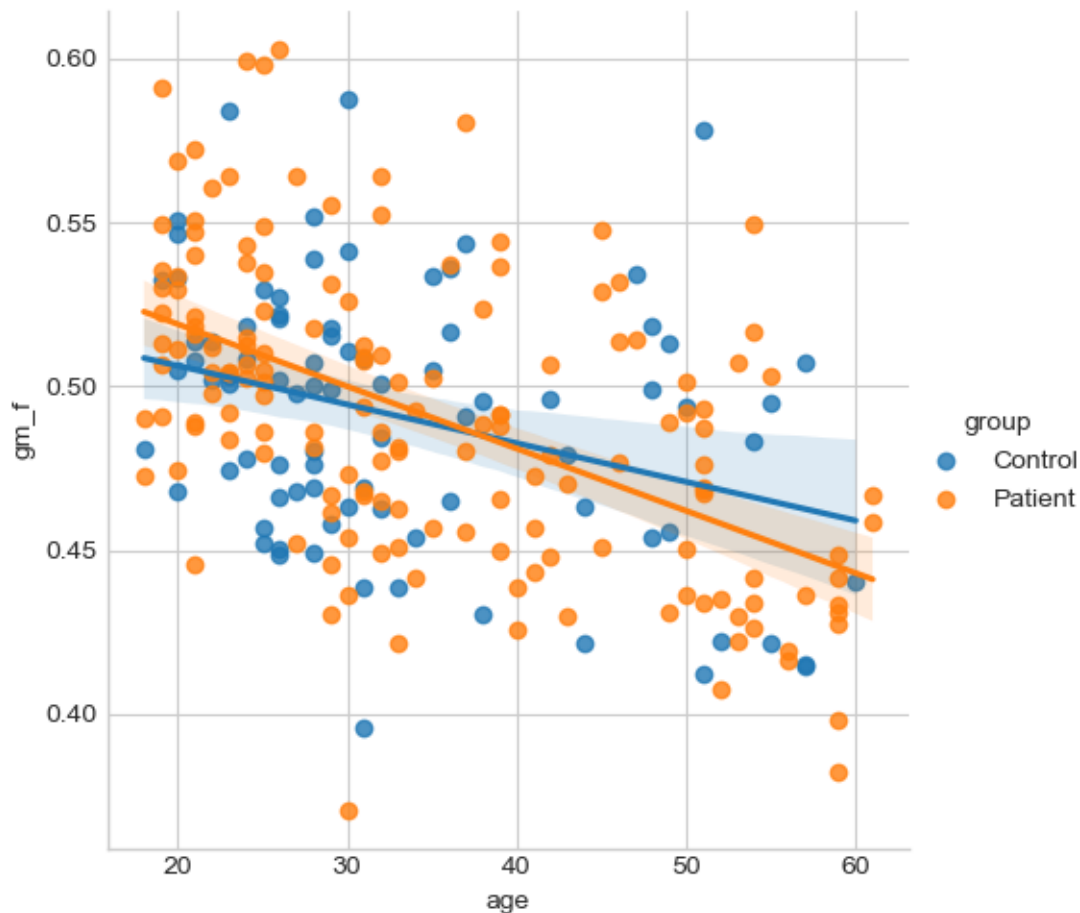
	sum_sq	df	F	PR(>F)
site	0.11	5.00	14.82	0.00
Residual	0.35	237.00	NaN	NaN

2. Test the association between the age and gray matter atrophy in the control and patient population independently.

Plot

```
sns.lmplot(x="age", y="gm_f", hue="group", data=brain_vol1)

brain_vol1_ctl = brain_vol1[brain_vol1.group == "Control"]
brain_vol1_pat = brain_vol1[brain_vol1.group == "Patient"]
```



Stats with scipy

```
print("--- In control population ---")
beta, beta0, r_value, p_value, std_err = \
    scipy.stats.linregress(x=brain_vol1_ctl.age, y=brain_vol1_ctl.gm_f)

print("gm_f = %f * age + %f" % (beta, beta0))
print("Corr: %f, r-squared: %f, p-value: %f, std_err: %f" \
      % (r_value, r_value**2, p_value, std_err))

print("--- In patient population ---")
beta, beta0, r_value, p_value, std_err = \
    scipy.stats.linregress(x=brain_vol1_pat.age, y=brain_vol1_pat.gm_f)

print("gm_f = %f * age + %f" % (beta, beta0))
print("Corr: %f, r-squared: %f, p-value: %f, std_err: %f" \
      % (r_value, r_value**2, p_value, std_err))

print("Decrease seems faster in patient than in control population")
```

```
--- In control population ---
gm_f = -0.001181 * age + 0.529829
Corr: -0.325122, r-squared: 0.105704, p-value: 0.002255, std_err: 0.000375
```

(continues on next page)

(continued from previous page)

```

--- In patient population ---
gm_f = -0.001899 * age + 0.556886
Corr: -0.528765, r-squared: 0.279592, p-value: 0.000000, std_err: 0.000245
Decrease seems faster in patient than in control population

```

Stats with statsmodels

```

print("--- In control population ---")
lr = smfmla.ols("gm_f ~ age", data=brain_vol1_ctl).fit()
print(lr.summary())
print("Age explains %.2f%% of the grey matter fraction variance" %
      (lr.rsquared * 100))

print("--- In patient population ---")
lr = smfmla.ols("gm_f ~ age", data=brain_vol1_pat).fit()
print(lr.summary())
print("Age explains %.2f%% of the grey matter fraction variance" %
      (lr.rsquared * 100))

```

```

--- In control population ---

```

OLS Regression Results						
Dep. Variable:	gm_f	R-squared:	0.106			
Model:	OLS	Adj. R-squared:	0.095			
Method:	Least Squares	F-statistic:	9.929			
Date:	jeu., 08 mai 2025	Prob (F-statistic):	0.00226			
Time:	02:16:34	Log-Likelihood:	159.34			
No. Observations:	86	AIC:	-314.7			
Df Residuals:	84	BIC:	-309.8			
Df Model:	1					
Covariance Type:	nonrobust					
	coef	std err	t	P> t	[0.025	0.975]
Intercept	0.5298	0.013	40.350	0.000	0.504	0.556
age	-0.0012	0.000	-3.151	0.002	-0.002	-0.000
Omnibus:	0.946	Durbin-Watson:	1.628			
Prob(Omnibus):	0.623	Jarque-Bera (JB):	0.782			
Skew:	0.233	Prob(JB):	0.676			
Kurtosis:	2.962	Cond. No.	111.			

```

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
    specified.
Age explains 10.57% of the grey matter fraction variance
--- In patient population ---

```

OLS Regression Results						

(continues on next page)

(continued from previous page)

=====						
Dep. Variable:	gm_f	R-squared:	0.280			
Model:	OLS	Adj. R-squared:	0.275			
Method:	Least Squares	F-statistic:	60.16			
Date:	jeu., 08 mai 2025	Prob (F-statistic):	1.09e-12			
Time:	02:16:34	Log-Likelihood:	289.38			
No. Observations:	157	AIC:	-574.8			
Df Residuals:	155	BIC:	-568.7			
Df Model:	1					
Covariance Type:	nonrobust					
=====						
	coef	std err	t	P> t	[0.025	0.975]

Intercept	0.5569	0.009	60.817	0.000	0.539	0.575
age	-0.0019	0.000	-7.756	0.000	-0.002	-0.001
=====						
Omnibus:	2.310	Durbin-Watson:	1.325			
Prob(Omnibus):	0.315	Jarque-Bera (JB):	1.854			
Skew:	0.230	Prob(JB):	0.396			
Kurtosis:	3.268	Cond. No.	111.			
=====						

Notes:

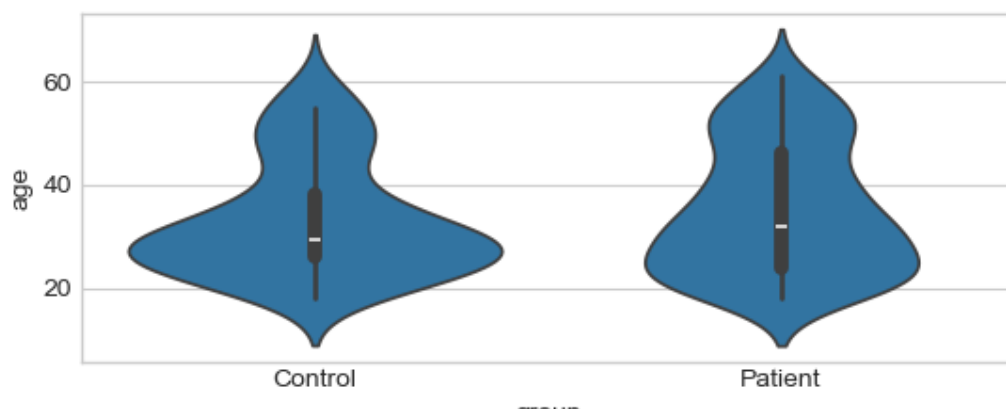
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

Age explains 27.96% of the grey matter fraction variance

Before testing for differences of atrophy between the patients and the controls **Preliminary tests for age x group effect** (patients would be older or younger than Controls)

Plot

```
sns.violinplot(x="group", y="age", data=brain_vol1)
```



```
<Axes: xlabel='group', ylabel='age'>
```

Stats with scipy

```
print(scipy.stats.ttest_ind(brain_vol1_ctl.age, brain_vol1_pat.age))
```

```
TtestResult(statistic=np.float64(-1.2155557697674162), pvalue=np.float64(0.
↪225343592508479), df=np.float64(241.0))
```

Stats with statsmodels

```
print(smfrmla.ols("age ~ group", data=brain_vol1).fit().summary())
print("No significant difference in age between patients and controls")
```

OLS Regression Results						
=====						
Dep. Variable:	age	R-squared:	0.006			
Model:	OLS	Adj. R-squared:	0.002			
Method:	Least Squares	F-statistic:	1.478			
Date:	jeu., 08 mai 2025	Prob (F-statistic):	0.225			
Time:	02:16:34	Log-Likelihood:	-949.69			
No. Observations:	243	AIC:	1903.			
Df Residuals:	241	BIC:	1910.			
Df Model:	1					
Covariance Type:	nonrobust					
=====						
	coef	std err	t	P> t	[0.025	0.
↪975]						

↪--						
Intercept	33.2558	1.305	25.484	0.000	30.685	35.
↪826						
group[T.Patient]	1.9735	1.624	1.216	0.225	-1.225	5.
↪172						
=====						
Omnibus:	35.711	Durbin-Watson:	2.096			
Prob(Omnibus):	0.000	Jarque-Bera (JB):	20.726			
Skew:	0.569	Prob(JB):	3.16e-05			
Kurtosis:	2.133	Cond. No.	3.12			
=====						
Notes:						
[1] Standard Errors assume that the covariance matrix of the errors is correctly_						
↪specified.						
No significant difference in age between patients and controls						

Preliminary tests for sex x group (more/less males in patients than in Controls)

```
crosstab = pd.crosstab(brain_vol1.sex, brain_vol1.group)
print("Observed contingency table")
print(crosstab)

chi2, pval, dof, expected = scipy.stats.chi2_contingency(crosstab)
```

(continues on next page)

(continued from previous page)

```
print("Chi2 = %f, pval = %f" % (chi2, pval))
print("No significant difference in sex between patients and controls")
```

Observed contingency table

group Control Patient

sex

F 33 55

M 53 102

Chi2 = 0.143253, pval = 0.705068

No significant difference in sex between patients and controls

3. Test for differences of atrophy between the patients and the controls

```
print(sm.stats.anova_lm(smfrmla.ols("gm_f ~ group", data=brain_vol1).fit(),
                                typ=2))
print("No significant difference in atrophy between patients and controls")
```

	sum_sq	df	F	PR(>F)
group	0.00	1.00	0.01	0.92
Residual	0.46	241.00	NaN	NaN

No significant difference in atrophy between patients and controls

This model is simplistic we should adjust for age and site

```
print(sm.stats.anova_lm(smfrmla.ols(
    "gm_f ~ group + age + site", data=brain_vol1).fit(), typ=2))
print("No significant difference in GM between patients and controls")
```

	sum_sq	df	F	PR(>F)
group	0.00	1.00	1.82	0.18
site	0.11	5.00	19.79	0.00
age	0.09	1.00	86.86	0.00
Residual	0.25	235.00	NaN	NaN

No significant difference in GM between patients and controls

Observe age effect

4. Test for interaction between age and clinical status, ie: is the brain atrophy process in patient population faster than in the control population.

```
ancova = smfrmla.ols("gm_f ~ group:age + age + site", data=brain_vol1).fit()
print(sm.stats.anova_lm(ancova, typ=2))

print("= Parameters =")
print(ancova.params)

print("%.3f%% of grey matter loss per year (almost %.1f%% per decade)" %
      (ancova.params.age * 100, ancova.params.age * 100 * 10))
```

(continues on next page)

(continued from previous page)

```
print("grey matter loss in patients is accelerated by %.3f%% per decade" %
      (ancova.params['group[T.Patient]:age'] * 100 * 10))
```

	sum_sq	df	F	PR(>F)
site	0.11	5.00	20.28	0.00
age	0.10	1.00	89.37	0.00
group:age	0.00	1.00	3.28	0.07
Residual	0.25	235.00	NaN	NaN

= Parameters =

Intercept	0.52
site[T.S3]	0.01
site[T.S4]	0.03
site[T.S5]	0.01
site[T.S7]	0.06
site[T.S8]	0.02
age	-0.00
group[T.Patient]:age	-0.00

dtype: float64
 -0.148% of grey matter loss per year (almost -1.5% per decade)
 grey matter loss in patients is accelerated by -0.232% per decade

Total running time of the script: (0 minutes 2.668 seconds)

5.3 Linear Mixed Models

Acknowledgements: Firstly, it's right to pay thanks to the blogs and sources I have used in writing this tutorial. Many parts of the text are quoted from the brilliant book from Brady T. West, Kathleen B. Welch and Andrzej T. Galecki, see [Brady et al. 2014] in the references section below.

5.3.1 Introduction

Quoted from [Brady et al. 2014]: A linear mixed model (LMM) is a parametric linear model for **clustered, longitudinal, or repeated-measures** data that quantifies the relationships between a continuous dependent variable and various predictor variables. An LMM may include both **fixed-effect** parameters associated with one or more continuous or categorical covariates and **random effects** associated with one or more random factors. The mix of fixed and random effects gives the linear mixed model its name. Whereas fixed-effect parameters describe the relationships of the covariates to the dependent variable for an entire population, random effects are specific to clusters or subjects within a population. LMM is closely related with hierarchical linear model (HLM).

Clustered/structured datasets

Quoted from [Bruin 2006]: Random effects, are used when there is non independence in the data, such as arises from a hierarchical structure with clustered data. For example, students could be sampled from within classrooms, or patients from within doctors. When there are multiple levels, such as patients seen by the same doctor, the variability in the outcome can be thought of as being either within group or between group. Patient level observations are not

independent, as within a given doctor patients are more similar. Units sampled at the highest level (in our example, doctors) are independent.

The continuous outcome variables is **structured or clustered** into **units** within **observations are not independent**. Types of clustered data:

1. studies with clustered data, such as students in classrooms, or experimental designs with random blocks, such as batches of raw material for an industrial process
2. **longitudinal or repeated-measures** studies, in which subjects are measured repeatedly over time or under different conditions.

Mixed effects = fixed + random effects

Fixed effects may be associated with continuous covariates, such as weight, baseline test score, or socioeconomic status, which take on values from a continuous (or sometimes a multivalued ordinal) range, or with factors, such as gender or treatment group, which are categorical. Fixed effects are unknown constant parameters associated with either continuous covariates or the levels of categorical factors in an LMM. Estimation of these parameters in LMMs is generally of intrinsic interest, because they indicate the relationships of the covariates with the continuous outcome variable.

Example: Suppose we want to study the relationship between the height of individuals and their gender. We will: sample individuals in a population (first source of randomness), measure their height (second source of randomness), and consider their gender (fixed for a given individual). Finally, these measures are modeled in the following linear model:

$$\text{height}_i = \beta_0 + \beta_1 \text{gender}_i + \varepsilon_i$$

- height: is the quantitative dependant (outcome, prediction) variable,
- gender: is an independant factor. It is known for a given individual. It is assumed that it has the same effect on all sampled individuals.
- ε is the noise. The sampling and measurement hazards are confounded at the individual level in this random variable. It is a random effect at the individual level.

Random effect When the levels of a factor can be thought of as having been sampled from a sample space, such that each particular level is not of intrinsic interest (e.g., classrooms or clinics that are randomly sampled from a larger population of classrooms or clinics), the effects associated with the levels of those factors can be modeled as random effects in an LMM. In contrast to fixed effects, which are represented by constant parameters in an LMM, random effects are represented by (unobserved) random variables, which are usually assumed to follow a normal distribution.

Example: Suppose now that we want to study the same effect on a global scale but by randomly sampling countries (j) and then individuals (i) in these countries. The model will be the following:

$$\text{height}_{ij} = \beta_0 + \beta_1 \text{gender}_{ij} + u_j \text{country}_{ij} + \varepsilon_{ij}$$

- $\text{country}_{ij} = \{1 \text{ if individual } i \text{ belongs to country } j, 0 \text{ otherwise}\}$, is an independant random factor which has three important properties:
 1. has been **sampled** (third source of randomness)
 2. **is not of interest**

- creates **clusters** of individuals within the same country whose heights is likely to be **correlated**. u_j will be the random effect associated to country j . It can be modeled as a random country-specific shift in height, a.k.a. a random intercept.

5.3.2 Random intercept

The `score_parentedu_byclass` dataset measure a score obtained by 60 students, indexed by i , within 3 classroom (with different teacher), indexed by j , given the education level `edu` of their parents. We want to study the link between `score` and `edu`. Observations, `score` are structured by the sampling of classroom, see Fig below. `score` from the same classroom are not independent from each other: they shifted upward or backward thanks to a classroom or teacher effect. There is an **intercept** for each classroom. But this effect is not known given a student (unlike the age or the sex), it is a consequence of a random sampling of the classrooms. It is called a **random intercept**.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import statsmodels.api as sm
import statsmodels.formula.api as smf

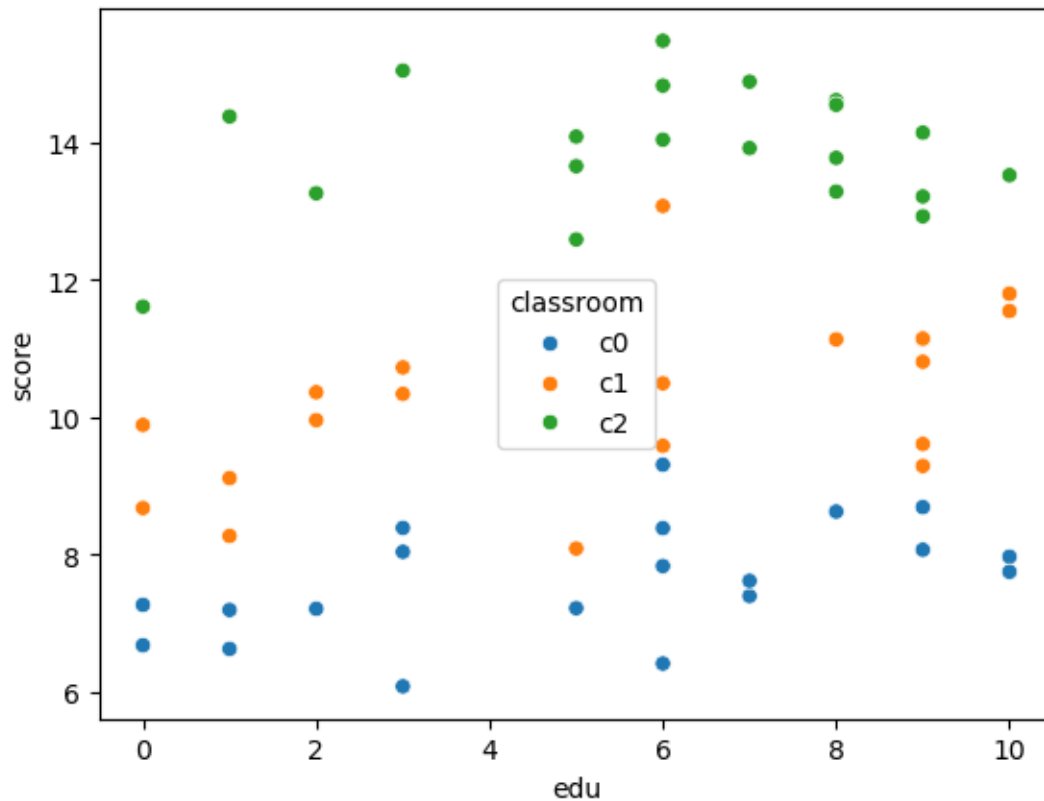
from stat_lmm_utils import rmse_coef_tstat_pval
from stat_lmm_utils import plot_lm_diagnosis
from stat_lmm_utils import plot_ancova_oneslope_grpintercept
from stat_lmm_utils import plot_lmm_oneslope_randintercept
from stat_lmm_utils import plot_ancova_fullmodel

results = pd.DataFrame(columns=["Model", "RMSE", "Coef", "Stat", "Pval"])

df = pd.read_csv('datasets/score_parentedu_byclass.csv')
print(df.head())

_ = sns.scatterplot(x="edu", y="score", hue="classroom", data=df)
```

	classroom	edu	score
0	c0	2	7.204352
1	c0	10	7.963083
2	c0	3	8.383137
3	c0	5	7.213047
4	c0	6	8.379630



Global fixed effect

Global effect regresses the the independant variable $y = \text{score}$ on the dependant variable $x = \text{edu}$ without considering the any classroom effect. For each individual i the model is:

$$y_{ij} = \beta_0 + \beta_1 x_{ij} + \varepsilon_{ij},$$

where, β_0 is the global intercept, β_1 is the slope associated with edu and ε_{ij} is the random error at the individual level. Note that the classroom, j index is not taken into account by the model and could be removed from the equation.

The general R formula is: $y \sim x$ which in this case is $\text{score} \sim \text{edu}$. This model is:

- **Not sensitive** since it does not model the classroom effect (high standard error).
- **Wrong** because, residuals are not normals, and it considers samples from the same classroom to be independant.

```
lm_glob = smf.ols('score ~ edu', df).fit()

#print(lm_glob.summary())
print(lm_glob.t_test('edu'))
print("MSE=%.3f" % lm_glob.mse_resid)
results.loc[len(results)] = ["LM-Global (biased)"] + \
    list(rmse_coef_tstat_pval(mod=lm_glob, var='edu'))
```

Test for Constraints

coef	std err	t	P> t	[0.025	0.975]

(continues on next page)

(continued from previous page)

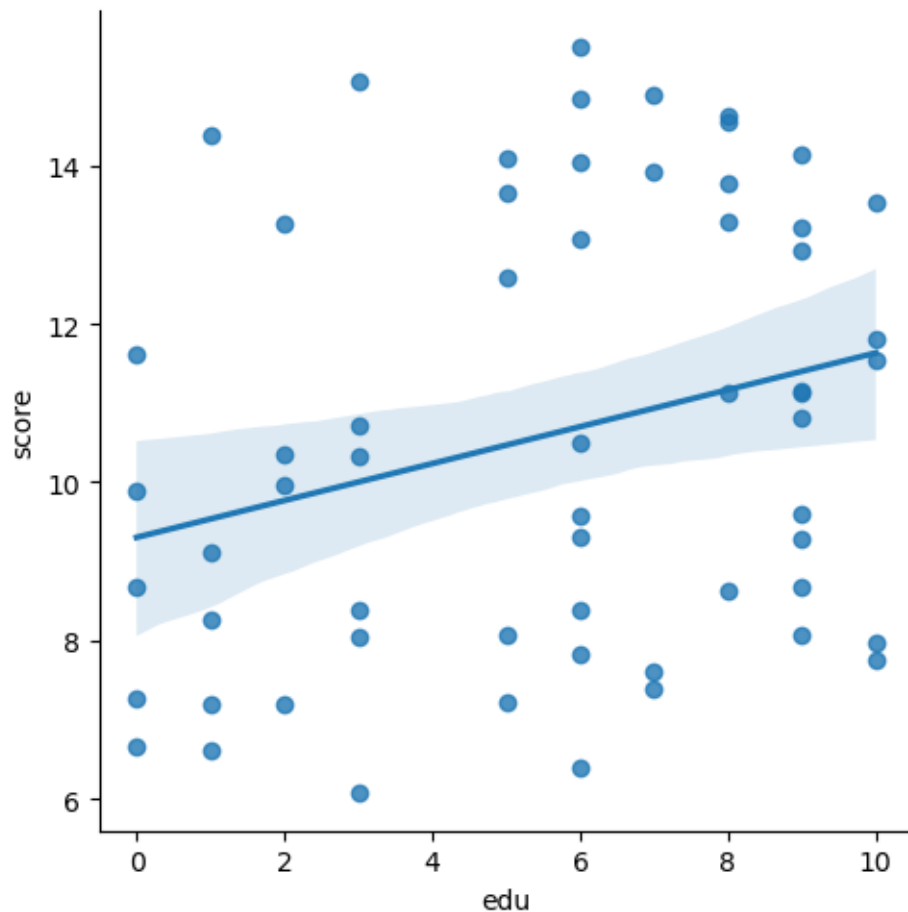
```

-----
c0      0.2328      0.109      2.139      0.037      0.015      0.451
=====
MSE=7.262

```

Plot

```
_ = sns.lmplot(x="edu", y="score", data=df)
```

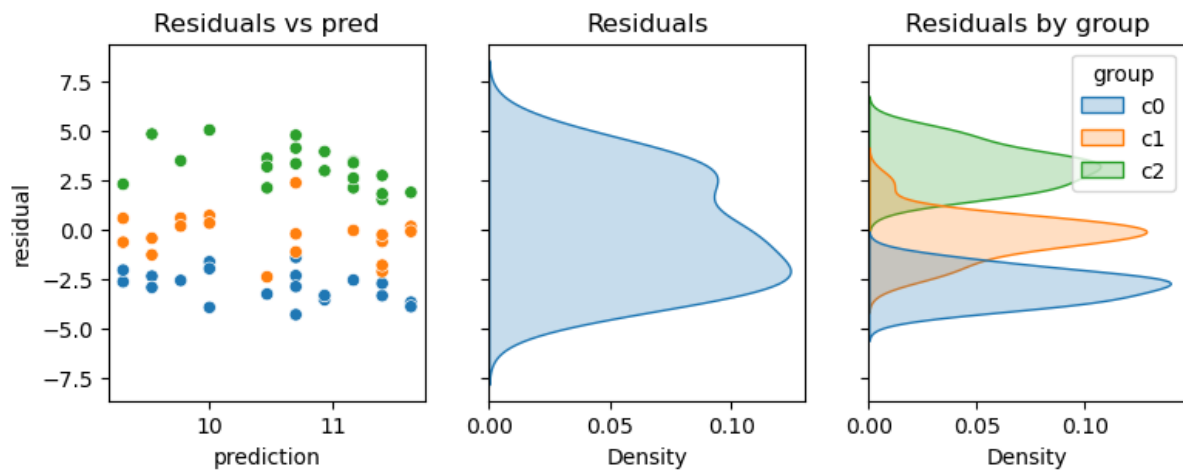


Model diagnosis: plot the normality of the residuals and residuals vs prediction.

```

plot_lm_diagnosis(residual=lm_glob.resid,
                  prediction=lm_glob.predict(df), group=df.classroom)

```

Model a classroom intercept as a fixed effect: ANCOVA

Remember ANCOVA = ANOVA with covariates. Model the classroom $z = \text{classroom}$ (as a fixed effect), ie a vertical shift for each classroom. The slope is the same for all classrooms. For each individual i and each classroom j the model is:

$$y_{ij} = \beta_0 + \beta_1 x_{ij} + u_j z_{ij} + \varepsilon_{ij},$$

where, u_j is the coefficient (an intercept, or a shift) associated with classroom j and $z_{ij} = 1$ if subject i belongs to classroom j else $z_{ij} = 0$.

The general R formula is: $y \sim x + z$ which in this case is $\text{score} \sim \text{edu} + \text{classroom}$.

This model is:

- **Sensitive** since it does not model the classroom effect (lower standard error). But,
- **questionable** because it considers the classroom to have a fixed constant effect without any uncertainty. However, those classrooms have been sampled from a larger samples of classrooms within the country.

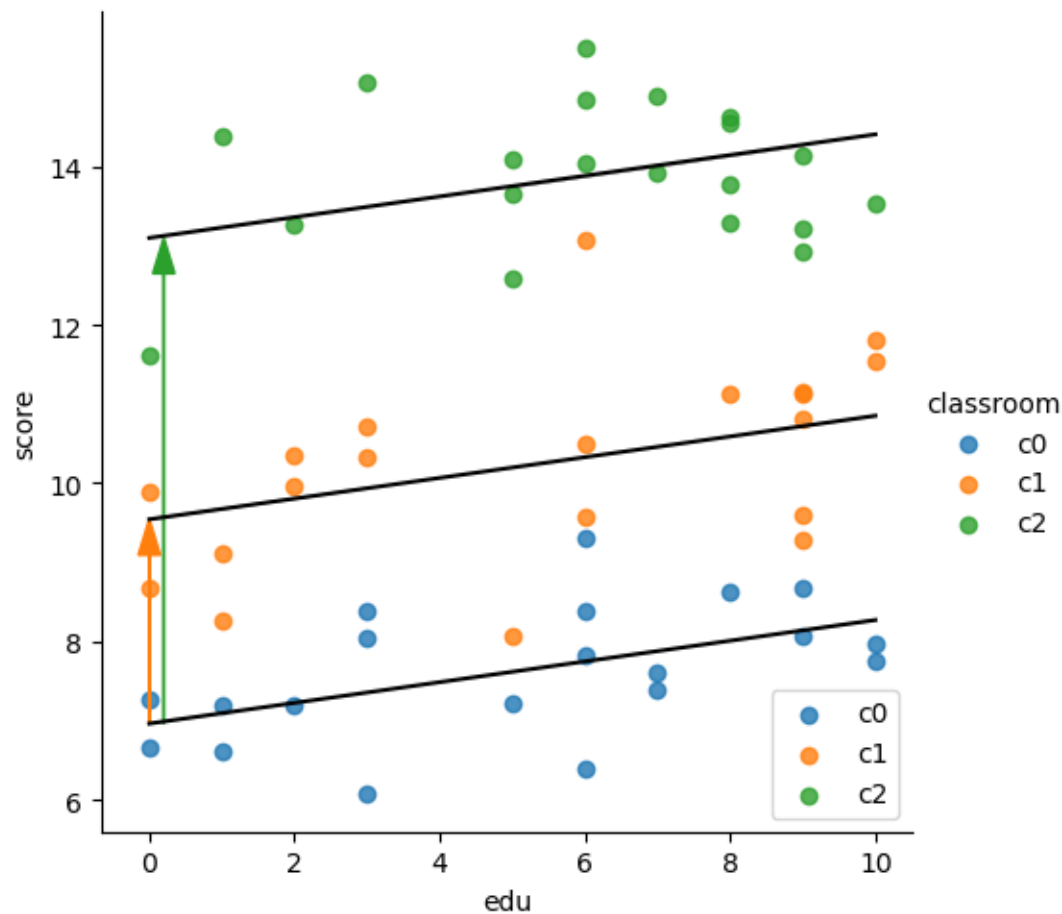
```
ancova_inter = smf.ols('score ~ edu + classroom', df).fit()
# print(sm.stats.anova_lm(ancova_inter, typ=3))
# print(ancova_inter.summary())
print(ancova_inter.t_test('edu'))

print("MSE=%.3f" % ancova_inter.mse_resid)
results.loc[len(results)] = ["ANCOVA-Inter (biased)"] + \
    list(rmse_coef_tstat_pval(mod=ancova_inter, var='edu'))
```

Test for Constraints						
	coef	std err	t	P> t	[0.025	0.975]
c0	0.1307	0.038	3.441	0.001	0.055	0.207
MSE=0.869						

Plot

```
plot_ancova_oneslope_grpintercept(x="edu", y="score",
                                  group="classroom", model=ancova_inter, df=df)
```



Explore the model

```
mod = ancova_inter

print("## Design matrix (independent variables):")
print(mod.model.exog_names)
print(mod.model.exog[:10])

print("## Outcome (dependant variable):")
print(mod.model.endog_names)
print(mod.model.endog[:10])

print("## Fitted model:")
print(mod.params)
sse_ = np.sum(mod.resid ** 2)
df_ = mod.df_resid
mod.df_model
print("MSE %f" % (sse_ / df_), "or", mod.mse_resid)

print("## Statistics:")
print(mod.tvalues, mod.pvalues)
```

```

## Design matrix (independant variables):
['Intercept', 'classroom[T.c1]', 'classroom[T.c2]', 'edu']
[[ 1.  0.  0.  2.]
 [ 1.  0.  0. 10.]
 [ 1.  0.  0.  3.]
 [ 1.  0.  0.  5.]
 [ 1.  0.  0.  6.]
 [ 1.  0.  0.  6.]
 [ 1.  0.  0.  3.]
 [ 1.  0.  0.  0.]
 [ 1.  0.  0.  6.]
 [ 1.  0.  0.  9.]]
## Outcome (dependant variable):
score
[7.20435162 7.96308267 8.38313712 7.21304665 8.37963003 6.40552793
 8.03417677 6.67164168 7.8268605  8.06401823]
## Fitted model:
Intercept          6.965429
classroom[T.c1]    2.577854
classroom[T.c2]    6.129755
edu                0.130717
dtype: float64
MSE 0.869278 or 0.8692776165530408
## Statistics:
Intercept          24.474487
classroom[T.c1]     8.736851
classroom[T.c2]    20.620005
edu                3.441072
dtype: float64 Intercept          1.377577e-31
classroom[T.c1]      4.815552e-12
classroom[T.c2]      7.876446e-28
edu                 1.102091e-03
dtype: float64

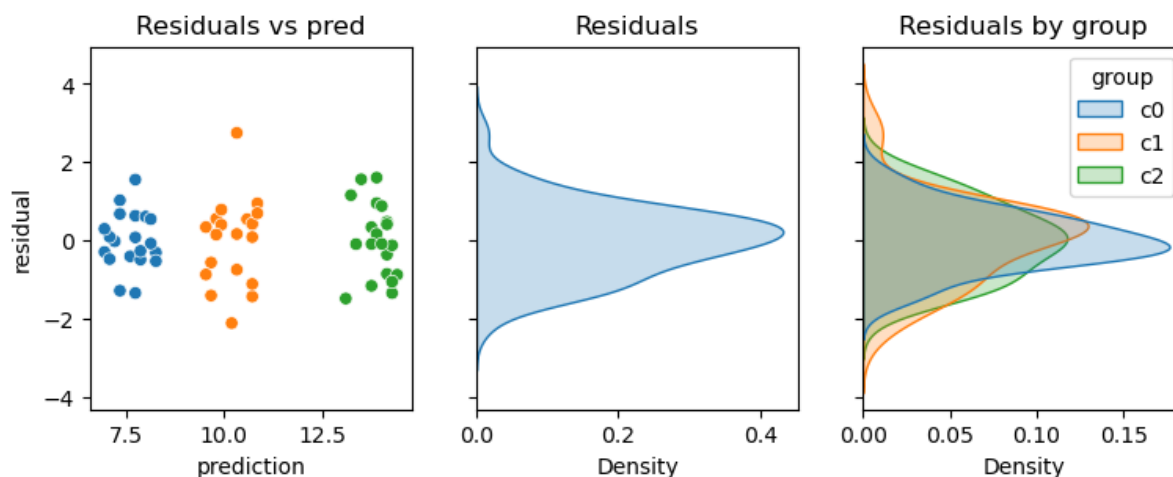
```

Normality of the residuals

```

plot_lm_diagnosis(residual=ancova_inter.resid,
                  prediction=ancova_inter.predict(df), group=df.classroom)

```



Fixed effect is the coefficient or parameter (β_1 in the model) that is associated with a continuous covariates (age, education level, etc.) or (categorical) factor (sex, etc.) that is known without uncertainty once a subject is sampled.

Random effect, in contrast, is the coefficient or parameter (u_j in the model below) that is associated with a continuous covariates or factor (classroom, individual, etc.) that is not known without uncertainty once a subject is sampled. It generally correspond to some random sampling. Here the classroom effect depends on the teacher which has been sampled from a larger samples of classrooms within the country. Measures are structured by units or a clustering structure that is possibly hierarchical. Measures within units are not independent. Measures between top level units are independent.

There are multiple ways to deal with structured data with random effect. One simple approach is to aggregate.

Aggregation of data into independent units

Aggregation of measure at classroom level: average all values within classrooms to perform statistical analysis between classroom. 1. **Level 1 (within unit)**: Average by classroom:

$$x_j = \text{mean}_i(x_{ij}), y_j = \text{mean}_i(y_{ij}), \text{ for } j \in \{1, 2, 3\}.$$

2. **Level 2 (between independent units)** Regress averaged score on a averaged edu:

$$y_j = \beta_0 + \beta_1 x_j + \varepsilon_j.$$

The general R formula is: $y \sim x$ which in this case is $\text{score} \sim \text{edu}$.

This model is:

- **Correct** because the aggregated data are independent.
- **Not sensitive** since all the within classroom association between edu and is lost. Moreover, at the aggregate level, there would only be three data points.

```

aggregate = df.groupby('classroom').mean()
lm_aggregate = smf.ols('score ~ edu', aggregate).fit()
#print(lm_aggregate.summary())
print(lm_aggregate.t_test('edu'))

```

(continues on next page)

(continued from previous page)

```
print("MSE=%.3f" % lm_aggregate.mse_resid)
results.loc[len(results)] = ["Aggregation"] + \
    list(rmse_coef_tstat_pval(mod=lm_aggregate, var='edu'))
```

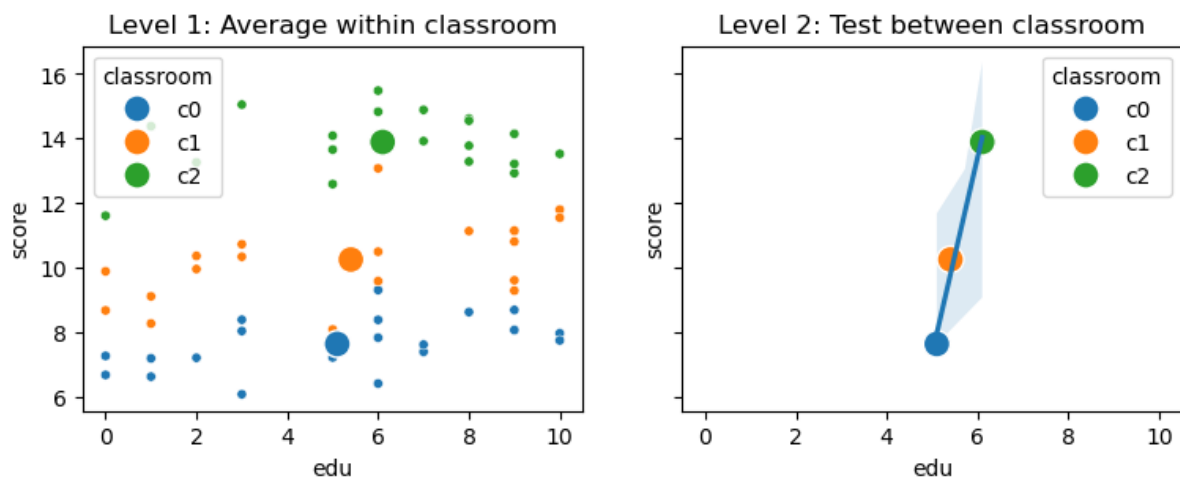
Test for Constraints						
	coef	std err	t	P> t	[0.025	0.975]
c0	6.0734	0.810	7.498	0.084	-4.219	16.366
MSE=0.346						

Plot

```
aggregate = aggregate.reset_index()
fig, axes = plt.subplots(1, 2, figsize=(9, 3), sharex=True, sharey=True)
sns.scatterplot(x='edu', y='score', hue='classroom',
                data=df, ax=axes[0], s=20, legend=False)
sns.scatterplot(x='edu', y='score', hue='classroom',
                data=aggregate, ax=axes[0], s=150)
axes[0].set_title("Level 1: Average within classroom")

sns.regplot(x="edu", y="score", data=aggregate, ax=axes[1])
sns.scatterplot(x='edu', y='score', hue='classroom',
                data=aggregate, ax=axes[1], s=150)
axes[1].set_title("Level 2: Test between classroom")
```

```
Text(0.5, 1.0, 'Level 2: Test between classroom')
```



Hierarchical/multilevel modeling

Another approach to hierarchical data is analyzing data from one unit at a time. Thus, we run three separate linear regressions - one for each classroom in the sample leading to three estimated parameters of the score vs edu association. Then the parameters are tested across the classrooms:

1. Run three separate linear regressions - one for each classroom

$$y_{ij} = \beta_{0j} + \beta_{1j}x_{ij} + \varepsilon_{ij}, \text{ for } j \in \{1, 2, 3\}$$

The general R formula is: $y \sim x$ which in this case is $\text{score} \sim \text{edu}$ within classrooms.

2. Test across the classrooms if is the mean $\beta_{1j} = \beta_0 \neq 0$:

$$\beta_{1j} = \beta_0 + \varepsilon_j$$

The general R formula is: $y \sim 1$ which in this case is $\text{beta\_edu} \sim 1$.

This model is:

- **Correct** because the individual estimated parameters are independent.
- **sensitive** since it allows to model different slopes for each classroom (see fixed interaction or random slope below). But it is but **not optimally designed** since there are many models, and each one does not take advantage of the information in data from other classroom. This can also make the results “noisy” in that the estimates from each model are not based on very much data

```
# Level 1 model within classes
x, y, group = 'edu', 'score', 'classroom'

lv1 = [[group_lab, smf.ols('%s ~ %s' % (y, x), group_df).fit().params[x]]
        for group_lab, group_df in df.groupby(group)]

lv1 = pd.DataFrame(lv1, columns=[group, 'beta'])
print(lv1)

# Level 2 model test beta_edu != 0
lm_hm = smf.ols('beta ~ 1', lv1).fit()
print(lm_hm.t_test('Intercept'))
print("MSE=%.3f" % lm_hm.mse_resid)

results.loc[len(results)] = ["Hierarchical"] + \
    list(rmse_coef_tstat_pval(mod=lm_hm, var='Intercept'))
```

classroom	beta
0	c0 0.129084
1	c1 0.177567
2	c2 0.055772

Test for Constraints						
	coef	std err	t	P> t	[0.025	0.975]
c0	0.1208	0.035	3.412	0.076	-0.032	0.273

MSE=0.004

Plot

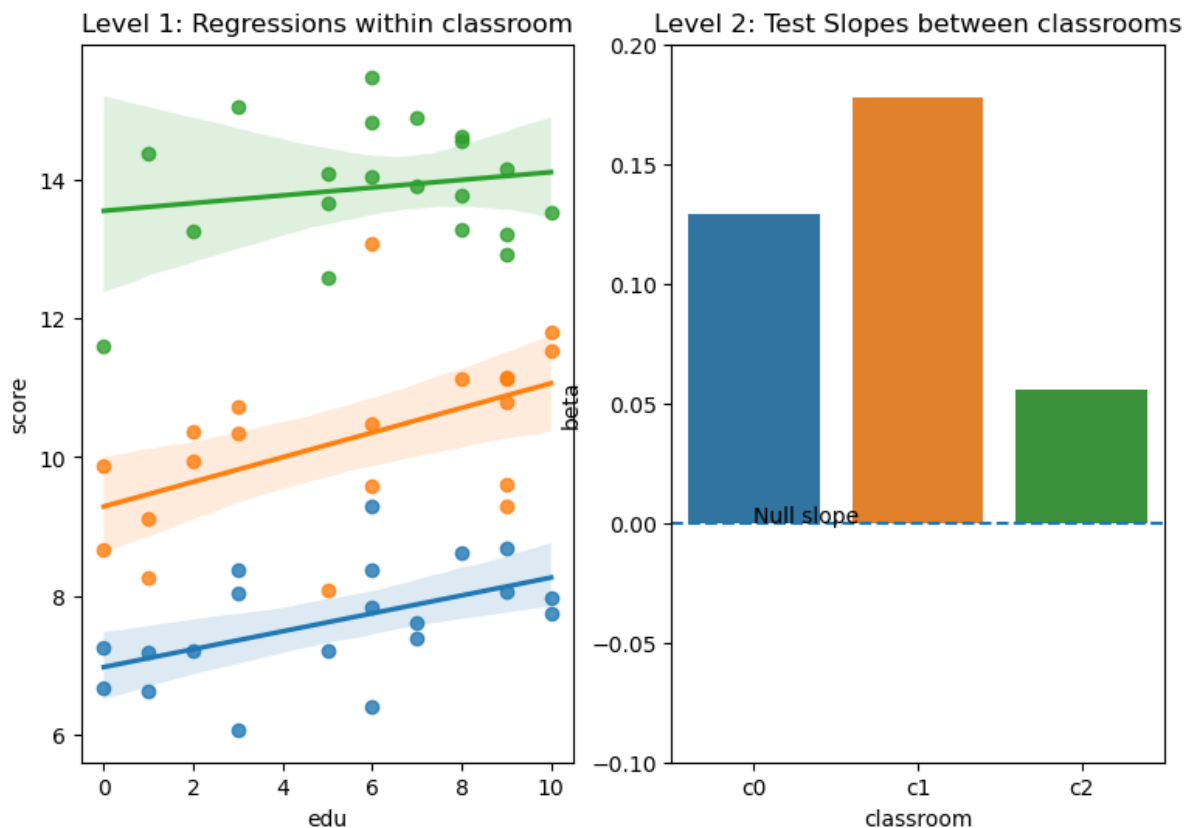
```

fig, axes = plt.subplots(1, 2, figsize=(9, 6))
for group_lab, group_df in df.groupby(group):
    sns.regplot(x=x, y=y, data=group_df, ax=axes[0])

axes[0].set_title("Level 1: Regressions within %s" % group)

_ = sns.barplot(x=group, y="beta", hue=group, data=lv1, ax=axes[1])
axes[1].axhline(0, ls='--')
axes[1].text(0, 0, "Null slope")
axes[1].set_ylim(-.1, 0.2)
_ = axes[1].set_title("Level 2: Test Slopes between classrooms")

```



Model the classroom random intercept: linear mixed model

Linear mixed models (also called multilevel models) can be thought of as a trade off between these two alternatives. The individual regressions has many estimates and lots of data, but is noisy. The aggregate is less noisy, but may lose important differences by averaging all samples within each classroom. LMMs are somewhere in between.

Model the classroom $z = \text{classroom}$ (as a random effect). For each individual i and each classroom j the model is:

$$y_{ij} = \beta_0 + \beta_1 x_{ij} + u_j z_{ij} + \varepsilon_{ij},$$

where, u_j is a **random intercept** following a normal distribution associated with classroom j .

The general R formula is: $y \sim x + (1|z)$ which in this case it is $\text{score} \sim \text{edu} + (1|\text{classroom})$. For python statmodels, the grouping factor |classroom is omitted and provided as groups parameter.

```
lmm_inter = smf.mixedlm("score ~ edu", df, groups=df["classroom"],
                        re_formula="~1").fit()
# But since the default use a random intercept for each group, the following
# formula would have provide the same result:
# lmm_inter = smf.mixedlm("score ~ edu", df, groups=df["classroom"]).fit()
print(lmm_inter.summary())

results.loc[len(results)] = ["LMM-Inter"] + \
    list(rmse_coef_tstat_pval(mod=lmm_inter, var='edu'))
```

Mixed Linear Model Regression Results

```
=====
Model:                MixedLM Dependent Variable: score
No. Observations:    60      Method:                REML
No. Groups:          3      Scale:                 0.8693
Min. group size:     20     Log-Likelihood:      -88.8676
Max. group size:     20     Converged:         Yes
Mean group size:     20.0

-----
                Coef. Std.Err.   z    P>|z| [0.025 0.975]
-----
Intercept      9.865      1.789  5.514  0.000   6.359 13.372
edu             0.131      0.038  3.453  0.001   0.057  0.206
Group Var      9.427     10.337

=====
```

Explore model

```
print("Fixed effect:")
print(lmm_inter.params)

print("Random effect:")
print(lmm_inter.random_effects)

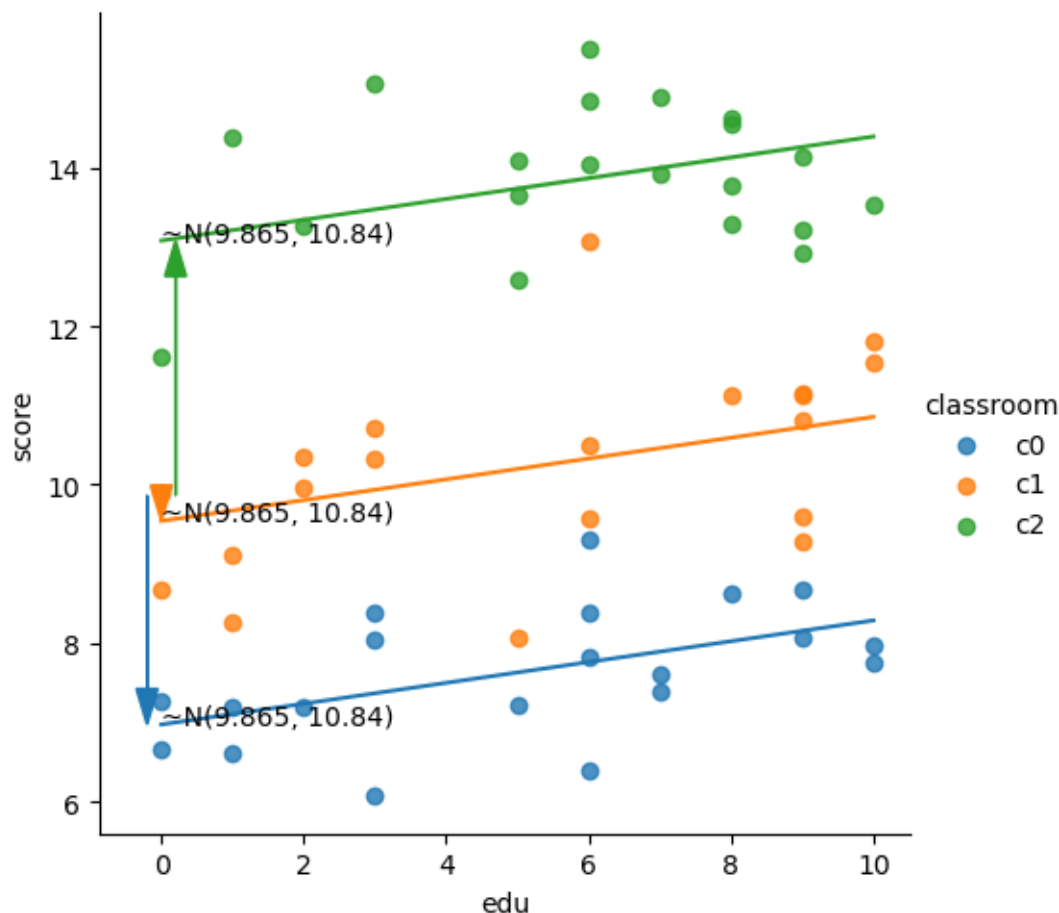
intercept = lmm_inter.params['Intercept']
var = lmm_inter.params["Group Var"]
```

```
Fixed effect:
Intercept      9.865327
edu             0.131193
Group Var     10.844222
dtype: float64
Random effect:
{'c0': Group   -2.889009
dtype: float64, 'c1': Group   -0.323129
dtype: float64, 'c2': Group    3.212138
dtype: float64}
```

Plot


```
plot_lmm_oneslope_randintercept(x='edu', y='score',
                                group='classroom', df=df, model=lmm_inter)
```

```
/home/ed203246/git/pystatsml/statistics/lmm/stat_lmm_utils.py:122: FutureWarning:
↳Series.__getitem__ treating keys as positions is deprecated. In a future
↳version, integer keys will always be treated as labels (consistent with
↳DataFrame behavior). To access a value by position, use ser.iloc[pos]
  group_offset = model.random_effects[group_lab][0]
/home/ed203246/git/pystatsml/statistics/lmm/stat_lmm_utils.py:122: FutureWarning:
↳Series.__getitem__ treating keys as positions is deprecated. In a future
↳version, integer keys will always be treated as labels (consistent with
↳DataFrame behavior). To access a value by position, use ser.iloc[pos]
  group_offset = model.random_effects[group_lab][0]
/home/ed203246/git/pystatsml/statistics/lmm/stat_lmm_utils.py:122: FutureWarning:
↳Series.__getitem__ treating keys as positions is deprecated. In a future
↳version, integer keys will always be treated as labels (consistent with
↳DataFrame behavior). To access a value by position, use ser.iloc[pos]
  group_offset = model.random_effects[group_lab][0]
```



5.3.3 Random slope

Now suppose that the classroom random effect is not just a vertical shift (random intercept) but that some teachers “compensate” or “amplify” educational disparity. The slope of the linear relation between score and edu for teachers that amplify will be larger. In the contrary, it will be smaller for teachers that compensate.

Model the classroom intercept and slope as a fixed effect: ANCOVA with interactions

1. Model the global association between edu and score: $y_{ij} = \beta_0 + \beta_1 x_{ij}$, in R: `score ~ edu`.
2. Model the classroom $z_j = \text{classroom}$ (as a fixed effect) as a vertical shift (intercept, u_j^1) for each classroom j indicated by z_{ij} : $y_{ij} = u_j^1 z_{ij}$, in R: `score ~ classroom`.
3. Model the classroom (as a fixed effect) specific slope (u_j^α): $y_i = u_j^\alpha x_i z_j$ `score ~ edu:classroom`. The $x_i z_j$ forms 3 new columns with values of x_i for each edu level, ie.: for z_j classroom 1, 2 and 3.
4. Put everything together:

$$y_{ij} = \beta_0 + \beta_1 x_{ij} + u_j^1 z_{ij} + u_j^\alpha z_{ij} x_{ij} + \varepsilon_{ij},$$

in R: `score ~ edu + classroom edu:classroom` or more simply `score ~ edu * classroom` that denotes the full model with the additive contribution of each regressor and all their interactions.

```
ancova_full = smf.ols('score ~ edu + classroom + edu:classroom', df).fit()
# Full model (including interaction) can use this notation:
# ancova_full = smf.ols('score ~ edu * classroom', df).fit()

# print(sm.stats.anova_lm(lm_fx, typ=3))
# print(lm_fx.summary())
print(ancova_full.t_test('edu'))
print("MSE=%.3f" % ancova_full.mse_resid)
results.loc[len(results)] = ["ANCOVA-Full (biased)"] + \
    list(rmse_coef_tstat_pval(mod=ancova_full, var='edu'))
```

Test for Constraints						
	coef	std err	t	P> t	[0.025	0.975]
c0	0.1291	0.065	1.979	0.053	-0.002	0.260
MSE=0.876						

The graphical representation of the model would be the same than the one provided for “Model a classroom intercept as a fixed effect: ANCOVA”. The same slope (associated to edu) with different intercept, depicted as dashed black lines. Moreover we added, as solid lines, the model’s prediction that account different slopes.

```
print("Model parameters:")
print(ancova_full.params)
```

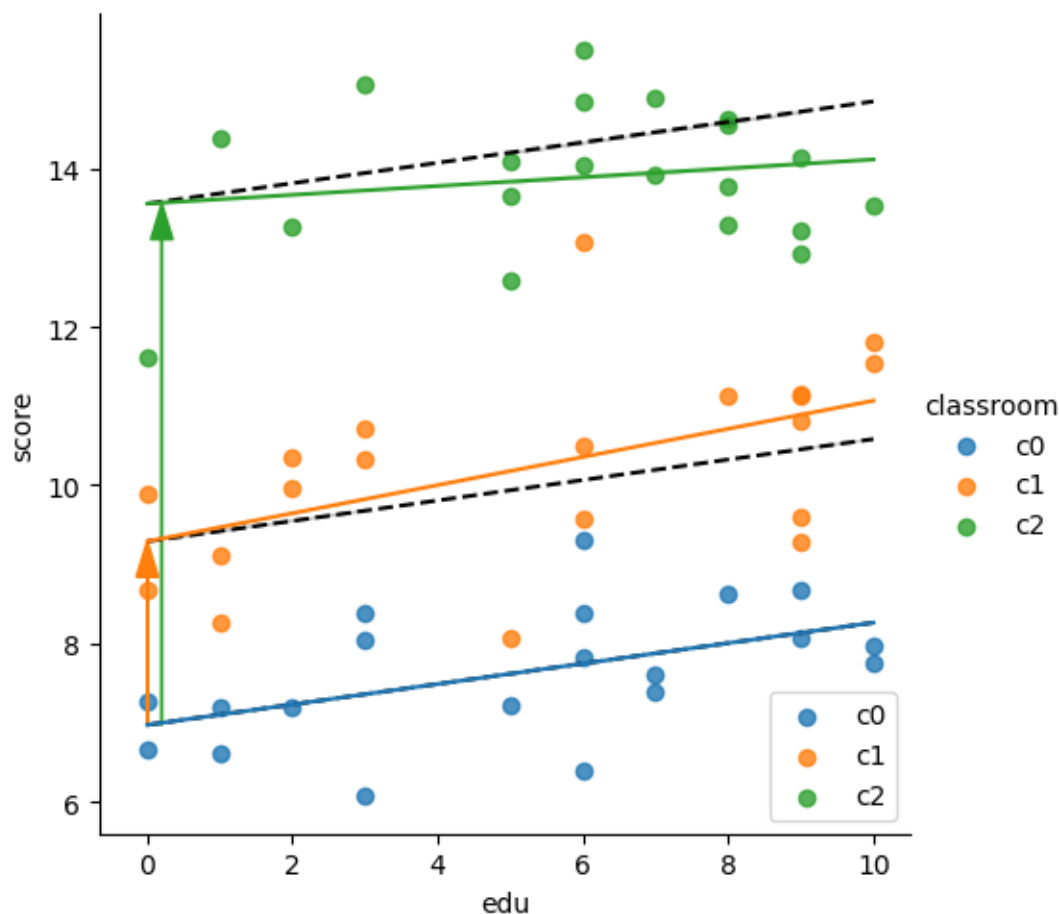
(continues on next page)

(continued from previous page)

```
plot_ancova_fullmodel(x='edu', y='score',
                      group='classroom', df=df, model=ancova_full)
```

Model parameters:

```
Intercept          6.973753
classroom[T.c1]     2.316540
classroom[T.c2]     6.578594
edu                0.129084
edu:classroom[T.c1] 0.048482
edu:classroom[T.c2] -0.073313
dtype: float64
```



Model the classroom random intercept and slope with LMM

The model looks similar to the ANCOVA with interactions:

$$y_{ij} = \beta_0 + \beta_1 x_{ij} + u_j^1 z_{ij} + u_j^\alpha z_{ij} x_{ij} + \varepsilon_{ij},$$

but:

- u_j^1 is a **random intercept** associated with classroom j following the same normal distribution for all classroom, $u_j^1 \sim \mathcal{N}(\mathbf{0}, \sigma^1)$.
- u_j^α is a **random slope** associated with classroom j following the same normal distribution for all classroom, $u_j^\alpha \sim \mathcal{N}(\mathbf{0}, \sigma^\alpha)$.

Note the difference with linear model: the variances parameters (σ^1, σ^α) should be estimated together with fixed effect ($\beta_0 + \beta_1$) and random effect (u^1, u_j^α , one pair of random intercept/slope per classroom). The R notation is: `score ~ edu + (edu | classroom)`. or `score ~ 1 + edu + (1 + edu | classroom)`, remember that intercepts are implicit. In statmodels, the notation is `~1+edu` or `~edu` since the groups is provided by the groups argument.

```
lmm_full = smf.mixedlm("score ~ edu", df, groups=df["classroom"],
                      re_formula="~1+edu").fit()
print(lmm_full.summary())
results.loc[len(results)] = ["LMM-Full (biased)"] + \
    list(rmse_coef_tstat_pval(mod=lmm_full, var='edu'))
```

Mixed Linear Model Regression Results

```
=====
Model:                MixedLM   Dependent Variable:   score
No. Observations:    60         Method:                REML
No. Groups:          3         Scale:              0.8609
Min. group size:     20        Log-Likelihood:     -88.5987
Max. group size:     20        Converged:          Yes
Mean group size:     20.0
=====
```

	Coef.	Std.Err.	z	P> z	[0.025	0.975]
Intercept	9.900	1.912	5.177	0.000	6.152	13.647
edu	0.127	0.046	2.757	0.006	0.037	0.218
Group Var	10.760	12.278				
Group x edu Cov	-0.121	0.318				
edu Var	0.001	0.012				

```
=====
```

```
/home/ed203246/git/pystatsml/.pixi/envs/default/lib/python3.13/site-packages/
↳ statsmodels/base/model.py:607: ConvergenceWarning: Maximum Likelihood_
↳ optimization failed to converge. Check mle_retvals
   warnings.warn("Maximum Likelihood optimization failed to "
/home/ed203246/git/pystatsml/.pixi/envs/default/lib/python3.13/site-packages/
↳ statsmodels/regression/mixed_linear_model.py:2200: ConvergenceWarning: Retrying_
↳ MixedLM optimization with lbfgs
   warnings.warn(
/home/ed203246/git/pystatsml/.pixi/envs/default/lib/python3.13/site-packages/
↳ statsmodels/regression/mixed_linear_model.py:1634: UserWarning: Random effects_
↳ covariance is singular
   warnings.warn(msg)
/home/ed203246/git/pystatsml/.pixi/envs/default/lib/python3.13/site-packages/
↳ statsmodels/regression/mixed_linear_model.py:2237: ConvergenceWarning: The MLE_
↳ may be on the boundary of the parameter space.
   warnings.warn(msg, ConvergenceWarning)
```

The warning results in a singular fit (correlation estimated at 1) caused by too little variance among the random slopes. It indicates that we should consider to remove random slopes.

5.3.4 Conclusion on modeling random effects

```
print(results)
```

	Model	RMSE	Coef	Stat	Pval
0	LM-Global (biased)	2.694785	0.232842	2.139165	0.036643
1	ANCOVA-Inter (biased)	0.932351	0.130717	3.441072	0.001102
2	Aggregation	0.587859	6.073401	7.497672	0.084411
3	Hierarchical	0.061318	0.120808	3.412469	0.076190
4	LMM-Inter	0.916211	0.131193	3.453472	0.000553
5	ANCOVA-Full (biased)	0.935869	0.129084	1.978708	0.052959
6	LMM-Full (biased)	0.911742	0.127270	2.757142	0.005831

Random intercepts

1. LM-Global is wrong (consider residuals to be independent) and has a large error (RMSE, Root Mean Square Error) since it does not adjust for classroom effect.
2. ANCOVA-Inter is “wrong” (consider residuals to be independent) but it has a small error since it adjusts for classroom effect.
3. Aggregation is ok (units average are independent) but it loses a lot of degrees of freedom ($df = 2 = 3 \text{ classroom} - 1 \text{ intercept}$) and a lot of informations.
4. Hierarchical model is ok (unit average are independent) and it has a reasonable error (look at the statistic, not the RMSE).
5. LMM-Inter (with random intercept) is ok (it models residuals non-independence) and it has a small error.
6. ANCOVA-Inter, Hierarchical model and LMM provide similar coefficients for the fixed effect. So if statistical significance is not the key issue, the “biased” ANCOVA is a reasonable choice.
7. Hierarchical and LMM with random intercept are the best options (unbiased and sensitive), with an advantage to LMM.

Random slopes

Modeling individual slopes in both ANCOVA-Full and LMM-Full decreased the statistics, suggesting that the supplementary regressors (one per classroom) do not significantly improve the fit of the model (see errors).

5.3.5 Theory of Linear Mixed Models

If we consider only 6 samples ($i \in \{1, 6\}$, two sample for each classroom $j \in \{c0, c1, c2\}$) and the random intercept model. Stacking the 6 observations, the equation $y_{ij} = \beta_0 + \beta_1 x_{ij} + u_j z_j + \varepsilon_{ij}$ gives :

$$\begin{bmatrix} \text{score} \\ 7.2 \\ 7.9 \\ 9.1 \\ 11.1 \\ 14.6 \\ 14.0 \end{bmatrix} = \begin{bmatrix} \text{Inter} & \text{Edu} \\ 1 & 2 \\ 1 & 10 \\ 1 & 1 \\ 1 & 9 \\ 1 & 8 \\ 1 & 5 \end{bmatrix} \begin{bmatrix} \text{Fix} \\ \beta_0 \\ \beta_1 \end{bmatrix} + \begin{bmatrix} c1 & c2 & c3 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \text{Rand} \\ u_1 \\ u_2 \\ u_3 \end{bmatrix} + \begin{bmatrix} \text{Err} \\ \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \\ \epsilon_6 \end{bmatrix}$$

where $\mathbf{u}_1 = u_1, u_2, u_3$ are the 3 parameters associated with the 3 level of the single random factor classroom.

This can be re-written in a more general form as:

$$\mathbf{y} = \mathbf{X}\beta + \mathbf{Z}\mathbf{u} + \varepsilon,$$

where: - $\mathbf{y}$ is the $N \times 1$ vector of the N observations. - $\mathbf{X}$ is the $N \times P$ design matrix, which represents the known values of the P covariates for the N observations. - β is a $P \times 1$ vector unknown regression coefficients (or fixed-effect parameters) associated with the P covariates. - ε is a $N \times 1$ vector of residuals $\varepsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{R})$, where $\mathbf{R}$ is a $N \times N$ matrix. - $\mathbf{Z}$ is a $N \times Q$ design matrix of random factors and covariates. In an LMM in which only the intercepts are assumed to vary randomly from Q units, the $\mathbf{Z}$ matrix would simply be Q columns of indicators 1 (if subject belong to unit q) or 0 otherwise. - $\mathbf{u}$ is a $Q \times 1$ vector of Q random effects associated with the Q covariates in the $\mathbf{Z}$ matrix. Note that one random factor of 3 levels will be coded by 3 coefficients in $\mathbf{u}$ and 3 columns $\mathbf{Z}$. $\mathbf{u} \sim \mathcal{N}(\mathbf{0}, \mathbf{D})$ where $\mathbf{D}$ plays a central role of the covariance structures associated with the mixed effect.

Covariance structures of the residuals covariance matrix: $\mathbf{R}$

Many different covariance structures are possible for the $\mathbf{R}$ matrix. The simplest covariance matrix for $\mathbf{R}$ is the diagonal structure, in which the residuals associated with observations on the same subject are assumed to be uncorrelated and to have equal variance: $\mathbf{R} = \sigma^2 \mathbf{I}_N$. Note that in this case, the correlation between observation within unit stem from mixed effects, and will be encoded in the $\mathbf{D}$ below. However, other model exists: popular models are the compound symmetry and first-order autoregressive structure, denoted by AR(1).

Covariance structures associated with the random effect

Many different covariance structures are possible for the $\mathbf{D}$ matrix. The usual practice associate a single variance parameter (a scalar, σ_k) to each random-effects factor k (eg. classroom). Hence $\mathbf{D}$ is simply parametrized by a set of scalars $\sigma_k, k \in \{1, K\}$ for the K random factors such the sum of levels of the K factors equals Q . In our case $K = 1$ with 3 levels ($Q = 3$), thus $\mathbf{D} = \sigma_k \mathbf{I}_Q$. Factors k define k **variance components** whose parameters σ_k should be estimated addition to the variance of the model errors σ . The σ_k and σ will define the overall covariance structure: $\mathbf{V}$, as define below.

In this model, the effect of a particular level (eg. classroom 0 c0) of a random factor is supposed to be sampled from a normal distribution of variance σ_k . This is a crucial aspect of LMM which is related to ℓ_2 -regularization or Bayes Baussian prior. Indeed, the estimator of associated to each level u_i of a random effect is shrinked toward 0 since $u_i \sim \mathcal{N}(0, \sigma_k)$. Thus it tends to be smaller than the estimated effects would be if they were computed by treating a random factor as if it were fixed.

Overall covariance structure as variance components $\mathbf{V}$

The overall covariance structure can be obtained by:

$$\mathbf{V} = \sum_k \sigma_k \mathbf{Z}\mathbf{Z}' + \mathbf{R}.$$

The $\sum_k \sigma_k \mathbf{Z}\mathbf{Z}'$ define the $N \times N$ variance structure, using k variance components, modeling the

non-independence between the observations. In our case with only one component we get:

$$\mathbf{V} = \begin{bmatrix} \sigma_k & \sigma_k & 0 & 0 & 0 & 0 \\ \sigma_k & \sigma_k & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_k & \sigma_k & 0 & 0 \\ 0 & 0 & \sigma_k & \sigma_k & 0 & 0 \\ 0 & 0 & 0 & 0 & \sigma_k & \sigma_k \\ 0 & 0 & 0 & 0 & \sigma_k & \sigma_k \end{bmatrix} + \begin{bmatrix} \sigma & 0 & 0 & 0 & 0 & 0 \\ 0 & \sigma & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma & 0 & 0 & 0 \\ 0 & 0 & 0 & \sigma & 0 & 0 \\ 0 & 0 & 0 & 0 & \sigma & 0 \\ 0 & 0 & 0 & 0 & 0 & \sigma \end{bmatrix}$$

$$= \begin{bmatrix} \sigma_k + \sigma & \sigma_k & 0 & 0 & 0 & 0 \\ \sigma_k & \sigma_k + \sigma & 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_k + \sigma & \sigma_k & 0 & 0 \\ 0 & 0 & \sigma_k & \sigma_k + \sigma & 0 & 0 \\ 0 & 0 & 0 & 0 & \sigma_k + \sigma & \sigma_k \\ 0 & 0 & 0 & 0 & \sigma_k & \sigma_k + \sigma \end{bmatrix}$$

The model to be minimized

Here σ_k and σ are called variance components of the model. Solving the problem consist in the estimation the fixed effect β and the parameters σ, σ_k of the variance-covariance structure. This is obtained by minizing the The likelihood of the sample:

$$l(\mathbf{y}, \beta, \sigma, \sigma_k) = \frac{1}{2\pi^{n/2} \det(\mathbf{V})^{1/2}} \exp -\frac{1}{2}(\mathbf{y} - \mathbf{X}\beta)\mathbf{V}^{-1}(\mathbf{y} - \mathbf{X}\beta)$$

LMM introduces the variance-covariance matrix $\mathbf{V}$ to reweight the residuals according to the non-independence between observations. If $\mathbf{V}$ is known, of. The optimal value of be can be obtained analytically using generalized least squares (GLS, minimisation of mean squared error associated with Mahalanobis metric):

$$\hat{\beta} = \mathbf{X}'\hat{\mathbf{V}}^{-1}\mathbf{X}^{-1}\mathbf{X}'\hat{\mathbf{V}}^{-1}\mathbf{y}$$

In the general case, $\mathbf{V}$ is unknown, therefore iterative solvers should be use to estimate the fixed effect β and the parameters $(\sigma, \sigma_k, \dots)$ of variance-covariance matrix $\mathbf{V}$. The ML Maximum Likelihood estimates provide biased solution for $\mathbf{V}$ because they do not take into account the loss of degrees of freedom that results from estimating the fixed-effect parameters in β . For this reason, REML (restricted (or residual, or reduced) maximum likelihood) is often preferred to ML estimation.

Tests for Fixed-Effect Parameters

Quoted from [Brady et al. 2014]: “The approximate methods that apply to both t-tests and F-tests take into account the presence of random effects and correlated residuals in an LMM. Several of these approximate methods (e.g., the **Satterthwaite** method, or the “between-within” method) involve different choices for the degrees of freedom used in” the approximate t-tests and F-tests”.

5.3.6 Checking model assumptions (Diagnostics)

Residuals plotted against predicted values represents a random pattern or not.

These residual vs. fitted plots are used to verify model assumptions and to detect outliers and potentially influential observations.

5.3.7 References

- Brady et al. 2014: Brady T. West, Kathleen B. Welch, Andrzej T. Galecki, [Linear Mixed Models: A Practical Guide Using Statistical Software \(2nd Edition\)](#), 2014
- Bruin 2006: [Introduction to Linear Mixed Models](#), UCLA, Statistical Consulting Group.
- [Statsmodel: Linear Mixed Effects Models](#)
- [Comparing R lmer to statsmodels MixedLM](#)
- [Statsmoels: Variance Component Analysis with nested groups](#)

5.4 Multivariate Statistics

Multivariate statistics includes all statistical techniques for analyzing samples made of two or more variables. The data set (a $N \times P$ matrix $\mathbf{X}$) is a collection of N points (or observations) with P variables, i.e., in a P -dimensional space:

$$\mathbf{X} = \begin{bmatrix} x_{11} & \cdots & x_{1j} & \cdots & x_{1P} \\ \vdots & & \vdots & & \vdots \\ x_{i1} & \cdots & x_{ij} & \cdots & x_{iP} \\ \vdots & & \vdots & & \vdots \\ x_{N1} & \cdots & x_{Nj} & \cdots & x_{NP} \end{bmatrix} = \begin{bmatrix} x_{11} & \cdots & x_{1P} \\ \vdots & & \vdots \\ & \mathbf{X} & \\ \vdots & & \vdots \\ x_{N1} & \cdots & x_{NP} \end{bmatrix}_{N \times P}.$$

- Each row i is a P -dimensional vector of the coordinates of the i 'th observation or point.
- Each column j is a N -dimensional vector of values of the points for the j 'th variable.

5.4.1 Basic Linear Algebra

The **Euclidean norm** of a vector $\mathbf{x} \in \mathbb{R}^P$ is denoted

$$\|\mathbf{x}\|_2 = \sqrt{\sum_i^P x_i^2}$$

The **Euclidean distance** between two point $\mathbf{x}, \mathbf{y} \in \mathbb{R}^P$ is

$$\|\mathbf{x} - \mathbf{y}\|_2 = \sqrt{\sum_i^P (x_i - y_i)^2}$$

The **dot product**, denoted “.” of two P -dimensional vectors $\mathbf{x} = [x_1, x_2, \dots, x_P]$ and $\mathbf{y} = [y_1, y_2, \dots, y_P]$ is defined as

$$\mathbf{x} \cdot \mathbf{y} = \mathbf{x}^T \mathbf{y} = \sum_i x_i y_i = \begin{bmatrix} x_1 & \cdots & \mathbf{x}^T & \cdots & x_P \end{bmatrix} \begin{bmatrix} y_1 \\ \vdots \\ \mathbf{y} \\ \vdots \\ y_P \end{bmatrix}.$$

Note that the Euclidean norm of a vector is the square root of the dot product of the vector with itself:

$$\|\mathbf{x}\|_2 = \sqrt{\mathbf{x} \cdot \mathbf{x}}.$$

Geometric interpretation: In Euclidean space, a Euclidean vector is a geometrical object that possesses both a norm (magnitude) and a direction. A vector can be pictured as an arrow. Its magnitude is its length, and its direction is the direction that the arrow points. The norm of a vector $\mathbf{x}$ is denoted by $\|\mathbf{x}\|_2$. The dot product of two Euclidean vectors $\mathbf{x}$ and $\mathbf{y}$ is defined by

$$\mathbf{x} \cdot \mathbf{y} = \|\mathbf{x}\|_2 \|\mathbf{y}\|_2 \cos \theta,$$

where θ is the angle between $\mathbf{x}$ and $\mathbf{y}$.

In particular, if $\mathbf{x}$ and $\mathbf{y}$ are orthogonal, then the angle between them is 90° and $\mathbf{x} \cdot \mathbf{y} = 0$. At the other extreme, if they are codirectional, then the angle between them is 0° and $\mathbf{x} \cdot \mathbf{y} = \|\mathbf{x}\|_2 \|\mathbf{y}\|_2$.

The (scalar) projection of a vector $\mathbf{x}$ (a point) in the direction of $\mathbf{v}$ is given by

$$\begin{aligned} x_v &= \|\mathbf{x}\|_2 \cos \theta, \\ &= \frac{\mathbf{x} \cdot \mathbf{v}}{\|\mathbf{v}\|_2}, \end{aligned}$$

where θ is the angle between $\mathbf{x}$ and $\mathbf{y}$. Note that x_v is a scalar measuring the length of x projected on v .

When we want to project on direction $\mathbf{v}$, we only care about its direction. Therefore vector $\mathbf{v}$ can be normalized to $\|\mathbf{v}\|_2 = 1$ dividing by the vector norm (without affecting its direction.). With $\|\mathbf{v}\|_2 = 1$ the projection of any point $\mathbf{x}$ toward a direction $\mathbf{v}$ becomes an simple dot product:

$$x_v = \mathbf{x} \cdot \mathbf{v}$$

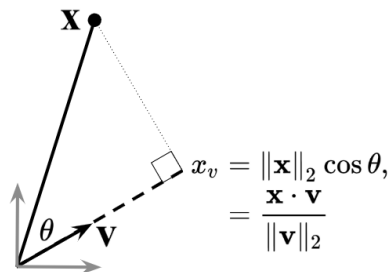


Fig. 4: Projection of point x on vector v

```
import numpy as np
import scipy
import pandas as pd

# Plot
import matplotlib.pyplot as plt
from matplotlib import cm # color map
import seaborn as sns
import pystatsml.plot_utils

# Plot parameters
plt.style.use('seaborn-v0_8-whitegrid')
```

(continues on next page)

(continued from previous page)

```
fig_w, fig_h = plt.rcParams.get('figure.figsize')
plt.rcParams['figure.figsize'] = (fig_w, fig_h * 1.)
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
```

```
import numpy as np
np.random.seed(42)

a = np.random.randn(10)
b = np.random.randn(10)

print(np.dot(a, b))
```

```
-4.085788532659923
```

5.4.2 Mean vector

The mean ($P \times 1$) column-vector μ whose estimator is

$$\bar{\mathbf{x}} = \frac{1}{N} \sum_{i=1}^N \mathbf{x}_i = \frac{1}{N} \sum_{i=1}^N \begin{bmatrix} x_{i1} \\ \vdots \\ x_{ij} \\ \vdots \\ x_{iP} \end{bmatrix} = \begin{bmatrix} \bar{x}_1 \\ \vdots \\ \bar{x}_j \\ \vdots \\ \bar{x}_P \end{bmatrix}.$$

5.4.3 Covariance matrix

- The covariance matrix $\Sigma_{\mathbf{X}\mathbf{X}}$ is a **symmetric** positive semi-definite matrix whose element in the j, k position is the covariance between the j^{th} and k^{th} elements of a random vector i.e. the j^{th} and k^{th} columns of $\mathbf{X}$.
- The covariance matrix generalizes the notion of covariance to multiple dimensions.
- The covariance matrix describe the shape of the sample distribution around the mean assuming an elliptical distribution:

$$\Sigma_{\mathbf{X}\mathbf{X}} = E(\mathbf{X} - E(\mathbf{X}))^T E(\mathbf{X} - E(\mathbf{X})),$$

whose estimator $\mathbf{S}_{\mathbf{X}\mathbf{X}}$ is a $P \times P$ matrix given by

$$\mathbf{S}_{\mathbf{X}\mathbf{X}} = \frac{1}{N-1} (\mathbf{X} - \mathbf{1}\bar{\mathbf{x}}^T)^T (\mathbf{X} - \mathbf{1}\bar{\mathbf{x}}^T).$$

If we assume that $\mathbf{X}$ is centered, i.e. $\mathbf{X}$ is replaced by $\mathbf{X} - \mathbf{1}\bar{\mathbf{x}}^T$ then the estimator is

$$\begin{aligned} \mathbf{S}_{\mathbf{X}\mathbf{X}} &= \frac{1}{N-1} \mathbf{X}^T \mathbf{X} \\ &= \frac{1}{N-1} \begin{bmatrix} x_{11} & \cdots & x_{N1} \\ x_{1j} & \cdots & x_{Nj} \\ \vdots & & \vdots \\ x_{1P} & \cdots & x_{NP} \end{bmatrix} \begin{bmatrix} x_{11} & \cdots & x_{1k} & x_{1P} \\ \vdots & & \vdots & \vdots \\ x_{N1} & \cdots & x_{Nk} & x_{NP} \end{bmatrix} \\ &= \begin{bmatrix} s_1 & \cdots & s_{1k} & s_{1P} \\ & \ddots & s_{jk} & \vdots \\ & & s_k & s_{kP} \\ & & & s_P \end{bmatrix}, \end{aligned}$$

where

$$s_{jk} = s_{kj} = \frac{1}{N-1} \mathbf{x}_j^T \mathbf{x}_k = \frac{1}{N-1} \sum_{i=1}^N x_{ij} x_{ik}$$

is an estimator of the covariance between the j^{th} and k^{th} variables.

```
np.random.seed(42)
colors = sns.color_palette()

n_samples, n_features = 100, 2

mean, Cov, X = [None] * 4, [None] * 4, [None] * 4
mean[0] = np.array([-2.5, 2.5])
Cov[0] = np.array([[1, 0],
                  [0, 1]])

mean[1] = np.array([2.5, 2.5])
Cov[1] = np.array([[1, .5],
                  [.5, 1]])

mean[2] = np.array([-2.5, -2.5])
Cov[2] = np.array([[1, .9],
                  [.9, 1]])

mean[3] = np.array([2.5, -2.5])
Cov[3] = np.array([[1, -.9],
                  [-.9, 1]])

# Generate dataset
for i in range(len(mean)):
    X[i] = np.random.multivariate_normal(mean[i], Cov[i], n_samples)

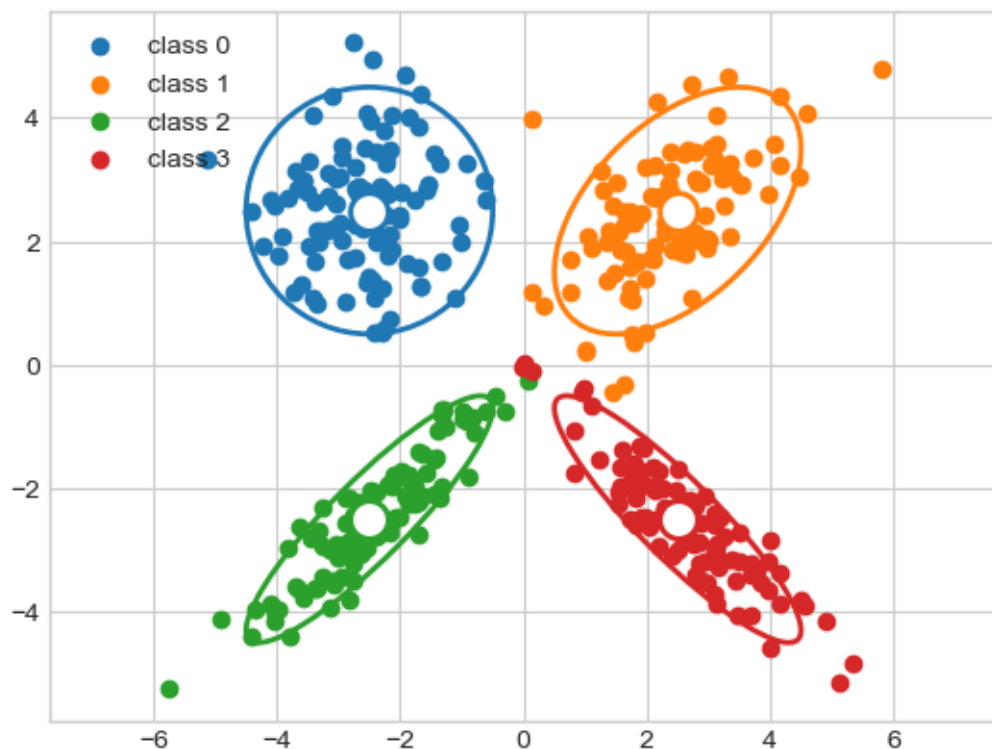
# Plot
for i in range(len(mean)):
    # Points
    plt.scatter(X[i][:, 0], X[i][:, 1], color=colors[i], label="class %i" % i)
    # Means
```

(continues on next page)

(continued from previous page)

```
plt.scatter(mean[i][0], mean[i][1], marker="o", s=200, facecolors='w',
            edgecolors=colors[i], linewidth=2)
# Ellipses representing the covariance matrices
pystatsml.plot_utils.plot_cov_ellipse(Cov[i], pos=mean[i], facecolor='none',
                                     linewidth=2, edgecolor=colors[i])

plt.axis('equal')
_ = plt.legend(loc='upper left')
```



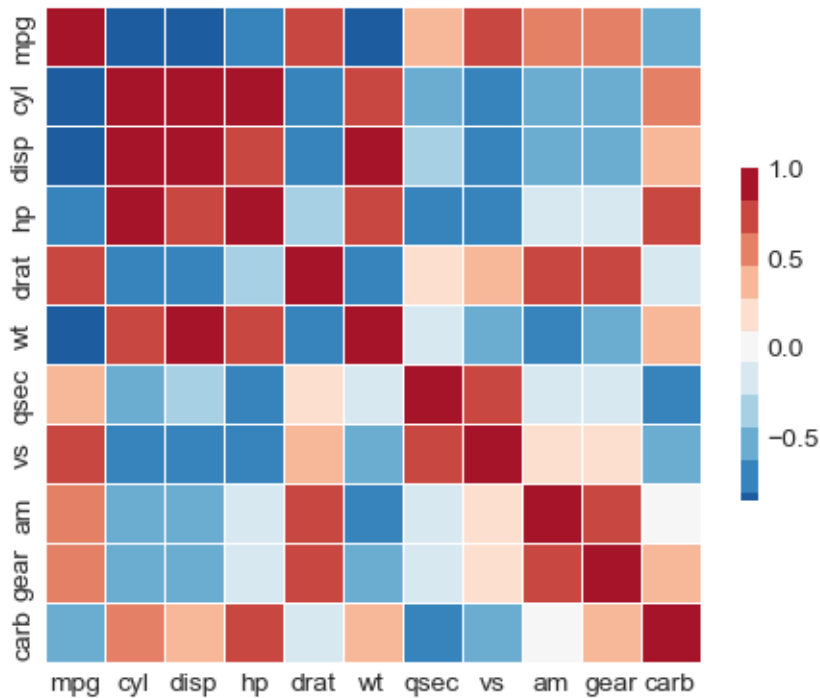
5.4.4 Correlation matrix

```
url = 'https://raw.githubusercontent.com/plotly/datasets/master/mtcars.csv'
df = pd.read_csv(url)
df = df.drop('manufacturer', axis=1)

# Compute the correlation matrix
corr = df.corr()

# Generate a mask for the upper triangle
mask = np.zeros_like(corr, dtype=np.bool)
mask[np.triu_indices_from(mask)] = True

f, ax = plt.subplots(figsize=(5.5, 4.5))
cmap = sns.color_palette("RdBu_r", 11)
# Draw the heatmap with the mask and correct aspect ratio
_ = sns.heatmap(corr, mask=None, cmap=cmap, vmax=1, center=0,
               square=True, linewidths=.5, cbar_kws={"shrink": .5})
```



Re-order correlation matrix using AgglomerativeClustering

```
# convert correlation to distances
from sklearn.cluster import AgglomerativeClustering
d = 2 * (1 - np.abs(corr))

clustering = AgglomerativeClustering(
    n_clusters=3, linkage='single', metric="precomputed").fit(d)
lab = 0

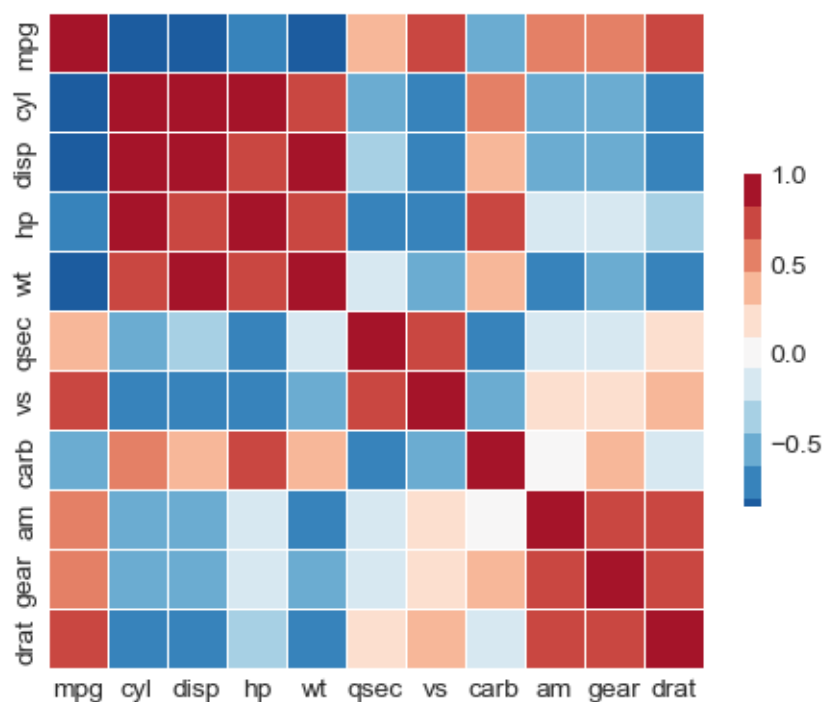
clusters = [list(corr.columns[clustering.labels_ == lab])
             for lab in set(clustering.labels_)]
print(clusters)

reordered = np.concatenate(clusters)

R = corr.loc[reordered, reordered]

f, ax = plt.subplots(figsize=(5.5, 4.5))
# Draw the heatmap with the mask and correct aspect ratio
_ = sns.heatmap(R, mask=None, cmap=cmap, vmax=1, center=0,
               square=True, linewidths=.5, cbar_kws={"shrink": .5})
```

```
[[ 'mpg', 'cyl', 'disp', 'hp', 'wt', 'qsec', 'vs', 'carb'], ['am', 'gear'], ['drat
↪ ']]
```



5.4.5 Precision matrix

In statistics, precision is the reciprocal of the variance, and the precision matrix is the matrix inverse of the covariance matrix.

It is related to **partial correlations** that measures the degree of association between two variables, while controlling the effect of other variables.

```
import numpy as np

Cov = np.array([[1.0, 0.9, 0.9, 0.0, 0.0, 0.0],
                [0.9, 1.0, 0.9, 0.0, 0.0, 0.0],
                [0.9, 0.9, 1.0, 0.0, 0.0, 0.0],
                [0.0, 0.0, 0.0, 1.0, 0.9, 0.0],
                [0.0, 0.0, 0.0, 0.9, 1.0, 0.0],
                [0.0, 0.0, 0.0, 0.0, 0.0, 1.0]])

print("# Precision matrix:")
Prec = np.linalg.inv(Cov)
print(Prec.round(2))

print("# Partial correlations:")
Pcor = np.zeros(Prec.shape)
Pcor[:] = np.nan

for i, j in zip(*np.triu_indices_from(Prec, 1)):
    Pcor[i, j] = - Prec[i, j] / np.sqrt(Prec[i, i] * Prec[j, j])

print(Pcor.round(2))
```

```
# Precision matrix:
[[ 6.79 -3.21 -3.21  0.    0.    0. ]
 [-3.21  6.79 -3.21  0.    0.    0. ]
 [-3.21 -3.21  6.79  0.    0.    0. ]
 [ 0.    0.    0.    5.26 -4.74  0. ]
 [ 0.    0.    0.   -4.74  5.26  0. ]
 [ 0.    0.    0.    0.    0.    1. ]]

# Partial correlations:
[[ nan  0.47  0.47 -0.   -0.   -0. ]
 [ nan  nan  0.47 -0.   -0.   -0. ]
 [ nan  nan  nan -0.   -0.   -0. ]
 [ nan  nan  nan  nan  0.9  -0. ]
 [ nan  nan  nan  nan  nan -0. ]
 [ nan  nan  nan  nan  nan  nan]]
```

5.4.6 Mahalanobis distance

- The Mahalanobis distance is a measure of the distance between two points $\mathbf{x}$ and μ where the dispersion (i.e. the covariance structure) of the samples is taken into account.
- The dispersion is considered through covariance matrix.

This is formally expressed as

$$D_M(\mathbf{x}, \mu) = \sqrt{(\mathbf{x} - \mu)^T \Sigma^{-1} (\mathbf{x} - \mu)}.$$

Intuitions

- Distances along the principal directions of dispersion are contracted since they correspond to likely dispersion of points.
- Distances orthogonal to the principal directions of dispersion are dilated since they correspond to unlikely dispersion of points.

For example, $D_M(\mathbf{1}) = \sqrt{\mathbf{1}^T \Sigma^{-1} \mathbf{1}}$:

```
ones = np.ones(Cov.shape[0])
d_euc = np.sqrt(np.dot(ones, ones))
d_mah = np.sqrt(np.dot(np.dot(ones, Prec), ones))

print("Euclidean norm of ones=%.2f. Mahalanobis norm of ones=%.2f" %
      (d_euc, d_mah))
```

```
Euclidean norm of ones=2.45. Mahalanobis norm of ones=1.77
```

The first dot product that distances along the principal directions of dispersion are contracted:

```
print(np.dot(ones, Prec))
```

```
[0.35714286 0.35714286 0.35714286 0.52631579 0.52631579 1.    ]
```

```

import numpy as np
import scipy
import matplotlib.pyplot as plt
import seaborn as sns
import pystatsml.plot_utils
%matplotlib inline
np.random.seed(40)
colors = sns.color_palette()

mean = np.array([0, 0])
Cov = np.array([[1, .8],
                [.8, 1]])
samples = np.random.multivariate_normal(mean, Cov, 100)
x1 = np.array([0, 2])
x2 = np.array([2, 2])

plt.scatter(samples[:, 0], samples[:, 1], color=colors[0])
plt.scatter(mean[0], mean[1], color=colors[0], s=200, label="mean")
plt.scatter(x1[0], x1[1], color=colors[1], s=200, label="x1")
plt.scatter(x2[0], x2[1], color=colors[2], s=200, label="x2")

# plot covariance ellipsis
pystatsml.plot_utils.plot_cov_ellipse(Cov, pos=mean, facecolor='none',
                                      linewidth=2, edgecolor=colors[0])

# Compute distances
d2_m_x1 = scipy.spatial.distance.euclidean(mean, x1)
d2_m_x2 = scipy.spatial.distance.euclidean(mean, x2)

Covi = scipy.linalg.inv(Cov)
dm_m_x1 = scipy.spatial.distance.mahalanobis(mean, x1, Covi)
dm_m_x2 = scipy.spatial.distance.mahalanobis(mean, x2, Covi)

# Plot distances
vm_x1 = (x1 - mean) / d2_m_x1
vm_x2 = (x2 - mean) / d2_m_x2
jitter = .1
plt.plot([mean[0] - jitter, d2_m_x1 * vm_x1[0] - jitter],
         [mean[1], d2_m_x1 * vm_x1[1]], color='k')
plt.plot([mean[0] - jitter, d2_m_x2 * vm_x2[0] - jitter],
         [mean[1], d2_m_x2 * vm_x2[1]], color='k')

plt.plot([mean[0] + jitter, dm_m_x1 * vm_x1[0] + jitter],
         [mean[1], dm_m_x1 * vm_x1[1]], color='r')
plt.plot([mean[0] + jitter, dm_m_x2 * vm_x2[0] + jitter],
         [mean[1], dm_m_x2 * vm_x2[1]], color='r')

plt.legend(loc='lower right')
plt.text(-6.1, 3,
        'Euclidian:  d(m, x1) = %.1f<d(m, x2) = %.1f' % (d2_m_x1, d2_m_x2),
        color='k')

```

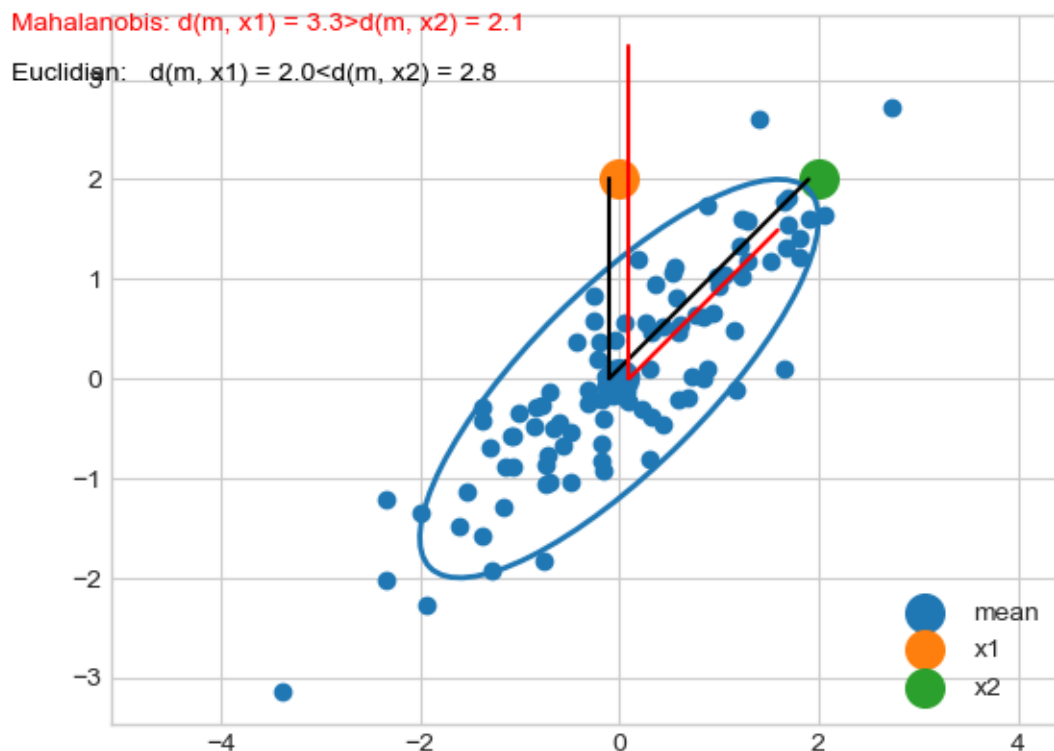
(continues on next page)

(continued from previous page)

```
plt.text(-6.1, 3.5,
        'Mahalanobis: d(m, x1) = %.1f>d(m, x2) = %.1f' % (dm_m_x1, dm_m_x2),
        color='r')

plt.axis('equal')
print('Euclidian   d(m, x1) = %.2f < d(m, x2) = %.2f' % (d2_m_x1, d2_m_x2))
print('Mahalanobis d(m, x1) = %.2f > d(m, x2) = %.2f' % (dm_m_x1, dm_m_x2))
```

```
Euclidian   d(m, x1) = 2.00 < d(m, x2) = 2.83
Mahalanobis d(m, x1) = 3.33 > d(m, x2) = 2.11
```



If the covariance matrix is the identity matrix, the Mahalanobis distance reduces to the Euclidean distance. If the covariance matrix is diagonal, then the resulting distance measure is called a normalized Euclidean distance.

More generally, the Mahalanobis distance is a measure of the distance between a point $\mathbf{x}$ and a distribution $\mathcal{N}(\mathbf{x}|\mu, \Sigma)$. It is a multi-dimensional generalization of the idea of measuring how many standard deviations away $\mathbf{x}$ is from the mean. This distance is zero if $\mathbf{x}$ is at the mean, and grows as $\mathbf{x}$ moves away from the mean: along each principal component axis, it measures the number of standard deviations from $\mathbf{x}$ to the mean of the distribution.

5.4.7 Multivariate normal distribution

The distribution, or probability density function (PDF) (sometimes just density), of a continuous random variable is a function that describes the relative likelihood for this random variable to take on a given value.

The multivariate normal distribution, or multivariate Gaussian distribution, of a P -dimensional

random vector $\mathbf{x} = [x_1, x_2, \dots, x_P]^T$ is

$$\mathcal{N}(\mathbf{x}|\mu, \Sigma) = \frac{1}{(2\pi)^{P/2}|\Sigma|^{1/2}} \exp\left\{-\frac{1}{2}(\mathbf{x} - \mu)^T \Sigma^{-1}(\mathbf{x} - \mu)\right\}.$$

```
import numpy as np
import matplotlib.pyplot as plt
import scipy.stats
from scipy.stats import multivariate_normal
#from mpl_toolkits.mplot3d import Axes3D

def multivariate_normal_pdf(X, mean, sigma):
    """Multivariate normal probability density function over X
    (n_samples x n_features)"""
    P = X.shape[1]
    det = np.linalg.det(sigma)
    norm_const = 1.0 / (((2*np.pi) ** (P/2)) * np.sqrt(det))
    X_mu = X - mu
    inv = np.linalg.inv(sigma)
    d2 = np.sum(np.dot(X_mu, inv) * X_mu, axis=1)
    return norm_const * np.exp(-0.5 * d2)

# mean and covariance
mu = np.array([0, 0])
sigma = np.array([[1, -.5],
                  [-.5, 1]])

# x, y grid
x, y = np.mgrid[-3:3:.1, -3:3:.1]
X = np.stack((x.ravel(), y.ravel())).T
norm = multivariate_normal_pdf(X, mean, sigma).reshape(x.shape)

# Do it with scipy
norm_scipy = multivariate_normal(mu, sigma).pdf(np.stack((x, y), axis=2))
assert np.allclose(norm, norm_scipy)

# Plot
fig, ax = plt.subplots(subplot_kw={"projection": "3d"})
surf = ax.plot_surface(x, y, norm, rstride=3,
                      cstride=3, cmap=plt.cm.coolwarm,
                      linewidth=1, antialiased=False
                      )

ax.set_zlim(0, 0.2)
ax.zaxis.set_major_locator(plt.LinearLocator(10))
ax.zaxis.set_major_formatter(plt.FormatStrFormatter('%0.2f'))

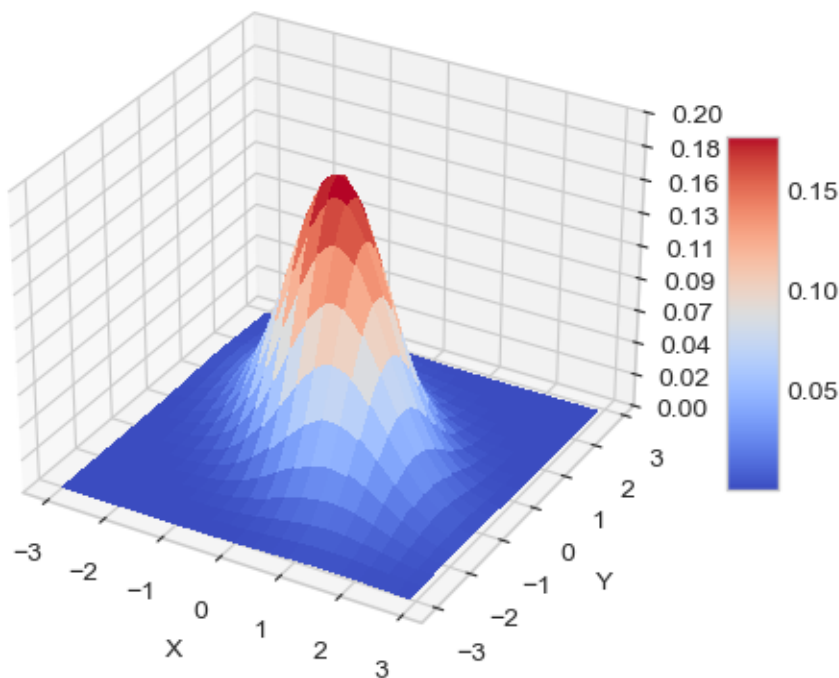
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('p(x)')
```

(continues on next page)

(continued from previous page)

```
plt.title('Bivariate Normal/Gaussian distribution')
fig.colorbar(surf, shrink=0.5, aspect=7, cmap=plt.cm.coolwarm)
plt.show()
```

Bivariate Normal/Gaussian distribution



5.4.8 Exercises

Dot product and Euclidean norm

Given $\mathbf{x} = [2, 1]^T$ and $\mathbf{y} = [1, 1]^T$

1. Write a function `euclidean(x)` that computes the Euclidean norm of vector, $\mathbf{x}$.
2. Compute the Euclidean norm of $\mathbf{x}$.
3. Compute the Euclidean distance of $\|\mathbf{x} - \mathbf{y}\|_2$.
4. Compute the projection of $\mathbf{y}$ in the direction of vector $\mathbf{x}$: y_a .
5. Simulate a dataset $\mathbf{X}$ of $N = 100$ samples of 2-dimensional vectors.
6. Project all samples in the direction of the vector $\mathbf{x}$.

Covariance matrix and Mahalanobis norm

1. Sample a dataset $\mathbf{X}$ of $N = 100$ samples of 2-dimensional vectors from the bivariate normal distribution $\mathcal{N}(\mu, \Sigma)$ where $\mu = [1, 1]^T$ and $\Sigma = \begin{bmatrix} 1 & 0.8 \\ 0.8, 1 \end{bmatrix}$.
2. Compute the mean vector $\bar{\mathbf{x}}$ and center $\mathbf{X}$. Compare the estimated mean $\bar{\mathbf{x}}$ to the true mean, μ .

3. Compute the empirical covariance matrix S . Compare the estimated covariance matrix S to the true covariance matrix, Σ .
4. Compute S^{-1} (`Sinv`) the inverse of the covariance matrix by using `scipy.linalg.inv(S)`.
5. Write a function `mahalanobis(x, xbar, Sinv)` that computes the Mahalanobis distance of a vector x to the mean, $\bar{x}$.
6. Compute the Mahalanobis and Euclidean distances of each sample x_i to the mean $\bar{x}$. Store the results in a 100×2 dataframe.

5.5 Resampling and Monte Carlo Methods

Sources:

- [Scipy Resampling and Monte Carlo Methods](#)

```
# Manipulate data
import numpy as np
import pandas as pd

# Statistics
import scipy.stats
import statsmodels.api as sm
# import statsmodels.stats.api as sms
import statsmodels.formula.api as smf
from statsmodels.stats.stattools import jarque_bera

# Plot
import matplotlib.pyplot as plt
import seaborn as sns
import pystatsml.plot_utils

# Plot parameters
plt.style.use('seaborn-v0_8-whitegrid')
fig_w, fig_h = plt.rcParams.get('figure.figsize')
plt.rcParams['figure.figsize'] = (fig_w, fig_h * .5)
colors = sns.color_palette()
```

5.5.1 Monte-Carlo simulation of Random Walk Process

One-dimensional random walk (Brownian motion)

More information: [Random Walks](#), [Central Limit Theorem](#)

At each step i the process moves with $+1$ or -1 with equal probability, ie, $X_i \in \{+1, -1\}$ with $P(X_i = +1) = P(X_i = -1) = 1/2$. Steps X_i 's are i.i.d..

Let $S_n = \sum_i^n X_i$, or S_i (at time i) is $S_i = S_{i-1} + X_i$

Realizations of random walks obtained by Monte Carlo simulation Plot Few random walks (trajectories), ie, S_n for $n = 0$ to 200

```

np.random.seed(seed=42) # make the example reproducible

n = 200 # trajectory depth
nsamp = 50000 # nb of trajectories

# X: each row (axis 0) contains one trajectory axis 1
# Xn = np.array([np.random.choice(a=[-1, +1], size=n,
#                                replace=True, p=np.ones(2) / 2)
#                                for i in range(nsamp)])

Xn = np.array([np.random.choice(a=np.array([-1, +1]), size=n,
                                replace=True, p=np.ones(2)/2)
                                for i in range(nsamp)])

# Sum of random walks (trajectories)
Sn = Xn.sum(axis=1)

print("True Stat. Mean={:.03f}, Sd={:.02f}".
      format(0, np.sqrt(n) * 1))

print("Est. Stat. Mean={:.03f}, Sd={:.02f}".
      format(Sn.mean(), Sn.std()))

```

True Stat. Mean=0.000, Sd=14.14

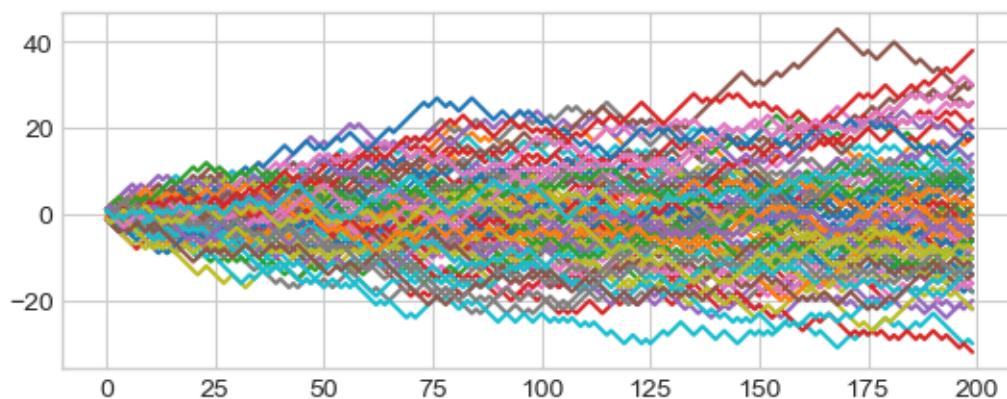
Est. Stat. Mean=0.010, Sd=14.09

Plot cumulative sum of 100 random walks (trajectories)

```

Sn_traj = Xn[:100, :].cumsum(axis=1)
_ = pd.DataFrame(Sn_traj.T).plot(legend=False)

```



Distribution of S_n vs $\mathcal{N}(0, \sqrt{n})$

```

x_low, x_high = Sn.mean()-3*Sn.std(), Sn.mean()+3*Sn.std()
h_ = plt.hist(Sn, range=(x_low, x_high), density=True, bins=43, fill=False,
              label="Histogram")

x_range = np.linspace(x_low, x_high, 30)

```

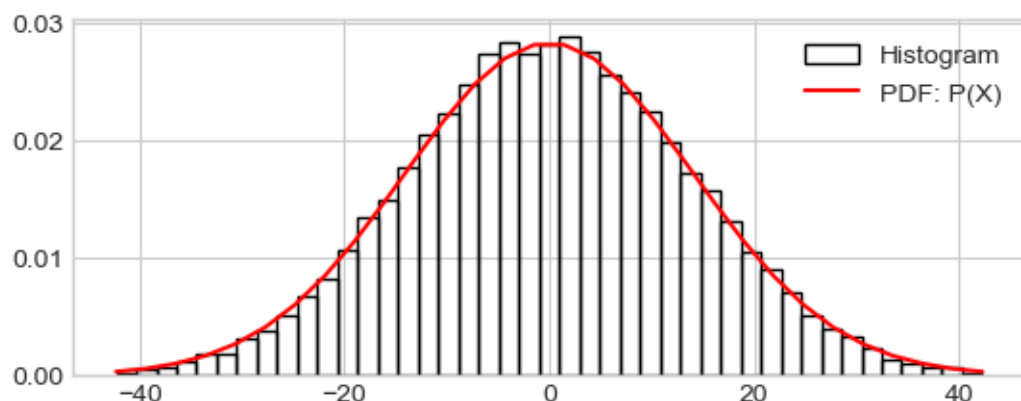
(continues on next page)

(continued from previous page)

```

prob_x_range = scipy.stats.norm.pdf(x_range, loc=Sn.mean(), scale=Sn.std())
plt.plot(x_range, prob_x_range, 'r-', label="PDF: P(X)")
_ = plt.legend()
# print(h_)

```



5.5.2 Permutation Tests

Permutation test:

- The test involves two or more samples assuming that values can be **randomly permuted** under the **null hypothesis**.
- The test is **Resampling procedure to estimate the distribution of a parameter or any statistic under the null hypothesis**, i.e., calculated on the permuted data. This parameter or any statistic is called the **estimator**.
- **Statistical inference** is conducted by computing the proportion of permuted values of the estimator that are “more extreme” than the true one, providing an estimation of the *p-value*.
- Permutation tests are a subset of non-parametric statistics, useful when the distribution of the estimator (under H_0) is unknown or requires complicated formulas.

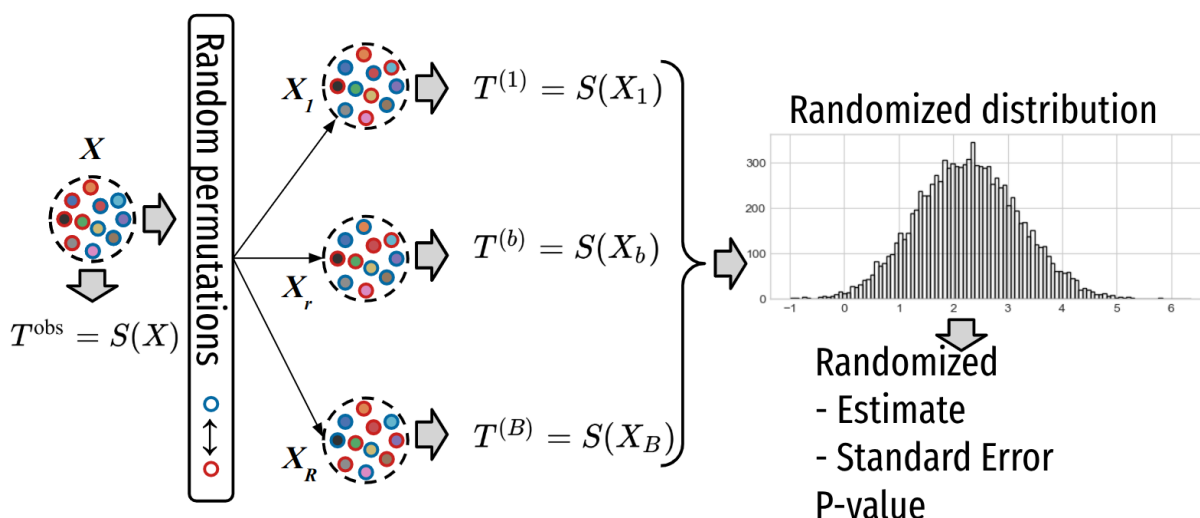


Fig. 5: Permutation test procedure

1. Estimate the observed parameter or statistic:

$$T^{\text{obs}} = S(X)$$

on the initial dataset X of size N . We call it the observed statistic.

Application to mean of one sample. Note that for genericity purposes, the proposed Python implementation can take an iterable list of arrays and calculate several statistics for several variables at once.

```
def mean(*x):
    return np.mean(x[0], axis=0)

# Can manage two variables, and return two statistics
x = np.array([[1, -1], [2, -2], [3, -3], [4, -4]])
print(mean(x))

# In our case with one variable
x = np.array([-1, 0, +1])
stat_obs = mean(x)
print(stat_obs)
```

```
[ 2.5 -2.5]
0.0
```

2. Generate B samples (called randomized samples) $[X_1, \dots, X_b, \dots, X_B]$ from the initial dataset using an adapted permutation scheme and compute the estimator (under H_0):

$$T^{(b)} = S(X_b).$$

First, we need a permutation scheme:

```
def onesample_sign_permutation(*x):
    """Randomly change the sign of all datasets"""
    return [np.random.choice([-1, 1], size=len(x_), replace=True) * x_
            for x_ in x]

x = np.array([-1, 0, +1])
# 10 random sign permutation of [-1, 0, +1] and mean calculation
# Mean([-1,0,1])= 0 or Mean([1,0,1])= 0.66 or Mean([-1,0,-1])= -0.66
np.random.seed(42)
stats_rnd = [float(mean(*onesample_sign_permutation(x)).round(3))
             for perm in range(10)]
print(stats_rnd)
```

```
[0.0, 0.667, 0.0, -0.667, 0.667, 0.667, 0.0, 0.667, 0.0, 0.0]
```

Using a parallelized permutation procedure using [Joblib](#) whose code is available [here](#)

```
from pystatsml import resampling

stats_rnd = np.array(resampling.resample_estimate(x, estimator=mean,
                                                resample=onesample_sign_permutation, n_resamples=10))

print(stats_rnd.round(3))
```

```
[ 0.    0.667  0.667  0.    0.    0.667  0.667 -0.667  0.    0.    ]
```

3. Permutation test: Compute statistics of the estimator under the null hypothesis using randomized estimates $T^{(b)}$'s.

- Mean (under H_0):

$$\bar{T}_B = \frac{1}{B} \sum_{b=1}^B T^{(b)}$$

- Standard Error σ_{T_B} (under H_0):

$$\sigma_{T_B} = \sqrt{\frac{1}{B-1} \sum_{b=1}^B (\bar{T}_B - T^{(b)})^2}$$

- $T^{(b)}$'s provides an estimate the distribution of $P(T|H_0)$ necessary to compute p-value using the distribution (under H_0):

- One-sided p-value:

$$P(T \geq T^{\text{obs}}|H_0) \approx \frac{\text{card}(T^{(b)} \geq T^{\text{obs}})}{B}$$

- Two-sided p-value:

$$P(T \leq T^{\text{obs}} \text{ or } T \geq T^{\text{obs}}|H_0) \approx \frac{\text{card}(T^{(b)} \leq T^{\text{obs}}) + \text{card}(T^{(b)} \geq T^{\text{obs}})}{B}$$

Let randomized samples) $[X_1, \dots X_r, \dots X_{rnd}]$ from the initial dataset using a adapted permutation scheme and compute the estimator (under H_0): $T^{(b)} = S(X_r)$.

3. Permutation test: Compute statistics of the estimator under the null hypothesis using randomized estimates $T^{(b)}$'s.

- Mean (under H_0):

$$\bar{T}_B = \frac{1}{r} \sum_{r=1}^R T^{(b)}$$

- Standard Error σ_{T_B} (under H_0):

$$\sigma_{T_B} = \sqrt{\frac{1}{R-1} \sum_{r=1}^K (\bar{T}_B - T^{(b)})^2}$$

- $T^{(b)}$'s provides an estimate the distribution of $P(T|H_0)$ necessary to compute p-value using the distribution (under H_0):
 - One-sided p-value:

$$P(T \geq T^{\text{obs}}|H_0) \approx \frac{\text{card}(T^{(b)} \geq T^{\text{obs}})}{R}$$

- Two-sided p-value:

$$P(T \leq T^{\text{obs}} \text{ or } T \geq T^{\text{obs}}|H_0) \approx \frac{\text{card}(T^{(b)} \leq T^{\text{obs}}) + \text{card}(T^{(b)} \geq T^{\text{obs}})}{R}$$

```
# 3. Permutation test: Compute statistics under H0, estimates distribution and
# compute p-values
```

```
def permutation_test(stat_obs, stats_rnd):
    """Compute permutation test using statistic under H1 (true statistic)
    and statistics under H0 (randomized statistics) for several statistics.

    Parameters
    -----
    stat_obs : array of shape (nb_statistics)
               statistic under H1 (true statistic)
    stats_rnd : array of shape (nb_permutations, nb_statistics)
               statistics under H0 (randomized statistics)

    Returns
    -----
    tuple
        p-value, statistics mean under H0, statistics stan under H0

    Example
    -----
    >>> np.random.seed(42)
    >>> stats_rnd = np.random.randn(1000, 5)
    >>> stat_obs = np.arange(5)
    >>> pval, stat_mean_rnd, stat_se_rnd = permutation_test(stat_obs, stats_rnd)
    >>> print(pval)
    [1.    0.315 0.049 0.002 0.    ]
    """

    n_perms = stats_rnd.shape[0]

    # 1. Randomized Statistics

    # Mean of randomized estimates
    stat_mean_rnd = np.mean(stats_rnd, axis=0)

    # Standard Error of randomized estimates
    stat_se_rnd = np.sqrt(1 / (n_perms - 1) *
                          np.sum((stat_mean_rnd - stats_rnd) ** 2, axis=0))
```

(continues on next page)

(continued from previous page)

```

# 2. Compute two-sided p-value using the distribution under H0:

extreme_vals = (stats_rnd <= -np.abs(stat_obs)
                ) | (stats_rnd >= np.abs(stat_obs))
pval = np.sum(extreme_vals, axis=0) / n_perms
# We could use:
# (np.sum(stats_rnd <= -np.abs(stat_obs)) + \
#  np.sum(stats_rnd >= np.abs(stat_obs))) / n_perms

# 3. Distribution of the parameter or statistic under H0
# stat_density_rnd, bins = np.histogram(stats_rnd, bins=50, density=True)
# dx = np.diff(bins)

return pval, stat_mean_rnd, stat_se_rnd

pval, stat_mean_rnd, stat_se_rnd = permutation_test(stat_obs, stats_rnd)
print("Estimate: {:.2f}, Mean(Estimate|H0): {:.4f}, p-val: {:e}, SE: {:.3f}".
      format(stat_obs, stat_mean_rnd, pval, stat_se_rnd))

```

```
Estimate: 0.00, Mean(Estimate|H0): 0.2000, p-val: 1.000000e+00, SE: 0.450
```

One Sample Sign Permutation

Consider the monthly revenue figures of for 100 stores before and after a marketing campaigns. We compute the difference ($x_i = x_i^{\text{after}} - x_i^{\text{before}}$) for each store i . Under the null hypothesis, i.e., no effect of the campaigns, x_i^{after} and x_i^{before} could be permuted, which is equivalent to randomly switch the sign of x_i . Here we will focus on the sample mean $T^{\text{obs}} = S(X) = 1/n \sum_i x_i$ as statistic of interest.

Read data:

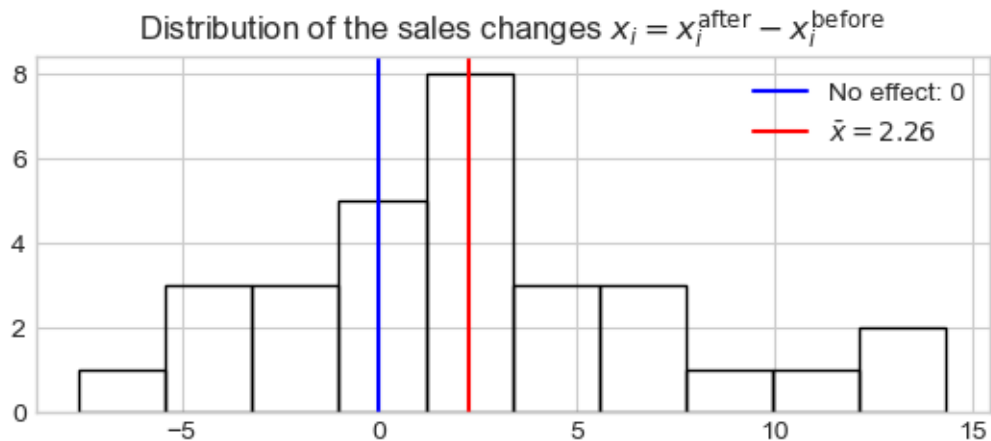
```

df = pd.read_csv("../datasets/Monthly Revenue (in thousands).csv")
# Reshape Keep only the 30 first samples
df = df.pivot(index='store_id', columns='time', values='revenue')[:30]
df.after -= 3 # force to smaller effect size

x = df.after - df.before

plt.hist(x, fill=False)
plt.axvline(x=0, color="b", label=r'No effect: 0')
plt.axvline(x=x.mean(), color="r", ls='--', label=r'$\bar{x}=%.2f$' % x.mean())
plt.legend()
_ = plt.title(
    r'Distribution of the sales changes $x_i = x_i^{\text{after}} - x_i^{\text{before}}$')

```



```
# 1. Estimate the observed parameter or statistic
stat_obs = mean(x)

# 2. Generate randomized samples and compute the estimator (under H0)
stats_rnd = np.array(resampling.resample_estimate(x, estimator=mean,
                                                resample=onesample_sign_permutation, n_resamples=1000))

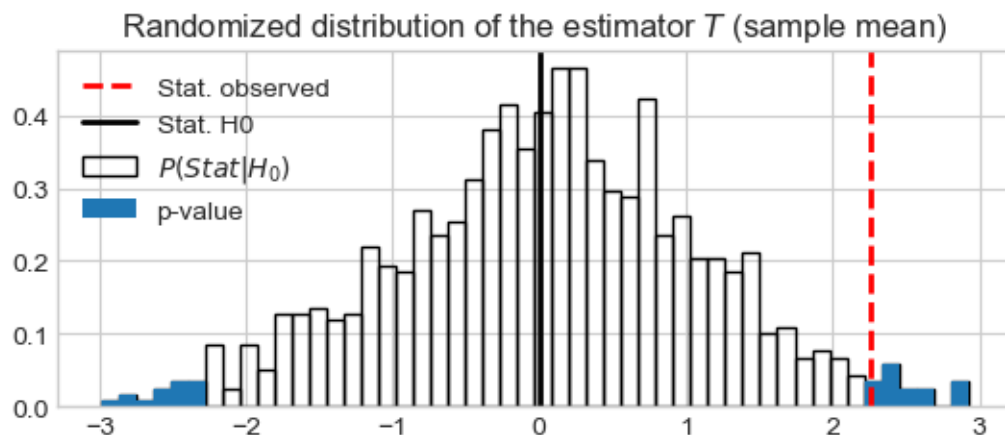
# 3. Permutation test: Compute stats. under H0, and p-values
pval, stat_mean_rnd, stat_se_rnd = permutation_test(stat_obs, stats_rnd)
print("Estimate: {:.2f}, Mean(Estimate|H0): {:.4f}, p-val: {:.e}, SE: {:.3f}".
      format(stat_obs, stat_mean_rnd, pval, stat_se_rnd))
```

```
Estimate: 2.26, Mean(Estimate|H0): -0.0093, p-val: 3.900000e-02, SE: 1.051
```

Plot density

```
stats_density_rnd, bins = np.histogram(stats_rnd, bins=50, density=True)
dx = np.diff(bins)

pystatsml.plot_utils.plot_pvalue_under_h0(stat_vals=bins[1:],
                                          stat_probs=stats_density_rnd,
                                          stat_obs=stat_obs, stat_h0=0, bar_width=np.diff(bins),
                                          thresh_low=-np.abs(stat_obs), thresh_high=np.abs(stat_obs))
_ = plt.title(r'Randomized distribution of the estimator  $\bar{x}$  (sample mean)')
```



Similar procedure can be conducted with many statistic e.g., the t-statistic (more sensitive than the mean):

```
def ttest_1samp(*x):
    x = x[0]
    return (np.mean(x, axis=0) - 0) / np.std(x, ddof=1, axis=0) * np.sqrt(len(x))

# 1. Estimate the observed parameter or statistic
stat_obs = ttest_1samp(x)

# 2. Generate randomized samples and compute the estimator (under H0)
stats_rnd = np.array(resampling.resample_estimate(x, estimator=ttest_1samp,
        resample=onesample_sign_permutation, n_resamples=1000))

# 3. Permutation test: Compute stats. under H0, and p-values
pval, stat_mean_rnd, stat_se_rnd = permutation_test(stat_obs, stats_rnd)
print("Estimate: {:.2f}, Mean(Estimate|H0): {:.4f}, p-val: {:e}, SE: {:.3f}".
        format(stat_obs, stat_mean_rnd, pval, stat_se_rnd))
```

Estimate: 2.36, Mean(Estimate|H0): 0.0130, p-val: 2.800000e-02, SE: 1.064

Or the median (less sensitive than the mean):

```
def median(*x):
    x = x[0]
    return np.median(x, axis=0)

# 1. Estimate the observed parameter or statistic
stat_obs = median(x)

# 2. Generate randomized samples and compute the estimator (under H0)
stats_rnd = np.array(resampling.resample_estimate(x, estimator=median,
        resample=onesample_sign_permutation, n_resamples=1000))

# 3. Permutation test: Compute stats. under H0, and p-values
```

(continues on next page)

(continued from previous page)

```
pval, stat_mean_rnd, stat_se_rnd = permutation_test(stat_obs, stats_rnd)
print("Estimate: {:.2f}, Mean(Estimate|H0): {:.4f}, p-val: {:e}, SE: {:.3f}".
      format(stat_obs, stat_mean_rnd, pval, stat_se_rnd))
```

```
Estimate: 1.85, Mean(Estimate|H0): 0.0255, p-val: 1.030000e-01, SE: 1.081
```

Two Samples Permutation

x is a variable $y \in \{0, 1\}$ is a sample label for two groups. To obtain sample under H_0 we just have to permute the group's labels y :

```
# Dataset
from pystatsml import datasets

x, y = datasets.make_twosamples(n_samples=30, n_features=1, n_informative=1,
                                group_scale=1., noise_scale=1., shared_scale=0,
                                random_state=0)

print(scipy.stats.ttest_ind(x[y == 0], x[y == 1], equal_var=True))

# Label permutation, expect label_permutation(x, label)

def label_permutation(*x):
    x, label = x[0], x[1]
    label = np.random.permutation(label)
    return x, label

xr, yr = label_permutation(x, y)
assert np.all(x == xr)
print("Original labels:", y)
print("Permuted labels:", yr)
```

```
TtestResult(statistic=np.float64(-1.2845532458346598), pvalue=np.float64(0.
↪2094747870423826), df=np.float64(28.0))
Original labels: [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 1. 1.
↪1. 1. 1.
1. 1. 1. 1. 1. 1.]
Permuted labels: [0. 1. 0. 1. 0. 0. 0. 0. 1. 1. 0. 0. 1. 0. 0. 0. 0. 1. 1. 1. 1.
↪0. 1. 1.
0. 1. 1. 0. 1. 1.]
```

As statistic we use the student statistic of two sample test with equal variance

```
def twosample_ttest(*x):
    x, label = x[0], x[1]
    ttest = scipy.stats.ttest_ind(x[label == 0], x[label == 1],
                                   equal_var=True, axis=0)
```

(continues on next page)

(continued from previous page)

```
return ttest.statistic
```

```
# 1. Estimate the observed parameter or statistic
stat_obs = twosample_ttest(x, y)

# 2. Generate randomized samples and compute the estimator (under H0)

# Sequential version:
# stats_rnd = np.array([twosample_ttest(*label_permutation(x, y))
#                       for perm in range(1000)])

# Parallel version:
stats_rnd = np.array(resampling.resample_estimate(x, y, estimator=twosample_ttest,
                                                  resample=label_permutation, n_resamples=1000))

# 3. Permutation test: Compute stats. under H0, and p-values
pval, stat_mean_rnd, stat_se_rnd = permutation_test(stat_obs, stats_rnd)
print("Estimate: {:.2f}, Mean(Estimate|H0): {:.4f}, p-val: {:.e}, SE: {:.3f}".
      format(stat_obs, stat_mean_rnd, pval, stat_se_rnd))

print("\nNon permuted t-test:")
ttest = scipy.stats.ttest_ind(x[y == 0], x[y == 1], equal_var=True)
print("Estimate: {:.2f}, p-val: {:.e}".
      format(ttest.statistic, ttest.pvalue))

print("\nPermutation using scipy.stats.ttest_ind")
ttest = scipy.stats.ttest_ind(x[y == 0], x[y == 1], equal_var=True,
                              method=scipy.stats.PermutationMethod())
print("Estimate: {:.2f}, p-val: {:.e}".
      format(ttest.statistic, ttest.pvalue))
```

```
Estimate: -1.28, Mean(Estimate|H0): 0.0015, p-val: 1.970000e-01, SE: 1.023
```

```
Non permuted t-test:
```

```
Estimate: -1.28, p-val: 2.094748e-01
```

```
Permutation using scipy.stats.ttest_ind
```

```
Estimate: -1.28, p-val: 1.962000e-01
```

5.5.3 Permutation-Based Correction for Multiple Testing: Westfall and Young Correction

Westfall and Young (1993) Correction for Multiple Testing, also known as maxT, controls the **Family-Wise Error Rate (FWER)** in the context of **multiple hypothesis testing**, particularly in high-dimensional settings (e.g., genomics), where traditional methods like Bonferroni are overly conservative.

- The method relies on **permutation testing** to empirically estimate the distribution of test

statistics under the **global null hypothesis**.

- It accounts for the **dependence structure** between tests, improving power compared to Bonferroni-type corrections.

Procedure

Let there be P hypotheses $H_1, \dots, H_P$ and corresponding test statistics $T_1, \dots, T_P$:

1. **Compute observed test statistics** $\{T_j^{\text{obs}}\}_{j=1}^P$, for each hypothesis.
2. **Permute the data** B times (e.g., shuffle labels), recomputing all P test statistics each time yielding to a $(B \times P)$ table with all permuted statistics $\{T_j^{(b)}\}_{(b,j)}^{(B,P)}$.
3. **For each permutation**, record the **maximum test statistic** across all hypotheses, yielding to a vector of B maximum which value is given by:

$$M^{(b)} = \max_{j=1, \dots, P} T_j^{(b)}$$

4. For each hypothesis j , calculate the adjusted p-value, yielding to a vector of P p-values which value is given by:

$$\tilde{p}_j = \frac{1}{B} \sum_{b=1}^B \text{card}(T_j^{\text{obs}} \leq M^{(b)})$$

(For one-sided tests. For two-sided, take $|T_j|$.)

5. Reject hypotheses with $\tilde{p}_j \leq \alpha$.

Advantages

- **Strong FWER control**, even under arbitrary dependence.
- **More powerful** than Bonferroni or Holm in many dependent testing scenarios.
- **Non-parametric**: does not rely on distributional assumptions.

Limitations

- **Computationally intensive**, especially with large m and many permutations.
- Requires **exchangeability** of data under the null (a key assumption for valid permutations).

Dataset

```
from pystatsml import datasets

n_informative = 100 # Number of features with signal (Positive features)
n_features = 1000
x, y = datasets.make_twosamples(n_samples=30, n_features=n_features, n_
    ↪informative=n_informative,
                                group_scale=1., noise_scale=1., shared_scale=1.5,
                                random_state=42)
```

```

# 1. **Compute observed test statistics**
stat_obs = twosample_ttest(x, y)

# 2. Randomized statistics
stats_rnd = np.array(resampling.resample_estimate(x, y, estimator=twosample_ttest,
                                                resample=label_permutation, n_resamples=1000))

def multipletests_westfall_young(stats_rnd, stat_obs):
    """Westfall and Young FWER correction for multiple comparisons.

    Parameters
    -----
    stats_rnd : array (n_resamples, n_features)
        Randomized Statistics
    stats_true : array (n_features)
        Observed Statistics
    alpha : float, optional
        by default 0.05

    Return
    -----
    Array (n_features) P-values corrected for multiple comparison
    """

    # 3. **For each permutation**, record the **maximum test statistic**
    stat_rnd_max = np.max(np.abs(stats_rnd), axis=1)

    # 4. For each hypothesis $j$, calculate the adjusted p-value
    pvalues_ws = np.array([np.sum(stat_rnd_max >= np.abs(
        stat)) / stat_rnd_max.shape[0] for stat in stat_obs])

    return pvalues_ws

pvalues_ws = multipletests_westfall_young(stats_rnd, stat_obs)

```

Compute p-values using:

1. Un-corrected p-values: High false positives rate.
2. Bonferroni correction: Reduced sensitivity (low True positives rate).
3. Westfall and Young correction: Improved sensitivity with controlled false positive:

```

# Usual t-test with uncorrected p-values:
import statsmodels.sandbox.stats.multicomp as multicomp
ttest = scipy.stats.ttest_ind(x[y == 0], x[y == 1], equal_var=True)

# Classical Bonferroni correction
_, pvals_bonf, _, _ = multicomp.multipletests(ttest.pvalue, alpha=0.05,
                                              method='bonferroni')

def print_positive(title, pvalues, n_informative, n_features):

```

(continues on next page)

(continued from previous page)

```

print('%s \nPositive: %i/%i, True Positive: %i/%i, False Positive: %i/%i' %
      (title,
       np.sum(pvalues <= 0.05), n_features,
       np.sum(pvalues[:n_informative] <= 0.05), n_informative,
       np.sum(pvalues[n_informative:] <= 0.05), (n_features-n_informative)))

print_positive("No correction", ttest.pvalue, n_informative, n_features)
print_positive("Bonferroni correction", pvals_bonf, n_informative, n_features)
print_positive("Westfall and Young", pvalues_ws, n_informative, n_features)

```

No correction

Positive: 209/1000, True Positive: 94/100, False Positive: 115/900

Bonferroni correction

Positive: 0/1000, True Positive: 0/100, False Positive: 0/900

Westfall and Young

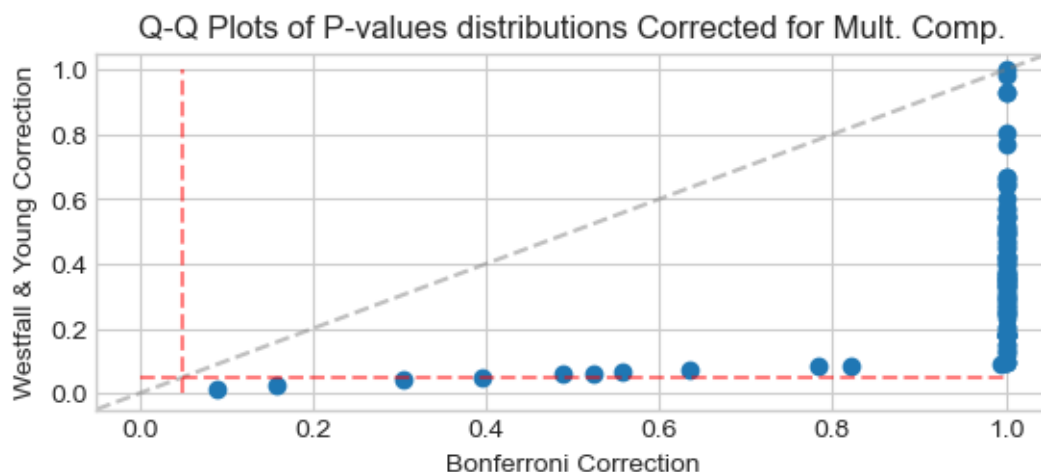
Positive: 4/1000, True Positive: 4/100, False Positive: 0/900

Compare P-Values distribution with Bonferroni correction for multiple comparison:

```

from scipy import stats
_, q_bonferroni = stats.probplot(pvals_bonf[:n_informative], fit=False)
_, q_ws = stats.probplot(pvalues_ws[:n_informative], fit=False)
ax = plt.scatter(q_bonferroni, q_ws, marker='o', color=colors[0])
ax = plt.axline([0, 0], slope=1, color="gray", ls='--', alpha=0.5)
ax = plt.vlines(0.05, 0, 1, color="r", ls='--', alpha=0.5)
ax = plt.hlines(0.05, 0, 1, color="r", ls='--', alpha=0.5)
ax = plt.xlabel('Bonferroni Correction')
ax = plt.ylabel('Westfall & Young Correction')
ax = plt.title('Q-Q Plots of P-values distributions Corrected for Mult. Comp.')

```



5.5.4 Bootstrapping

Bootstrapping:

- **Resampling procedure** to estimate the distribution of a statistic or parameter of interest, called the estimator.

- Derive estimates of **variability for estimator** (bias, standard error, confidence intervals, etc.).
- **Statistical inference** is conducted by looking the **confidence interval (CI) contains the null hypothesis**.
- Nonparametric approach statistical inference, useful when model assumptions is in doubt, unknown or requires complicated formulas.
- **Bootstrapping with replacement** has favorable performances (Efron 1979, and 1986) compared to prior methods like the jackknife that sample without replacement
- **Regularize models** by fitting several models on bootstrap samples and averaging their predictions (see Baging and random-forest). See machine learning chapter.

Note that compared to random permutation, bootstrapping sample the distribution under the **alternative hypothesis**, it doesn't consider the distribution under the null hypothesis. A great advantage of bootstrap is its **simplicity of the procedure**:

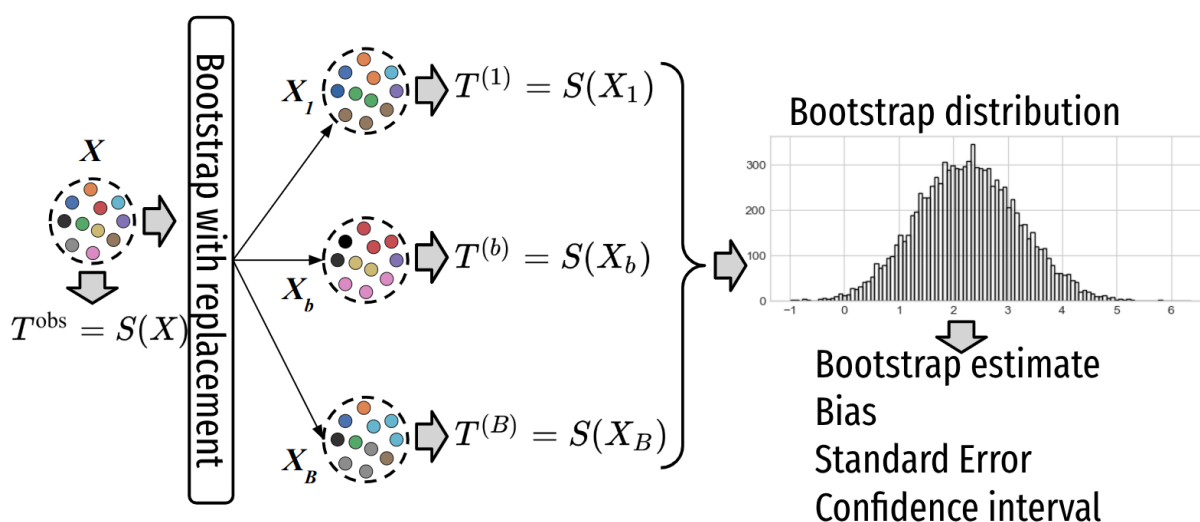


Fig. 6: Bootstrapping procedure

1. Compute the estimator $T^{\text{obs}} = S(X)$ on the initial dataset X of size N .
2. Generate B samples (called bootstrap samples) $[X_1, \dots, X_b, \dots, X_B]$ from the initial dataset by randomly drawing **with replacement** B of the N observations.
3. For each sample b compute the estimator $\{T^{(b)} = S(X_b)\}_{b=1}^B$, which provides the bootstrap distribution $P(T_B|H1)$ of the estimator (under the alternative hypothesis $H1$).
4. Compute statistics of the estimator on bootstrap estimates $T^{(b)}$'s:
 - Bootstrap estimate (of the parameters):

$$\bar{T}_B = \frac{1}{B} \sum_{b=1}^B T^{(b)}$$

- Bias = bootstrap estimate - estimate:

$$\hat{b}_{T_B} = \bar{T}_B - T^{\text{obs}}$$

- Standard error $\sigma_{\hat{T}_B}$:

$$\sigma_{\hat{T}_B} = \sqrt{\frac{1}{B-1} \sum_{b=1}^B (\bar{T}_B - T^{(b)})^2}$$

- Confidence interval using the estimated bootstrapped distribution of the estimator:

$CI_{95\%} = [T_1^{\text{obs}} = Q_{T^{(b)}}(2.5\%), T_2^{\text{obs}} = Q_{T^{(b)}}(97.5\%)]$ i.e., the 2.5%, 97.5% quantiles estimators out of the $\{T^{(b)}\}$

Application using the monthly revenue of 100 stores before and after a marketing campaigns, using the difference ($x_i = x_i^{\text{after}} - x_i^{\text{before}}$) for each store i . If the average difference $\bar{x} = 1/n \sum_i x_i$ is positive (resp. negative), then the marketing campaigns will be considered as efficient (resp. detrimental). We will use bootstrapping to compute the confidence interval (CI) and see if 0 is comprised in the CI.

```
x = df.after - df.before
S = np.mean

# 1. Model parameters
stat_hat = S(x)

np.random.seed(15) # set the seed for reproducible results
B = 1000 # Number of bootstrap

# 2. Bootstrapped samples

x_B = [np.random.choice(x, size=len(x), replace=True) for boot_i in range(B)]

# 3. Bootstrap estimates and distribution

stat_hats_B = np.array([S(x_b) for x_b in x_B])
stat_density_B, bins = np.histogram(stat_hats_B, bins=50, density=True)
dx = np.diff(bins)

# 4. Bootstrap Statistics

# Bootstrap estimate
stat_bar_B = np.mean(stat_hats_B)

# Bias = bootstrap estimate - estimate
bias_hat_B = stat_bar_B - stat_hat

# Standard Error
se_hat_B = np.sqrt(1 / (B - 1) * np.sum((stat_bar_B - stat_hats_B) ** 2))

# Confidence interval using the estimated bootstrapped distribution of estimator

ci95 = np.quantile(a=stat_hats_B, q=[0.025, 0.975])

print(
    "Est.: {:.2f}, Boot Est.: {:.2f}, bias: {:e},\n"
```

(continues on next page)

(continued from previous page)

```

Boot SE: {:.2f}, CI: [{:.5f}, {:.5f}]"
.format(stat_hat, stat_bar_B, bias_hat_B, se_hat_B, ci95[0], ci95[1]))

```

```

Est.: 2.26, Boot Est.: 2.19, bias: -7.201946e-02,      Boot SE: 0.95, CI: [0.45256,
↪ 4.08465]

```

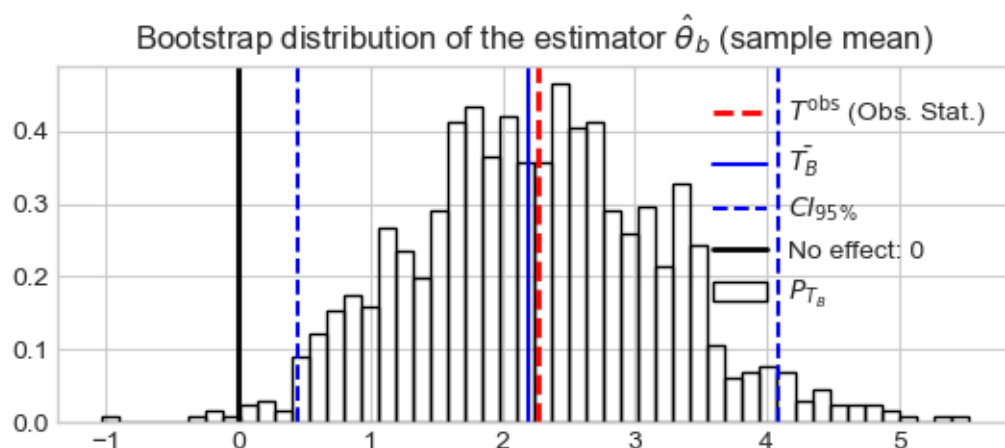
Conclusion: Zero is outside the CI, moreover $\bar{X}$ is positive. Thus we can conclude the marketing campaign produced a significant increase of the sales.

Plot

```

plt.bar(bins[1:], stat_density_B, width=dx, fill=False, label=r'$P_{T_B}$')
plt.axvline(x=stat_hat, color='r', ls='--', lw=2,
            label=r'$T^{\text{obs}}$ (Obs. Stat.)')
plt.axvline(x=stat_bar_B, color="b", ls='-', lw=2, label=r'$\bar{T}_B$')
plt.axvline(x=ci95[0], color="b", ls='--', label=r'$CI_{95\%}$')
plt.axvline(x=ci95[1], color="b", ls='--')
plt.axvline(x=0, color="k", lw=2, label=r'No effect: 0')
plt.legend()
_ = plt.title(
    r'Bootstrap distribution of the estimator $\hat{\theta}_b$ (sample mean)'
)

```

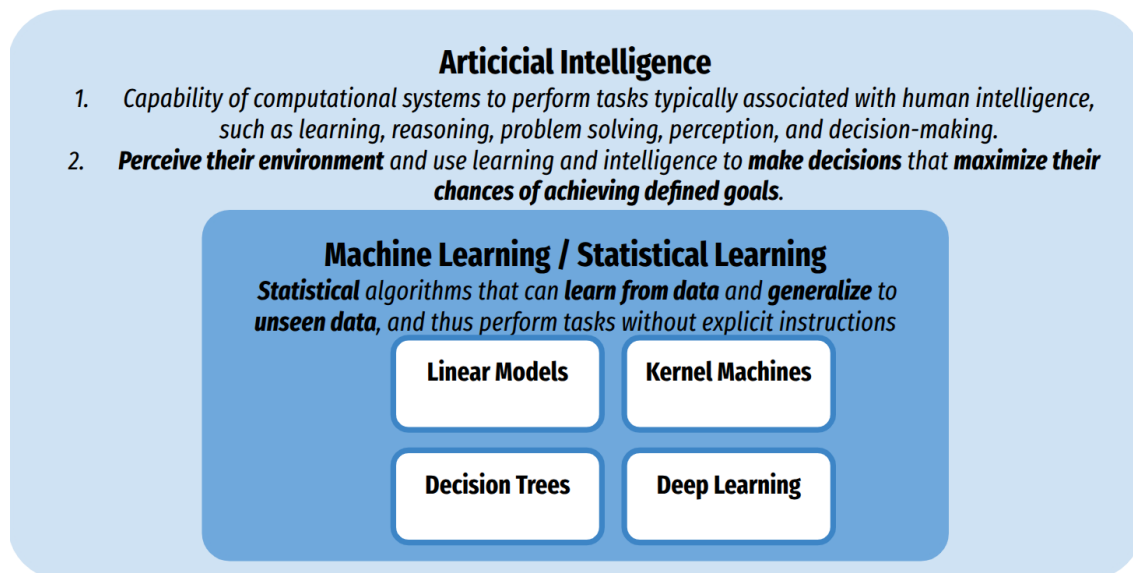


MACHINE LEARNING OVERVIEW

6.1 Introduction

6.1.1 Artificial Intelligence & Machine Learning

Machine learning is a branch of artificial intelligence (AI) focused on developing algorithms and models that enable computers to **learn patterns** and make data-based decisions. Instead of being explicitly programmed to perform a task, a machine learning system improves its performance through experience by analyzing data and recognizing patterns.



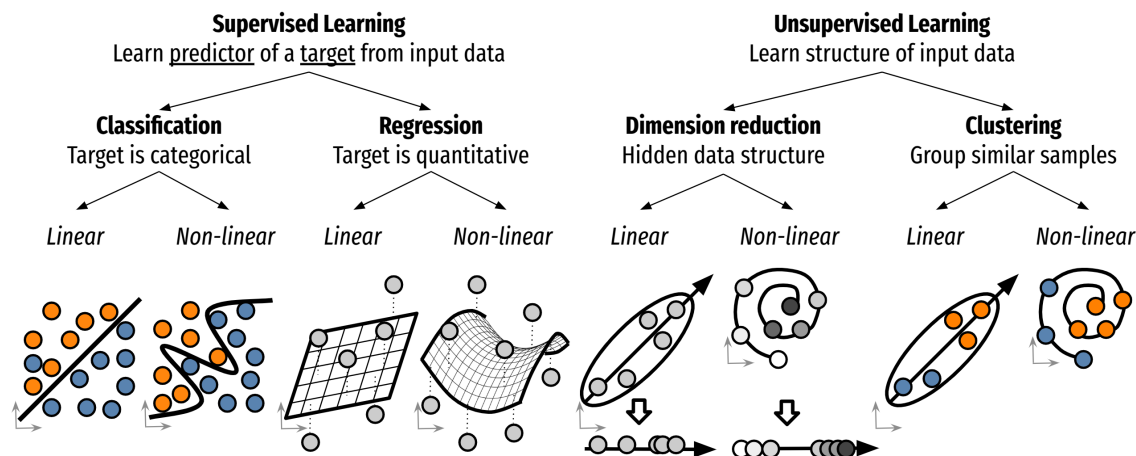
In machine learning, a **model** is a **mathematical representation** of the data using set **parameters**. An **estimator** refers to any algorithm that learns or estimates the parameters from data, i.e., fitting the model to the data.

Machine learning models can **linear or non-linear**, see [ML map](#). Machine learning can address various types of problems, including:

6.1.2 Typology of Machine Learning Problems

1. **Supervised learning** Learn a function to predict output or target y given input X .
 - a. **Regression** problems: y is quantitative.
 - b. **Classification** problems y is qualitative/categorical, i.e., (labels).
2. **Unsupervised learning**: Learn the hidden structure of the data X

- a. **Dimensionality reduction** (or feature extraction) in machine learning refers to techniques that reduce the number of input variables or features in a dataset while preserving essential information and exploiting redundant or irrelevant features. It can be helpful for visualization or analysis of high-dimension data.
- b. **Clustering** groups similar data points together based on their features. It helps identify patterns or structures within the data by organizing it into clusters, where points within the same cluster are more similar to each other than to those in different clusters.



6.1.3 Simplified Typology of Machine Learning Algorithms

Linear Models (scikit-learn)

- Output is a Linear combination of input features: (One neuron)
- Domains: Tabular data, basic computer vision (with feature extraction) and NLP
- Robust to overfit but limited power, requires few samples (>100)

Kernel Machine (Linear or non-linear) (scikit-learn)

- Output is a Linear combination of input samples
- Domains: Tabular data, basic computer vision (with feature extraction) and NLP
- Requires limited samples (>500)

Decision Tree (non-linear) (scikit-learn)

- Output is a Linear combination of input samples
- Domains: Tabular data, basic computer vision (with feature extraction) and NLP
- Requires limited samples (>500)

Deep Learning (Non Linear) (Pytorch or TensorFlow with scikit-learn)

- Stack layers of neurons
- Domains: Computer vision, NLP
- Powerful, requires limited very large sample size samples (>1000)

6.2 Overfitting and Regularization

In statistics and machine learning, overfitting occurs when a statistical model describes random errors or noise instead of the underlying relationships. Overfitting generally occurs when a model is **excessively complex**, such as having **too many parameters relative to the number of observations**. A model that has been overfit will generally have poor predictive performance, as it can exaggerate minor fluctuations in the data.

A learning algorithm is trained using some set of training samples. If the learning algorithm has the capacity to overfit the training samples the performance on the **training sample set** will improve while the performance on unseen **test sample set** will decline.

The overfitting phenomenon has three main explanations:

- excessively complex models,
- multicollinearity, and
- high dimensionality.

6.2.1 Causes of Overfitting

Multicollinearity

Predictors are highly correlated, meaning that one can be linearly predicted from the others. In this situation the coefficient estimates of the multiple regression may change erratically in response to small changes in the model or the data. Multicollinearity does not reduce the predictive power or reliability of the model as a whole, at least not within the sample data set; it only affects computations regarding individual predictors. That is, a multiple regression model with correlated predictors can indicate how well the entire bundle of predictors predicts the outcome variable, but it may not give valid results about any individual predictor, or about which predictors are redundant with respect to others. In case of perfect multicollinearity the predictor matrix is singular and therefore cannot be inverted. Under these circumstances, for a general linear model $\mathbf{y} = \mathbf{X}\mathbf{w} + \varepsilon$, the ordinary least-squares estimator, $\mathbf{w}_{OLS} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$, does not exist.

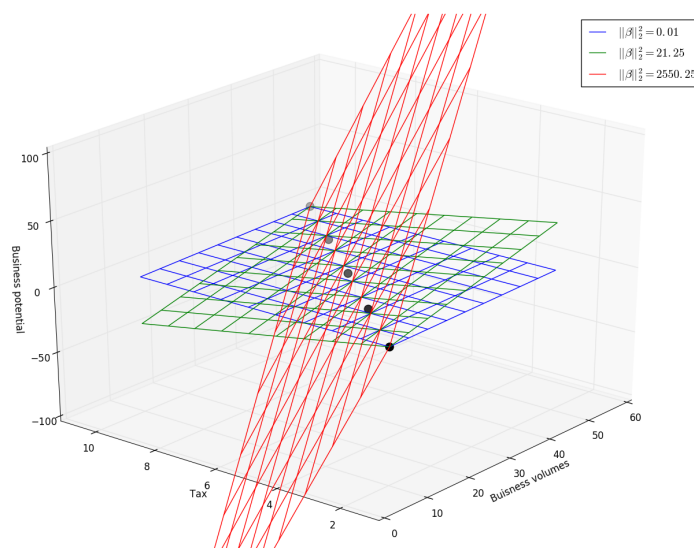
```
import numpy as np

# Plot
import matplotlib.pyplot as plt
import seaborn as sns

# Plot parameters
plt.style.use('seaborn-v0_8-whitegrid')
fig_w, fig_h = plt.rcParams.get('figure.figsize')
plt.rcParams['figure.figsize'] = (fig_w, fig_h * .5)
```

An example where correlated predictor may produce an unstable model follows: We want to predict the business potential (pb) of some companies given their business volume (bv) and the taxes (tx) they are paying. Here $\text{pb} \sim 10\%$ of bv . However, $\text{taxes} = 20\%$ of bv (tax and bv are highly collinear), therefore there is an infinite number of linear combinations of tax and bv that lead to the same prediction. Solutions with very large coefficients will produce excessively large predictions.

Multicollinearity between the predictors: business volumes and tax produces unstable models



with arbitrary large coefficients.

Dealing with multicollinearity:

- Regularization by e.g. ℓ_2 shrinkage: Introduce a bias in the solution by making $(X^T X)^{-1}$ non-singular. See ℓ_2 shrinkage.
- Feature selection: select a small number of features. See: Isabelle Guyon and André Elisseeff *An introduction to variable and feature selection* The Journal of Machine Learning Research, 2003.
- Feature selection: select a small number of features using ℓ_1 shrinkage.
- Extract few independent (uncorrelated) features using e.g. principal components analysis (PCA), partial least squares regression (PLS-R) or regression methods that cut the number of predictors to a smaller set of uncorrelated components.

```

bv = np.array([10, 20, 30, 40, 50])          # business volume
tax = .2 * bv                                # Tax
bp = .1 * bv + np.array([-1, .2, .1, -.2, .1]) # business potential

X = np.column_stack([bv, tax])
beta_star = np.array([.1, 0]) # true solution

'''
Since tax and bv are correlated, there is an infinite number of linear
combinations leading to the same prediction.
'''

# 10 times the bv then subtract it 9 times using the tax variable:
beta_medium = np.array([.1 * 10, -.1 * 9 * (1/.2)])
# 100 times the bv then subtract it 99 times using the tax variable:
beta_large = np.array([.1 * 100, -.1 * 99 * (1/.2)])

print("L2 norm of coefficients: small:%.2f, medium:%.2f, large:%.2f." %
      (np.sum(beta_star ** 2), np.sum(beta_medium ** 2), np.sum(beta_large ** 2)))

print("However all models provide the exact same predictions.")

```

(continues on next page)

(continued from previous page)

```
assert np.all(np.dot(X, beta_star) == np.dot(X, beta_medium))
assert np.all(np.dot(X, beta_star) == np.dot(X, beta_large))
```

L2 norm of coefficients: small:0.01, medium:21.25, large:2550.25.
However `all` models provide the exact same predictions.

Model complexity

Model complexity impacts both bias and variance:

- Low-complexity models (underfitting):
 - High bias (priors or assumptions about data are too strong, leading to oversimplified models).
 - Low variance (consistent predictions across different datasets but with poor accuracy).
 - Example: A linear regression model trying to fit a highly non-linear dataset or an under regularized model.
 - Poor performance on both training and test data.
- High-complexity models (overfitting):
 - Low bias (can fit training data very well).
 - High variance (small fluctuations in training data lead to large changes in predictions).
 - Example: A deep neural network with too many parameters trained on limited data or an over regularized model.
 - Good training performance but poor test performance.

The **bias-variance tradeoff** states that as complexity increases, bias decreases, but variance increases. The goal is to find the optimal model complexity that balances both.

Complex learners with too many parameters relative to the number of observations may overfit the training dataset.

The Challenges of High Dimensionality

High-dimensional data refers to datasets with a large number of input features (P). In linear models, each feature corresponds to a parameter, so when the number of features P is large compared to the number of training samples N (the “**large P, small N**” problem), the model tends to overfit the training data. This phenomenon is part of the **curse of dimensionality**, which describes the difficulties that arise when working in high-dimensional spaces.

One of the most critical factors in selecting a machine learning algorithm is the relationship between P and N , as it significantly impacts model performance. Below are three key problems associated with high-dimensionality:

Infinite Solutions and Ill-Conditioned Matrices

In linear models, the covariance matrix $\mathbf{X}^T \mathbf{X}$ is of size $P \times P$ and has rank $\min(N, P)$. When $P > N$, the system of equations becomes **overparameterized**, meaning there are infinitely

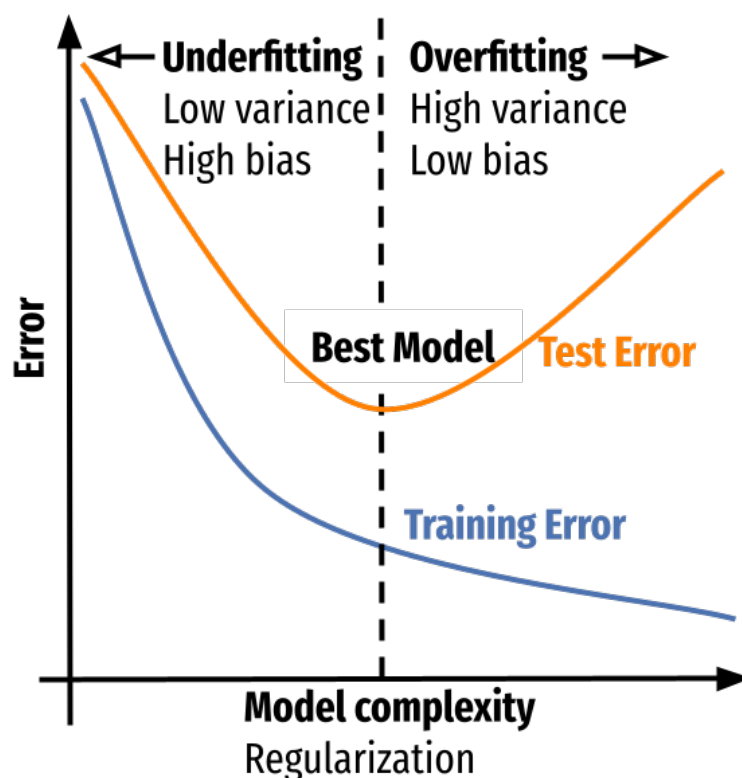


Fig. 1: Model complexity

many possible solutions that fit the training data. This leads to poor generalization, as the learned solutions may be highly specific to the dataset. In such cases, the covariance matrix is singular or ill-conditioned, making it unstable for inversion in methods like ordinary least squares regression.

Exponential Growth of Sample Requirements

The density of data points in a high-dimensional space decreases exponentially with increasing P . Specifically, the effective sampling density of N points in a P -dimensional space is proportional to $N^{1/P}$. As a result, the data becomes increasingly sparse as P grows, making it difficult to estimate distributions or learn meaningful patterns. To maintain a constant density, the number of required samples grows exponentially. For example:

- In **1D**, 50 samples provide reasonable coverage.
- In **2D**, approximately **2,500** samples are needed for equivalent density.
- In **3D**, around **125,000** samples are required.

This illustrates why high-dimensional problems often suffer from a lack of sufficient training data.

Most Data Points Lie on the Edge of the Space

In high-dimensional spaces, most data points are **closer to the boundary of the sample space** than to any other data point. Consider N points uniformly distributed in a P -dimensional unit ball. The median distance from the origin to the nearest neighbor is given by:

$$d(P, N) = \left(1 - \frac{1}{2}^{1/N}\right)^{1/P}$$

For example, with $N = 500$ and $P = 10$, this distance is approximately **0.52**, meaning that most data points are more than halfway to the boundary. This has severe consequences for prediction:

- In lower dimensions, models **interpolate** between data points.
- In high dimensions, models must **extrapolate**, which is significantly harder and less reliable.

This explains why many machine learning algorithms perform poorly in high-dimensional settings and why dimensionality reduction techniques (e.g., PCA, feature selection) are essential.

Conclusion

The curse of dimensionality creates fundamental challenges for machine learning, including overparameterization, data sparsity, and unreliable predictions. Addressing these issues requires strategies such as **dimensionality reduction**, **regularization**, and **feature selection** to ensure that models generalize well and remain computationally efficient.

(Source: T. Hastie, R. Tibshirani, J. Friedman. The Elements of Statistical Learning: Data Mining, Inference, and Prediction.* Second Edition, 2009.)*

Measure of overfitting risk: Vapnik–Chervonenkis (VC) Dimension

The Vapnik–Chervonenkis (VC) dimension is a fundamental concept in statistical learning theory that measures the capacity of a hypothesis class (i.e., the set of functions a model can learn). It provides a way to quantify a model’s ability to fit data and generalize to unseen examples.

VC dimension (for Vapnik–Chervonenkis dimension) is a measure of the **capacity** (complexity, expressive power, richness, or flexibility) of a statistical classification algorithm, defined as the cardinality of the largest set of points that the algorithm can shatter.

Theorem: Linear classifier in R^P have VC dimension of $P + 1$. Hence in dimension two ($P = 2$) any random partition of 3 points can be learned.



Fig. 2: In 2D we can shatter any three non-collinear points

6.2.2 Regularization Approaches to Mitigate Overfitting

Regularization techniques help prevent overfitting by constraining the complexity of machine learning models. Below is a categorized enumeration of common regularization methods, along with their summaries.

Norm-Based Regularization (Penalty Methods)

These techniques add constraints to the model’s parameters to prevent excessive complexity.

- **L2 Regularization (Ridge Regression / Weight Decay / Shrinkage)**
 - Adds a squared penalty: $\lambda \sum w_i^2$.
 - Shrinks weights but does not eliminate them, reducing model sensitivity to noise.
 - Common in linear regression, logistic regression, and deep learning.

- **L1 Regularization (Lasso Regression)**
 - Adds an absolute penalty: $\lambda \sum |w_i|$.
 - Promote sparsity by setting some weights to zero, effectively selecting features.
 - Used in high-dimensional datasets to perform feature selection.
- **Elastic Net Regularization**
 - Combines L1 and L2 penalties: $\lambda_1 \sum |w_i| + \lambda_2 \sum w_i^2$
 - Used when dealing with correlated features.

Ensemble Learning approaches

- **Bagging & Boosting**
 - **Bagging** (e.g., Random Forests) reduces overfitting by averaging multiple models trained on different data subsets.
 - **Boosting** (e.g., XGBoost) adds weak learners sequentially with a learning rate to control overfitting.
 - **Stacking** reduces overfitting by averaging multiple models trained on same data subsets.

Data-Filtering/or preprocessing Regularization

- **Feature Selection**
 - Reduces model complexity by removing redundant or irrelevant features.
 - Methods include univariate filter (SelectKBest) or recursive feature elimination (RFE) and mutual information filtering.
- **Unsupervised Dimension Reduction as preprocessing step**
 - Reduces model complexity by reducing the dimension of the input data.
 - Methods include Linear Dimension reduction or Manifold Learning.
 - Unsupervised approaches are generally not efficient, as they tend to overfill the data before the supervised stage.

Regularization for Probabilistic Models

- **Bayesian Regularization**
 - Introduces priors over model parameters, effectively acting as L2 regularization.
 - Used in Bayesian Neural Networks, Gaussian Processes, and Bayesian Ridge Regression.

Regularization in Kernel Methods (SVM, Gaussian Processes)

- **Margin-Based Regularization (SVM)**
 - The **soft margin** parameter C controls the trade-off between a large margin and misclassification.
 - A smaller C encourages more regularization, preventing overfitting.

- **Kernel Regularization**

- Kernel methods (e.g., Gaussian RBF, polynomial kernels) use hyperparameters like kernel bandwidth to control model complexity.
- A wider kernel bandwidth smooths the decision boundary, reducing variance.

Regularization in Deep Learning

- **Dropout**

- Randomly disables a fraction of neurons during training to reduce reliance on specific features.
- Helps improve generalization in fully connected and convolutional networks.

- **Batch Normalization**

- Normalizes activations across mini-batches, reducing internal covariate shift.
- Acts as an implicit regularizer by smoothing the optimization landscape.

- **Early Stopping**

- Monitors validation loss and stops training when it stops decreasing.
- Prevents the model from overfitting to the training data.

- **Weight Decay (L2 Regularization in Neural Networks)**

- Reduces the magnitude of neural network weights to prevent overfitting.
- Equivalent to L2 regularization in traditional machine learning models.

Data-Centric Regularization

- **Data Augmentation**

- Artificially increases the dataset size using transformations (e.g., rotations, scaling, flipping).
- Particularly useful in image and text processing tasks.

- **Adding Noise to Inputs or Weights**

- Introduces small random noise to training data or network weights to improve robustness.
- Common in deep learning and reinforcement learning.

Regularization for Tree-Based Models

- **Pruning (Decision Trees, Random Forests, Gradient Boosting)**

- Removes branches that have low importance to reduce complexity.
- Prevents trees from memorizing noise.

Summary

Category	Method	Description
Norm-Based Regularization	L2 Regularization (Ridge, Weight Decay, Shrinkage)	Adds a squared penalty on coefficients; shrinks weights but does not eliminate them, reducing noise sensitivity.
	L1 Regularization (Lasso)	Adds an absolute penalty on coefficients; promotes sparsity by setting some weights to zero (feature selection).
	Elastic Net	Combines L1 and L2 penalties; useful for correlated features.
Ensemble Learning Regularization	Bagging	Reduces overfitting by averaging multiple models trained on different subsets (e.g., Random Forests).
	Boosting	Sequentially adds weak learners with a learning rate to control overfitting (e.g., XGBoost).
	Stacking	Combines multiple models trained on the same data and uses a meta-learner for final predictions.
Data-Filtering / Preprocessing Regularization	Feature Selection	Reduces model complexity by removing redundant or irrelevant features (e.g., SelectKBest, RFE).
	Unsupervised Dimension Reduction	Reduces data dimensionality before modeling (e.g., PCA, Manifold Learning); less efficient for supervised learning.
Regularization for Probabilistic Models	Bayesian Regularization	Introduces priors over parameters, acting as L2 regularization (e.g., Bayesian Ridge Regression, Gaussian Processes).
Regularization in Kernel Methods	Margin-Based Regularization (SVM)	Soft margin parameter (C) controls the trade-off between margin size and misclassification.
	Kernel Regularization	Kernel hyperparameters (e.g., RBF bandwidth) control decision boundary complexity.
Regularization in Deep Learning	Dropout	Randomly disables neurons during training to reduce reliance on specific features.
	Batch Normalization	Normalizes activations across mini-batches, reducing internal covariate shift and smoothing optimization.
	Early Stopping	Stops training when validation loss stops improving to prevent overfitting.
	Weight Decay (L2 in Deep Learning)	Applies L2 regularization to neural network weights.
Data-Centric Regularization	Data Augmentation	Increases dataset size using transformations (e.g., rotations, scaling, flipping) to improve generalization.
	Adding Noise	Introduces noise to inputs or weights to improve model robustness.
Regularization for Tree-Based Models	Pruning	Removes low-importance branches in decision trees and ensemble models to reduce complexity.

UNSUPERVISED LEARNING

7.1 Linear Dimensionality Reduction and Feature Extraction

In machine learning and statistics, dimensionality reduction or dimension reduction is the process of reducing the number of features under consideration, and can be divided into feature selection (not addressed here) and feature extraction.

Feature extraction starts from an initial set of measured data and builds derived values (features) intended to be informative and non-redundant, facilitating the subsequent learning and generalization steps, and in some cases leading to better human interpretations. Feature extraction is related to dimensionality reduction.

Objective: Given a input matrix $\mathbf{X}$, of dimensions $N \times P$, where the rows represent the samples and columns represent the variables. The goal is to learn a linear transformation $\mathbf{V}$ (of dimension $P \times K$), with $K < P$ principle directions, transforming the P -dimensional data into lower K -dimensional space, leading to the transformed dataset $\mathbf{C}$ (of dimension $N \times K$):

$$\begin{bmatrix} c_{11} & \dots & c_{1K} \\ \vdots & \mathbf{C} & \vdots \\ c_{N1} & \dots & c_{NK} \end{bmatrix}_{N \times K} = \begin{bmatrix} x_{11} & \dots & \dots & \dots & x_{1P} \\ \vdots & & \mathbf{X} & & \vdots \\ x_{N1} & \dots & \dots & \dots & x_{NP} \end{bmatrix}_{N \times P} \begin{bmatrix} v_{11} & \dots & v_{1K} \\ \vdots & & \mathbf{V} & & \vdots \\ v_{P1} & \dots & v_{NK} \end{bmatrix}_{P \times K}$$

Keys aspects of linear dimension reduction (or matrix decomposition/factorization):

- Find new axes (components) that best describe the variance or structure of the data.
- Exploit the covariance $\Sigma_{\mathbf{X}\mathbf{X}}$ between the input features.
- Dimension reduction is obtained by a rotation (linear transformation) in the input space.
- SVD or PCA find orthogonal directions of maximum variance.

7.1.1 Principal components analysis (PCA)

Sources:

- [PCA with scikit-learn](#)
- C. M. Bishop *Pattern Recognition and Machine Learning*, Springer, 2006
- [Everything you did and didn't know about PCA](#)
- [Principal Component Analysis in 3 Simple Steps](#)

Principles

- Principal components analysis is the main method used for linear dimension reduction.
- The idea of principal component analysis is to find the K **principal components directions** (called the **loadings**) $\mathbf{V}_{K \times P}$ that capture the variation in the data as much as possible.
- It converts a set of N P -dimensional observations $\mathbf{N}_{N \times P}$ of possibly correlated variables into a set of N K -dimensional samples $\mathbf{C}_{N \times K}$, where the $K < P$. The new variables are linearly uncorrelated. The columns of $\mathbf{C}_{N \times K}$ are called the **principal components**.
- The dimension reduction is obtained by using only $K < P$ components that exploit correlation (covariance) among the original variables.
- PCA is mathematically defined as an orthogonal linear transformation $\mathbf{V}_{K \times P}$ that transforms the data to a new coordinate system such that the greatest variance by some projection of the data comes to lie on the first coordinate (called the first principal component), the second greatest variance on the second coordinate, and so on.

$$\mathbf{C}_{N \times K} = \mathbf{X}_{N \times P} \mathbf{V}_{P \times K}$$

- PCA can be thought of as fitting a P -dimensional ellipsoid to the data, where each axis of the ellipsoid represents a principal component. If some axis of the ellipse is small, then the variance along that axis is also small, and by omitting that axis and its corresponding principal component from our representation of the dataset, we lose only a commensurately small amount of information.
- Finding the K largest axes of the ellipse will permit to project the data onto a space having dimensionality $K < P$ while maximizing the variance of the projected data.

```
import numpy as np
import scipy
from sklearn.decomposition import PCA

# Plot
import matplotlib.pyplot as plt
import seaborn as sns

# Plot parameters
plt.style.use('seaborn-v0_8-whitegrid')
fig_w, fig_h = plt.rcParams.get('figure.figsize')
plt.rcParams['figure.figsize'] = (fig_w, fig_h * .5)
colors = sns.color_palette()

# dataset
np.random.seed(42)
n_samples = 100
experience = np.random.normal(size=n_samples)
salary = 1500 + experience + np.random.normal(size=n_samples, scale=.5)
X = np.column_stack([experience, salary])
print("Input features Std-dev.:", X.std(axis=0))
print("Input features mean:", X.mean(axis=0))
```

(continues on next page)

(continued from previous page)

```

# PCA with scikit-learn
pca = PCA(n_components=2)
pca.fit(X)
print("Explained variance ratios:", pca.explained_variance_ratio_)

PC = pca.transform(X)

plt.subplot(121)
plt.scatter(X[:, 0], X[:, 1])
V = pca.components_ # (n_components, n_features)
print("Principle Directions:\n", V)

m = pca.mean_

for i in range(V.shape[0]):
    plt.arrow(x=m[0], y=m[1], dx=V[i, 0], dy=V[i, 1], head_width=0.2,
              head_length=0.2, linewidth=2, fc='r', ec='r')
    plt.text(m[0] + V[i, 0], m[1] + V[i, 1], 'v%i' % (i+1), color="r", fontsize=15,
             horizontalalignment='left', verticalalignment='top')

plt.axis('equal')
plt.xlabel("x1"); plt.ylabel("x2")

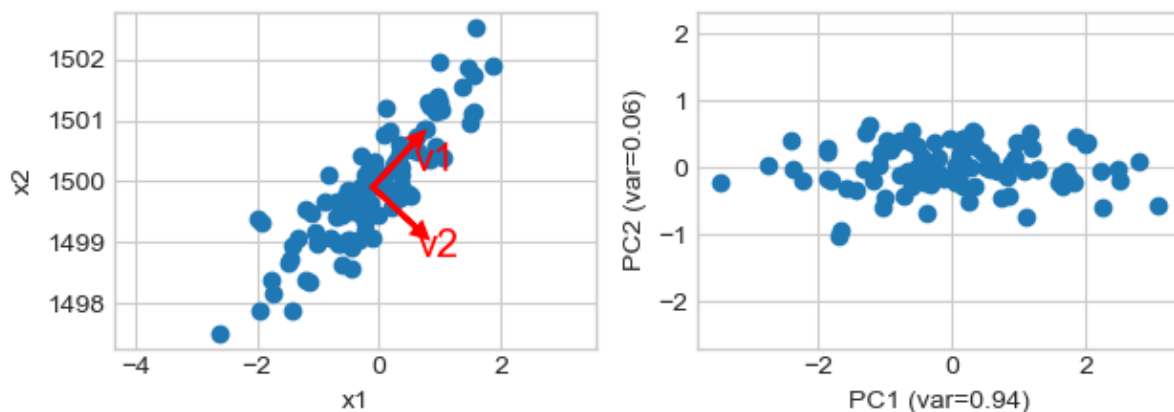
plt.subplot(122)
plt.scatter(PC[:, 0], PC[:, 1])
plt.xlabel("PC1 (var=%.2f)" % pca.explained_variance_ratio_[0])
plt.ylabel("PC2 (var=%.2f)" % pca.explained_variance_ratio_[1])
plt.axis('equal')
plt.tight_layout()

```

```

Input features Std-dev.: [0.90361618 0.96158565]
Input features mean: [-1.03846517e-01 1.49990731e+03]
Explained variance ratios: [0.93646607 0.06353393]
Principle Directions:
[[ 0.68149162  0.73182591]
 [ 0.73182591 -0.68149162]]

```



Dataset preprocessing

- **Centering:** Consider a data matrix, $\mathbf{X}$, with column-wise zero empirical mean (the sample mean of each column has been shifted to zero), ie. $\mathbf{X}$ is replaced by $\mathbf{X} - \mathbf{1}\bar{\mathbf{x}}^\top$.
- **Standardizing:** Optionally, standardize the columns, i.e., scale them by their standard-deviation. Without standardization, a variable with a high variance will capture most of the effect of the PCA. The principal direction will be aligned with this variable. Standardization will, however, raise noise variables to the same level as informative variables. The covariance matrix of centered standardized data is the correlation matrix.

Projection in the directions that capture the greatest variance

To begin with, consider the projection onto a one-dimensional space ($K = 1$). We can define the direction of this space using a P -dimensional vector $\mathbf{v}$, which for convenience (and without loss of generality) we shall choose to be a unit vector so that $\|\mathbf{v}\|_2 = 1$ (note that we are only interested in the direction defined by $\mathbf{v}$, not in the magnitude of $\mathbf{v}$ itself). PCA consists of two main steps:

Each P -dimensional data point $\mathbf{x}_i$ is then projected onto $\mathbf{v}$, where the coordinate (in the coordinate system of $\mathbf{v}$) is a scalar value, namely $\mathbf{x}_i^\top \mathbf{v}$. I.e., we want to find the vector $\mathbf{v}$ that maximizes these coordinates along $\mathbf{v}$, which we will see corresponds to maximizing the variance of the projected data. This is equivalently expressed as

$$\mathbf{v} = \arg \max_{\|\mathbf{v}\|=1} \frac{1}{N} \sum_i (\mathbf{x}_i^\top \mathbf{v})^2.$$

Mathematical details

We can write this in matrix form as

$$\mathbf{v} = \arg \max_{\|\mathbf{v}\|=1} \frac{1}{N} \|\mathbf{X}\mathbf{v}\|^2 = \frac{1}{N} \mathbf{v}^\top \mathbf{X}^\top \mathbf{X} \mathbf{v} = \mathbf{v}^\top \mathbf{S}_{\mathbf{X}\mathbf{X}} \mathbf{v},$$

where $\mathbf{S}_{\mathbf{X}\mathbf{X}}$ is a biased estimate of the covariance matrix of the data, i.e.

$$\mathbf{S}_{\mathbf{X}\mathbf{X}} = \frac{1}{N} \mathbf{X}^\top \mathbf{X}.$$

We now maximize the projected variance $\mathbf{v}^\top \mathbf{S}_{\mathbf{X}\mathbf{X}} \mathbf{v}$ with respect to $\mathbf{v}$. Clearly, this has to be a constrained maximization to prevent $\|\mathbf{v}\|_2 \rightarrow \infty$. The appropriate constraint comes from the normalization condition $\|\mathbf{v}\|_2 \equiv \|\mathbf{v}\|_2^2 = \mathbf{v}^\top \mathbf{v} = 1$. To enforce this constraint, we introduce a [Lagrange multiplier](#) that we shall denote by λ , and then make an unconstrained maximization of

$$\mathbf{v}^\top \mathbf{S}_{\mathbf{X}\mathbf{X}} \mathbf{v} - \lambda(\mathbf{v}^\top \mathbf{v} - 1).$$

By setting the gradient with respect to $\mathbf{v}$ equal to zero, we see that this quantity has a stationary point when

$$\mathbf{S}_{\mathbf{X}\mathbf{X}} \mathbf{v} = \lambda \mathbf{v}.$$

We note that $\mathbf{v}$ is an eigenvector of $\mathbf{S}_{\mathbf{X}\mathbf{X}}$.

If we left-multiply the above equation by $\mathbf{v}^\top$ and make use of $\mathbf{v}^\top \mathbf{v} = 1$, we see that the variance is given by

$$\mathbf{v}^\top \mathbf{S}_{\mathbf{X}\mathbf{X}} \mathbf{v} = \lambda,$$

and so the variance will be at a maximum when $\mathbf{v}$ is equal to the eigenvector corresponding to the largest eigenvalue, λ . This eigenvector is known as the first principal component.

We can define additional principal components in an incremental fashion by choosing each new direction to be that which maximizes the projected variance amongst all possible directions that are orthogonal to those already considered. If we consider the general case of a K -dimensional projection space, the optimal linear projection for which the variance of the projected data is maximized is now defined by the K eigenvectors, $\mathbf{v}_1, \dots, \mathbf{v}_K$, of the data covariance matrix $\mathbf{S}_{\mathbf{X}\mathbf{X}}$ that corresponds to the K largest eigenvalues, $\lambda_1 \geq \lambda_2 \geq \dots \geq \lambda_K$.

Implementation with Singular Value Decomposition (SVD)

The sample covariance matrix of **centered data** $\mathbf{X}$ is given by

$$\mathbf{S}_{\mathbf{X}\mathbf{X}} = \frac{1}{N-1} \mathbf{X}^T \mathbf{X}.$$

We rewrite $\mathbf{X}^T \mathbf{X}$ using the SVD decomposition of $\mathbf{X}$ as

$$\begin{aligned} \mathbf{X}^T \mathbf{X} &= (\mathbf{U} \mathbf{D} \mathbf{V}^T)^T (\mathbf{U} \mathbf{D} \mathbf{V}^T) \\ &= \mathbf{V} \mathbf{D}^T \mathbf{U}^T \mathbf{U} \mathbf{D} \mathbf{V}^T \\ &= \mathbf{V} \mathbf{D}^2 \mathbf{V}^T \\ \mathbf{V}^T \mathbf{X}^T \mathbf{X} \mathbf{V} &= \mathbf{D}^2 \\ \frac{1}{N-1} \mathbf{V}^T \mathbf{X}^T \mathbf{X} \mathbf{V} &= \frac{1}{N-1} \mathbf{D}^2 \\ \mathbf{V}^T \mathbf{S}_{\mathbf{X}\mathbf{X}} \mathbf{V} &= \frac{1}{N-1} \mathbf{D}^2. \end{aligned}$$

Considering only the k^{th} right-singular vectors $\mathbf{v}_k$ associated to the singular value d_k

$$\mathbf{v}_k^T \mathbf{S}_{\mathbf{X}\mathbf{X}} \mathbf{v}_k = \frac{1}{N-1} d_k^2,$$

It turns out that if you have done the singular value decomposition then you already have the Eigenvalue decomposition for $\mathbf{X}^T \mathbf{X}$. Where - The eigenvectors of $\mathbf{S}_{\mathbf{X}\mathbf{X}}$ are equivalent to the right singular vectors, $\mathbf{V}$, of $\mathbf{X}$. - The eigenvalues, λ_k , of $\mathbf{S}_{\mathbf{X}\mathbf{X}}$, i.e. the variances of the components, are equal to $\frac{1}{N-1}$ times the squared singular values, d_k .

Moreover computing PCA with SVD do not require to form the matrix $\mathbf{X}^T \mathbf{X}$, so computing the SVD is now the standard way to calculate a principal components analysis from a data matrix, unless only a handful of components are required.

PCA outputs

The SVD or the eigendecomposition of the data covariance matrix provides three main quantities:

1. **Principal component directions** or **loadings** are the **eigenvectors** of $\mathbf{X}^T \mathbf{X}$. The $\mathbf{V}_{K \times P}$ or the **right-singular vectors** of an SVD of $\mathbf{X}$ are called principal component directions of $\mathbf{X}$. They are generally computed using the SVD of $\mathbf{X}$.
2. **Principal components** is the $N \times K$ matrix $\mathbf{C}$ which is obtained by projecting $\mathbf{X}$ onto the principal components directions, i.e.

$$\mathbf{C}_{N \times K} = \mathbf{X}_{N \times P} \mathbf{V}_{P \times K}.$$

Since $\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^T$ and $\mathbf{V}$ is orthogonal ($\mathbf{V}^T\mathbf{V} = \mathbf{I}$):

$$\begin{aligned}\mathbf{C}_{N \times K} &= \mathbf{U}\mathbf{D}\mathbf{V}_{N \times P}^T \mathbf{V}_{P \times K} \\ \mathbf{C}_{N \times K} &= \mathbf{U}\mathbf{D}_{N \times K}^T \mathbf{I}_{K \times K} \\ \mathbf{C}_{N \times K} &= \mathbf{U}\mathbf{D}_{N \times K}^T\end{aligned}$$

Thus $\mathbf{c}_j = \mathbf{X}\mathbf{v}_j = \mathbf{u}_j d_j$, for $j = 1, \dots, K$. Hence $\mathbf{u}_j$ is simply the projection of the row vectors of $\mathbf{X}$, i.e., the input predictor vectors, on the direction $\mathbf{v}_j$, scaled by d_j .

$$\mathbf{c}_1 = \begin{bmatrix} x_{1,1}v_{1,1} + \dots + x_{1,P}v_{1,P} \\ x_{2,1}v_{1,1} + \dots + x_{2,P}v_{1,P} \\ \vdots \\ x_{N,1}v_{1,1} + \dots + x_{N,P}v_{1,P} \end{bmatrix}$$

3. The **variance** of each component is given by the eigen values $\lambda_k, k = 1, \dots, K$. It can be obtained from the singular values:

$$\begin{aligned}var(\mathbf{c}_k) &= \frac{1}{N-1} (\mathbf{X}\mathbf{v}_k)^2 \\ &= \frac{1}{N-1} (\mathbf{u}_k d_k)^2 \\ &= \frac{1}{N-1} d_k^2\end{aligned}$$

Determining the number of PCs

We must choose $K^* \in [1, \dots, K]$, the number of required components. This can be done by calculating the explained variance ratio of the K^* first components and by choosing K^* such that the **cumulative explained variance** ratio is greater than some given threshold (e.g., $\approx 90\%$). This is expressed as

$$\text{cumulative explained variance}(\mathbf{c}_k) = \frac{\sum_j^{K^*} var(\mathbf{c}_k)}{\sum_j^K var(\mathbf{c}_k)}.$$

Interpretation and visualization

- **PCs** Plot the samples projected on first the principal components as e.g. PC1 against PC2.
- **PC directions** Exploring the loadings associated with a component provides the contribution of each original variable in the component. Remark: The loadings (PC directions) are the coefficients of multiple regression of PC on original variables:

$$\begin{aligned}\mathbf{c} &= \mathbf{X}\mathbf{v} \\ \mathbf{X}^T \mathbf{c} &= \mathbf{X}^T \mathbf{X} \mathbf{v} \\ (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{c} &= \mathbf{v}\end{aligned}$$

Another way to evaluate the contribution of the original variables in each PC can be obtained by computing the correlation between the PCs and the original variables, i.e. columns of $\mathbf{X}$, denoted $\mathbf{x}_j$, for $j = 1, \dots, P$. For the k^{th} PC, compute and plot the correlations with all original variables

$$cor(\mathbf{c}_k, \mathbf{x}_j), j = 1 \dots K, j = 1 \dots K.$$

These quantities are sometimes called the *correlation loadings*.

7.1.2 Singular value decomposition and matrix factorization

Sources: [Matrix factorization problems with scikit-learn](#)

Matrix factorization principles

Decompose the data matrix $\mathbf{X}_{N \times P}$ into a product of a mixing matrix $\mathbf{U}_{N \times K}$ and a dictionary matrix $\mathbf{V}_{P \times K}$.

$$\mathbf{X} = \mathbf{U}\mathbf{V}^T,$$

If we consider only a subset of components $K < \text{rank}(\mathbf{X}) < \min(P, N - 1)$, $\mathbf{X}$ is approximated by a matrix $\hat{\mathbf{X}}$:

$$\mathbf{X} \approx \hat{\mathbf{X}} = \mathbf{U}\mathbf{V}^T,$$

Each line of $\mathbf{x}_i$ is a linear combination (mixing $\mathbf{u}_i$) of dictionary items $\mathbf{V}$.

N P -dimensional data points lie in a space whose dimension is less than $N - 1$ (2 dots lie on a line, 3 on a plane, etc.).

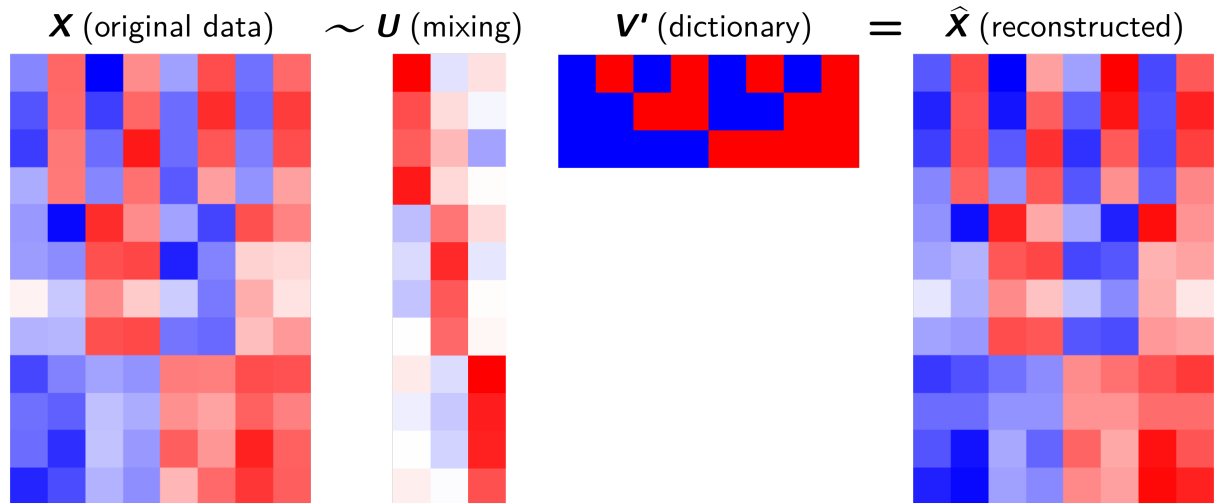


Fig. 1: Matrix factorization

Singular value decomposition (SVD) principles

Singular-value decomposition (SVD) factorises the data matrix $\mathbf{X}_{N \times P}$ into a product:

$$\mathbf{X} = \mathbf{U}\mathbf{D}\mathbf{V}^T,$$

where

$$\begin{bmatrix} x_{11} & & x_{1P} \\ & \mathbf{X} & \\ x_{N1} & & x_{NP} \end{bmatrix} = \begin{bmatrix} u_{11} & & u_{1K} \\ & \mathbf{U} & \\ u_{N1} & & u_{NK} \end{bmatrix} \begin{bmatrix} d_1 & & 0 \\ & \mathbf{D} & \\ 0 & & d_K \end{bmatrix} \begin{bmatrix} v_{11} & & v_{1P} \\ & \mathbf{V}^T & \\ v_{K1} & & v_{KP} \end{bmatrix}.$$

U: right-singular

- $\mathbf{V} = [\mathbf{v}_1, \dots, \mathbf{v}_K]$ is a $P \times K$ orthogonal matrix.

- It is a **dictionary** of patterns to be combined (according to the mixing coefficients) to reconstruct the original samples.
- $\mathbf{V}$ performs the initial **rotations (projection)** along the $K = \min(N, P)$ **principal component directions**, also called **loadings**.
- Each $\mathbf{v}_j$ performs the linear combination of the variables that has maximum sample variance, subject to being uncorrelated with the previous $\mathbf{v}_{j-1}$.

D: singular values

- $\mathbf{D}$ is a $K \times K$ diagonal matrix made of the singular values of $\mathbf{X}$ with $d_1 \geq d_2 \geq \dots \geq d_K \geq 0$.
- $\mathbf{D}$ scale the projection along the coordinate axes by $d_1, d_2, \dots, d_K$.
- Singular values are the square roots of the eigenvalues of $\mathbf{X}^T \mathbf{X}$.

V: left-singular vectors

- $\mathbf{U} = [\mathbf{u}_1, \dots, \mathbf{u}_K]$ is an $N \times K$ orthogonal matrix.
- Each row $\mathbf{v}_i$ provides the **mixing coefficients** of dictionary items to reconstruct the sample $\mathbf{x}_i$
- It may be understood as the coordinates on the new orthogonal basis (obtained after the initial rotation) called **principal components** in the PCA.

SVD for variables transformation

$\mathbf{V}$ transforms correlated variables ($\mathbf{X}$) into a set of uncorrelated ones ($\mathbf{UD}$) that better expose the various relationships among the original data items.

$$\begin{aligned}\mathbf{X} &= \mathbf{UDV}^T, \\ \mathbf{XV} &= \mathbf{UDV}^T \mathbf{V}, \\ \mathbf{XV} &= \mathbf{UD}, \\ \mathbf{XV} &= \mathbf{UD}\end{aligned}$$

At the same time, SVD is a method for identifying and ordering the dimensions along which data points exhibit the most variation.

PCA using SVD

```
X -= X.mean(axis=0) # Centering is required
U, s, Vh = scipy.linalg.svd(X, full_matrices=False)
# U : Unitary matrix having left singular vectors as columns.
#     Of shape (n_samples,n_samples) or (n_samples,n_comps), depending on
#     full_matrices.
#
# s : The singular values, sorted in non-increasing order. Of shape (n_comps,),
#     with n_comps = min(n_samples, n_features).
#
# Vh: Unitary matrix having right singular vectors as rows.
#     Of shape (n_features, n_features) or (n_comps, n_features) depending
#     on full_matrices.
```

(continues on next page)

(continued from previous page)

```

plt.figure(figsize=(9, 3))

# Original data with principle directions
plt.subplot(131)
plt.scatter(X[:, 0], X[:, 1], s=50)
for i in range(Vh.shape[0]):
    plt.arrow(x=0, y=0, dx=Vh[i, 0], dy=Vh[i, 1], head_width=0.2,
              head_length=0.2, linewidth=2, fc='r', ec='r')
    plt.text(Vh[i, 0], Vh[i, 1], 'v%i' % (i+1), color="r", fontsize=15,
             horizontalalignment='right', verticalalignment='top')
plt.axis('equal')
plt.ylim(-4, 4)

plt.title("X: original data (v1, v2:PC dir.)")
plt.xlabel("experience")
plt.ylabel("salary")

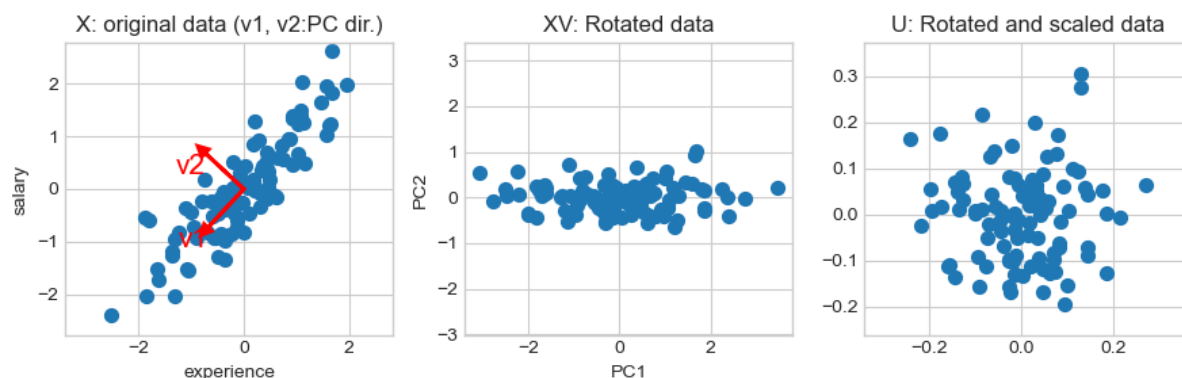
# Project data onto principle directions
plt.subplot(132)
PC = np.dot(X, Vh.T)
plt.scatter(PC[:, 0], PC[:, 1], s=50)
plt.axis('equal')
plt.title("XV: Rotated data")
plt.xlabel("PC1")
plt.ylabel("PC2")

# Rotated and scaled data
plt.subplot(133)
plt.scatter(U[:, 0], U[:, 1], s=50)
plt.axis('equal')
plt.title("U: Rotated and scaled data")

plt.tight_layout()

```

Ignoring fixed y limits to fulfill fixed data aspect **with** adjustable data limits.



7.1.3 Eigen faces

Sources: [Scikit learn Faces decompositions](#)

Load data

```
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_olivetti_faces
from sklearn import decomposition

n_row, n_col = 2, 3
n_components = n_row * n_col
image_shape = (64, 64)

faces, _ = fetch_olivetti_faces(return_X_y=True, shuffle=True,
                                random_state=1)
n_samples, n_features = faces.shape

# Utils function
def plot_gallery(title, images, n_col=n_col, n_row=n_row, cmap=plt.cm.gray):
    plt.figure(figsize=(2. * n_col, 2.26 * n_row))
    plt.suptitle(title, size=16)
    for i, comp in enumerate(images):
        plt.subplot(n_row, n_col, i + 1)
        vmax = max(comp.max(), -comp.min())
        plt.imshow(comp.reshape(image_shape), cmap=cmap,
                    interpolation='nearest',
                    vmin=-vmax, vmax=vmax)
        plt.xticks(())
        plt.yticks(())
    plt.subplots_adjust(0.01, 0.05, 0.99, 0.93, 0.04, 0.)
```

Preprocessing

```
# global centering
faces_centered = faces - faces.mean(axis=0)

# local centering
faces_centered -= faces_centered.mean(axis=1).reshape(n_samples, -1)
```


First centered Olivetti faces

```
plot_gallery("First centered Olivetti faces", faces_centered[:n_components])
```

```
pca = decomposition.PCA(n_components=n_components)
pca.fit(faces_centered)
plot_gallery("PCA first %i loadings" % n_components, pca.components_[:n_
↪ components])
```

7.1.4 Exercises

Write a basic PCA class

Write a class BasicPCA with two methods:

- `fit(X)` that estimates the data mean, principal components directions $\mathbf{V}$ and the explained variance of each component.
- `transform(X)` that projects the data onto the principal components.

Check that your BasicPCA gave similar results, compared to the results from sklearn.

Apply your Basic PCA on the iris dataset

Get [iris.csv](#).

- Describe the data set. Should the dataset been standardized?
- Describe the structure of correlations among variables.
- Compute a PCA with the maximum number of components.
- Compute the cumulative explained variance ratio. Determine the number of components K by your computed values.
- Print the K principal components directions and correlations of the K principal components with the original variables. Interpret the contribution of the original variables into the PC.
- Plot the samples projected into the K first PCs.
- Color samples by their species.

Run scikit-learn examples

Load the notebook or python file at the end of each examples

- [Faces dataset decompositions](#)
- [Faces recognition example using eigenfaces and SVMs](#)

7.2 Manifold learning: non-linear dimension reduction

Sources:

- [Scikit-learn documentation](#)
- [Wikipedia](#)

Nonlinear dimensionality reduction or **manifold learning** cover unsupervised methods that attempt to identify low-dimensional manifolds within the original P -dimensional space that represent high data density. Then those methods provide a mapping from the high-dimensional space to the low-dimensional embedding.

7.2.1 Multi-dimensional Scaling (MDS)

Resources:

- [wikipedia](#)
- Hastie, Tibshirani and Friedman (2009). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. New York: Springer, Second Edition.

The purpose of MDS is to find a low-dimensional projection of the data in which the pairwise distances between data points is preserved, as closely as possible (in a least-squares sense).

- Let $\mathbf{D}$ be the $(N \times N)$ pairwise distance matrix where d_{ij} is a distance between points i and j .
- The MDS concept can be extended to a wide variety of data types specified in terms of a similarity matrix.

Given the dissimilarity (distance) matrix $\mathbf{D}_{N \times N} = [d_{ij}]$, MDS attempts to find K -dimensional projections of the N points $\mathbf{x}_1, \dots, \mathbf{x}_N \in \mathbb{R}^K$, concatenated in an $\mathbf{X}_{N \times K}$ matrix, so that $d_{ij} \approx \|\mathbf{x}_i - \mathbf{x}_j\|$ are as close as possible. This can be obtained by the minimization of a loss function called the **stress function**

$$\text{stress}(\mathbf{X}) = \sum_{i \neq j} (d_{ij} - \|\mathbf{x}_i - \mathbf{x}_j\|)^2.$$

This loss function is known as *least-squares* or *Kruskal-Shepard* scaling.

A modification of *least-squares* scaling is the *Sammon mapping*

$$\text{stress}_{\text{Sammon}}(\mathbf{X}) = \sum_{i \neq j} \frac{(d_{ij} - \|\mathbf{x}_i - \mathbf{x}_j\|)^2}{d_{ij}}.$$

The Sammon mapping performs better at preserving small distances compared to the *least-squares* scaling.

Classical multidimensional scaling

Also known as *principal coordinates analysis*, PCoA.

- The distance matrix, $\mathbf{D}$, is transformed to a *similarity matrix*, $\mathbf{B}$, often using centered inner products.
- The loss function becomes

$$\text{stress}_{\text{classical}}(\mathbf{X}) = \sum_{i \neq j} (b_{ij} - \langle \mathbf{x}_i, \mathbf{x}_j \rangle)^2.$$

- The stress function in classical MDS is sometimes called *strain*.
- The solution for the classical MDS problems can be found from the eigenvectors of the similarity matrix.
- If the distances in $\mathbf{D}$ are Euclidean and double centered inner products are used, the results are equivalent to PCA.

Example

The eurodist dataset provides the road distances (in kilometers) between 21 cities in Europe. Given this matrix of pairwise (non-Euclidean) distances $\mathbf{D} = [d_{ij}]$, MDS can be used to recover the coordinates of the cities in *some* Euclidean referential whose orientation is arbitrary.

```
import pandas as pd
import numpy as np

# Plot
import matplotlib.pyplot as plt
import seaborn as sns
import pystatsml.plot_utils

# Plot parameters
plt.style.use('seaborn-v0_8-whitegrid')
fig_w, fig_h = plt.rcParams.get('figure.figsize')
plt.rcParams['figure.figsize'] = (fig_w, fig_h * .5)
%matplotlib inline

np.random.seed(42)

# Pairwise distance between European cities
try:
    url = '../datasets/eurodist.csv'
    df = pd.read_csv(url)
except:
    url = 'https://github.com/duchesnay/pystatsml/raw/master/datasets/eurodist.csv'
    df = pd.read_csv(url)

print(df.iloc[:5, :5])

city = df["city"]
D = np.array(df.iloc[:, 1:]) # Distance matrix

# Arbitrary choice of K=2 components
from sklearn.manifold import MDS
mds = MDS(dissimilarity='precomputed', n_components=2, random_state=40, max_
    iter=3000, eps=1e-9)
X = mds.fit_transform(D)
```

	city	Athens	Barcelona	Brussels	Calais
0	Athens	0	3313	2963	3175
1	Barcelona	3313	0	1318	1326
2	Brussels	2963	1318	0	204
3	Calais	3175	1326	204	0
4	Cherbourg	3339	1294	583	460

Recover coordinates of the cities in Euclidean referential whose orientation is arbitrary:

```
from sklearn import metrics
Deuclidean = metrics.pairwise.pairwise_distances(X, metric='euclidean')
print(np.round(Deuclidean[:5, :5]))
```

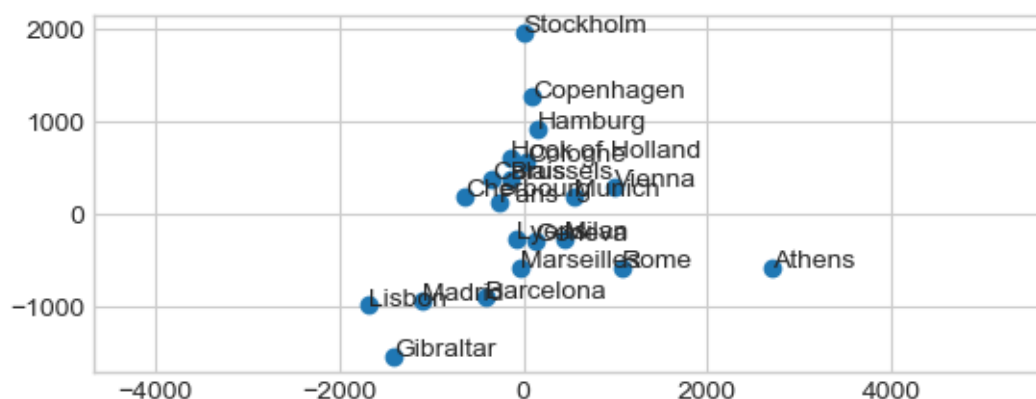
```
[[ 0. 3116. 2994. 3181. 3428.]
 [3116. 0. 1317. 1289. 1128.]
 [2994. 1317. 0. 198. 538.]
 [3181. 1289. 198. 0. 358.]
 [3428. 1128. 538. 358. 0.]]
```

Plot the results:

```
# Plot: apply some rotation and flip
theta = 80 * np.pi / 180.
rot = np.array([[np.cos(theta), -np.sin(theta)],
                [np.sin(theta), np.cos(theta)]])
Xr = np.dot(X, rot)
# flip x
Xr[:, 0] *= -1
plt.scatter(Xr[:, 0], Xr[:, 1])

for i in range(len(city)):
    plt.text(Xr[i, 0], Xr[i, 1], city[i])
plt.axis('equal')
```

```
(np.float64(-1894.091917806915),
 np.float64(2914.3554370871243),
 np.float64(-1712.973369719749),
 np.float64(2145.4370687880146))
```



Determining the number of components

We must choose $K^* \in \{1, \dots, K\}$ the number of required components. Plotting the values of the stress function, obtained using $k \leq N - 1$ components. In general, start with $1, \dots, K \leq 4$. Choose K^* where you can clearly distinguish an *elbow* in the stress curve.

Thus, in the plot below, we choose to retain information accounted for by the first *two* components, since this is where the *elbow* is in the stress curve.

```

k_range = range(1, min(5, D.shape[0]-1))
stress = [MDS(dissimilarity='precomputed', n_components=k,
              random_state=42, max_iter=300, eps=1e-9).fit(D).stress_ for k in k_
↪range]

print(stress)
plt.plot(k_range, stress)
plt.xlabel("k")
plt.ylabel("stress")

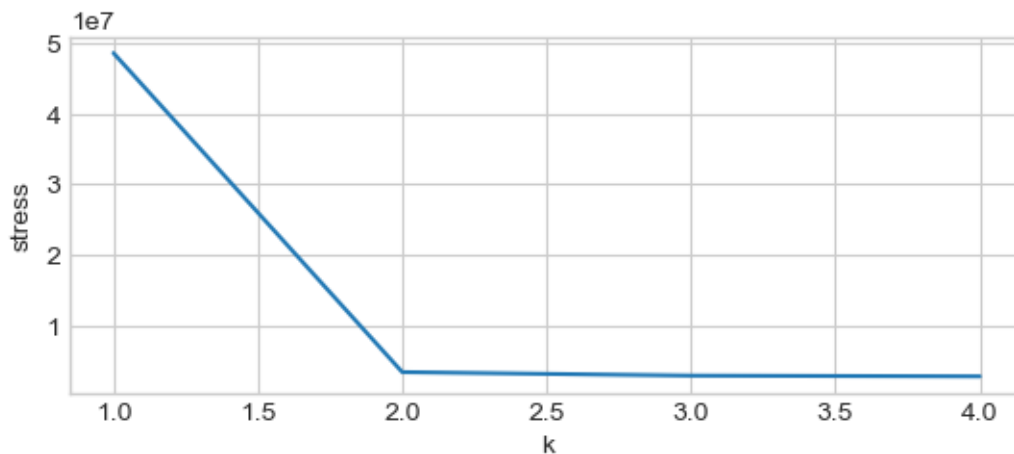
```

```

[np.float64(48644495.28571428), np.float64(3356497.365752386), np.float64(2858455.
↪495887962), np.float64(2756310.6376280114)]

```

```
Text(0, 0.5, 'stress')
```



Exercises

Apply MDS from sklearn on the iris dataset available at:

<https://github.com/duchesnay/pystatsml/raw/master/datasets/iris.csv>

- Center and scale the dataset.
- Compute Euclidean pairwise distances matrix.
- Select the number of components.
- Show that classical MDS on Euclidean pairwise distances matrix is equivalent to PCA.

Manifold learning

Dataset S curve:

```

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from sklearn import manifold, datasets

X, color = datasets.make_s_curve(1000, random_state=42)

```

7.2.2 Isomap

Isomap is a nonlinear dimensionality reduction method that combines a procedure to compute the distance matrix with MDS. The distances calculation is based on geodesic distances evaluated on neighborhood graph:

1. Determine the neighbors of each point. All points in some fixed radius or K nearest neighbors.
2. Construct a neighborhood graph. Each point is connected to other if it is a K nearest neighbor. Edge length equal to Euclidean distance.
3. Compute shortest path between pairwise of points d_{ij} to build the distance matrix $\mathbf{D}$.
4. Apply MDS on $\mathbf{D}$.

```
isomap = manifold.Isomap(n_neighbors=10, n_components=2)
X_isomap = isomap.fit_transform(X)
```

7.2.3 t-SNE

Sources:

- [Wikipedia](#)
- [scikit-learn](#)

Principles

1. Construct a (Gaussian) probability distribution between pairs of object in input (high-dimensional) space.
2. Construct a (student) probability distribution between pairs of object in embedded (low-dimensional) space.
3. Minimize the Kullback–Leibler divergence (KL divergence) between the two distributions.

Features

- Isomap, LLE and variants are best suited to unfold a single continuous low dimensional manifold
- t-SNE will focus on the **local structure** of the data and will tend to extract clustered **local groups** of samples

```
tsne = manifold.TSNE(n_components=2, init='pca', random_state=0)
X_tsne = tsne.fit_transform(X)
```

```
fig = plt.figure(figsize=(15, 5))
plt.suptitle("Manifold Learning", fontsize=14)

ax = fig.add_subplot(131, projection='3d')
ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=color, cmap=plt.cm.Spectral)
ax.view_init(4, -72)
plt.title('2D "S shape" manifold in 3D')

ax = fig.add_subplot(132)
```

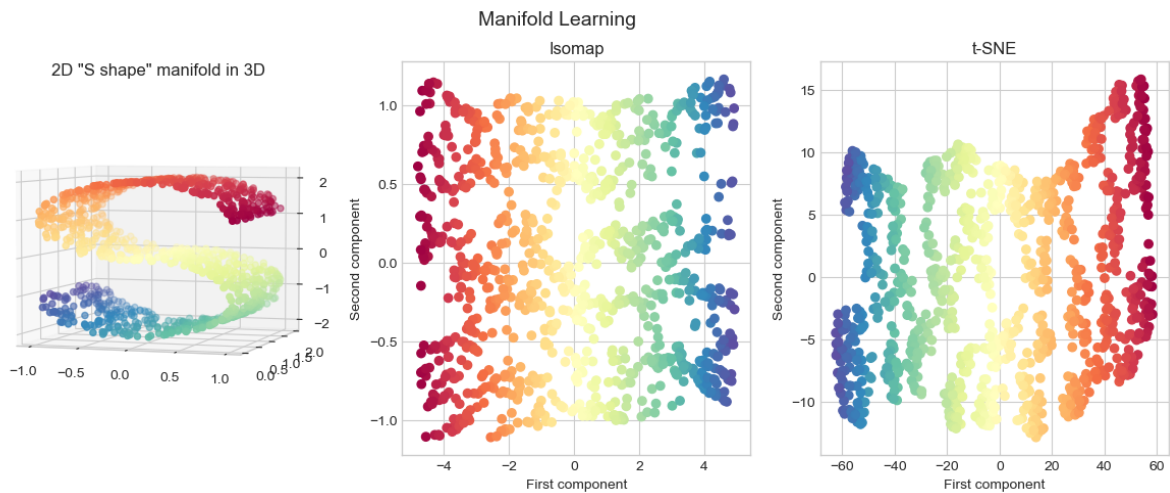
(continues on next page)

(continued from previous page)

```
plt.scatter(X_isomap[:, 0], X_isomap[:, 1], c=color, cmap=plt.cm.Spectral)
plt.title("Isomap")
plt.xlabel("First component")
plt.ylabel("Second component")

ax = fig.add_subplot(133)
plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=color, cmap=plt.cm.Spectral)
plt.title("t-SNE")
plt.xlabel("First component")
plt.ylabel("Second component")
plt.axis('tight')
```

```
(np.float64(-68.37603721618652),
 np.float64(64.30499229431152),
 np.float64(-14.287820672988891),
 np.float64(17.26294598579407))
```



7.2.4 Exercises

Run Manifold learning on handwritten digits: Locally Linear Embedding, Isomap with scikit-learn

7.3 Clustering

Wikipedia: Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense or another) to each other than to those in other groups (clusters). Clustering is one of the main task of exploratory data mining, and a common technique for statistical data analysis, used in many fields, including machine learning, pattern recognition, image analysis, information retrieval, and bioinformatics.

Source: [Clustering with Scikit-learn](#).

7.3.1 K-means clustering

Source: C. M. Bishop *Pattern Recognition and Machine Learning*, Springer, 2006

Suppose we have a data set $X = \{x_1, \dots, x_N\}$ that consists of N observations of a random D -dimensional Euclidean variable x . Our goal is to partition the data set into some number, K , of clusters, where we shall suppose for the moment that the value of K is given. Intuitively, we might think of a cluster as comprising a group of data points whose inter-point distances are small compared to the distances to points outside of the cluster. We can formalize this notion by first introducing a set of D -dimensional vectors μ_k , where $k = 1, \dots, K$, in which μ_k is a **prototype** associated with the k^{th} cluster. As we shall see shortly, we can think of the μ_k as representing the centres of the clusters. Our goal is then to find an assignment of data points to clusters, as well as a set of vectors $\{\mu_k\}$, such that the sum of the squares of the distances of each data point to its closest prototype vector μ_k , is at a minimum.

It is convenient at this point to define some notation to describe the assignment of data points to clusters. For each data point x_i , we introduce a corresponding set of binary indicator variables $r_{ik} \in \{0, 1\}$, where $k = 1, \dots, K$, that describes which of the K clusters the data point x_i is assigned to, so that if data point x_i is assigned to cluster k then $r_{ik} = 1$, and $r_{ij} = 0$ for $j \neq k$. This is known as the 1-of- K coding scheme. We can then define an objective function, denoted **inertia**, as

$$J(r, \mu) = \sum_i^N \sum_k^K r_{ik} \|x_i - \mu_k\|_2^2$$

which represents the sum of the squares of the Euclidean distances of each data point to its assigned vector μ_k . Our goal is to find values for the $\{r_{ik}\}$ and the $\{\mu_k\}$ so as to minimize the function J . We can do this through an iterative procedure in which each iteration involves two successive steps corresponding to successive optimizations with respect to the r_{ik} and the μ_k . First we choose some initial values for the μ_k . Then in the first phase we minimize J with respect to the r_{ik} , keeping the μ_k fixed. In the second phase we minimize J with respect to the μ_k , keeping r_{ik} fixed. This two-stage optimization process is then repeated until convergence. We shall see that these two stages of updating r_{ik} and μ_k correspond respectively to the expectation (E) and maximization (M) steps of the expectation-maximisation (EM) algorithm, and to emphasize this we shall use the terms E step and M step in the context of the K -means algorithm.

Consider first the determination of the r_{ik} . Because J is a linear function of r_{ik} , this optimization can be performed easily to give a closed form solution. The terms involving different i are independent and so we can optimize for each i separately by choosing r_{ik} to be 1 for whichever value of k gives the minimum value of $\|x_i - \mu_k\|^2$. In other words, we simply assign the i th data point to the closest cluster centre. More formally, this can be expressed as

$$r_{ik} = \begin{cases} 1, & \text{if } k = \arg \min_j \|x_i - \mu_j\|^2. \\ 0, & \text{otherwise.} \end{cases}$$

Now consider the optimization of the μ_k with the r_{ik} held fixed. The objective function J is a quadratic function of μ_k , and it can be minimized by setting its derivative with respect to μ_k to zero giving

$$2 \sum_i r_{ik} (x_i - \mu_k) = 0$$

which we can easily solve for μ_k to give

$$\mu_k = \frac{\sum_i r_{ik} x_i}{\sum_i r_{ik}}.$$

The denominator in this expression is equal to the number of points assigned to cluster k , and so this result has a simple interpretation, namely set μ_k equal to the mean of all of the data points x_i assigned to cluster k . For this reason, the procedure is known as the K -means algorithm.

The two phases of re-assigning data points to clusters and re-computing the cluster means are repeated in turn until there is no further change in the assignments (or until some maximum number of iterations is exceeded). Because each phase reduces the value of the objective function J , convergence of the algorithm is assured. However, it may converge to a local rather than global minimum of J .

```
from sklearn import cluster, datasets

# Plot
import matplotlib.pyplot as plt
import seaborn as sns
import pystatsml.plot_utils

# Plot parameters
plt.style.use('seaborn-v0_8-whitegrid')
fig_w, fig_h = plt.rcParams.get('figure.figsize')
plt.rcParams['figure.figsize'] = (fig_w, fig_h * .5)
colors = sns.color_palette()

iris = datasets.load_iris()
X = iris.data[:, :2] # use only 'sepal length and sepal width'
y_iris = iris.target

km2 = cluster.KMeans(n_clusters=2).fit(X)
km3 = cluster.KMeans(n_clusters=3).fit(X)
km4 = cluster.KMeans(n_clusters=4).fit(X)

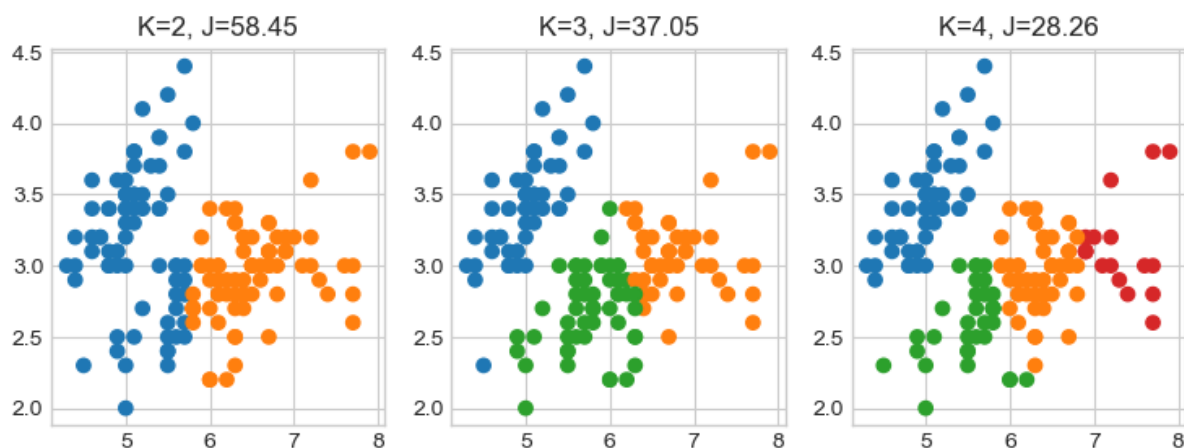
plt.figure(figsize=(9, 3))
plt.subplot(131)

plt.scatter(X[:, 0], X[:, 1],
            c=[colors[lab] for lab in km2.predict(X)])
plt.title("K=2, J=%.2f" % km2.inertia_)

plt.subplot(132)
plt.scatter(X[:, 0], X[:, 1],
            c=[colors[lab] for lab in km3.predict(X)])
plt.title("K=3, J=%.2f" % km3.inertia_)

plt.subplot(133)
plt.scatter(X[:, 0], X[:, 1],
            c=[colors[lab] for lab in km4.predict(X)])
plt.title("K=4, J=%.2f" % km4.inertia_)

Text(0.5, 1.0, 'K=4, J=28.26')
```



Exercises

1. Analyse clusters

- Analyse the plot above visually. What would a good value of K be?
- If you instead consider the inertia, the value of J , what would a good value of K be?
- Explain why there is such difference.
- For $K = 2$ why did K -means clustering not find the two “natural” clusters? See the assumptions of K -means: [See sklearn doc](#).

2. Re-implement the K -means clustering algorithm (homework)

Write a function `kmeans(X, K)` that return an integer vector of the samples' labels.

7.3.2 Gaussian mixture models

The Gaussian mixture model (GMM) is a simple linear superposition of Gaussian components over the data, aimed at providing a rich class of density models. We turn to a formulation of Gaussian mixtures in terms of discrete latent variables: the K hidden classes to be discovered.

Differences compared to K -means:

- Whereas the K -means algorithm performs a hard assignment of data points to clusters, in which each data point is associated uniquely with one cluster, the GMM algorithm makes a soft assignment based on posterior probabilities.
- Whereas the classic K -means is only based on Euclidean distances, classic GMM use a Mahalanobis distances that can deal with non-spherical distributions. It should be noted that Mahalanobis could be plugged within an improved version of K -Means clustering. The Mahalanobis distance is unitless and scale-invariant, and takes into account the correlations of the data set.

The Gaussian mixture distribution can be written as a linear superposition of K Gaussians in the form:

$$p(x) = \sum_{k=1}^K \mathcal{N}(x | \mu_k, \Sigma_k) p(k),$$

where:

- The $p(k)$ are the mixing coefficients also known as the class probability of class k , and they sum to one: $\sum_{k=1}^K p(k) = 1$.
- $\mathcal{N}(x | \mu_k, \Sigma_k) = p(x | k)$ is the conditional distribution of x given a particular class k . It is the multivariate Gaussian distribution defined over a P -dimensional vector x of continuous variables.

The goal is to maximize the log-likelihood of the GMM:

$$\ln \prod_{i=1}^N p(x_i) = \ln \prod_{i=1}^N \left\{ \sum_{k=1}^K \mathcal{N}(x_i | \mu_k, \Sigma_k) p(k) \right\} = \sum_{i=1}^N \ln \left\{ \sum_{k=1}^K \mathcal{N}(x_i | \mu_k, \Sigma_k) p(k) \right\}.$$

To compute the classes parameters: $p(k)$, μ_k , Σ_k we sum over all samples, by weighting each sample i by its responsibility or contribution to class k : $p(k | x_i)$ such that for each point its contribution to all classes sum to one $\sum_k p(k | x_i) = 1$. This contribution is the conditional probability of class k given x : $p(k | x)$ (sometimes called the posterior). It can be computed using Bayes' rule:

$$\begin{aligned} p(k | x) &= \frac{p(x | k) p(k)}{p(x)} \\ &= \frac{\mathcal{N}(x | \mu_k, \Sigma_k) p(k)}{\sum_{k=1}^K \mathcal{N}(x | \mu_k, \Sigma_k) p(k)} \end{aligned}$$

Since the class parameters, $p(k)$, μ_k and Σ_k , depend on the responsibilities $p(k | x)$ and the responsibilities depend on class parameters, we need a two-step iterative algorithm: the expectation-maximization (EM) algorithm. We discuss this algorithm next.

The expectation-maximization (EM) algorithm for Gaussian mixtures

Given a Gaussian mixture model, the goal is to maximize the likelihood function with respect to the parameters (comprised of the means and covariances of the components and the mixing coefficients).

Initialize the means μ_k , covariances Σ_k and mixing coefficients $p(k)$

1. **E step.** For each sample i , evaluate the responsibilities for each class k using the current parameter values

$$p(k | x_i) = \frac{\mathcal{N}(x_i | \mu_k, \Sigma_k) p(k)}{\sum_{k=1}^K \mathcal{N}(x_i | \mu_k, \Sigma_k) p(k)}$$

2. **M step.** For each class, re-estimate the parameters using the current responsibilities

$$\begin{aligned} \mu_k^{\text{new}} &= \frac{1}{N_k} \sum_{i=1}^N p(k | x_i) x_i \\ \Sigma_k^{\text{new}} &= \frac{1}{N_k} \sum_{i=1}^N p(k | x_i) (x_i - \mu_k^{\text{new}})(x_i - \mu_k^{\text{new}})^T \\ p^{\text{new}}(k) &= \frac{N_k}{N} \end{aligned}$$

3. Evaluate the log-likelihood

$$\sum_{i=1}^N \ln \left\{ \sum_{k=1}^K \mathcal{N}(x_i | \mu_k, \Sigma_k) p(k) \right\},$$

and check for convergence of either the parameters or the log-likelihood. If the convergence criterion is not satisfied return to step 1.

```

import numpy as np
from sklearn import datasets
import sklearn
from sklearn.mixture import GaussianMixture

import pystatsml.plot_utils

colors = sns.color_palette()

iris = datasets.load_iris()
X = iris.data[:, :2] # 'sepal length (cm)' 'sepal width (cm)'
y_iris = iris.target

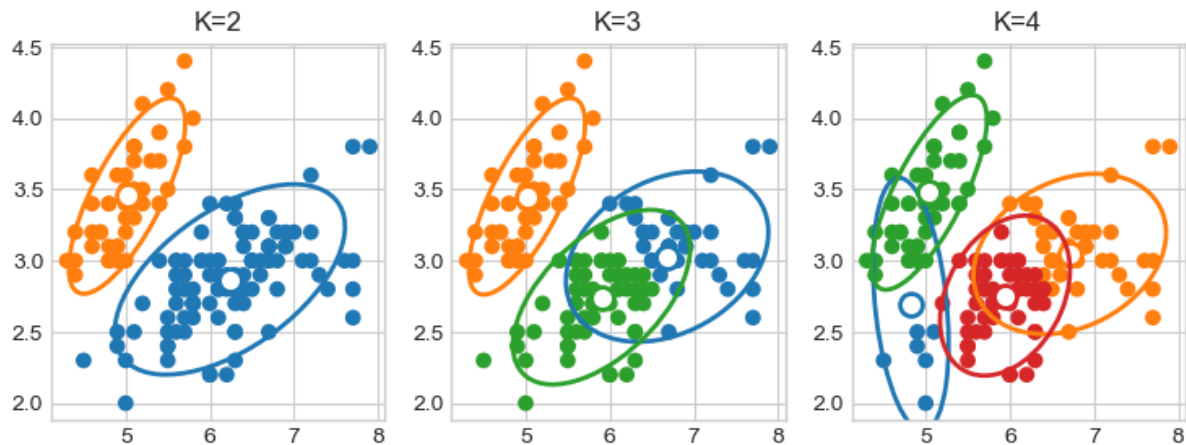
gmm2 = GaussianMixture(n_components=2, covariance_type='full').fit(X)
gmm3 = GaussianMixture(n_components=3, covariance_type='full').fit(X)
gmm4 = GaussianMixture(n_components=4, covariance_type='full').fit(X)

plt.figure(figsize=(9, 3))
plt.subplot(131)
plt.scatter(X[:, 0], X[:, 1], c=[colors[lab] for lab in gmm2.predict(X)])
for i in range(gmm2.covariances_.shape[0]):
    pystatsml.plot_utils.plot_cov_ellipse(cov=gmm2.covariances_[i, :],
                                          pos=gmm2.means_[i, :],
                                          facecolor='none', linewidth=2, edgecolor=colors[i])
    plt.scatter(gmm2.means_[i, 0], gmm2.means_[i, 1], edgecolor=colors[i],
               marker="o", s=100, facecolor="w", linewidth=2)
plt.title("K=2")

plt.subplot(132)
plt.scatter(X[:, 0], X[:, 1], c=[colors[lab] for lab in gmm3.predict(X)])
for i in range(gmm3.covariances_.shape[0]):
    pystatsml.plot_utils.plot_cov_ellipse(cov=gmm3.covariances_[i, :],
                                          pos=gmm3.means_[i, :],
                                          facecolor='none', linewidth=2, edgecolor=colors[i])
    plt.scatter(gmm3.means_[i, 0], gmm3.means_[i, 1], edgecolor=colors[i],
               marker="o", s=100, facecolor="w", linewidth=2)
plt.title("K=3")

plt.subplot(133)
plt.scatter(X[:, 0], X[:, 1], c=[colors[lab] for lab in gmm4.predict(X)])
for i in range(gmm4.covariances_.shape[0]):
    pystatsml.plot_utils.plot_cov_ellipse(cov=gmm4.covariances_[i, :],
                                          pos=gmm4.means_[i, :],
                                          facecolor='none', linewidth=2, edgecolor=colors[i])
    plt.scatter(gmm4.means_[i, 0], gmm4.means_[i, 1], edgecolor=colors[i],
               marker="o", s=100, facecolor="w", linewidth=2)
_ = plt.title("K=4")

```



Models of covariances: parameter `covariance_type` see [Sklearn doc](#). K-means is almost a GMM with spherical covariance.

Model selection using Bayesian Information Criterion

In statistics, the Bayesian information criterion (BIC) is a criterion for model selection among a finite set of models; the model with the lowest BIC is preferred. It is based, in part, on the likelihood function and it is closely related to the Akaike information criterion (AIC).

```
X = iris.data
y_iris = iris.target

bic = list()
ks = np.arange(1, 10)

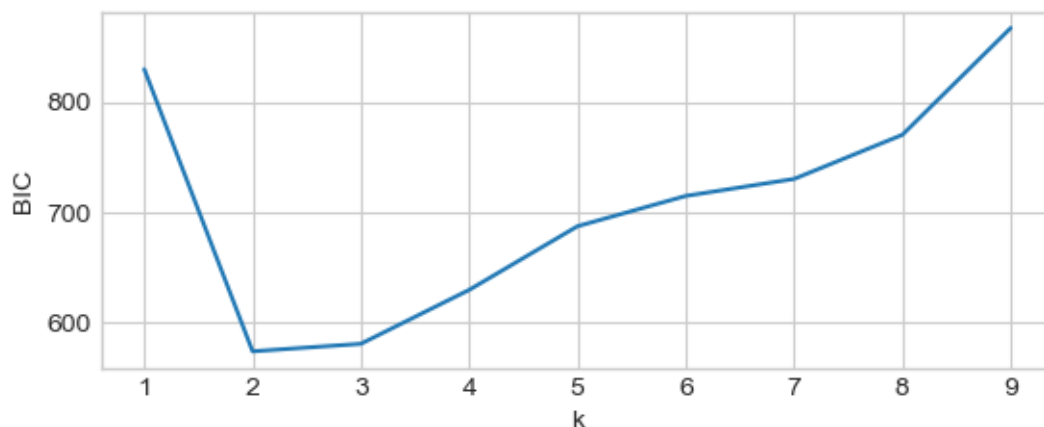
for k in ks:
    gmm = GaussianMixture(n_components=k, covariance_type='full')
    gmm.fit(X)
    bic.append(gmm.bic(X))

k_chosen = ks[np.argmin(bic)]

plt.plot(ks, bic)
plt.xlabel("k")
plt.ylabel("BIC")

print("Choose k=", k_chosen)
```

Choose k= 2



7.3.3 Hierarchical clustering

Sources:

- [Hierarchical clustering with Scikit-learn](#)
- [Comparing different hierarchical linkage](#)

Hierarchical clustering is an approach to clustering that build hierarchies of clusters in two main approaches:

- **Agglomerative:** A *bottom-up* strategy, where each observation starts in their own cluster, and pairs of clusters are merged upwards in the hierarchy.
- **Divisive:** A *top-down* strategy, where all observations start out in the same cluster, and then the clusters are split recursively downwards in the hierarchy.

In order to decide which clusters to merge or to split, a measure of dissimilarity between clusters is introduced. More specific, this comprise a *distance* measure and a *linkage* criterion. The distance measure is just what it sounds like, and the linkage criterion is essentially a function of the distances between points, for instance the minimum distance between points in two clusters, the maximum distance between points in two clusters, the average distance between points in two clusters, etc. One particular linkage criterion, the Ward criterion, will be discussed next.

The Agglomerative clustering use four main linkage strategies:

- **Single Linkage:** The distance between two clusters is defined as the shortest distance between any two points in the clusters. This can create elongated, chain-like clusters organized on manifolds that cannot be summarized by distribution around a center. However, it is sensitive to noise.
- **Complete Linkage:** The distance between two clusters is the maximum distance between any two points in the clusters. This tends to produce compact and well-separated clusters.
- **Average Linkage:** The distance between two clusters is the average distance between all pairs of points in the two clusters. This provides a balance between single and complete linkage.
- **Ward's Linkage:** Merges clusters by minimizing the increase in within-cluster variance. This often results in more evenly sized clusters and is commonly used for hierarchical clustering.

Application to “non-linear” manifolds: Ward vs. single linkage.

```

from sklearn.datasets import make_moons
from sklearn.cluster import AgglomerativeClustering

# Generate synthetic dataset
X, y = make_moons(n_samples=300, noise=0.05, random_state=42)

# Define clustering models with different linkage strategies
clustering_ward = AgglomerativeClustering(n_clusters=2, linkage="ward")
clustering_single = AgglomerativeClustering(n_clusters=2, linkage="single")

# Fit and predict cluster labels
colors_ward = [colors[lab] for lab in clustering_ward.fit_predict(X)]
colors_single = [colors[lab] for lab in clustering_single.fit_predict(X)]

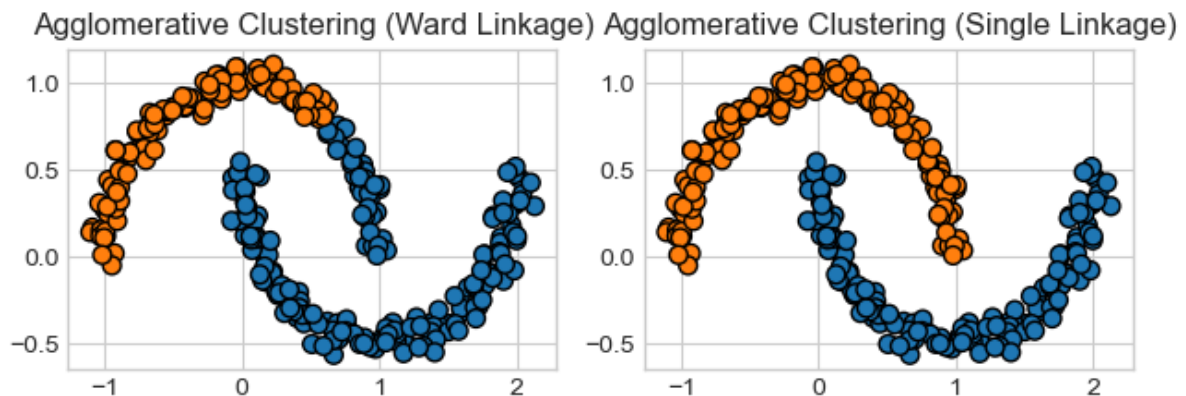
# Plot results
fig, axes = plt.subplots(1, 2)

# Ward linkage clustering
axes[0].scatter(X[:, 0], X[:, 1], c=colors_ward, edgecolors='k', s=50)
axes[0].set_title("Agglomerative Clustering (Ward Linkage)")

# Single linkage clustering
axes[1].scatter(X[:, 0], X[:, 1], c=colors_single, edgecolors='k', s=50)
axes[1].set_title("Agglomerative Clustering (Single Linkage)")

plt.tight_layout()
plt.show()

```



Application to Gaussian-like distribution: use Ward linkage.

```

from sklearn import cluster, datasets
import matplotlib.pyplot as plt
import seaborn as sns # nice color

iris = datasets.load_iris()
X = iris.data[:, :2] # 'sepal length (cm)' 'sepal width (cm)'
y_iris = iris.target

```

(continues on next page)

(continued from previous page)

```

ward2 = cluster.AgglomerativeClustering(n_clusters=2, linkage='ward').fit(X)
ward3 = cluster.AgglomerativeClustering(n_clusters=3, linkage='ward').fit(X)
ward4 = cluster.AgglomerativeClustering(n_clusters=4, linkage='ward').fit(X)

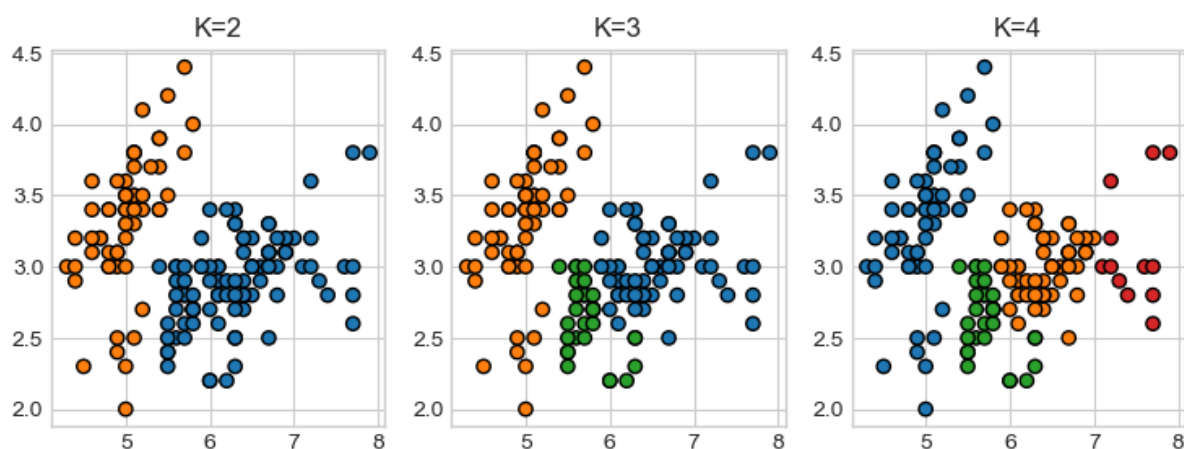
plt.figure(figsize=(9, 3))
plt.subplot(131)
plt.scatter(X[:, 0], X[:, 1], edgecolors='k',
            c=[colors[lab] for lab in ward2.fit_predict(X)])
plt.title("K=2")

plt.subplot(132)
plt.scatter(X[:, 0], X[:, 1], edgecolors='k',
            c=[colors[lab] for lab in ward3.fit_predict(X)])
plt.title("K=3")

plt.subplot(133)
plt.scatter(X[:, 0], X[:, 1], edgecolors='k',
            c=[colors[lab] for lab in ward4.fit_predict(X)]) # .astype(np.float))
plt.title("K=4")

```

```
Text(0.5, 1.0, 'K=4')
```



7.3.4 Exercises

Perform clustering of the iris dataset based on all variables using Gaussian mixture models. Use PCA to visualize clusters.

SUPERVISED LEARNING

8.1 Linear Models for Regression

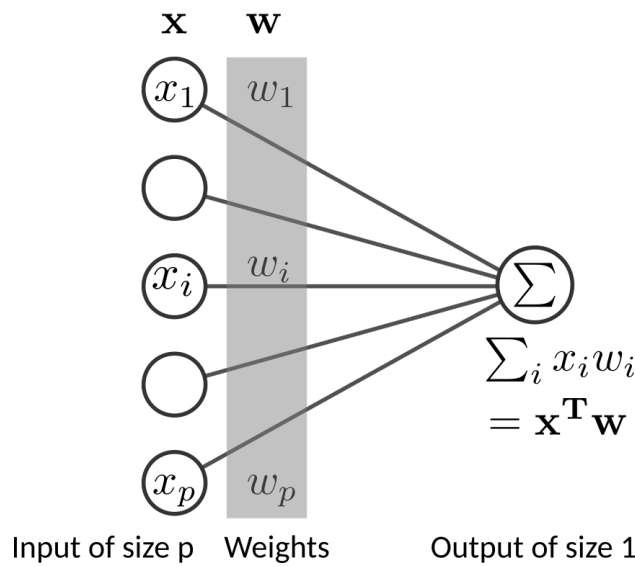


Fig. 1: Linear regression

8.1.1 Minimizing the Sum of Squared Errors (SSE) Loss

A linear model assumes a linear relationship between input features of observation i $\mathbf{x}_i \in \mathbb{R}^P$ and an output $y \in \mathbb{R}$ using:

$$y_i = \mathbf{x}_i^T \mathbf{w} + \varepsilon_i,$$

where $\varepsilon_i \in \mathbb{R}$ is the **residual** or the error of the prediction. $\mathbf{w}$ is the weight vector (model's parameters) of coefficients. $\mathbf{w}$ is found by minimizing an **objective function**, which is the **loss function**, $L(\mathbf{w})$, i.e. the error measured on the data. This error is the **sum of squared errors (SSE) loss**:

$$\text{SSE}(\mathbf{w}) = \sum_i^N (y_i - \mathbf{x}_i^T \mathbf{w})^2$$

The equivalent matrix notation is:

$$\mathbf{y} = \mathbf{X}\mathbf{w} + \boldsymbol{\varepsilon},$$

where $\mathbf{X}$ be the $N \times P$ matrix with each row an input vector and similarly let $\mathbf{y}$ the N -dimensional vector of outputs.

The loss is given by

$$\begin{aligned} \text{SSE}(\mathbf{w}) &= (\mathbf{y} - \mathbf{X}\mathbf{w})^\top (\mathbf{y} - \mathbf{X}\mathbf{w}) \\ &= \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2, \end{aligned}$$

Minimizing the SSE is the Ordinary Least Square **OLS** regression as objective function. which is a simple **ordinary least squares (OLS)** minimization whose analytic solution is:

$$\mathbf{w}_{\text{OLS}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$$

The solution could also be found using gradient descent using the gradient of the loss:

$$\partial \frac{\text{SSE}(\mathbf{w})}{\partial \mathbf{w}} = 2 \sum_i \mathbf{x}_i (\mathbf{x}_i \cdot \mathbf{w} - y_i)$$

Linear regression of Advertising.csv dataset with TV and Radio advertising as input features and Sales as target. The linear model that minimizes the MSE is a plan (2 input features) defined as: Sales = 0.05 TV + .19 Radio + 3:

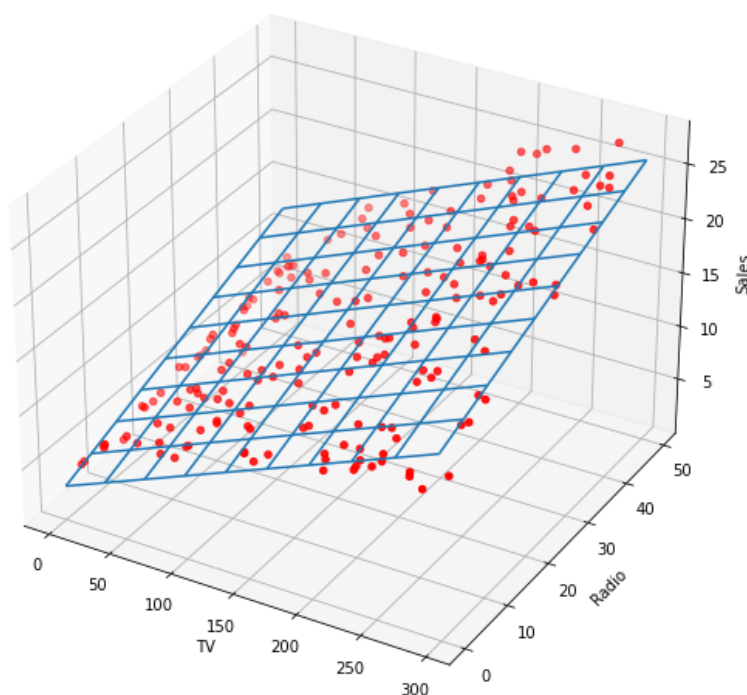


Fig. 2: Linear regression

8.1.2 Linear regression with scikit-learn

Scikit-learn offers many models for supervised learning, and they all follow the same **Application Programming Interface (API)**, namely:

```
est = Estimator()
est.fit(X, y)
predictions = est.predict(X)
```

```

import numpy as np
import pandas as pd
# Plot
import matplotlib.pyplot as plt
import seaborn as sns

# Plot parameters
plt.style.use('seaborn-v0_8-whitegrid')
fig_w, fig_h = plt.rcParams.get('figure.figsize')
plt.rcParams['figure.figsize'] = (fig_w, fig_h * .5)

from sklearn import datasets
import sklearn.linear_model as lm
import sklearn.metrics as metrics

# Dataset with some correlation
X, y, coef = datasets.make_regression(n_samples=100, n_features=10,
                                     n_informative=5, random_state=0,
                                     effective_rank=3, coef=True)

lr = lm.LinearRegression().fit(X, y)

```

8.1.3 Regularization using penalization of coefficients

Regarding linear models, overfitting generally leads to excessively complex solutions (coefficient vectors), accounting for noise or spurious correlations within predictors. **Regularization** aims to alleviate this phenomenon by constraining (biasing or reducing) the capacity of the learning algorithm in order to promote simple solutions. Regularization penalizes “large” solutions forcing the coefficients to be small, i.e. to shrink them toward zeros.

The objective function $J(\mathbf{w})$ to minimize with respect to $\mathbf{w}$ is composed of a loss function $L(\mathbf{w})$ for goodness-of-fit and a penalty term $\Omega(\mathbf{w})$ (regularization to avoid overfitting). This is a trade-off where the respective contribution of the loss and the penalty terms is controlled by the regularization parameter λ .

Therefore the **loss function** $L(\mathbf{w})$ is combined with a **penalty function** $\Omega(\mathbf{w})$ leading to the general form:

$$J(\mathbf{w}) = \sum_i^N (y_i - \mathbf{x}_i^\top \mathbf{w})^2 + \lambda \Omega(\mathbf{w})$$

The respective contribution of the loss and the penalty is controlled by the **regularization hyper-parameter** λ .

Regularization exploits Occam’s razor is a principle

Occam’s razor is a principle stating that, among competing hypotheses, the one with the fewest assumptions should be selected. This is also known as the principle of parsimony, favoring simpler explanations or models when possible. In machine learning, this translates to preferring models with similar loss but lower complexity.

8.1.4 Ridge Regression (ℓ_2 or L2 regularization)

- **Penalty:** $\Omega(\mathbf{w}) = \|\mathbf{w}\|_2^2 = \sum_j w_j^2$
- **Objective function:**

$$\text{Ridge}(\mathbf{w}) = \sum_i^N (y_i - \mathbf{x}_i^\top \mathbf{w})^2 + \lambda \|\mathbf{w}\|_2^2$$

- **Effect:** Shrinks all coefficients smoothly towards zero, but **does not enforce sparsity** (no coefficients become exactly zero).
- The gradient of the loss:

$$\partial \frac{L(\mathbf{w}, \mathbf{X}, \mathbf{y})}{\partial \mathbf{w}} = 2 \left(\sum_i \mathbf{x}_i (\mathbf{x}_i \cdot \mathbf{w} - y_i) + \lambda \mathbf{w} \right)$$

Numerical Interpretation

Using the matrix notation, the Ridge objective function can be rewritten as

$$\text{Ridge}(\mathbf{w}) = \|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 + \lambda \|\mathbf{w}\|_2^2$$

The $\mathbf{w}$ that minimizes $F_{\text{Ridge}}(\mathbf{w})$ can be found by the following derivation:

$$\begin{aligned} \nabla_{\mathbf{w}} \text{Ridge}(\mathbf{w}) &= 0 \\ \nabla_{\mathbf{w}} ((\mathbf{y} - \mathbf{X}\mathbf{w})^\top (\mathbf{y} - \mathbf{X}\mathbf{w}) + \lambda \mathbf{w}^\top \mathbf{w}) &= 0 \\ \nabla_{\mathbf{w}} (\mathbf{y}^\top \mathbf{y} - 2\mathbf{w}^\top \mathbf{X}^\top \mathbf{y} + \mathbf{w}^\top \mathbf{X}^\top \mathbf{X} \mathbf{w} + \lambda \mathbf{w}^\top \mathbf{w}) &= 0 \\ -2\mathbf{X}^\top \mathbf{y} + 2\mathbf{X}^\top \mathbf{X} \mathbf{w} + 2\lambda \mathbf{w} &= 0 \\ -\mathbf{X}^\top \mathbf{y} + (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}) \mathbf{w} &= 0 \\ (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I}) \mathbf{w} &= \mathbf{X}^\top \mathbf{y} \\ \mathbf{w} &= (\mathbf{X}^\top \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y} \end{aligned}$$

- The solution adds a positive constant to the diagonal of $\mathbf{X}^\top \mathbf{X}$ before inversion. This makes the problem nonsingular, even if $\mathbf{X}^\top \mathbf{X}$ is not of full rank, and was the main motivation behind ridge regression.
- Increasing λ shrinks the $\mathbf{w}$ coefficients toward 0.
- Solutions with large coefficients become unattractive.

8.1.5 Lasso Regression (ℓ_1 or L1 regularization)

- **Penalty:** $\Omega(\mathbf{w}) = \|\mathbf{w}\|_1 = \sum_j |w_j|$
- **Objective function:**

$$\text{Lasso}(\mathbf{w}) = \sum_i^N (y_i - \mathbf{x}_i^\top \mathbf{w})^2 + \lambda \sum_j |w_j|$$

- **Effect:** Promotes **sparsity**, i.e., forces some coefficients to be exactly zero, ie. performs feature selection.
- **No closed-form solution**, It is convex but not differentiable. Requires specific optimization algorithms, such as the fast iterative shrinkage-thresholding algorithm (FISTA): Amir Beck and Marc Teboulle, *A Fast Iterative Shrinkage-Thresholding Algorithm for Linear Inverse Problems* SIAM J. Imaging Sci., 2009.

Geometric interpretation

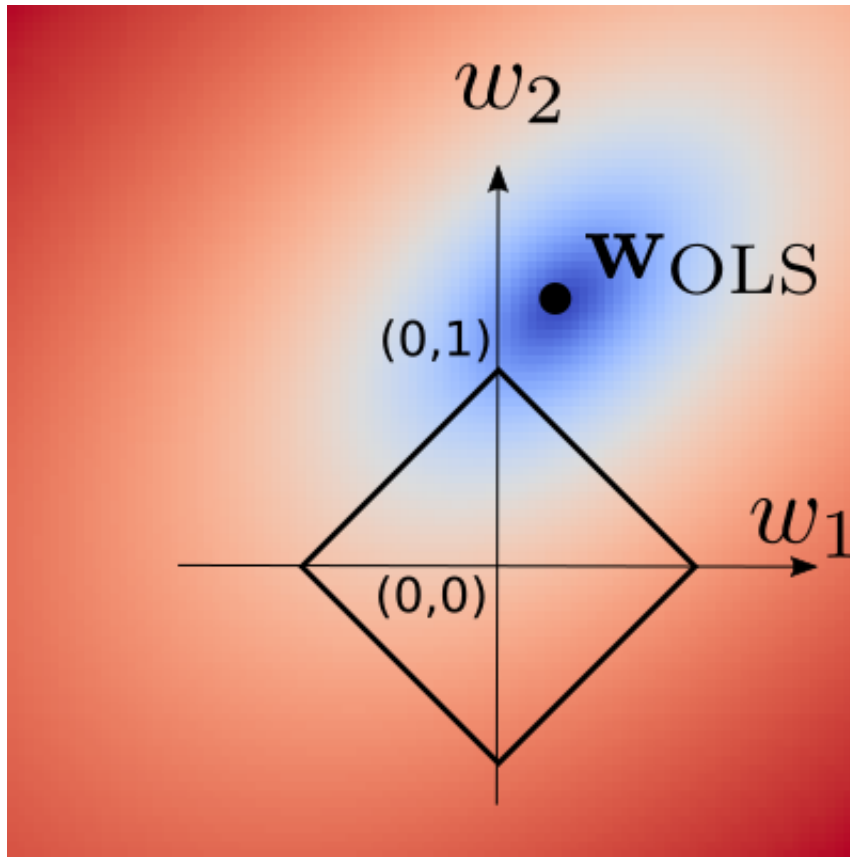


Fig. 3: Sparsity of L1 norm

8.1.6 Elastic Net

- **Penalty:** Combination of L1 and L2: $\Omega(\mathbf{w}) = \alpha \|\mathbf{w}\|_1 + (1 - \alpha) \|\mathbf{w}\|_2^2$ where $\alpha \in [0, 1]$ controls the mix of Lasso and Ridge.
- **Effect:** Balances sparsity (L1) and stability (L2), especially useful when features are correlated.

The Elastic-net estimator combines the ℓ_1 and ℓ_2 penalties, and results in the problem to

$$\text{Enet}(\mathbf{w}) = \sum_i^N (y_i - \mathbf{x}_i^\top \mathbf{w})^2 + \alpha (\rho \|\mathbf{w}\|_1 + (1 - \rho) \|\mathbf{w}\|_2^2),$$

where α acts as a global penalty and ρ as an ℓ_1/ℓ_2 ratio.

Rational

- If there are groups of highly correlated variables, Lasso tends to arbitrarily select only one from each group. These models are difficult to interpret because covariates that are strongly associated with the outcome are not included in the predictive model. Conversely, the elastic net encourages a grouping effect, where strongly correlated predictors tend to be in or out of the model together.
- Studies on real world data and simulation studies show that the elastic net often outperforms the lasso, while enjoying a similar sparsity of representation.

Penalization: summary and Application with scikit-learn

Summary

Method	Penalty Type	Promotes Spar-	Handles M multi-	Closed-Form Solution
Ridge	L2: $\sum_j w_j^2$	No	Yes	Yes
Lasso	L1: $\sum_j w_j $	Yes	No (can be unstable)	No
Elastic Net	L1 + L2	Yes (less than Lasso)	Yes	No

Effect on problems with increasing dimension

The next figure shows the predicted performance (r-squared) on train and test sets with an increasing number of input features. The number of predictive features is always 10% of the total number of input features. Therefore, the signal to noise ratio (SNR) increases by increasing the number of input features. The performances on the training set rapidly reach 100% ($R^2=1$). However, the performance on the test set decreases with the increase of the input dimensionality. The difference between the train and test performances (blue shaded region) depicts the overfitting phenomena. Regularization using penalties of the coefficient vector norm greatly limits the overfitting phenomena.

Effect on solution finding

The ridge penalty shrinks the coefficients toward zero. The figure illustrates: the OLS solution on the left. The ℓ_1 and ℓ_2 penalties in the middle pane. The penalized OLS in the right pane. The right pane shows how the penalties shrink the coefficients toward zero. The black points are the minimum found in each case, and the white points represents the true solution used to generate the data.

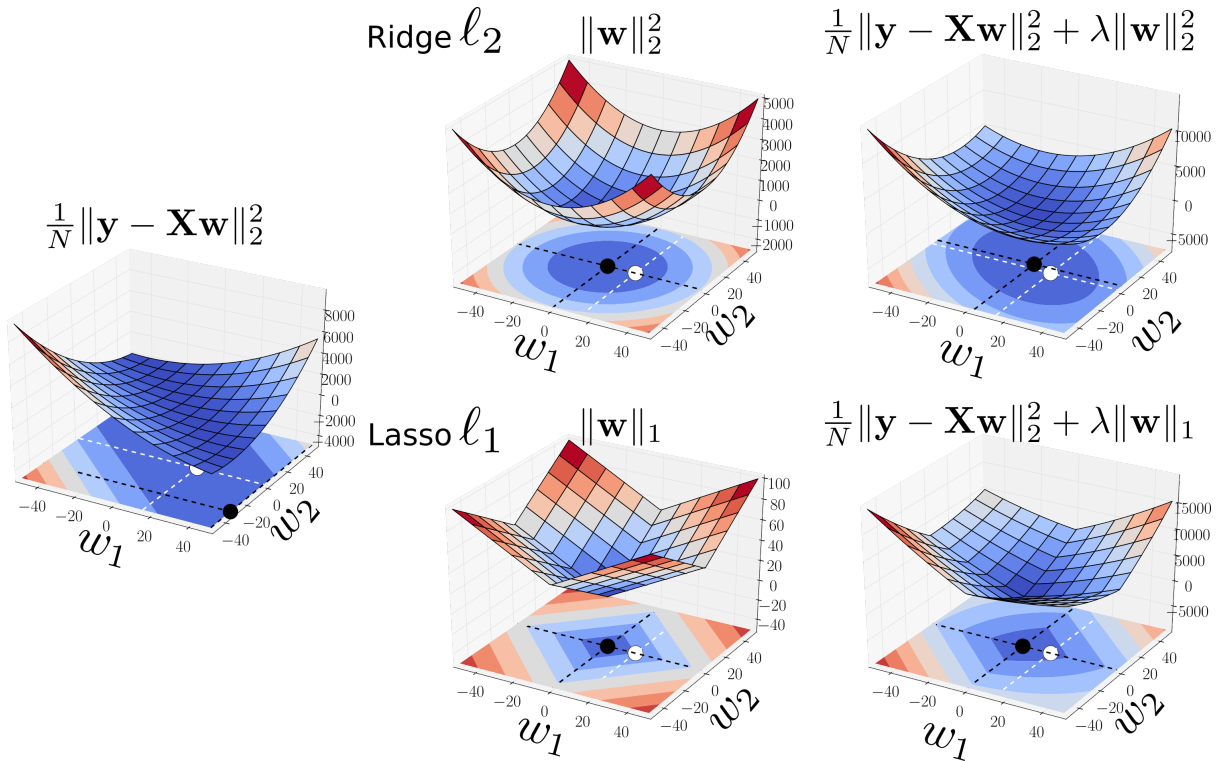
Application with scikit-learn

```
l2 = lm.Ridge(alpha=10).fit(X, y) # lambda is alpha!
l1 = lm.Lasso(alpha=.1).fit(X, y) # lambda is alpha !
l1l2 = lm.ElasticNet(alpha=.1, l1_ratio=.9).fit(X, y)
print(pd.DataFrame(np.vstack((coef, lr.coef_, l2.coef_, l1.coef_, l1l2.coef_)),
                    index=['True', 'lr', 'l2', 'l1', 'l1l2']).round(2))
```

```

      0      1      2      3      4      5      6      7      8      9
True 28.49  0.00 13.17  0.00 48.97 70.44 39.70  0.00  0.00  0.00
lr   28.49  0.00 13.17  0.00 48.97 70.44 39.70  0.00  0.00 -0.00
```

(continues on next page)


 Fig. 4: ℓ_1 and ℓ_2 shrinkages

(continued from previous page)

12	1.03	0.21	0.93	-0.32	1.82	1.57	2.10	-1.14	-0.84	-1.02
11	0.00	-0.00	0.00	-0.00	24.40	25.16	25.36	-0.00	-0.00	-0.00
1112	0.78	0.00	0.51	-0.00	7.20	5.71	8.95	-1.38	-0.00	-0.40

Sparsity of the ℓ_1 norm

Occam's razor

Occam's razor (also written as Ockham's razor, and **lex parsimoniae** in Latin, which means law of parsimony) is a problem solving principle attributed to William of Ockham (1287-1347), who was an English Franciscan friar and scholastic philosopher and theologian. The principle can be interpreted as stating that **among competing hypotheses, the one with the fewest assumptions should be selected.**

Principle of parsimony

The simplest of two competing theories is to be preferred. Definition of parsimony: Economy of explanation in conformity with Occam's razor.

Among possible models with similar loss, choose the simplest one:

- Choose the model with the smallest coefficient vector, i.e. smallest ℓ_2 ($\|\mathbf{w}\|_2$) or ℓ_1 ($\|\mathbf{w}\|_1$) norm of $\mathbf{w}$, i.e. ℓ_2 or ℓ_1 penalty. See also bias-variance tradeoff.
- Choose the model that uses the smallest number of predictors. In other words, choose the model that has many predictors with zero weights. Two approaches are available to obtain this: (i) Perform a feature selection as a preprocessing prior to applying the learning algorithm, or (ii) embed the feature selection procedure within the learning process.

8.1.7 Regression performance evaluation metrics: R-squared, MSE and MAE

Common regression [Scikit-learn Metrics](#) are:

- R^2 : R-squared
- MSE: Mean Squared Error
- MAE: Mean Absolute Error

R-squared

The goodness of fit of a statistical model describes how well it fits a set of observations. Measures of goodness of fit typically summarize the discrepancy between observed values and the values expected under the model in question. We will consider the **explained variance** also known as the coefficient of determination, denoted R^2 pronounced **R-squared**.

The total sum of squares, SS_{tot} is the sum of the sum of squares explained by the regression, SS_{reg} , plus the sum of squares of residuals unexplained by the regression, SS_{res} , also called the SSE, i.e. such that

$$SS_{\text{tot}} = SS_{\text{reg}} + SS_{\text{res}}$$

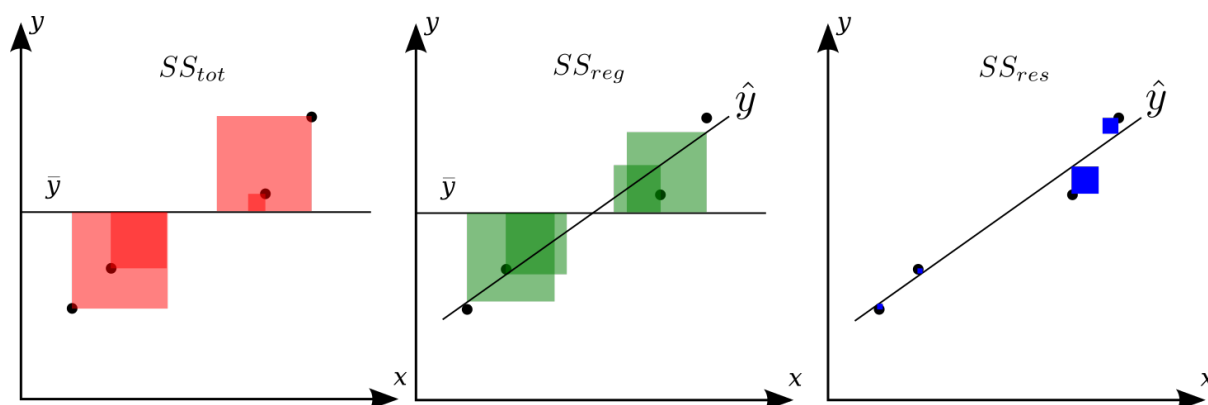


Fig. 5: title

The mean of y is

$$\bar{y} = \frac{1}{n} \sum_i y_i.$$

The total sum of squares is the total squared sum of deviations from the mean of y , i.e.

$$SS_{\text{tot}} = \sum_i (y_i - \bar{y})^2$$

The regression sum of squares, also called the explained sum of squares:

$$SS_{\text{reg}} = \sum_i (\hat{y}_i - \bar{y})^2,$$

where $\hat{y}_i = \beta x_i + \beta_0$ is the estimated value of salary $\hat{y}_i$ given a value of experience x_i .

The sum of squares of the residuals (**SSE, Sum Squared Error**), also called the residual sum of squares (RSS) is:

$$SS_{\text{res}} = \sum_i (y_i - \hat{y}_i)^2.$$

R^2 is the explained sum of squares of errors. It is the variance explain by the regression divided by the total variance, i.e.

$$R^2 = \frac{\text{explained SS}}{\text{total SS}} = \frac{SS_{\text{reg}}}{SS_{\text{tot}}} = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}}.$$

Test

Let $\hat{\sigma}^2 = SS_{\text{res}}/(n - 2)$ be an estimator of the variance of ϵ . The 2 in the denominator stems from the 2 estimated parameters: intercept and coefficient.

- **Unexplained variance:** $\frac{SS_{\text{res}}}{\hat{\sigma}^2} \sim \chi_{n-2}^2$
- **Explained variance:** $\frac{SS_{\text{reg}}}{\hat{\sigma}^2} \sim \chi_1^2$. The single degree of freedom comes from the difference between $\frac{SS_{\text{tot}}}{\hat{\sigma}^2} (\sim \chi_{n-1}^2)$ and $\frac{SS_{\text{res}}}{\hat{\sigma}^2} (\sim \chi_{n-2}^2)$, i.e. $(n - 1) - (n - 2)$ degree of freedom.

The Fisher statistics of the ratio of two variances:

$$F = \frac{\text{Explained variance}}{\text{Unexplained variance}} = \frac{SS_{\text{reg}}/1}{SS_{\text{res}}/(n - 2)} \sim F(1, n - 2)$$

Using the F -distribution, compute the probability of observing a value greater than F under H_0 , i.e.: $P(x > F|H_0)$, i.e. the survival function ($1 - \text{Cumulative Distribution Function}$) at x of the given F -distribution.

```
import sklearn.metrics as metrics
from sklearn.model_selection import train_test_split

X, y = datasets.make_regression(random_state=0)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=1)

lr = lm.LinearRegression()
lr.fit(X_train, y_train)
yhat = lr.predict(X_test)

r2 = metrics.r2_score(y_test, yhat)
mse = metrics.mean_squared_error(y_test, yhat)
mae = metrics.mean_absolute_error(y_test, yhat)

print("r2: %.3f, mae: %.3f, mse: %.3f" % (r2, mae, mse))
```

```
r2: 0.053, mae: 71.712, mse: 7866.759
```

In pure numpy:

```
res = y_test - lr.predict(X_test)

y_mu = np.mean(y_test)
```

(continues on next page)

(continued from previous page)

```

ss_tot = np.sum((y_test - y_mu) ** 2)
ss_res = np.sum(res ** 2)

r2 = (1 - ss_res / ss_tot)
mse = np.mean(res ** 2)
mae = np.mean(np.abs(res))

print("r2: %.3f, mae: %.3f, mse: %.3f" % (r2, mae, mse))

```

```

r2: 0.053, mae: 71.712, mse: 7866.759

```

8.2 Linear Models for Classification

Linear vs. non-linear classifier: See [Scikit-learn Classifier comparison](#).

```

import numpy as np
import pandas as pd

from sklearn import datasets
import sklearn.linear_model as lm
import sklearn.metrics as metrics
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split

# Plot
import matplotlib.pyplot as plt
import seaborn as sns

# Plot parameters
plt.style.use('seaborn-v0_8-whitegrid')
fig_w, fig_h = plt.rcParams.get('figure.figsize')
plt.rcParams['figure.figsize'] = (fig_w, fig_h * .5)

```

8.2.1 Geometric Method: Naive Method

Principles:

- Compute classes means $\mu_1, \mu_2, \mu_k, \dots$
- Classify new point $\mathbf{x}$ to the closest mean, i.e.: class $\arg \min_k \|\mathbf{x} - \mu_k\|_2$

For binary classification, this is equivalent to compute the most discriminative direction as the vector between class mean:

$$\mathbf{w}_{\text{naive}} = \mu_1 - \mu_2$$

And projecting a new point $\mathbf{x}_i$ along this direction to obtain a score z_i :

$$z_i = \mathbf{x}_i^\top \mathbf{w}_{\text{naive}}$$

Finally the Classify the point to the closest projected class mean.

Illustration:

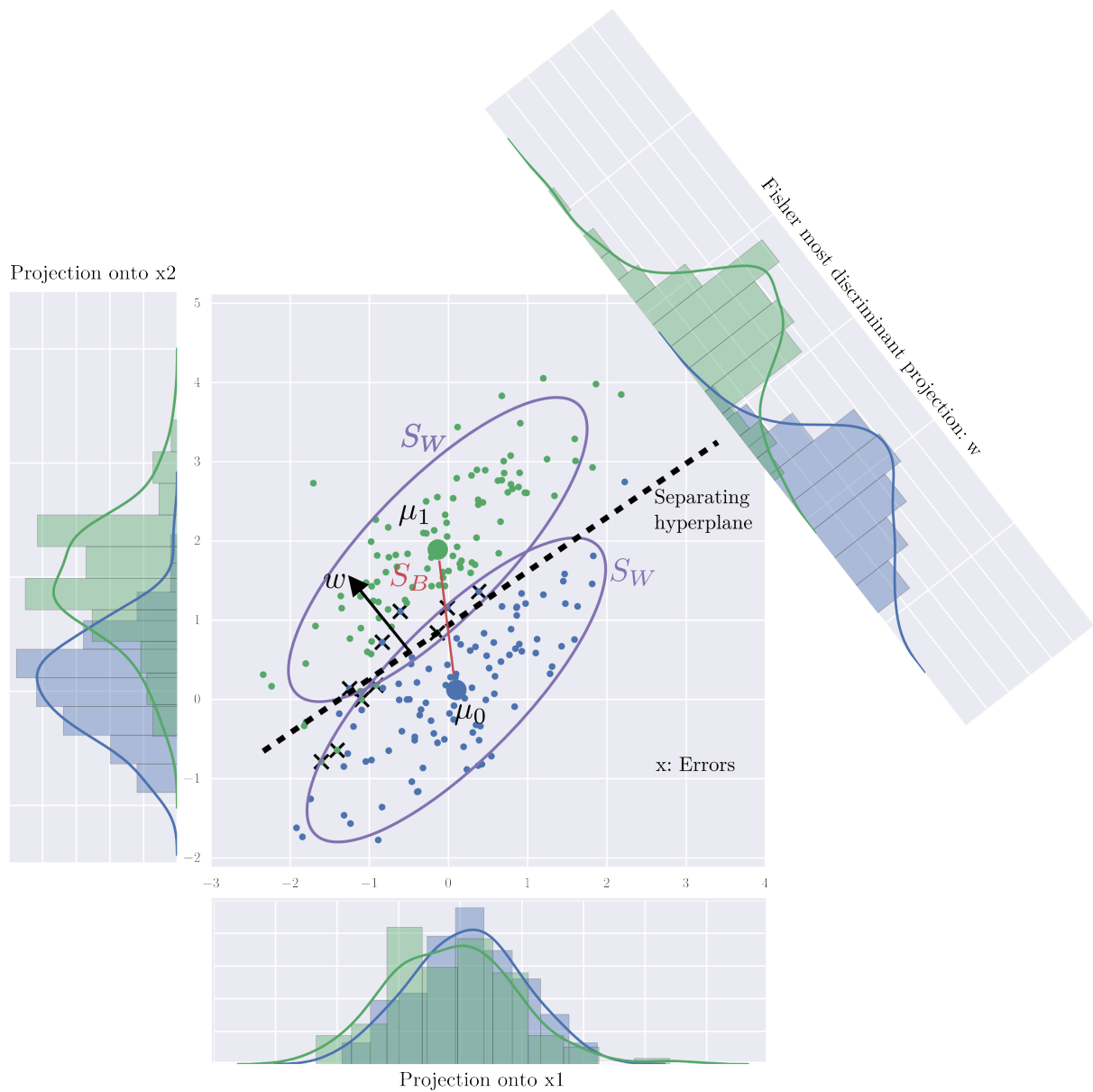


Fig. 6: Most discriminant projections, Naive and Fisher methods

8.2.2 Geometric Method: Fisher's Linear Discriminant

Principles:

- Dimensionality reduction before later classification.
- Find the most discriminant axis.
- Taking account the distribution, assuming same normal distribution for all classes.

Simply compute the **within class covariance** $\mathbf{S}_W$ to rotate the projection direction according to the point (elliptic) distribution:

$$\mathbf{w}_{\text{Fisher}} = \mathbf{S}_W^{-1}(\mu_1 - \mu_0).$$

This geometric method does not make any probabilistic assumptions, instead it relies on distances. It looks for the **linear projection** of the data points onto a vector, $\mathbf{w}$, that maximizes the between/within variance ratio, denoted $F(\mathbf{w})$. Under a few assumptions, it will provide the same results as linear discriminant analysis (LDA), explained below.

Suppose two classes of observations, C_0 and C_1 , have means μ_0 and μ_1 and the same total within-class scatter ("covariance") matrix,

$$\begin{aligned} \mathbf{S}_W &= \sum_{i \in C_0} (\mathbf{x}_i - \mu_0)(\mathbf{x}_i - \mu_0)^T + \sum_{j \in C_1} (\mathbf{x}_j - \mu_1)(\mathbf{x}_j - \mu_1)^T \\ &= \mathbf{X}_c^T \mathbf{X}_c, \end{aligned}$$

where $\mathbf{X}_c$ is the $(N \times P)$ matrix of data centered on their respective means:

$$\mathbf{X}_c = \begin{bmatrix} \mathbf{X}_0 - \mu_0 \\ \mathbf{X}_1 - \mu_1 \end{bmatrix},$$

where $\mathbf{X}_0$ and $\mathbf{X}_1$ are the $(N_0 \times P)$ and $(N_1 \times P)$ matrices of samples of classes C_0 and C_1 .

Let $\mathbf{S}_B$ being the scatter "between-class" matrix, given by

$$\mathbf{S}_B = (\mu_1 - \mu_0)(\mu_1 - \mu_0)^T.$$

The linear combination of features $\mathbf{w}^T x$ have means $\mathbf{w}^T \mu_i$ for $i = 0, 1$, and variance $\mathbf{w}^T \mathbf{X}_c^T \mathbf{X}_c \mathbf{w}$. Fisher defined the separation between these two distributions to be the ratio of the variance between the classes to the variance within the classes:

$$\begin{aligned} F_{\text{Fisher}}(\mathbf{w}) &= \frac{\sigma_{\text{between}}^2}{\sigma_{\text{within}}^2} \\ &= \frac{(\mathbf{w}^T \mu_1 - \mathbf{w}^T \mu_0)^2}{\mathbf{w}^T \mathbf{X}_c^T \mathbf{X}_c \mathbf{w}} \\ &= \frac{(\mathbf{w}^T (\mu_1 - \mu_0))^2}{\mathbf{w}^T \mathbf{X}_c^T \mathbf{X}_c \mathbf{w}} \\ &= \frac{\mathbf{w}^T (\mu_1 - \mu_0)(\mu_1 - \mu_0)^T \mathbf{w}}{\mathbf{w}^T \mathbf{X}_c^T \mathbf{X}_c \mathbf{w}} \\ &= \frac{\mathbf{w}^T \mathbf{S}_B \mathbf{w}}{\mathbf{w}^T \mathbf{S}_W \mathbf{w}}. \end{aligned}$$

In the two-class case, the maximum separation occurs by a projection on the $(\mu_1 - \mu_0)$ using the Mahalanobis metric $\mathbf{S}_W^{-1}$, so that

$$\mathbf{w} \propto \mathbf{S}_W^{-1}(\mu_1 - \mu_0).$$

Demonstration

Differentiating $F_{\text{Fisher}}(w)$ with respect to w gives

$$\begin{aligned}\nabla_{\mathbf{w}} F_{\text{Fisher}}(\mathbf{w}) &= 0 \\ \nabla_{\mathbf{w}} \left(\frac{\mathbf{w}^T \mathbf{S}_B \mathbf{w}}{\mathbf{w}^T \mathbf{S}_W \mathbf{w}} \right) &= 0 \\ (\mathbf{w}^T \mathbf{S}_W \mathbf{w})(2\mathbf{S}_B \mathbf{w}) - (\mathbf{w}^T \mathbf{S}_B \mathbf{w})(2\mathbf{S}_W \mathbf{w}) &= 0 \\ (\mathbf{w}^T \mathbf{S}_W \mathbf{w})(\mathbf{S}_B \mathbf{w}) &= (\mathbf{w}^T \mathbf{S}_B \mathbf{w})(\mathbf{S}_W \mathbf{w}) \\ \mathbf{S}_B \mathbf{w} &= \frac{\mathbf{w}^T \mathbf{S}_B \mathbf{w}}{\mathbf{w}^T \mathbf{S}_W \mathbf{w}} (\mathbf{S}_W \mathbf{w}) \\ \mathbf{S}_B \mathbf{w} &= \lambda (\mathbf{S}_W \mathbf{w}) \\ \mathbf{S}_W^{-1} \mathbf{S}_B \mathbf{w} &= \lambda \mathbf{w}.\end{aligned}$$

Since we do not care about the magnitude of $\mathbf{w}$, only its direction, we replaced the scalar factor $(\mathbf{w}^T \mathbf{S}_B \mathbf{w})/(\mathbf{w}^T \mathbf{S}_W \mathbf{w})$ by λ .

In the multiple-class case, the solutions w are determined by the eigenvectors of $\mathbf{S}_W^{-1} \mathbf{S}_B$ that correspond to the $K - 1$ largest eigenvalues.

However, in the two-class case (in which $\mathbf{S}_B = (\mu_1 - \mu_0)(\mu_1 - \mu_0)^T$) it is easy to show that $\mathbf{w} = \mathbf{S}_W^{-1}(\mu_1 - \mu_0)$ is the unique eigenvector of $\mathbf{S}_W^{-1} \mathbf{S}_B$:

$$\begin{aligned}\mathbf{S}_W^{-1}(\mu_1 - \mu_0)(\mu_1 - \mu_0)^T \mathbf{w} &= \lambda \mathbf{w} \\ \mathbf{S}_W^{-1}(\mu_1 - \mu_0)(\mu_1 - \mu_0)^T \mathbf{S}_W^{-1}(\mu_1 - \mu_0) &= \lambda \mathbf{S}_W^{-1}(\mu_1 - \mu_0),\end{aligned}$$

where here $\lambda = (\mu_1 - \mu_0)^T \mathbf{S}_W^{-1}(\mu_1 - \mu_0)$. Which leads to the result

$$\mathbf{w} \propto \mathbf{S}_W^{-1}(\mu_1 - \mu_0).$$

The separating hyperplane

The separating hyperplane is a $P - 1$ -dimensional hyper surface, orthogonal to the projection vector, w . There is no single best way to find the origin of the plane along w , or equivalently the classification threshold that determines whether a point should be classified as belonging to C_0 or to C_1 . However, if the projected points have roughly the same distribution, then the threshold can be chosen as the hyperplane exactly between the projections of the two means, i.e. as

$$T = \mathbf{w} \cdot \frac{1}{2}(\mu_1 - \mu_0).$$

8.2.3 Generative Model: Linear Discriminant Analysis (LDA)

- Probabilistic generalization of Fisher's linear discriminant.
 - Generative model of the **conditional distribution** of the input data $\mathbf{x}$ given the label k : $p(\mathbf{x}|y = k)$.
 - Uses Bayes' rule to provide the **posterior distribution** of the label k given the input data $\mathbf{x}$: $p(y = k|\mathbf{x})$.
 - Uses Bayes' rule to fix the threshold based on prior probabilities of classes.
1. First compute the class-**conditional distributions** of $\mathbf{x}$ given class C_k : $p(\mathbf{x}|C_k) = \mathcal{N}(\mathbf{x}|\mu_k, \mathbf{S}_W)$. Where $\mathcal{N}(\mathbf{x}|\mu_k, \mathbf{S}_W)$ is the multivariate Gaussian distribution defined over a P -dimensional vector x of continuous variables, which is given by

$$\mathcal{N}(\mathbf{x}|\mu_{\mathbf{k}}, \mathbf{S}_{\mathbf{W}}) = \frac{1}{(2\pi)^{P/2} |\mathbf{S}_{\mathbf{W}}|^{1/2}} \exp\left\{-\frac{1}{2}(\mathbf{x} - \mu_{\mathbf{k}})^T \mathbf{S}_{\mathbf{W}}^{-1} (\mathbf{x} - \mu_{\mathbf{k}})\right\}$$

2. Estimate the **prior probabilities** of class k , $p(C_k) = N_k/N$.
3. Compute **posterior probabilities** (ie. the probability of a each class given a sample) combining conditional with priors using Bayes' rule:

$$p(C_k|\mathbf{x}) = \frac{p(C_k)p(\mathbf{x}|C_k)}{p(\mathbf{x})}$$

Where $p(x)$ is the marginal distribution obtained by summing of classes: As usual, the denominator in Bayes' theorem can be found in terms of the quantities appearing in the numerator, because

$$p(x) = \sum_k p(\mathbf{x}|C_k)p(C_k)$$

4. Classify $\mathbf{x}$ using the Maximum-a-Posteriori probability: $C_k = \arg \max_{C_k} p(C_k|\mathbf{x})$

LDA is a **generative model** since the class-conditional distributions can be used to generate samples of each classes.

LDA is useful to deal with imbalanced group sizes (eg.: $N_1 \gg N_0$) since priors probabilities can be used to explicitly re-balance the classification by setting $p(C_0) = p(C_1) = 1/2$ or whatever seems relevant.

LDA can be generalized to the multiclass case with $K > 2$.

With $N_1 = N_0$, LDA lead to the same solution than Fisher's linear discriminant.

Question: How many parameters are required to estimate to perform a LDA?

Application with scikit-learn

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA

# Dataset 2 two multivariate normal
n_samples, n_features = 100, 2
mean0, mean1 = np.array([0, 0]), np.array([0, 2])
Cov = np.array([[1, .8], [.8, 1]])
np.random.seed(42)
X0 = np.random.multivariate_normal(mean0, Cov, n_samples)
X1 = np.random.multivariate_normal(mean1, Cov, n_samples)
X = np.vstack([X0, X1])
y = np.array([0] * X0.shape[0] + [1] * X1.shape[0])
m = X.mean(axis=0)

# Naive rule
w_naive = mean1 - mean0
w_naive = w_naive / np.linalg.norm(w_naive, ord=2) * 2
score_naive = np.dot(X, w_naive)

# Fischer rule
Xc = np.vstack([X0 - mean0, X1 - mean1])
Sw = np.dot(Xc.T, Xc)
w_fisher = np.dot(np.linalg.inv(Sw), mean1 - mean0)
```

(continues on next page)

(continued from previous page)

```

w_fisher = w_fisher / np.linalg.norm(w_fisher, ord=2) * 2
score_fisher = np.dot(X, w_fisher)

# LDA with scikit-learn
lda = LDA()
score_lda = lda.fit(X, y).transform(X).ravel()
w_lda = lda.coef_.ravel()
w_lda = w_lda / np.linalg.norm(w_lda, ord=2) * 2
y_pred_lda = lda.predict(X)

errors = y_pred_lda != y
print("Nb errors=%i, error rate=%.2f" %
      (errors.sum(), errors.sum() / len(y_pred_lda)))

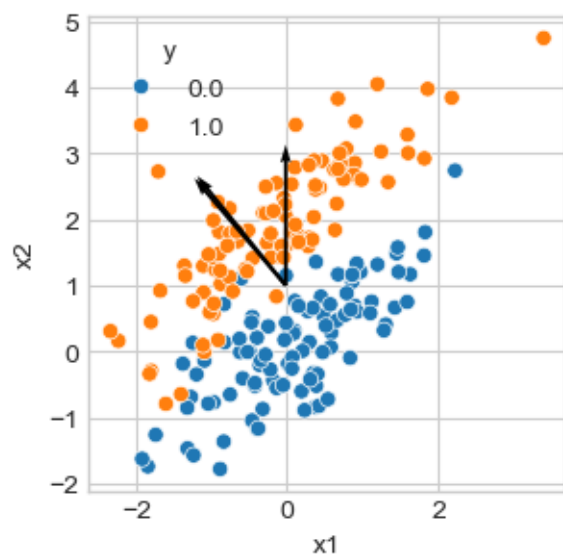
# Plot
plt.figure(figsize=(fig_w * 0.5, fig_w * 0.5))
data = pd.DataFrame(np.hstack([X, y.reshape(-1, 1)]), columns=("x1", "x2", "y"))
ax_ = sns.scatterplot(data=data, x="x1", y="x2", hue="y")
ax_.quiver([m[0]] * 3, [m[1]] * 3,
           [w_naive[0], w_fisher[0], w_lda[0]],
           [w_naive[1], w_fisher[1], w_lda[1]],
           units='xy', scale=1)

scores = [("Naive", score_naive), ("Fisher", score_fisher), ("LDA", score_lda)]
colors = plt.rcParams['axes.prop_cycle'].by_key()['color']
#colors = [colors[i] for i in [0, 2]]

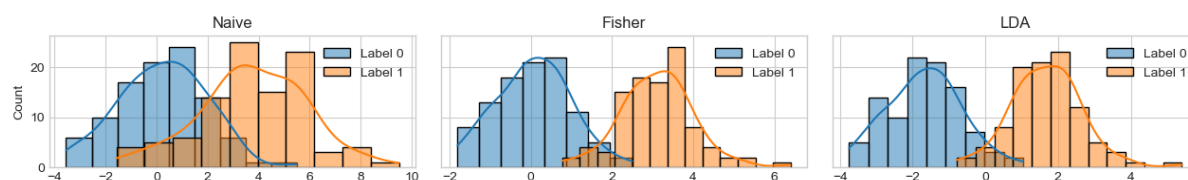
#Plot
fig, axes = plt.subplots(1, 3, figsize=(fig_w * 2, fig_h * .5), sharey=True)
for ax, (title, score) in zip(axes, scores):
    for lab in np.unique(y):
        sns.histplot(score[y == lab], ax=ax, label=f"Label {lab}", kde=True,
        ↪color=colors[lab])
    ax.set_title(title)
    ax.legend()
fig.suptitle('Projection on discriminative directions', fontsize=16)
plt.tight_layout()
plt.show()

```

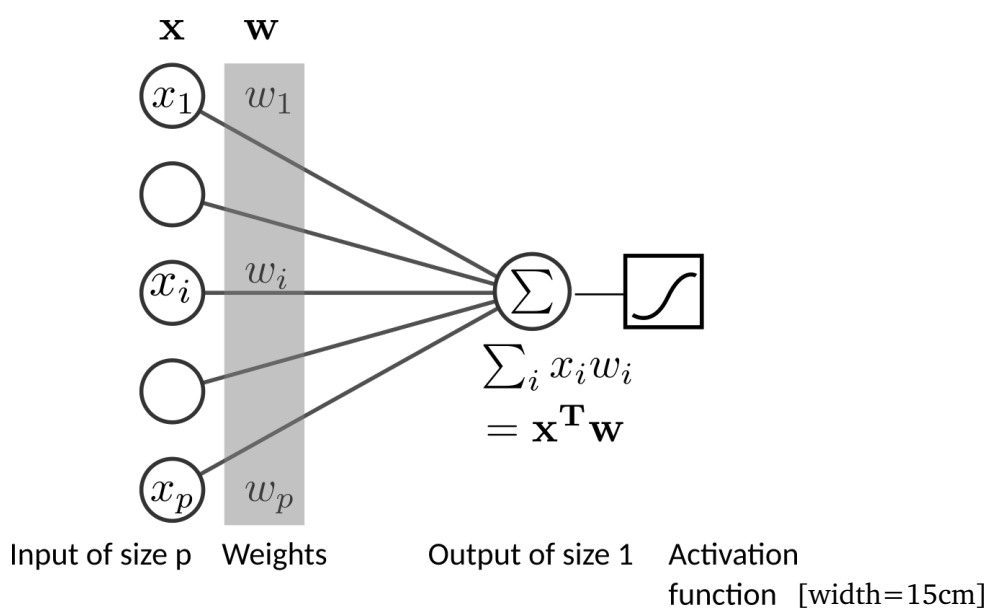
```
Nb errors=10, error rate=0.05
```



Projection on discriminative directions



8.2.4 Logistic Regression



Principles:

1. Map the output of a linear model: $\mathbf{w}^T \mathbf{x}$ into a score z . This step performs a **projection** or a **rotation** of input sample into a good discriminative one-dimensional sub-space, that best discriminate sample of current class vs sample of other classes.
2. Using an activation function $\sigma(\cdot)$, this score (a.k.a decision function) is transformed, to a “posterior probabilities” of a class k : $P(y = k | \mathbf{x}) = \sigma(\mathbf{w}^T \mathbf{x})$.

Properties:

- Consider only the posterior probability of a class k : $P(y = k|\mathbf{x})$
- Multiclass Classification problems can be seen as several binary classification problems $y_i \in \{0, 1\}$ where the classifier aims to discriminate the sample of the current class (label 1) versus the samples of other classes (label 0).
- The decision surfaces (orthogonal hyperplan to $\mathbf{w}$) correspond to $\sigma(z) = \text{constant}$, so that $\mathbf{x}^T \mathbf{w} = \text{constant}$ and hence the decision surfaces are linear functions of $\mathbf{x}$, even if the function $\sigma(\cdot)$ is nonlinear.
- A thresholding of the activation (shifted by the bias or intercept) provides the predicted class label.

Linear Discriminant Analysis (LDA) vs. Logistic Regression

Feature	Linear Discriminant Analysis (LDA)	Logistic Regression
Model Type	Generative model	Discriminative model
What It Models	Joint probability: $P(x, y) = P(x \text{ mid } y) P(y)$	Posterior probability: $P(y x)$
Ass umptions	Gaussian class-conditional distributions with equal covariance	No distributional assumption on features
Decision Boundary	Linear (under equal covariance assumption)	Linear
Probability Estimation	Uses Bayes' rule + Gaussian likelihood	Uses sigmoid of linear function
Interpre tability	Less intuitive, based on data distribution	Coefficients directly reflect impact on class log-odds
Per formance	Can outperform logistic regression when assumptions hold	More robust when assumptions (e.g., normality) are violated
Use in Mu lti-class	Naturally extends to multiclass	Extends via one-vs-rest or softmax (multinomial logistic)
Regula riza-tion	Not built-in (requires extensions like shrinkage LDA)	Easily regularized (e.g., with L1/L2 penalties)

Key Insights

- **LDA is generative**: it models how the data was generated (distribution of features given class), then uses Bayes' theorem to classify.
- **Logistic regression is discriminative**: it models the boundary between classes directly.

Activation Functions for Classification (Sigmoid and Softmax)

The **sigmoid** and **softmax** functions are closely related—they both transform real-valued inputs (*logits*) into probabilities, but they are used in different settings:

The **sigmoid function** maps real-valued inputs (called *logits*) into the interval $[0, 1]$, making them interpretable as probabilities (for scalar z):

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

It has the following properties:

- $\sigma(z) \rightarrow 1$ as $z \rightarrow +\infty$

- $\sigma(z) \rightarrow 0$ as $z \rightarrow -\infty$
- $\sigma(0) = 0.5$

In binary classification, we use $\sigma(\mathbf{w}^\top \mathbf{x})$ to estimate the probability of the positive class. This function is differentiable, which is essential for optimization via gradient descent.

The **softmax function** converts raw scores z_k into probabilities:

$$\hat{p}_k = \frac{e^{z_k}}{\sum_{j=1}^K e^{z_j}}$$

It ensures that output probabilities are positive and sum to 1, making it suitable for use with cross-entropy.

Softmax function (for vector $\mathbf{z} \in \mathbb{R}^K$):

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} \quad \text{for } i = 1, \dots, K$$

The **sigmoid function** is a special case of the **softmax** function when you have **two classes (binary classification)**. Given two logits z_0 and z_1 , the softmax for class 1 is:

$$\text{softmax}(z_1) = \frac{e^{z_1}}{e^{z_0} + e^{z_1}}$$

If we define $z = z_1 - z_0$, then:

$$\text{softmax}(z_1) = \frac{1}{1 + e^{-(z_1 - z_0)}} = \sigma(z)$$

So: **Sigmoid = Softmax over 2 logits, if one logit is fixed as 0 (reference class)**

Intuition

- **Sigmoid** gives the probability of one class (positive class) vs. its complement.
- **Softmax** generalizes this to $K > 2$ classes, ensuring the sum of probabilities is 1.

Summary

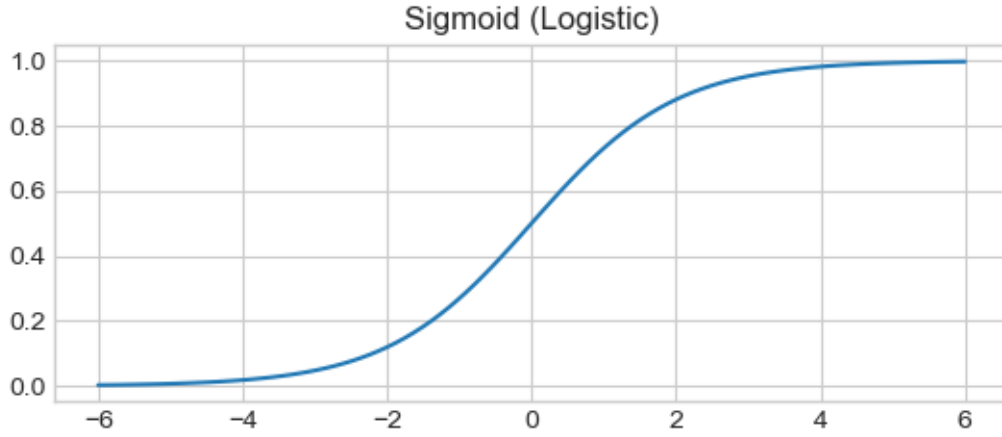
F unction	Use Case	Output
Sigmoid	Binary classification	Scalar in (0, 1)
Softmax	Multiclass classification	Vector of probabilities summing to 1 over classes

- Sigmoid and Softmax maps logits to probabilities over classes
- The **sigmoid function is equivalent to a 2-class softmax**.
- Both map logits to probabilities but are used in different classification settings.
- Softmax ensures a **normalized probability distribution** over multiple classes.

```
def sigmoid(x): return 1 / (1 + np.exp(-x))

x = np.linspace(-6, 6, 100)
plt.plot(x, sigmoid(x))
plt.grid(True)
plt.title('Sigmoid (Logistic)')
```

```
Text(0.5, 1.0, 'Sigmoid (Logistic)')
```



Loss Functions for Classification

Negative Log-Likelihood (NLL)

The Negative Log-Likelihood (NLL) for a dataset of observations $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$, where each x_i is an input and y_i is the corresponding label.

General Formulation of NLL

Given a **probabilistic model** that predicts $P(y_i | x_i; \theta)$, where θ are the model parameters (e.g., weights in a neural network or logistic regression), the **Negative Log-Likelihood** over the dataset is:

$$\mathcal{L}_{\text{NLL}}(\theta) = - \sum_{i=1}^n \log P(y_i | x_i; \theta)$$

This expression encourages the model to assign **high probability** to the **correct labels**.

Binary Classification (Sigmoid Output) a.k.a. Logistic or Binary Cross-Entropy Loss

For binary labels $y_i \in \{0, 1\}$, and using:

$$P(y_i = 1 | x_i; \theta) = \hat{p}_i = \sigma(z_i)$$

where $\hat{p}$ is the predicted probability of the positive class (obtained via a sigmoid activation, $\sigma(z_i)$). The NLL becomes:

$$\mathcal{L}_{\text{NLL}} = - \log \prod_{i=1}^n [\hat{p}_i^{y_i} + (1 - \hat{p}_i)^{(1-y_i)}]$$

$$\mathcal{L}_{\text{NLL}} = - \sum_{i=1}^n [y_i \log(\hat{p}_i) + (1 - y_i) \log(1 - \hat{p}_i)]$$

or

$$\mathcal{L}_{\text{NLL}}(\mathbf{w}) = \sum_{i=1}^n [\log(1 + e^{-z_i}) + (1 - y_i) z_i] \quad \text{with } y_i \in \{0, 1\}.$$

Recoding output $y_i \in \{-1, +1\}$, the expression simplifies to

$$\mathcal{L}_{\text{NLL}}(\mathbf{w}) = \sum_{i=1}^n \log(1 + e^{-y_i \cdot z_i}) \quad \text{with } y_i \in \{-1, +1\}.$$

For linear models: $z_i = \mathbf{w}^\top \mathbf{x}_i$. For more details, see the [Demonstration of Negative Log-Likelihood \(NLL\) Loss](#).

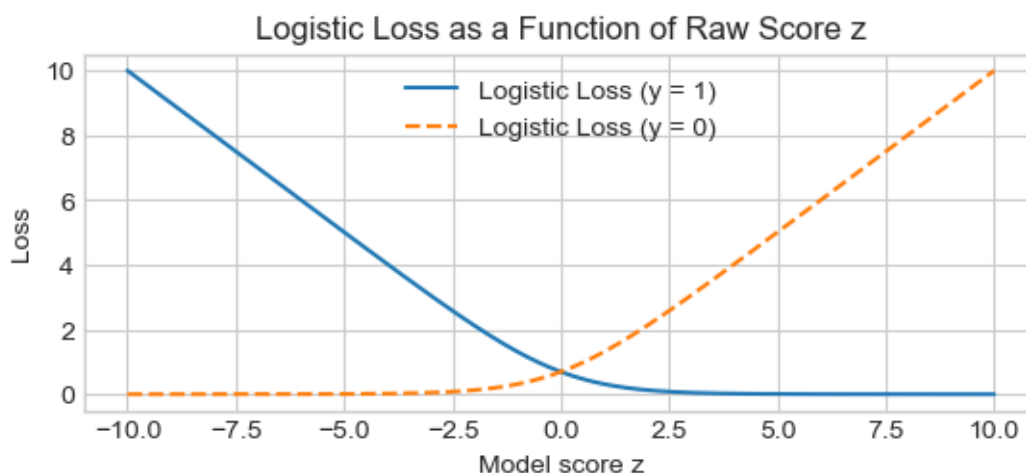
This expression is also known as the **logistic loss** or **binary cross-entropy**. It penalizes confident but incorrect predictions very heavily (e.g., predicting $\hat{p} = 0.99$ when $y = 0$).

```
# Define the logistic loss for binary classification
def logistic_loss(z, y):
    return np.log(1 + np.exp(-y * z))

# Input range (raw scores from the model)
z = np.linspace(-10, 10, 400)

# Logistic loss for y = 1 and y = 0
loss_y1 = logistic_loss(z, 1)
loss_y0 = logistic_loss(z, -1) # equivalent to logistic loss for y=0

# Plotting
plt.figure()
plt.plot(z, loss_y1, label="Logistic Loss (y = 1)")
plt.plot(z, loss_y0, label="Logistic Loss (y = 0)", linestyle="--")
plt.title("Logistic Loss as a Function of Raw Score z")
plt.xlabel("Model score z")
plt.ylabel("Loss")
plt.legend()
plt.show()
```



Gradient Logistic or Binary Cross-Entropy Loss for Linear Models

For linear models where $z_i = \mathbf{w}^\top \mathbf{x}_i$ the gradient of the NLL according to the coefficients vector $\mathbf{w}$ is:

$$\nabla_{\mathbf{w}} \mathcal{L}_{\text{NLL}} = \sum_{i=1}^n (\sigma(\mathbf{w}^\top \mathbf{x}_i) - y_i) \mathbf{x}_i$$

Multiclass Classification (Softmax Output) a.k.a. Cross-Entropy Loss

Assume $y_i \in \{1, 2, \dots, K\}$, and the model outputs softmax probabilities $\hat{p}_{i,k} = P(y_i = k \mid x_i; \theta)$. Then:

$$\mathcal{L}_{\text{NLL}} = - \sum_{i=1}^n \log \hat{p}_{i,y_i}$$

If you use one-hot encoded labels y_i , the NLL is also written as:

$$\mathcal{L}_{\text{NLL}} = - \sum_{i=1}^n \sum_{k=1}^K y_{i,k} \log(\hat{p}_{i,k})$$

This is equivalent to the **cross-entropy loss** when combined with softmax.

Summary

- Negative Log-Likelihood = general loss for probabilistic models
- Logistic loss = binary cross-entropy
- Cross-Entropy loss = NLL + Softmax (for multiclass problems)
- These losses are convex (for linear models), interpretable, and widely used in training classifiers like logistic regression and neural networks.

See also [Scikit learn doc](#)

Hinge loss or ℓ_1 loss

TODO

Logistic Regression Summary

Logistic regression minimizes the Cross-Entropy loss i.e. NLL with Sigmoid (binary problems) or NLL with Softmax (multiclass problems):

$$\min_{\mathbf{w}} \text{Logistic}(\mathbf{w}) = \mathcal{L}_{\text{NLL}}(\mathbf{w})$$

Logistic Regression with Scikit-Learn

```
import sklearn.linear_model as lm
logreg = lm.LogisticRegression(penalty=None).fit(X, y)

logreg.fit(X, y)
y_pred_logreg = logreg.predict(X)

errors = y_pred_logreg != y
print("Nb errors=%i, error rate=%.2f" %
      (errors.sum(), errors.sum() / len(y_pred_logreg)))
print(logreg.coef_.round(2))
```

```
Nb errors=10, error rate=0.05
[[-5.15  5.57]]
```

8.2.5 Regularization using penalization of coefficients

The penalties use in regression are also used in classification. The only difference is the loss function generally the negative log likelihood (cross-entropy) or the hinge loss. We will explore:

Summary

- Ridge (also called ℓ_2) penalty: $\|\mathbf{w}\|_2^2$. It shrinks coefficients toward 0.
- Lasso (also called ℓ_1) penalty: $\|\mathbf{w}\|_1$. It performs feature selection by setting some coefficients to 0.
- ElasticNet (also called $\ell_1\ell_2$) penalty: $\alpha (\rho \|\mathbf{w}\|_1 + (1 - \rho) \|\mathbf{w}\|_2^2)$. It performs selection of group of correlated features by setting some coefficients to 0.

```
# Dataset with some correlation
X, y = make_classification(n_samples=100, n_features=10,
                          n_informative=5, n_redundant=3,
                          n_classes=2, random_state=3, shuffle=False)

# Logistic Regression unpenalized
lr = lm.LogisticRegression(penalty=None).fit(X, y)

# Logistic Regression with L2 penalty
l2 = lm.LogisticRegression(penalty='l2', C=.1).fit(X, y) # lambda = 1 / C!

# Logistic Regression with L1 penalty
# use solver 'saga' to handle L1 penalty. lambda = 1 / C
l1 = lm.LogisticRegression(penalty='l1', C=.1, solver='saga').fit(X, y)

# Logistic Regression with L1/L2 penalties
l1l2 = lm.LogisticRegression(penalty='elasticnet', C=.1, l1_ratio=0.5,
                             solver='saga').fit(X, y) # lambda = 1 / C!

print(pd.DataFrame(np.vstack((lr.coef_, l2.coef_, l1.coef_, l1l2.coef_)),
                    index=['lr', 'l2', 'l1', 'l1l2']).round(2))
```

	0	1	2	3	4	5	6	7	8	9
lr	0.04	1.14	-0.28	0.57	0.55	-0.03	0.17	0.37	-0.42	0.39
l2	-0.05	0.52	-0.21	0.34	0.26	-0.05	0.14	0.27	-0.25	0.21
l1	0.00	0.31	0.00	0.10	0.00	0.00	0.00	0.26	0.00	0.00
l1l2	-0.01	0.41	-0.15	0.29	0.12	0.00	0.00	0.20	-0.10	0.06

Understanding the effect of penalty using ℓ_2 -regularization Fisher's linear classification

When the matrix $\mathbf{S}_W$ is not full rank or $P \gg N$, the The Fisher most discriminant projection estimate of the is not unique. This can be solved using a biased version of $\mathbf{S}_W$:

$$\mathbf{S}_W^{\text{Ridge}} = \mathbf{S}_W + \lambda \mathbf{I}$$

where $\mathbf{I}$ is the $P \times P$ identity matrix. This leads to the regularized (ridge) estimator of the Fisher's linear discriminant analysis:

$$\mathbf{w}^{\text{Ridge}} \propto (\mathbf{S}_W + \lambda \mathbf{I})^{-1}(\mu_1 - \mu_0)$$

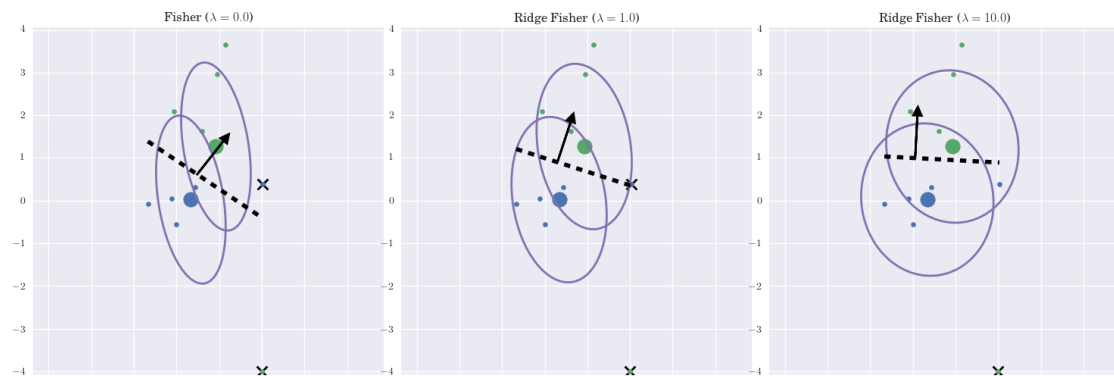


Fig. 7: The Ridge Fisher most discriminant projection

Increasing λ will:

- Shrinks the coefficients toward zero.
- The covariance will converge toward the diagonal matrix, reducing the contribution of the pairwise covariances.

8.2.6 ℓ_2 -regularized logistic regression

The **objective function** to be minimized is now the combination of the logistic loss (Negative Log Likelihood) with a penalty of the L2 norm of the weights vector. In the two-class case, using the 0/1 coding we obtain:

$$\min_{\mathbf{w}} \text{Logistic L2}(\mathbf{w}) = \mathcal{L}_{\text{NLL}}(\mathbf{w}) + \lambda \|\mathbf{w}\|^2$$

```
import sklearn.linear_model as lm
lrl2 = lm.LogisticRegression(penalty='l2', C=.1)
# This class implements regularized logistic regression.
# C is the Inverse of regularization strength. Large value => no regularization.

lrl2.fit(X, y)
y_pred_l2 = lrl2.predict(X)
prob_pred_l2 = lrl2.predict_proba(X)

print("Probas of 5 first samples for class 0 and class 1:")
print(prob_pred_l2[:5, :].round(2))

print("Coef vector:")
print(lrl2.coef_.round(2))

# Retrieve proba from coef vector
probas = 1 / (1 + np.exp(-(np.dot(X, lrl2.coef_.T) + lrl2.intercept_))).ravel()
print("Diff", np.max(np.abs(prob_pred_l2[:, 1] - probas)))

errors = y_pred_l2 != y
print("Nb errors=%i, error rate=%.2f" % (errors.sum(), errors.sum() / len(y)))
```

```

Probabs of 5 first samples for class 0 and class 1:
[[0.89 0.11]
 [0.72 0.28]
 [0.73 0.27]
 [0.75 0.25]
 [0.48 0.52]]
Coef vector:
[[-0.05  0.52 -0.21  0.34  0.26 -0.05  0.14  0.27 -0.25  0.21]]
Diff 0.0
Nb errors=24, error rate=0.24

```

8.2.7 Lasso logistic regression (ℓ_1 -regularization)

The **objective function** to be minimized is now the combination of the logistic loss $-\log \mathcal{L}(\mathbf{w})$ with a penalty of the L1 norm of the weights vector. In the two-class case, using the 0/1 coding we obtain:

$$\min_{\mathbf{w}} \text{Logistic Lasso}(\mathbf{w}) = \mathcal{L}_{\text{NLL}}(\mathbf{w}) + \lambda \|\mathbf{w}\|_1$$

```

import sklearn.linear_model as lm
lrl1 = lm.LogisticRegression(penalty='l1', C=.1, solver='saga') # lambda = 1 / C!

# This class implements regularized logistic regression. C is the Inverse of
↳ regularization strength.
# Large value => no regularization.

lrl1.fit(X, y)
y_pred_lrl1 = lrl1.predict(X)

errors = y_pred_lrl1 != y
print("Nb errors=%i, error rate=%.2f" % (errors.sum(), errors.sum() / len(y_pred_
↳ lrl1)))

print("Coef vector:")
print(lrl1.coef_.round(2))

```

```

Nb errors=27, error rate=0.27
Coef vector:
[[0.    0.31 0.    0.1  0.    0.    0.    0.26 0.    0.   ]]

```

8.2.8 Linear Support Vector Machine (ℓ_2 -regularization with Hinge loss)

Support Vector Machine seek for separating hyperplane with maximum margin to enforce robustness against noise. Like logistic regression it is a **discriminative method** that only focuses of predictions.

Here we present the non separable case of Maximum Margin Classifiers with ± 1 coding (ie.: $y_i \in \{-1, +1\}$).

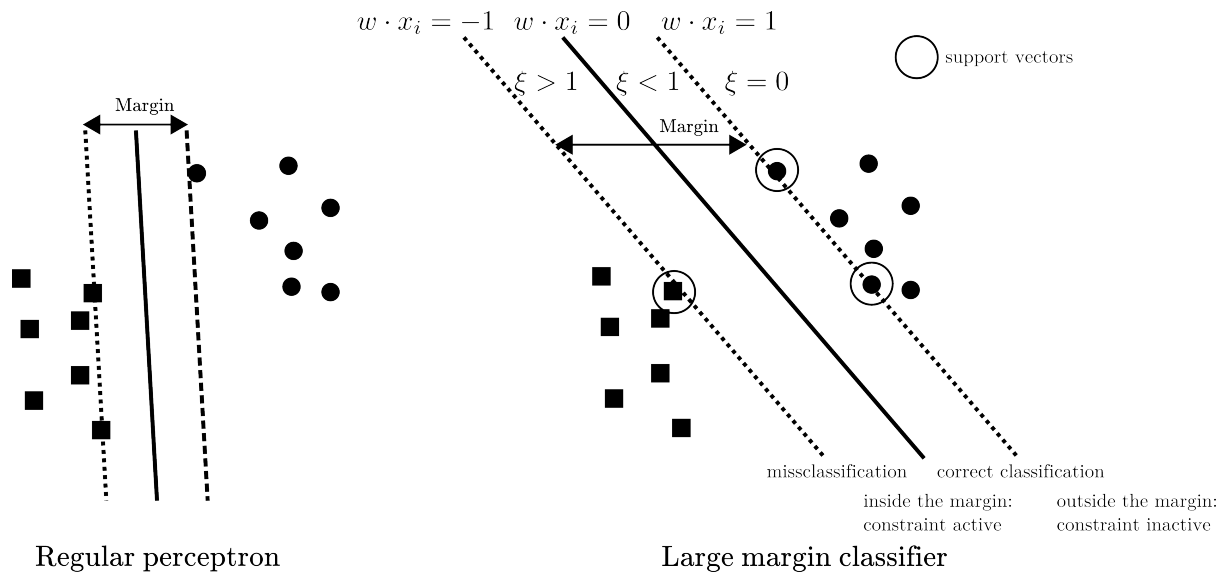


Fig. 8: Linear lar margin classifiers

Linear SVM for classification (also called SVM-C or SVC) minimizes:

$$\min_{\mathbf{w}} \text{Linear SVM}(\mathbf{w}) = \|\mathbf{w}\|_2^2 + C \sum_i^n \max(0, 1 - y_i (\mathbf{w} \cdot \mathbf{x}_i))$$

where $\max(0, 1 - y_i (\mathbf{w} \cdot \mathbf{x}_i))$ is the **hinge loss**.

Here we introduced the slack variables: ξ_i , with $\xi_i = 0$ for points that are on or inside the correct margin boundary and $\xi_i = |y_i - (\mathbf{w} \cdot \mathbf{x}_i)|$ for other points. Thus:

1. If $y_i(\mathbf{w} \cdot \mathbf{x}_i) \geq 1$ then the point lies outside the margin but on the correct side of the decision boundary. In this case the constraint is thus not active for this point. It does not contribute to the prediction.
2. If $1 > y_i(\mathbf{w} \cdot \mathbf{x}_i) \geq 0$ then the point lies inside the margin and on the correct side of the decision boundary. In this case the constraint is active for this point. It does contribute to the prediction as a support vector.
3. If $0 < y_i(\mathbf{w} \cdot \mathbf{x}_i)$ then the point is on the wrong side of the decision boundary (missclassification). In this case the constraint is active for this point. It does contribute to the prediction as a support vector.

So linear SVM is closed to Ridge logistic regression, using the hinge loss instead of the logistic loss. Both will provide very similar predictions.

```
from sklearn import svm

svmlin = svm.LinearSVC(C=.1)
# Remark: by default LinearSVC uses squared_hinge as loss
svmlin.fit(X, y)
y_pred_svmlin = svmlin.predict(X)

errors = y_pred_svmlin != y
print("Nb errors=%i, error rate=%.2f" %
      (errors.sum(), errors.sum() / len(y_pred_svmlin)))
```

(continues on next page)

(continued from previous page)

```
print("Coef vector:")
print(svmmlin.coef_.round(2))
```

```
Nb errors=20, error rate=0.20
Coef vector:
[[-0.    0.32 -0.09  0.17  0.16 -0.01  0.06  0.13 -0.16  0.13]]
```

8.2.9 Lasso linear Support Vector Machine (ℓ_1 -regularization)

Linear SVM with l1-regularization:

$$\min_{\mathbf{w}} \text{Lasso Linear SVM}(\mathbf{w}) = \|\mathbf{w}\|_1 + C \sum_i^n \max(0, 1 - y_i (\mathbf{w} \cdot \mathbf{x}_i))$$

```
from sklearn import svm

svmlinl1 = svm.LinearSVC(penalty='l1', dual=False)
# Remark: by default LinearSVC uses squared_hinge as loss

svmlinl1.fit(X, y)
y_pred_svmlinl1 = svmlinl1.predict(X)

errors = y_pred_svmlinl1 != y
print("Nb errors=%i, error rate=%.2f" % (errors.sum(), errors.sum() / len(y_pred_
    ↪svmlinl1)))
print("Coef vector:")
print(svmlinl1.coef_.round(2))
```

```
Nb errors=20, error rate=0.20
Coef vector:
[[-0.01  0.37 -0.12  0.24  0.17  0.    0.    0.1 -0.16  0.13]]
```

8.2.10 Elastic-net classification ($\ell_1\ell_2$ -regularization)

The **objective function** to be minimized is now the combination of a loss with combination of L1 and L2 penalties:

- For the NLL loss:

$$\min \text{Logistic Enet}(\mathbf{w}) = \mathcal{L}_{\text{NLL}}(\mathbf{w}) + \alpha (\rho \|\mathbf{w}\|_1 + (1 - \rho) \|\mathbf{w}\|_2^2)$$

- For the Hinge loss

$$\min \text{Hinge Enet}(\mathbf{w}) = \text{Hinge loss}(\mathbf{w}) + \alpha (\rho \|\mathbf{w}\|_1 + (1 - \rho) \|\mathbf{w}\|_2^2)$$

```
# Use SGD solver
enetlog = lm.SGDClassifier(loss="log_loss", penalty="elasticnet",
    alpha=0.1, l1_ratio=0.5, random_state=42)
```

(continues on next page)

(continued from previous page)

```

enetlog.fit(X, y)

# Or saga solver:
# enetloglike = lm.LogisticRegression(penalty='elasticnet',
#                                     C=.1, l1_ratio=0.5, solver='saga')

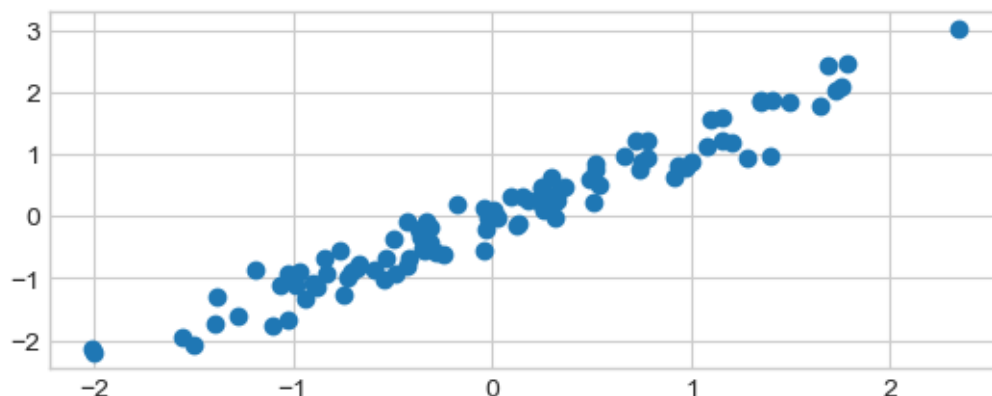
enethinge = lm.SGDClassifier(loss="hinge", penalty="elasticnet",
                             alpha=0.1, l1_ratio=0.5, random_state=42)
enethinge.fit(X, y)

print("Hinge loss and logistic loss provide almost the same predictions.")
print("Confusion matrix")
metrics.confusion_matrix(enetlog.predict(X), enethinge.predict(X))

print("Decision_function log x hinge losses:")
_ = plt.plot(enetlog.decision_function(X),
             enethinge.decision_function(X), "o")

```

Hinge loss **and** logistic loss provide almost the same predictions.
 Confusion matrix
 Decision_function log x hinge losses:



8.2.11 Classification performance evaluation metrics

[Wikipedia Sensitivity and specificity](#)

Imagine a study evaluating a new test that screens people for a disease. Each person taking the test either has or does not have the disease. The test outcome can be positive (classifying the person as having the disease) or negative (classifying the person as not having the disease). The test results for each subject may or may not match the subject's actual status. In that setting we compute the [Confusion Matrix](#) to derive the **classification metrics**:

Confusion Matrix

True / Call	Positive Call	Negative Call
P: Positive (Sick)	TP: True Positive. Sick people correctly identified as sick	FN: False Negative. Sick people incorrectly identified as healthy
N: Negative (Healthy)	FP: False Positive. Healthy people incorrectly identified as sick	TN: True Negative. Healthy people correctly identified as healthy

Derived Classification Metrics

See [Scikit-learn doc](#).

Metric	Definition	Formula
Accuracy (ACC)	Overall proportion of correct classifications	$ACC = (TP + TN) / (TP + FP + FN + TN)$
Sensitivity (SEN) (Recall, True Positive Rate, TPR)	Ability to correctly detect sick individuals	$SEN = TP / (TP + FN)$
Specificity (SPC) (True Negative Rate)	Ability to correctly identify healthy individuals	$SPC = TN / (TN + FP)$
Precision (PPV) (Positive Predictive Value)	Probability that predicted sick individuals are actually sick: True Positive / Positive call	$PPV = TP / (TP + FP)$
Negative Predictive Value (NPV)	Probability that predicted healthy individuals are actually healthy: True Negative / Negative call	$NPV = TN / (TN + FN)$
Balanced Accuracy (bACC)	Mean of sensitivity and specificity. To be used on imbalanced datasets	$bACC = 1/2 * (SEN + SPC)$

See also:

- F1 Score (or F-score) which is a weighted average of precision and recall are useful to deal with imbalanced datasets

```
from sklearn import metrics
y_pred = [0, 1, 0, 0]
y_true = [0, 1, 0, 1]

metrics.accuracy_score(y_true, y_pred)

# The overall precision an recall
metrics.precision_score(y_true, y_pred)
metrics.recall_score(y_true, y_pred)

# Recalls on individual classes: SEN & SPC
recalls = metrics.recall_score(y_true, y_pred, average=None)
recalls[0] # is the recall of class 0: specificity
recalls[1] # is the recall of class 1: sensitivity
```

(continues on next page)

(continued from previous page)

```
# Balanced accuracy
b_acc = recalls.mean()

# The overall precision an recall on each individual class
p, r, f, s = metrics.precision_recall_fscore_support(y_true, y_pred)
```

Area Under Curve (AUC) of Receiver operating characteristic (ROC)

Some classifier may have found a good discriminative projection w . However if the threshold to decide the final predicted class is poorly adjusted, the performances will highlight an high specificity and a low sensitivity or the contrary.

In this case it is recommended to use the AUC of a ROC analysis which basically provide a measure of overlap of the two classes when points are projected on the discriminative axis. See [Wikipedia: ROC and AUC](#).

```
score_pred = np.array([.1, .2, .3, .4, .5, .6, .7, .8])
y_true = np.array([0, 0, 0, 0, 1, 1, 1, 1])
thres = .9
y_pred = (score_pred > thres).astype(int)

print("With a threshold of %.2f, the rule always predict 0. Predictions:" % thres)
print(y_pred)
metrics.accuracy_score(y_true, y_pred)

# The overall precision an recall on each individual class
r = metrics.recall_score(y_true, y_pred, average=None)
print("Recalls on individual classes are:", r,
      "ie, 100% of specificity, 0% of sensitivity")

# However AUC=1 indicating a perfect separation of the two classes
auc = metrics.roc_auc_score(y_true, score_pred)
print("But the AUC of %.2f demonstrate a good classes separation." % auc)
```

```
With a threshold of 0.90, the rule always predict 0. Predictions:
[0 0 0 0 0 0 0 0]
Recalls on individual classes are: [1. 0.] ie, 100% of specificity, 0% of
↪sensitivity
But the AUC of 1.00 demonstrate a good classes separation.
```

8.2.12 Imbalanced classes

Learning with discriminative (logistic regression, SVM) methods is generally based on minimizing the misclassification of training samples, which may be unsuitable for imbalanced datasets where the recognition might be biased in favor of the most numerous class. This problem can be addressed with a generative approach, which typically requires more parameters to be determined leading to reduced performances in high dimension.

Dealing with imbalanced class may be addressed by three main ways (see Japkowicz and Stephen (2002) for a review), resampling, reweighting and one class learning.

In **sampling strategies**, either the minority class is oversampled or majority class is undersampled or some combination of the two is deployed. Undersampling (Zhang and Mani, 2003) the majority class would lead to a poor usage of the left-out samples. Sometime one cannot afford such strategy since we are also facing a small sample size problem even for the majority class. Informed oversampling, which goes beyond a trivial duplication of minority class samples, requires the estimation of class conditional distributions in order to generate synthetic samples. Here generative models are required. An alternative, proposed in (Chawla et al., 2002) generate samples along the line segments joining any/all of the k minority class nearest neighbors. Such procedure blindly generalizes the minority area without regard to the majority class, which may be particularly problematic with high-dimensional and potentially skewed class distribution.

Reweighting, also called cost-sensitive learning, works at an algorithmic level by adjusting the costs of the various classes to counter the class imbalance. Such reweighting can be implemented within SVM (Chang and Lin, 2001) or logistic regression (Friedman et al., 2010) classifiers. Most classifiers of Scikit learn offer such reweighting possibilities.

The `class_weight` parameter can be positioned into the "balanced" mode which uses the values of y to automatically adjust weights inversely proportional to class frequencies in the input data as $N/(2N_k)$.

```
# dataset
X, y = make_classification(n_samples=500,
                          n_features=5,
                          n_informative=2,
                          n_redundant=0,
                          n_repeated=0,
                          n_classes=2,
                          random_state=1,
                          shuffle=False)

print(*["#samples of class %i = %i;" % (lev, np.sum(y == lev))
        for lev in np.unique(y)])

print('# No Reweighting balanced dataset')
lr_inter = lm.LogisticRegression(C=1)
lr_inter.fit(X, y)
p, r, f, s = metrics.precision_recall_fscore_support(y, lr_inter.predict(X))
print("SPC: %.3f; SEN: %.3f" % tuple(r))
print('# => The predictions are balanced in sensitivity and specificity\n')

# Create imbalanced dataset, by subsampling sample of class 0: keep only 10% of
# class 0's samples and all class 1's samples.
n0 = int(np.rint(np.sum(y == 0) / 20))
subsample_idx = np.concatenate((np.where(y == 0)[0][:n0], np.where(y == 1)[0]))
Ximb = X[subsample_idx, :]
yimb = y[subsample_idx]
print(*["#samples of class %i = %i;" % (lev, np.sum(yimb == lev)) for lev in
        np.unique(yimb)])

print('# No Reweighting on imbalanced dataset')
lr_inter = lm.LogisticRegression(C=1)
lr_inter.fit(Ximb, yimb)
```

(continues on next page)

(continued from previous page)

```

p, r, f, s = metrics.precision_recall_fscore_support(
    yimb, lr_inter.predict(Ximb))
print("SPC: %.3f; SEN: %.3f" % tuple(r))
print('# => Sensitivity >> specificity\n')

print('# Reweighting on imbalanced dataset')
lr_inter_reweight = lm.LogisticRegression(C=1, class_weight="balanced")
lr_inter_reweight.fit(Ximb, yimb)
p, r, f, s = metrics.precision_recall_fscore_support(yimb,
                                                    lr_inter_reweight.
                                                    predict(Ximb))
print("SPC: %.3f; SEN: %.3f" % tuple(r))
print('# => The predictions are balanced in sensitivity and specificity\n')

```

```

#samples of class 0 = 250; #samples of class 1 = 250;
# No Reweighting balanced dataset
SPC: 0.940; SEN: 0.928
# => The predictions are balanced in sensitivity and specificity

#samples of class 0 = 12; #samples of class 1 = 250;
# No Reweighting on imbalanced dataset
SPC: 0.750; SEN: 0.996
# => Sensitivity >> specificity

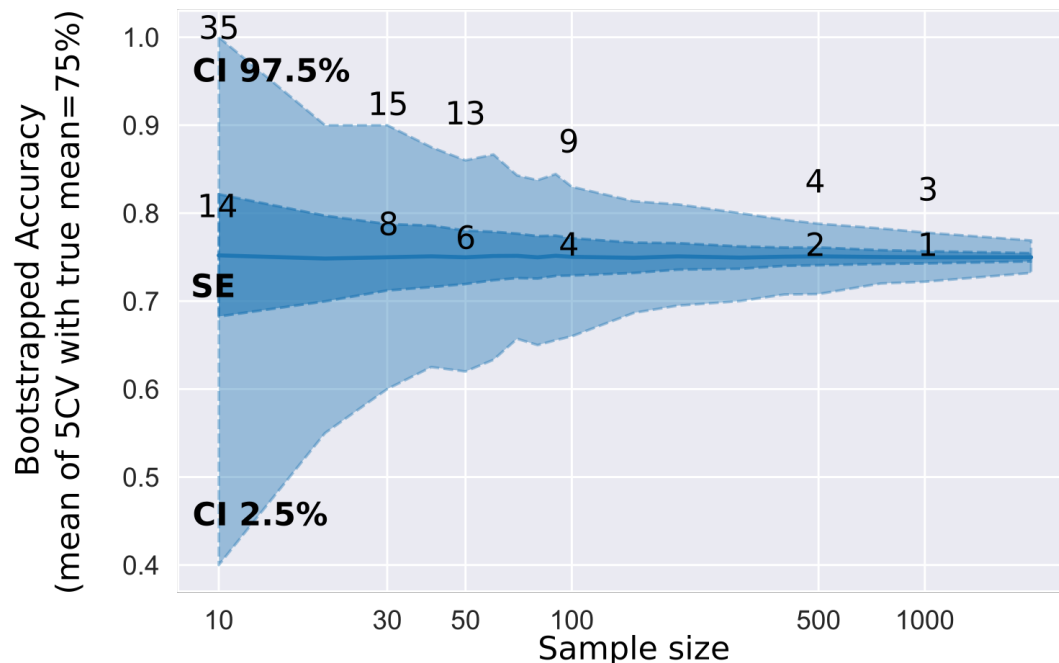
# Reweighting on imbalanced dataset
SPC: 1.000; SEN: 0.980
# => The predictions are balanced in sensitivity and specificity

```

8.2.13 Confidence interval cross-validation

Confidence interval CI classification accuracy measured by cross-validation:

Classif. accuracy Standard Error and 95% Confidence Interval



8.2.14 Significance of classification metrics

P-value of classification accuracy: Compare the number of correct classifications ($= \text{accuracy} \times N$) to the null hypothesis of Binomial distribution of parameters p (typically 50% of chance level) and N (Number of observations). Is 60% accuracy a significant prediction rate among 50 observations? Since this is an exact, **two-sided** test of the null hypothesis, the p-value can be divided by two since we test that the accuracy is superior to the chance level.

P-value of ROC-AUC: ROC-AUC measures the ranking of the two classes. Therefore non-parametric test can be used to assess the significance of the classes's separation. Mason and Graham (RMetS, 2002) show that the ROC area is equivalent to the Mann–Whitney U-statistic.

Mann–Whitney U test (also called the Mann–Whitney–Wilcoxon, Wilcoxon rank-sum test or Wilcoxon–Mann–Whitney test) is a nonparametric

```
import numpy as np
from sklearn import metrics
N_test = 50
X, y = make_classification(n_samples=200, random_state=42,
                          shuffle=False, class_sep=0.80)
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    test_size=N_test, random_state=42)

lr = lm.LogisticRegression().fit(X_train, y_train)

y_pred = lr.predict(X_test)
proba_pred = lr.predict_proba(X_test)[: , 1]
```

(continues on next page)

(continued from previous page)

```
acc = metrics.accuracy_score(y_test, y_pred)
bacc = metrics.balanced_accuracy_score(y_test, y_pred)
auc = metrics.roc_auc_score(y_test, proba_pred)

print("ACC=%.2f, bACC=%.2f, AUC=%.2f," % (acc, bacc, auc))
```

```
ACC=0.60, bACC=0.61, AUC=0.72,
```

```
from scipy import stats

# acc, N = 0.65, 70
k = int(acc * N_test)
acc_test = stats.binomtest(k=k, n=N_test, p=0.5, alternative='greater')
auc_pval = stats.mannwhitneyu(
    proba_pred[y_test == 0], proba_pred[y_test == 1]).pvalue

def is_significant(pval): return True if pval < 0.05 else False

print("ACC=%.2f (pval=%.3f, significance=%r)" %
      (acc, acc_test.pvalue, is_significant(acc_test.pvalue)))
print("AUC=%.2f (pval=%.3f, significance=%r)" %
      (auc, auc_pval, is_significant(auc_pval)))
```

```
ACC=0.60 (pval=0.101, significance=False)
AUC=0.72 (pval=0.009, significance=True)
```

8.2.15 Exercise

Fisher linear discriminant rule

Write a class `FisherLinearDiscriminant` that implements the Fisher's linear discriminant analysis. This class must be compliant with the scikit-learn API by providing two methods: - `fit(X, y)` which fits the model and returns the object itself; - `predict(X)` which returns a vector of the predicted values. Apply the object on the dataset presented for the LDA.

8.3 Non-Linear Kernel Methods and Support Vector Machines (SVM)

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.svm import SVC
from sklearn.preprocessing import StandardScaler
```

(continues on next page)

(continued from previous page)

```

from sklearn import datasets
from sklearn import metrics
from sklearn.model_selection import train_test_split

# Plot
import matplotlib.pyplot as plt
import seaborn as sns

# Plot parameters
plt.style.use('seaborn-v0_8-whitegrid')
fig_w, fig_h = plt.rcParams.get('figure.figsize')
plt.rcParams['figure.figsize'] = (fig_w, fig_h * .5)

```

8.3.1 Kernel algorithms

Kernel Machine are based kernel methods require only a user-specified kernel function $K(x_i, x_j)$, i.e., a **similarity function** over pairs of data points (x_i, x_j) into kernel (dual) space on which learning algorithms operate linearly, i.e. every operation on points is a linear combination of $K(x_i, x_j)$. Outline of the SVM algorithm:

1. **Map points x into kernel space** using a **kernel function**: $x \rightarrow K(x, \cdot)$. Learning algorithms operates linearly by dot product into high-kernel space: $K(\cdot, x_i) \cdot K(\cdot, x_j)$.
 - Using the kernel trick (Mercer's Theorem) replaces dot product in high dimensional space by a simpler operation such that $K(\cdot, x_i) \cdot K(\cdot, x_j) = K(x_i, x_j)$.
 - Thus we only need to compute a similarity measure $K(x_i, x_j)$ for each pairs of point and store in a $N \times N$ Gram matrix of.

8.3.2 SVM

2. **The learning process** consist of estimating the α_i of the decision function that maximizes the hinge loss (of $f(x)$) plus some penalty when applied on all training points.

3. **Prediction** of a new point x using the decision function.

$$f(x) = \text{sign} \left(\sum_i^N \alpha_i y_i K(x_i, x) \right).$$

8.3.3 Kernel function

One of the most commonly used kernel is the **Radial Basis Function (RBF) Kernel**. For a pair of points x_i, x_j the RBF kernel is defined as:

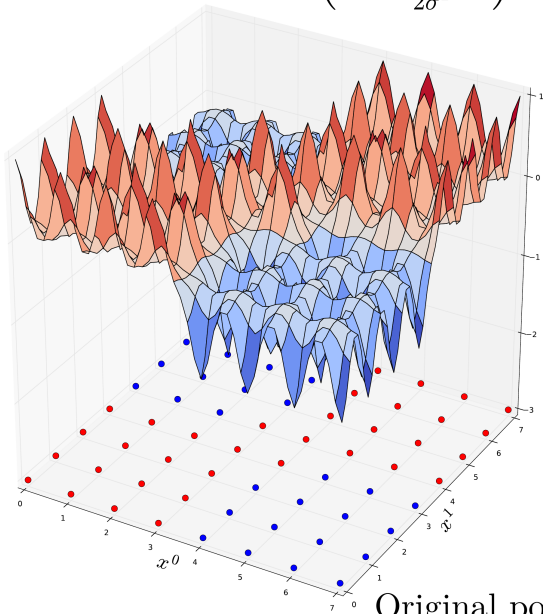
$$K(x_i, x_j) = \exp \left(-\frac{\|x_i - x_j\|^2}{2\sigma^2} \right) \quad (8.1)$$

$$= \exp \left(-\gamma \|x_i - x_j\|^2 \right) \quad (8.2)$$

Where σ (or γ) defines the kernel width parameter. Basically, we consider a Gaussian function centered on each training sample x_i . it has a ready interpretation as a similarity measure as it decreases with squared Euclidean distance between the two feature vectors.

(1) Kernel mapping:

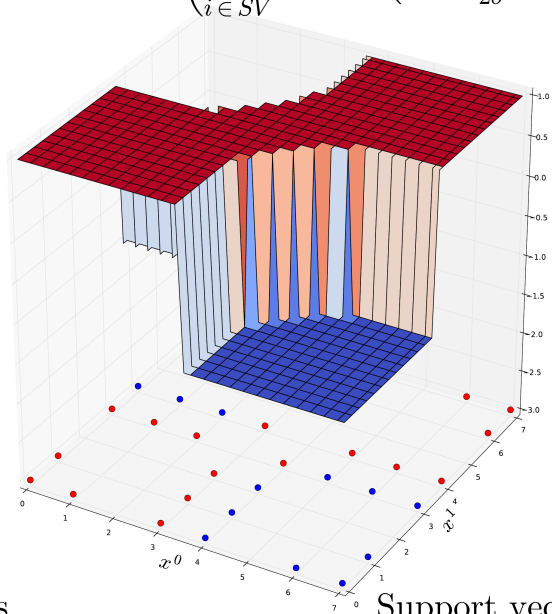
$$x \rightarrow K(x_i, x) = \exp\left(-\frac{\|x_i - x\|^2}{2\sigma^2}\right)$$



Original points

(2) Learn the decision function:

$$f(x) = \text{sign}\left(\sum_{i \in SV} \alpha_i y_i \exp\left(-\frac{\|x_i - x\|^2}{2\sigma^2}\right)\right)$$



Support vectors

Non linear SVM also exists for regression problems.

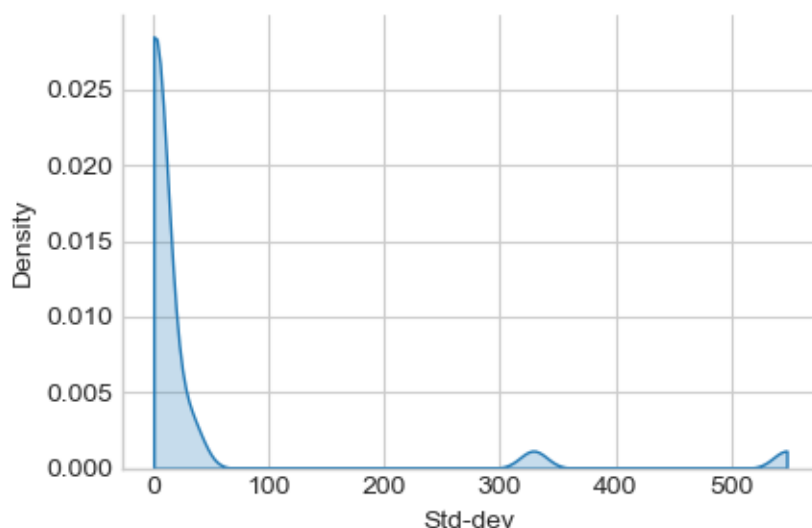
Dataset

```
X, y = datasets.load_breast_cancer(return_X_y=True)
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.5, stratify=y, random_state=42)
```

Preprocessing: unequal variance of input features, requires scaling for svm.

```
ax = sns.displot(x=X_train.std(axis=0), kind="kde", bw_adjust=.2, cut=0,
                 fill=True, height=3, aspect=1.5,)
_ = ax.set_xlabel("Std-dev").tight_layout()

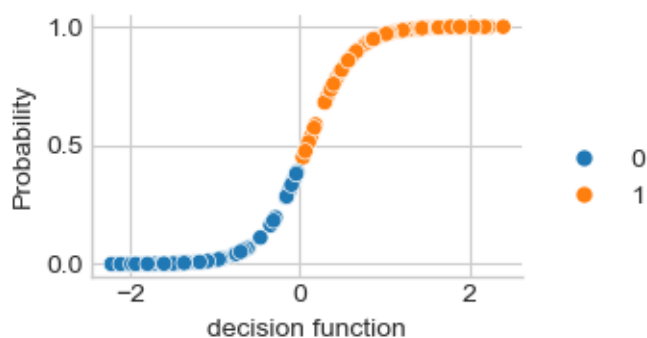
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```



Scikit-learn SVC (Support Vector Classification) with probability function applying a logistic of the decision_function

```
svm = SVC(kernel='rbf', probability=True).fit(X_train, y_train)
y_pred = svm.predict(X_test)
y_score = svm.decision_function(X_test)
y_prob = svm.predict_proba(X_test)[:, 1]

ax = sns.relplot(x=y_score, y=y_prob, hue=y_pred, height=2, aspect=1.5)
_ = ax.set_axis_labels("decision function", "Probability").tight_layout()
```



```
print("bAcc: %.2f, AUC: %.2f (AUC with proba: %.2f)" % (
    metrics.balanced_accuracy_score(y_true=y_test, y_pred=y_pred),
    metrics.roc_auc_score(y_true=y_test, y_score=y_score),
    metrics.roc_auc_score(y_true=y_test, y_score=y_prob)))

# Usefull internals: indices of support vectors within original X
np.all(X_train[svm.support_, :] == svm.support_vectors_)
```

```
bAcc: 0.96, AUC: 0.99 (AUC with proba: 0.99)
```

```
np.True_
```

Total running time of the script: (0 minutes 0.490 seconds)

8.4 Non-Linear Ensemble Learning

Sources:

- [Scikit-learn API](#)
- [Scikit-learn doc](#)

8.4.1 Introduction to Ensemble Learning

Ensemble learning is a powerful machine learning technique that combines multiple models to achieve better performance than any individual model. By aggregating predictions from diverse learners, ensemble methods enhance accuracy, reduce variance, and improve generalization. The main advantages of ensemble learning include:

- **Reduced overfitting:** By averaging multiple models, ensemble methods mitigate overfitting risks.
- **Increased robustness:** The diversity of models enhances stability, making the approach more resistant to noise and biases.

There are three main types of ensemble learning techniques: **Bagging**, **Boosting**, and **Stacking**. Each method follows a unique strategy to combine multiple models and improve overall performance.

Conclusion

Ensemble learning is a fundamental approach in machine learning that significantly enhances predictive performance. **Bagging** helps reduce variance, **boosting** improves bias, and **stacking** leverages multiple models to optimize performance. By carefully selecting and tuning ensemble techniques, practitioners can build powerful and robust machine learning models suitable for various real-world applications.

```
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import StackingClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
import matplotlib.pyplot as plt

import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

from sklearn.preprocessing import StandardScaler

from sklearn import datasets
from sklearn import metrics
from sklearn.model_selection import train_test_split
```

Breast cancer dataset

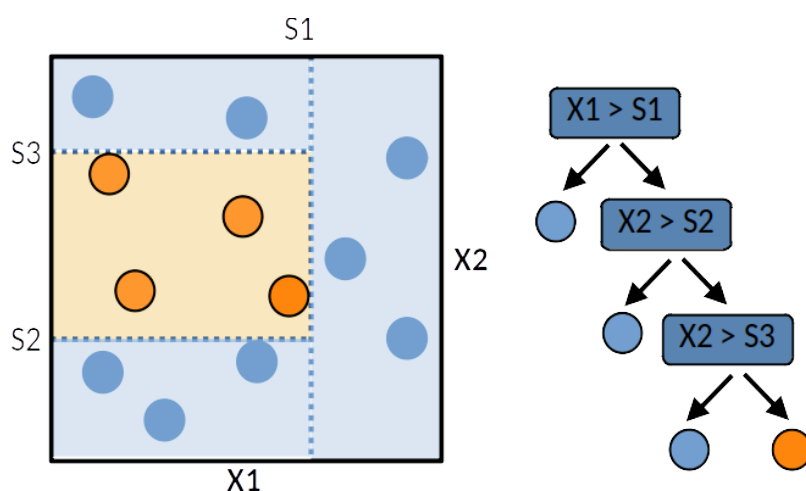
```
breast_cancer = datasets.load_breast_cancer()
X, y = breast_cancer.data, breast_cancer.target
print(breast_cancer.feature_names)

X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.5, stratify=y, random_state=42)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

```
['mean radius' 'mean texture' 'mean perimeter' 'mean area'
 'mean smoothness' 'mean compactness' 'mean concavity'
 'mean concave points' 'mean symmetry' 'mean fractal dimension'
 'radius error' 'texture error' 'perimeter error' 'area error'
 'smoothness error' 'compactness error' 'concavity error'
 'concave points error' 'symmetry error' 'fractal dimension error'
 'worst radius' 'worst texture' 'worst perimeter' 'worst area'
 'worst smoothness' 'worst compactness' 'worst concavity'
 'worst concave points' 'worst symmetry' 'worst fractal dimension']
```

8.4.2 Decision tree

A tree can be “learned” by splitting the training dataset into subsets based on an features value test. Each internal node represents a “test” on an feature resulting on the split of the current sample. At each step the algorithm selects the feature and a cutoff value that maximises a given metric. Different metrics exist for regression tree (target is continuous) or classification tree (the target is qualitative). This process is repeated on each derived subset in a recursive manner called recursive partitioning. The recursion is completed when the subset at a node has all the same value of the target variable, or when splitting no longer adds value to the predictions. This general principle is implemented by many recursive partitioning tree algorithms.



Decision trees are simple to understand and interpret however they tend to overfit the data. However decision trees tend to overfit the training set. Leo Breiman propose random forest to deal with this issue.

A single decision tree is usually overfits the data it is learning from because it learn from only one pathway of decisions. Predictions from a single decision tree usually don't make accurate

predictions on new data.

```
tree = DecisionTreeClassifier()
tree.fit(X_train, y_train)

y_pred = tree.predict(X_test)
y_prob = tree.predict_proba(X_test)[:, 1]
print("bAcc: %.2f, AUC: %.2f " % (
    metrics.balanced_accuracy_score(y_true=y_test, y_pred=y_pred),
    metrics.roc_auc_score(y_true=y_test, y_score=y_prob)))
```

```
bAcc: 0.90, AUC: 0.90
```

8.4.3 Bagging (Bootstrap Aggregating): Random forest

Bagging is an ensemble method that aims to reduce variance by training multiple models on different subsets of the training data. It follows these steps:

1. Generate multiple bootstrap samples (randomly drawn with replacement) from the original dataset.
2. Train an independent model (typically a weak learner like a decision tree) on each bootstrap sample.
3. Aggregate predictions using majority voting (for classification) or averaging (for regression).

Example: The **Random Forest** algorithm is a widely used bagging method that constructs multiple decision trees and combines their predictions.

Key Benefits:

- Reduces variance and improves stability.
- Works well with high-dimensional data.
- Effective for handling noisy datasets.

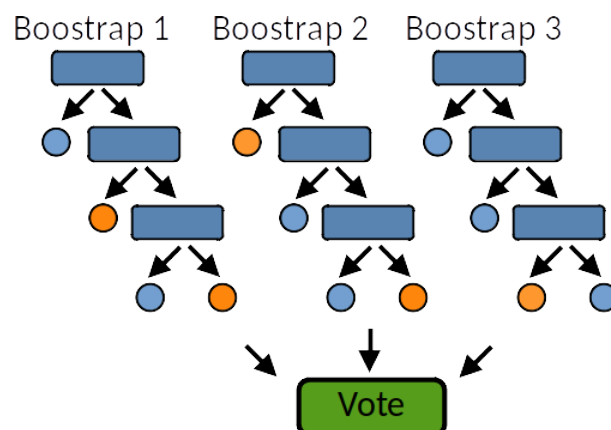
```
bagging_tree = BaggingClassifier(DecisionTreeClassifier())
bagging_tree.fit(X_train, y_train)

y_pred = bagging_tree.predict(X_test)
y_prob = bagging_tree.predict_proba(X_test)[:, 1]
print("bAcc: %.2f, AUC: %.2f " % (
    metrics.balanced_accuracy_score(y_true=y_test, y_pred=y_pred),
    metrics.roc_auc_score(y_true=y_test, y_score=y_prob)))
```

```
bAcc: 0.95, AUC: 0.98
```

Random Forest

A random forest is a meta estimator that fits a number of **decision tree learners** on various sub-samples of the dataset and use averaging to improve the predictive accuracy and control over-fitting. Random forest models reduce the risk of overfitting by introducing randomness by:



- building multiple trees (`n_estimators`)
- drawing observations with replacement (i.e., a bootstrapped sample)
- splitting nodes on the best split among a random subset of the features selected at every node

```
forest = RandomForestClassifier(n_estimators=100)
forest.fit(X_train, y_train)

y_pred = forest.predict(X_test)
y_prob = forest.predict_proba(X_test)[:, 1]
print("bAcc: %.2f, AUC: %.2f " % (
    metrics.balanced_accuracy_score(y_true=y_test, y_pred=y_pred),
    metrics.roc_auc_score(y_true=y_test, y_score=y_prob)))
```

```
bAcc: 0.95, AUC: 0.99
```

8.4.4 Boosting and Gradient boosting

Boosting is an ensemble method that focuses on reducing bias by training models sequentially, where each new model corrects the errors of its predecessors. The process includes:

1. Train an initial weak model on the training data.
2. Assign higher weights to misclassified instances to emphasize difficult cases.
3. Train a new model on the updated dataset, repeating the process iteratively.
4. Combine the predictions of all models using a weighted sum.

Gradient boosting

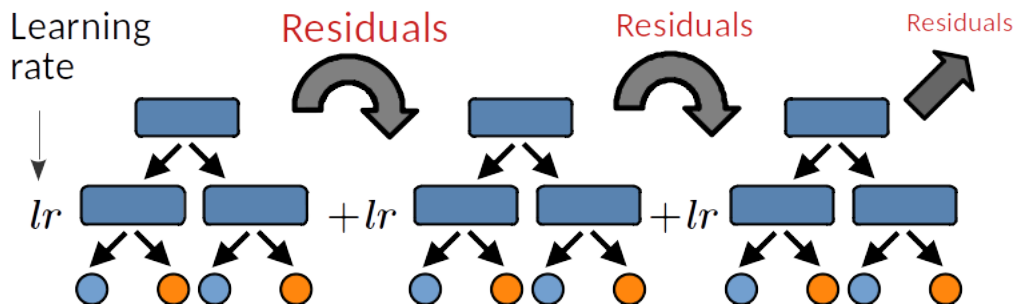
Popular boosting algorithms include **AdaBoost**, **Gradient Boosting Machines (GBM)**, **XG-Boost**, and **LightGBM**.

Key Benefits:

- Improves accuracy by focusing on difficult instances.
- Works well with structured data and tabular datasets.
- Reduces bias while maintaining interpretability.

The two main hyper-parameters are:

- The **learning rate** (lr) controls over-fitting: decreasing the lr limits the capacity of a learner to overfit the residuals, ie, it slows down the learning speed and thus increases the **regularization**.
- The **sub-sampling fraction** controls the fraction of samples to be used for fitting the learners. Values smaller than 1 leads to **Stochastic Gradient Boosting**. It thus controls for over-fitting reducing variance and increasing bias.



```
gb = GradientBoostingClassifier(n_estimators=100, learning_rate=0.1,
                               subsample=0.5, random_state=0)
gb.fit(X_train, y_train)

y_pred = gb.predict(X_test)
y_prob = gb.predict_proba(X_test)[:, 1]

print("bAcc: %.2f, AUC: %.2f " % (
    metrics.balanced_accuracy_score(y_true=y_test, y_pred=y_pred),
    metrics.roc_auc_score(y_true=y_test, y_score=y_prob)))
```

```
bAcc: 0.94, AUC: 0.99
```

8.4.5 Stacking

Stacking (or stacked generalization) is a more complex ensemble technique that combines predictions from multiple base models using a **meta-model**. The process follows:

1. Train several base models (e.g., decision trees, SVMs, neural networks) on the same dataset.
2. Collect predictions from all base models and use them as new features.
3. Train a meta-model (often a simple regression or classification model) to learn how to best combine the base predictions.

Example: Stacking can combine weak and strong learners, such as decision trees, logistic regression, and deep learning models, to create a robust final model.

Key Benefits:

- Allows different types of models to complement each other.
- Captures complex relationships between models.
- Can outperform traditional ensemble methods when well-tuned.

```
from sklearn.svm import LinearSVC
from sklearn.pipeline import make_pipeline

estimators = [
    ('rf', RandomForestClassifier(n_estimators=10, random_state=42)),
    ('svr', make_pipeline(StandardScaler(),
                          LinearSVC(random_state=42)))]

stacked_trees = StackingClassifier(estimators)
stacked_trees.fit(X_train, y_train)

y_pred = stacked_trees.predict(X_test)
y_prob = stacked_trees.predict_proba(X_test)[:, 1]
print("bAcc: %.2f, AUC: %.2f " % (
    metrics.balanced_accuracy_score(y_true=y_test, y_pred=y_pred),
    metrics.roc_auc_score(y_true=y_test, y_score=y_prob)))
```

```
bAcc: 0.97, AUC: 0.99
```

Total running time of the script: (0 minutes 1.056 seconds)

RESAMPLING METHODS AND MODEL EVALUATION

9.1 Out-of-sample Validation for Model Selection and Evaluation

Source scikit-learn model selection and evaluation

```
from sklearn.base import is_classifier, clone
from joblib import Parallel, delayed
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_validate
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn import datasets
import sklearn.linear_model as lm
from sklearn.model_selection import train_test_split, KFold, PredefinedSplit
from sklearn.model_selection import cross_val_score, GridSearchCV

import sklearn.metrics as metrics
X, y = datasets.make_regression(n_samples=100, n_features=100,
                               n_informative=10, random_state=42)
```

9.1.1 Train, validation and test sets

Machine learning algorithms tend to overfit training data. Predictive performances **MUST** be evaluated on independent hold-out dataset. A split of into a training test and an independent test set mandatory. However to set the hyperparameters the dataset is generally splitted into three sets:

1. Training Set (Fitting the Model and Learning Parameters)

- The training set is used to fit the model by learning its parameters (e.g., weights in a neural network, coefficients in a regression model).
- The algorithm adjusts its parameters to minimize a chosen loss function (e.g., MSE for regression, cross-entropy for classification).
- The model learns patterns from this data, but using only the training set risks overfitting—where the model memorizes data instead of generalizing.

- Role: Learn the parameters of the model.

2. Validation Set (Hyperparameter Tuning and Model Selection)

- The validation set is used to fine-tune the model's hyperparameters (e.g., learning rate, number of layers, number of clusters).
- Hyperparameters are not directly learned from data but are instead set before training.
- The validation set helps to assess different model configurations, preventing overfitting by ensuring that the model generalizes beyond the training set.
- If we see high performance on the training set but poor performance on the validation set, we are likely overfitting.
- The process of choosing the best hyperparameters based on the validation set is called **model selection**.
- Role: Tune hyperparameters and select the best model configuration.
- Data Leakage Risk: If we tune hyperparameters too much on the validation set, it essentially becomes part of training, leading to potential overfitting on it.

3. Test Set (Final Independent Evaluation)

- The test set is an independent dataset used to evaluate the final model after training and hyperparameter tuning.
- This provides an unbiased estimate of how the model will perform on completely new data.
- The model should never be trained or tuned using the test set to ensure a fair evaluation.
- Performance metrics (e.g., accuracy, F1-score, ROC-AUC) on the test set indicate how well the model is expected to perform in real-world scenarios.
- Role: Evaluate the final model's performance on unseen data.

Summary:

- **Training set**
 - Fits model parameters.
 - High risk of overfitting if the model is too complex.
- **Validation set**
 - Tunes hyperparameters and selects the best model.
 - Risk of overfitting if tuning too much.
- **Test set**
 - Provides a final evaluation on unseen data.

Split dataset in train/test sets to train and assess the final model after training and hyperparameter tuning.

```
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.25, shuffle=True, random_state=42)

mod = lm.Ridge(alpha=10)
```

(continues on next page)



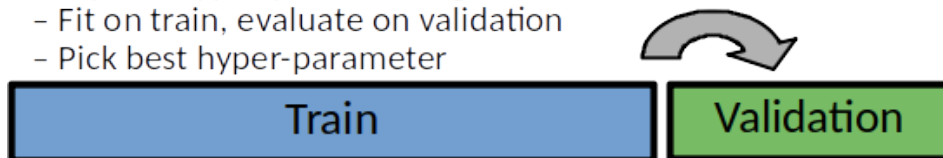
Model evaluation: hold-out a test set

- `mod.fit(X_train, y_train)`
- `mod.predic(X_test)`

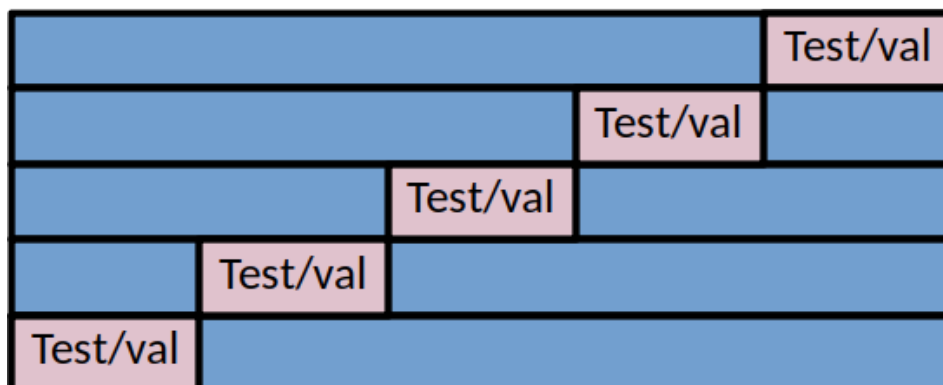


Model selection: hold-out a validation set

- Explore hyper-parameters grid
- Fit on train, evaluate on validation
- Pick best hyper-parameter



Cross-validation: when dataset is too small for to apply hold-out strategy. CV Can be used for model evaluation or/and model selection.



(continued from previous page)

```
mod.fit(X_train, y_train)

y_pred_test = mod.predict(X_test)
print("Test R2: %.2f" % metrics.r2_score(y_test, y_pred_test))
```

```
Test R2: 0.74
```

9.1.2 Cross-Validation (CV)

If sample size is limited, train/validation/test split may be impossible:

- Large training+validation set (80%) small test set (20%) might provide a poor estimation of the predictive performances on few test samples. The same argument stands for train vs validation samples.
- On the contrary, large test set and small training set might produce a poorly estimated learner.

Cross Validation (CV) ([Scikit-learn](#)) can be used to replace train/validation split and/or train+validation / test split. Main procedure:

1. The dataset is divided into k equal-sized subsets (folds).
2. The model is trained k times, each time using $k-1$ folds as the training set and 1 fold as the validation set.
3. The final performance is the average of the k validation scores.

For 10-fold we can either average over 10 values (Macro measure) or concatenate the 10 experiments and compute the micro measures.

Two strategies [micro vs macro estimates] (<https://stats.stackexchange.com/questions/34611/meanscores-vs-scoreconcatenation-in-cross-validation>):

- **Micro measure: average(individual scores)**: compute a score S for each sample and average over all samples. It is similar to **average score(concatenation)**: an averaged score computed over all concatenated samples.
- **Macro measure mean(CV scores)** (the most commonly used method): compute a score S on each fold k and average across folds:

These two measures (an average of average vs. a global average) are generally similar. They may differ slightly if folds are of different sizes. This validation scheme is known as the **K-Fold CV**. Typical choices of K are 5 or 10, [Kohavi 1995]. The extreme case where $K = N$ is known as **Leave-One-Out Cross-Validation, LOO-CV**.

CV for regression

Usually the error function $\mathcal{L}()$ is the r-squared score. However other function (MAE, MSE) can be used.

CV with explicit loop:

```

estimator = lm.Ridge(alpha=10)

cv = KFold(n_splits=5, shuffle=True, random_state=42)
r2_train, r2_test = list(), list()

for train, test in cv.split(X):
    estimator.fit(X[train, :], y[train])
    r2_train.append(metrics.r2_score(y[train], estimator.predict(X[train, :])))
    r2_test.append(metrics.r2_score(y[test], estimator.predict(X[test, :])))

print("Train r2:%.2f" % np.mean(r2_train))
print("Test  r2:%.2f" % np.mean(r2_test))

```

```

Train r2:0.99
Test  r2:0.67

```

Scikit-learn provides user-friendly function to perform CV

`cross_val_score`: single metric

```

scores = cross_val_score(estimator=estimator, X=X, y=y, cv=5)
print("Test  r2:%.2f" % scores.mean())

cv = KFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(estimator=estimator, X=X, y=y, cv=cv)
print("Test  r2:%.2f" % scores.mean())

```

```

Test  r2:0.73
Test  r2:0.67

```

`cross_validate`: multi metric, + time, etc.

```

scores = cross_validate(estimator=mod, X=X, y=y, cv=cv,
                        scoring=['r2', 'neg_mean_absolute_error'])

print("Test R2:%.2f; MAE:%.2f" % (scores['test_r2'].mean(),
                                  -scores['test_neg_mean_absolute_error'].mean()))

```

```

Test R2:0.67; MAE:55.27

```

CV for classification: stratify for the target label

With classification problems it is essential to sample folds where each set contains approximately the same percentage of samples of each target class as the complete set. This is called **stratification**. In this case, we will use `StratifiedKFold` which is a variation of k-fold which returns stratified folds. As error function we recommend:

- The `balanced accuracy`
- The `ROC-AUC`

CV with explicit loop:

```
X, y = datasets.make_classification(n_samples=100, n_features=100, shuffle=True,
                                   n_informative=10, random_state=42)

mod = lm.LogisticRegression(C=1, solver='lbfgs')

cv = StratifiedKFold(n_splits=5)

# Lists to store scores by folds (for macro measure only)
bacc, auc = [], []

for train, test in cv.split(X, y):
    mod.fit(X[train, :], y[train])
    bacc.append(metrics.roc_auc_score(
        y[test], mod.decision_function(X[test, :]))))
    auc.append(metrics.balanced_accuracy_score(
        y[test], mod.predict(X[test, :]))))

print("Test AUC:%.2f; bACC:%.2f" % (np.mean(bacc), np.mean(auc)))
```

```
Test AUC:0.86; bACC:0.79
```

`cross_val_score`: single metric

```
scores = cross_val_score(estimator=mod, X=X, y=y, cv=5)

print("Test ACC:%.2f" % scores.mean())
```

```
Test ACC:0.79
```

Provide your own CV and score

```
def balanced_acc(estimator, X, y, **kwargs):
    """Balanced accuracy scorer."""
    return metrics.recall_score(y, estimator.predict(X), average=None).mean()

scores = cross_val_score(estimator=mod, X=X, y=y, cv=cv,
                        scoring=balanced_acc)
print("Test bACC:%.2f" % scores.mean())
```

```
Test bACC:0.79
```

`cross_validate`: multi metric, + time, etc.

```
scores = cross_validate(estimator=mod, X=X, y=y, cv=cv,
                       scoring=['balanced_accuracy', 'roc_auc'])

print("Test AUC:%.2f; bACC:%.2f" % (scores['test_roc_auc'].mean(),
                                   scores['test_balanced_accuracy'].mean()))
```


Test AUC:0.86; bACC:0.79

Cross-validation for model selection (GridSearchCV)

Combine CV and grid search: `GridSearchCV` perform hyperparameter tuning (model selection) by systematically searching the best combination of hyperparameters evaluating all possible combinations (over a grid of possible values) using cross-validation:

1. Define the model: Choose a machine learning model (e.g., SVM, Random Forest).
2. Specify hyperparameters: Create a dictionary of hyperparameters and their possible values.
3. Perform exhaustive search: `GridSearchCV` trains the model with every possible combination of hyperparameters.
4. Cross-validation: For each combination, it uses k-fold cross-validation (default `cv=5`).
5. Select the best model: The combination with the highest validation performance is chosen. By default, refit an estimator using the best found parameters on the whole dataset.

```
# Outer, train/test, split:
X_train, X_test, y_train, y_test = \
    train_test_split(X, y, test_size=0.25, shuffle=True, random_state=42)

cv_inner = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Inner Cross-Validation (train/validation, splits) for model selection
lm_cv = GridSearchCV(lm.LogisticRegression(), {'C': 10. ** np.arange(-3, 3)},
                    cv=cv_inner, n_jobs=5)

# Fit, including model selection with internal CV
lm_cv.fit(X_train, y_train)

# Predict
y_pred_test = lm_cv.predict(X_test)
print("Test bACC: %.2f" % metrics.balanced_accuracy_score(y_test, y_pred_test))
```

Test bACC: 0.75

Cross-validation for both model (outer) evaluation and model (inner) selection

```
cv_outer = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
cv_inner = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Cross-validation for model (inner) selection
lm_cv = GridSearchCV(lm.Ridge(), {'alpha': 10. ** np.arange(-3, 3)},
                    cv=cv_inner, n_jobs=5)

# Cross-validation for model (outer) evaluation
scores = cross_validate(estimator=mod, X=X, y=y, cv=cv_outer,
                      scoring=['balanced_accuracy', 'roc_auc'])
```

(continues on next page)

(continued from previous page)

```
print("Test AUC:%.2f; bACC:%.2f, Time: %.2fs" % (scores['test_roc_auc'].mean(),
                                              scores['test_balanced_accuracy'].
→mean(
),
      scores['fit_time'].sum()))
```

```
Test AUC:0.85; bACC:0.74, Time: 0.03s
```

Models with built-in cross-validation

Let sklearn select the best parameters over a default grid.

Classification

```
print("== Logistic Ridge (L2 penalty) ==")
mod_cv = lm.LogisticRegressionCV(class_weight='balanced',
                                  scoring='balanced_accuracy',
                                  n_jobs=-1, cv=5)
scores = cross_val_score(estimator=mod_cv, X=X, y=y, cv=5)
print("Test ACC:%.2f" % scores.mean())
```

```
== Logistic Ridge (L2 penalty) ==
Test ACC:0.79
```

Regression

```
X, y, coef = datasets.make_regression(n_samples=50, n_features=100, noise=10,
                                     n_informative=2, random_state=42, coef=True)
```

```
print("== Ridge (L2 penalty) ==")
model = lm.RidgeCV(cv=3)
scores = cross_val_score(estimator=model, X=X, y=y, cv=5)
print("Test r2:%.2f" % scores.mean())

print("== Lasso (L1 penalty) ==")
model = lm.LassoCV(n_jobs=-1, cv=3)
scores = cross_val_score(estimator=model, X=X, y=y, cv=5)
print("Test r2:%.2f" % scores.mean())

print("== ElasticNet (L1 penalty) ==")
model = lm.ElasticNetCV(l1_ratio=[.1, .5, .9], n_jobs=-1, cv=3)
scores = cross_val_score(estimator=model, X=X, y=y, cv=5)
print("Test r2:%.2f" % scores.mean())
```

```
== Ridge (L2 penalty) ==
Test r2:0.16
== Lasso (L1 penalty) ==
Test r2:0.74
```

(continues on next page)

(continued from previous page)

```
== ElasticNet (L1 penalty) ==
Test  r2:0.58
```

9.2 Random Permutations: sample the null distribution

A permutation test is a type of non-parametric randomization test in which the null distribution of a test statistic is estimated by randomly permuting the observations.

Permutation tests are highly attractive because they make no assumptions other than that the observations are independent and identically distributed under the null hypothesis.

1. Compute a observed statistic t_{obs} on the data.
2. Use randomization to compute the distribution of t under the null hypothesis: Perform N random permutation of the data. For each sample of permuted data, i the data compute the statistic t_i . This procedure provides the distribution of t under the null hypothesis H_0 : $P(t|H_0)$
3. Compute the p-value = $P(t > t_{obs}|H_0) |\{t_i > t_{obs}\}|$, where t_i include : $math : 't_{obs}$.

Example Ridge regression

Sample the distributions of r-squared and coefficients of ridge regression under the null hypothesis. Simulated dataset:

```
# Regression dataset where first two features are predictive
np.random.seed(0)
n_features = 5
n_features_info = 2
n_samples = 100
X = np.random.randn(100, 5)
beta = np.zeros(n_features)
beta[:n_features_info] = 1
Xbeta = np.dot(X, beta)
eps = np.random.randn(n_samples)
y = Xbeta + eps

# Fit model on all data (!! risk of overfit)
model = lm.RidgeCV()
model.fit(X, y)
print("Coefficients on all data:")
print(model.coef_)

# Random permutation loop
nperm = 1000 # !! Should be at least 1000 (to assess a p-value at 1%)
scores_names = ["r2"]
scores_perm = np.zeros((nperm + 1, len(scores_names)))
coefs_perm = np.zeros((nperm + 1, X.shape[1]))

scores_perm[0, :] = metrics.r2_score(y, model.predict(X))
coefs_perm[0, :] = model.coef_
```

(continues on next page)

(continued from previous page)

```

orig_all = np.arange(X.shape[0])
for perm_i in range(1, nperm + 1):
    model.fit(X, np.random.permutation(y))
    y_pred = model.predict(X).ravel()
    scores_perm[perm_i, :] = metrics.r2_score(y, y_pred)
    coefs_perm[perm_i, :] = model.coef_

# One-tailed empirical p-value
pval_pred_perm = np.sum(scores_perm >= scores_perm[0]) / scores_perm.shape[0]
pval_coef_perm = np.sum(
    coefs_perm >= coefs_perm[0, :], axis=0) / coefs_perm.shape[0]

print("R2 p-value: %.3f" % pval_pred_perm)
print("Coefficients p-values:", np.round(pval_coef_perm, 3))

```

```

Coefficients on all data:
[ 1.01872179  1.05713711  0.20873888 -0.01784094 -0.05265821]
R2 p-value: 0.001
Coefficients p-values: [0.001 0.001 0.098 0.573 0.627]

```

Compute p-values corrected for multiple comparisons using FWER max-T (Westfall and Young, 1993) procedure.

```

pval_coef_perm_tmax = np.array([np.sum(coefs_perm.max(axis=1) >= coefs_perm[0, j])
                                for j in range(coefs_perm.shape[1])]) / coefs_
    perm.shape[0]
print("P-values with FWER (Westfall and Young) correction")
print(pval_coef_perm_tmax)

```

```

P-values with FWER (Westfall and Young) correction
[0.000999  0.000999  0.41058941 0.98001998 0.99200799]

```

Plot distribution of third coefficient under null-hypothesis Coeffitients 0 and 1 are significantly different from 0.

```

def hist_pvalue(perms, ax, name):
    """Plot statistic distribution as histogram.

    Paramters
    -----
    perms: 1d array, statistics under the null hypothesis.
           perms[0] is the true statistic .
    """
    # Re-weight to obtain distribution
    pval = np.sum(perms >= perms[0]) / perms.shape[0]
    weights = np.ones(perms.shape[0]) / perms.shape[0]
    ax.hist([perms[perms >= perms[0]], perms], histtype='stepfilled',
            bins=100, label="p-val<%.3f" % pval,

```

(continues on next page)

(continued from previous page)

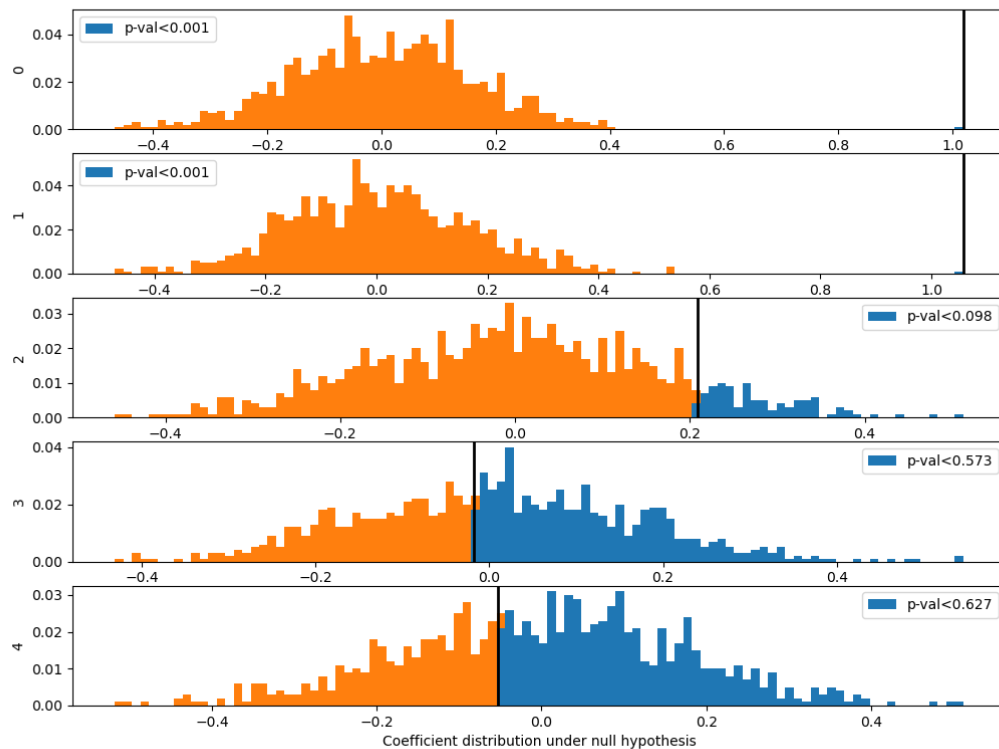
```

        weights=[weights[perms >= perms[0]], weights])
    # , label="observed statistic")
    ax.axvline(x=perms[0], color="k", linewidth=2)
    ax.set_ylabel(name)
    ax.legend()
    return ax

n_coef = coefs_perm.shape[1]
fig, axes = plt.subplots(n_coef, 1, figsize=(12, 9))
for i in range(n_coef):
    hist_pvalue(coefs_perm[:, i], axes[i], str(i))

_ = axes[-1].set_xlabel("Coefficient distribution under null hypothesis")

```



Exercise

Given the logistic regression presented above and its validation given a 5 folds CV.

1. Compute the p-value associated with the prediction accuracy measured with 5CV using a permutation test.
2. Compute the p-value associated with the prediction accuracy using a parametric test.

9.3 Bootstrapping

Bootstrapping is a statistical technique which consists in generating sample (called bootstrap samples) from an initial dataset of size N by randomly drawing with replacement N observations. It provides sub-samples with the same distribution than the original dataset. It aims to:

1. Assess the variability (standard error, **Confidence Intervals (CI)** of performances scores or estimated parameters (see Efron et al. 1986).
2. Regularize model by fitting several models on bootstrap samples and averaging their predictions (see Baging and random-forest).

A great advantage of bootstrap is its simplicity. It is a straightforward way to derive estimates of standard errors and confidence intervals for complex estimators of complex parameters of the distribution, such as percentile points, proportions, odds ratio, and correlation coefficients.

1. Perform B sampling, with replacement, of the dataset.
2. For each sample i fit the model and compute the scores.
3. Assess standard errors and confidence intervals of scores using the scores obtained on the B resampled dataset. Or, average models predictions.

References:

[Efron B, Tibshirani R. Bootstrap methods for standard errors, confidence intervals, and other measures of statistical accuracy. Stat Sci 1986;1:54–75](https://projecteuclid.org/download/pdf_1/euclid.ss/1177013815)

```
# Bootstrap loop
nboot = 100 # !! Should be at least 1000
scores_names = ["r2"]
scores_boot = np.zeros((nboot, len(scores_names)))
coefs_boot = np.zeros((nboot, X.shape[1]))

orig_all = np.arange(X.shape[0])
for boot_i in range(nboot):
    boot_tr = np.random.choice(orig_all, size=len(orig_all), replace=True)
    boot_te = np.setdiff1d(orig_all, boot_tr, assume_unique=False)
    Xtr, ytr = X[boot_tr, :], y[boot_tr]
    Xte, yte = X[boot_te, :], y[boot_te]
    model.fit(Xtr, ytr)
    y_pred = model.predict(Xte).ravel()
    scores_boot[boot_i, :] = metrics.r2_score(yte, y_pred)
    coefs_boot[boot_i, :] = model.coef_
```

Compute Mean, SE, CI Coeffitients 0 and 1 are significantly different from 0.

```
scores_boot = pd.DataFrame(scores_boot, columns=scores_names)
scores_stat = scores_boot.describe(percentiles=[.975, .5, .025])

print("r-squared: Mean=%.2f, SE=%.2f, CI=(%.2f %.2f)" %
      tuple(scores_stat.loc[["mean", "std", "2.5%", "97.5%"], "r2"]))
```

(continues on next page)

(continued from previous page)

```

coefs_boot = pd.DataFrame(coefs_boot)
coefs_stat = coefs_boot.describe(percentiles=[.975, .5, .025])
print("Coefficients distribution")
print(coefs_stat)

```

r-squared: Mean=0.59, SE=0.09, CI=(0.40 0.73)

Coefficients distribution

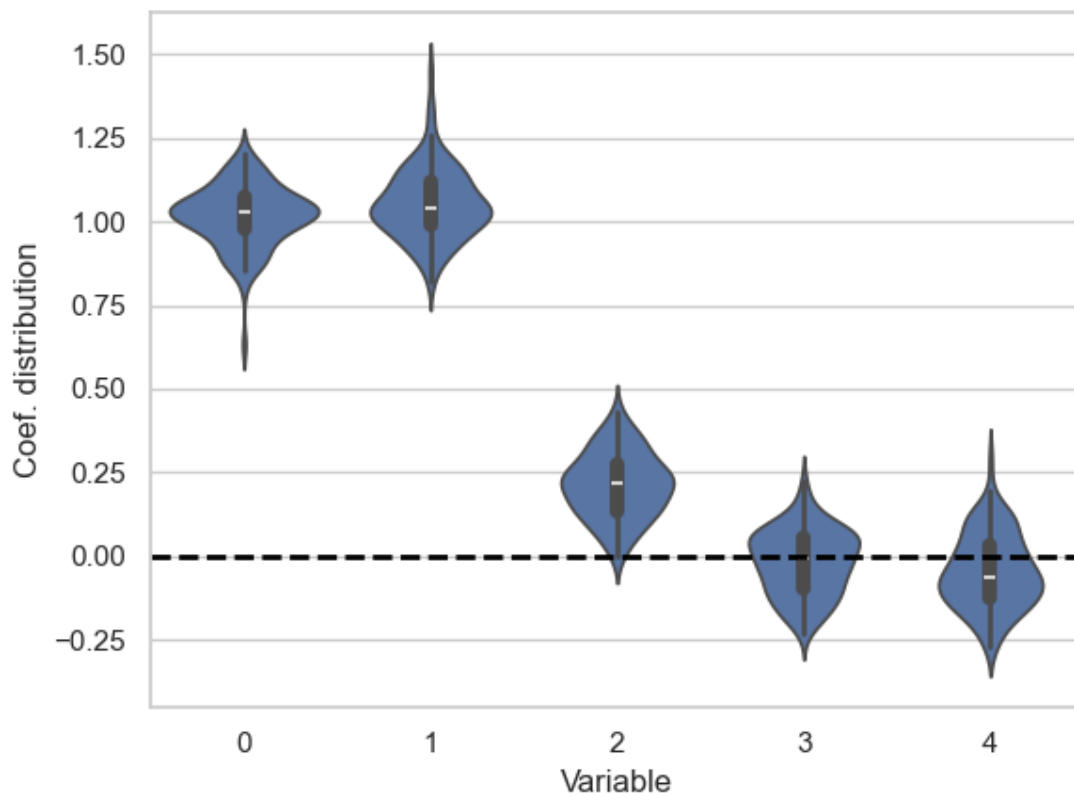
	0	1	2	3	4
count	100.000000	100.000000	100.000000	100.000000	100.000000
mean	1.017598	1.053832	0.212464	-0.018828	-0.045851
std	0.091508	0.105196	0.097532	0.097343	0.110555
min	0.631917	0.819190	-0.002689	-0.231580	-0.270810
2.5%	0.857418	0.883319	0.032672	-0.195018	-0.233241
50%	1.027161	1.038053	0.216531	-0.010023	-0.063331
97.5%	1.174707	1.289990	0.392701	0.150340	0.141587
max	1.204006	1.449672	0.432764	0.220711	0.290928

Plot coefficient distribution

```

df = pd.DataFrame(coefs_boot)
staked = pd.melt(df, var_name="Variable", value_name="Coef. distribution")
sns.set_theme(style="whitegrid")
ax = sns.violinplot(x="Variable", y="Coef. distribution", data=staked)
_ = ax.axhline(0, ls='--', lw=2, color="black")

```



9.4 Parallel Computation

Dataset

```
X, y = datasets.make_classification(
    n_samples=20, n_features=5, n_informative=2, random_state=42)
cv = StratifiedKFold(n_splits=5)
```

Classic sequential computation of CV:

```
estimator = lm.LogisticRegression(C=1, solver='lbfgs')
y_test_pred_seq = np.zeros(len(y)) # Store predictions in the original order
coefs_seq = list()
for train, test in cv.split(X, y):
    X_train, X_test, y_train, y_test = X[train,
                                       :], X[test, :], y[train], y[test]
    estimator.fit(X_train, y_train)
    y_test_pred_seq[test] = estimator.predict(X_test)
    coefs_seq.append(estimator.coef_)

test_accs = [metrics.accuracy_score(
    y[test], y_test_pred_seq[test]) for train, test in cv.split(X, y)]

# Accuracy
print(np.mean(test_accs), test_accs)

# Coef
coefs_cv = np.array(coefs_seq)
print("Mean of the coef")
print(coefs_cv.mean(axis=0).round(2))
print("Std Err of the coef")
print((coefs_cv.std(axis=0) / np.sqrt(coefs_cv.shape[0])).round(2))
```

```
0.8 [0.5, 0.5, 1.0, 1.0, 1.0]
Mean of the coef
[[-0.87  0.56  1.11 -0.08 -0.32]]
Std Err of the coef
[[0.03 0.02 0.02 0.09 0.03]]
```

Parallelization using `cross_validate` function

```
estimator = lm.LogisticRegression(C=1, solver='lbfgs')
cv_results = cross_validate(estimator, X, y, cv=cv, n_jobs=5)
print(np.mean(cv_results['test_score']), cv_results['test_score'])
```

```
0.8 [0.5 0.5 1.  1.  1. ]
```

Parallel computation with `joblib`:

```
def _split_fit_predict(estimator, X, y, train, test):
    X_train, X_test, y_train, y_test = X[train,
```

(continues on next page)

(continued from previous page)

```

                                :], X[test, :], y[train], y[test]
    estimator.fit(X_train, y_train)
    return [estimator.predict(X_test), estimator.coef_]

estimator = lm.LogisticRegression(C=1, solver='lbfgs')

```

```

parallel = Parallel(n_jobs=5)
cv_ret = parallel(
    delayed(_split_fit_predict)(
        clone(estimator), X, y, train, test)
    for train, test in cv.split(X, y))

y_test_pred_cv, coefs_cv = zip(*cv_ret)

```

```

y_test_pred = np.zeros(len(y))
for i, (train, test) in enumerate(cv.split(X, y)):
    y_test_pred[test] = y_test_pred_cv[i]

test_accs = [metrics.accuracy_score(
    y[test], y_test_pred[test]) for train, test in cv.split(X, y)]
print(np.mean(test_accs), test_accs)

```

```
0.8 [0.5, 0.5, 1.0, 1.0, 1.0]
```

Test same predictions and same coefficients

```

assert np.all(y_test_pred == y_test_pred_seq)
assert np.allclose(np.array(coefs_cv).squeeze(), np.array(coefs_seq).squeeze())

```

Total running time of the script: (0 minutes 9.988 seconds)

9.5 Hands-On: Validation of Supervised Classification Pipelines

9.5.1 Imports

```

# System
from sklearn.base import clone
from itertools import product
from joblib import Parallel, delayed
from joblib import Memory
from joblib import cpu_count

import os
import os.path
import tempfile
import time

# Scientific python

```

(continues on next page)

(continued from previous page)

```

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Univariate statistics
import statsmodels.api as sm
import statsmodels.formula.api as smf
import statsmodels.stats.api as sms

# Dataset
from sklearn.datasets import make_classification

# Models
from sklearn.decomposition import PCA
import sklearn.linear_model as lm
import sklearn.svm as svm
from sklearn.neural_network import MLPClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import StackingClassifier

# Metrics
import sklearn.metrics as metrics

# Resampling
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import cross_validate
from sklearn.model_selection import cross_val_predict
# from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
# from sklearn.model_selection import KFold
from sklearn.model_selection import StratifiedKFold
from sklearn import preprocessing
from sklearn.pipeline import make_pipeline

# Utils
from pystatsml.dataframe_utils import expand_key_value_column

```

9.5.2 Settings

Input/Output and working directory

```

WD = os.path.join(tempfile.gettempdir(), "ml_supervised_classif")
os.makedirs(WD, exist_ok=True)
INPUT_DIR = os.path.join(WD, "data")
OUTPUT_DIR = os.path.join(WD, "models")

```

(continues on next page)

(continued from previous page)

```
cachedir = os.path.join(OUTPUT_DIR, "cachedir")
os.makedirs(INPUT_DIR, exist_ok=True)
os.makedirs(OUTPUT_DIR, exist_ok=True)
```

Validation scheme here [cross_validation](#)

```
n_splits_test = 5
cv_test = StratifiedKFold(n_splits=n_splits_test,
                          shuffle=True, random_state=42)

n_splits_val = 5
cv_val = StratifiedKFold(n_splits=n_splits_val, shuffle=True, random_state=42)

metrics_names = ['accuracy', 'balanced_accuracy', 'roc_auc']
```

9.5.3 Dataset

```
X, y = make_classification(n_samples=200, n_features=100,
                          n_informative=10, n_redundant=10,
                          random_state=1)
```

9.5.4 Fit/Predict (Cached) Function for Parallel Execution

Principles:

1. Define dictionary of models including gridsearch model selection, where parallelization is controlled by a *n_jobs_grid_search* parameter.
2. Define a fit/predict (Cached) function `fit_predict()` for parallel execution. It returns everything required for further analysis, fitted models, test prediction, test indices, etc.
3. Run `fit_predict()` using a sequential `run_sequential()` or parallel `run_parallel()` execution:
 - Execution follows a “flat” map-reduce scheme of `fit_predict()` function (the mapper) over all combinations of models * CV-folds.
 - `run_xx()` return everything needed for further analysis, such as fitted model, test prediction, test indices, etc.
 - `run_xx()` also returns predictions into a DataFrame for further computation of metrics.
 - Predictions DataFrame over models * CV-folds can be save into a file.
 - Parallelization can be controlled at every steps: (i) inner loop of model selection using `make_models(n_jobs_grid_search)` (ii) outer loop of model x cv execution. Prediction are save into a csv file.
4. The computation of classification metrics is done afterwards, using `dataFrame`.

1. Models

```

def make_models(n_jobs_grid_search):
    mlp_param_grid = {"hidden_layer_sizes":
                      [(100, ), (50, ), (25, ), (10, ), (5, ),          # 1 hidden_
↪ layer
                      (100, 50, ), (50, 25, ), (25, 10,
                      # 2 hidden layers
                      ), (10, 5, ),
                      # 3 hidden layers
                      (100, 50, 25, ), (50, 25, 10, ), (25, 10, 5, )],
                      "activation": ["relu"], "solver": ["sgd"], 'alpha': [0.
↪ 0001]]

    models = dict(
        lrl2_cv=make_pipeline(
            preprocessing.StandardScaler(),
            # preprocessing.MinMaxScaler(),
            GridSearchCV(lm.LogisticRegression(),
                         {'C': 10. ** np.arange(-3, 1)},
                         cv=cv_val, n_jobs=n_jobs_grid_search)),

        lrenet_cv=make_pipeline(
            preprocessing.StandardScaler(),
            # preprocessing.MinMaxScaler(),
            GridSearchCV(estimator=lm.SGDClassifier(loss='log_loss',
                                                    penalty='elasticnet'),
                         param_grid={'alpha': 10. ** np.arange(-1, 3),
                                     'l1_ratio': [.1, .5, .9]},
                         cv=cv_val, n_jobs=n_jobs_grid_search)),

        svmbf_cv=make_pipeline(
            # preprocessing.StandardScaler(),
            preprocessing.MinMaxScaler(),
            GridSearchCV(svm.SVC(),
                         # {'kernel': ['poly', 'rbf'], 'C': 10. ** np.arange(-3,
↪ 3)},

                         {'kernel': ['rbf'], 'C': 10. ** np.arange(-1, 2)},
                         cv=cv_val, n_jobs=n_jobs_grid_search)),

        forest_cv=make_pipeline(
            # preprocessing.StandardScaler(),
            preprocessing.MinMaxScaler(),
            GridSearchCV(RandomForestClassifier(random_state=1),
                         {"n_estimators": [10, 100]},
                         cv=cv_val, n_jobs=n_jobs_grid_search)),

        gb_cv=make_pipeline(
            preprocessing.MinMaxScaler(),
            GridSearchCV(estimator=GradientBoostingClassifier(random_state=1),
                         param_grid={"n_estimators": [10, 100]},

```

(continues on next page)

(continued from previous page)

```

        cv=cv_val, n_jobs=n_jobs_grid_search)),

    # mlp_cv=make_pipeline(
    #     # preprocessing.StandardScaler(),
    #     preprocessing.MinMaxScaler(),
    #     GridSearchCV(estimator=MLPClassifier(random_state=1, max_iter=200,
    →tol=0.01),
    #                 param_grid=mlp_param_grid,
    #                 cv=cv_val, n_jobs=n_jobs_grid_search))
    )

    return models

```

(Flat) iterator over models and cv

```

def models_cv_iterator(models, cv):
    if not isinstance(cv, dict):
        cv = {fold_num: (train_idx, test_idx) for
              fold_num, (train_idx, test_idx) in enumerate(cv.split(X, y))}
    for (estimator_name, estimator), (fold_num, (train_idx, test_idx)) in\
        product(models.items(), cv.items()):
        yield estimator_name, estimator, fold_num, train_idx, test_idx

```

2. Define Fit/Predict (Cached) function for Parallel Execution

2.1 fit_predict: performs on demand recomputing using the

Joblib Memory class. Returns result_dict with fitted model, test predictions, and test indices.

```

memory = Memory(cachedir, verbose=0)

@memory.cache
def fit_predict_binary_classif(estimator_name, estimator, fold, X, y,
                              train_idx, test_idx,
                              verbose=0):
    """Fit and predict using on demand recomputing Joblib Memory class.

    Parameters
    -----
    estimator_name : str
        _description_
    estimator : Sklearn fitted estimator
        _description_
    fold : int/str
        _description_
    X : Array
        _description_
    y : Array
        _description_

```

(continues on next page)

(continued from previous page)

```

train_idx : _type_
    _description_
test_idx : _type_
    _description_
verbose : int
    verbose if >= 10
Returns
-----
_type_
    _description_
"""
if verbose >= 10:
    print("Fit %s" % estimator_name)
X_train, y_train = X[train_idx], y[train_idx]
X_test, y_test = X[test_idx], y[test_idx]
estimator = clone(estimator).fit(X_train, y_train)

# Predicted labels
pred_labs_test = estimator.predict(X_test)
pred_labs_train = estimator.predict(X_train)

# Predicted values before threshold
try:
    pred_proba_test = estimator.predict_proba(X_test)[: , 1]
    pred_proba_train = estimator.predict_proba(X_train)[: , 1]
except AttributeError:
    pred_proba_test = pred_proba_train = None
try:
    pred_decision_function_test = estimator.decision_function(X_test)
    pred_decision_function_train = estimator.decision_function(X_train)
except AttributeError:
    pred_decision_function_test = pred_decision_function_train = None

result_dict = dict(
    # Keys:
    estimator_name=estimator_name, fold=fold, test_idx=test_idx,
    # Estimator:
    estimator=estimator,
    # Test predictions:
    pred_labs_test=pred_labs_test,
    pred_proba_test=pred_proba_test,
    pred_decision_function_test=pred_decision_function_test,
    # Train predictions:
    pred_labs_train=pred_labs_train,
    pred_proba_train=pred_proba_train,
    pred_decision_function_train=pred_decision_function_train,
    train_idx=train_idx)

return result_dict

```

2.2 `predictions_dataframe(result_dict)`: Transform prediction result into DataFrame.

```
def predictions_dataframe(result_dict):
    """Transform prediction result into DataFrame.

    Parameters
    -----
    result_dict : dict
        Prediction result from fit_predict()

    Returns
    -----
    DataFrame
        _description_
    """
    cols = ['model', 'fold',
            'pred_labs_test', 'pred_proba_test', 'pred_decision_function_test',
            'test_idx']
    return pd.DataFrame({col: result_dict[col] for col in cols})
```

3. Run Fit/Predict

Sequential execution function

```
def run_sequential(fit_predict, models, cv_test, memory=None, clear_cache=False,
                  verbose=0):

    # Clear cache
    if memory and clear_cache:
        memory.clear()

    start_time = time.time()
    fitted_models_cv_list = [fit_predict(estimator_name, estimator,
                                         fold_num, X, y, train_idx, test_idx,
                                         verbose=verbose)
                             for estimator_name, estimator, fold_num, train_idx,
    ↪ test_idx
                             in models_cv_iterator(models, cv_test)]

    if verbose > 0:
        print('Sequential execution, Elapsed time: \t%.3f sec' %
              (time.time() - start_time))

    predictions_df = pd.concat([predictions_dataframe(r)
                               for r in fitted_models_cv_list])

    return fitted_models_cv_list, predictions_df
```

Parallel execution function

```
def run_parallel(fit_predict, models, cv_test, memory=None, clear_cache=False,
                 verbose=0):
```

(continues on next page)

(continued from previous page)

```

# Clear cache
if memory and clear_cache:
    memory.clear()

parallel = Parallel(n_jobs=cpu_count(only_physical_cores=True))

start_time = time.time()
fitted_models_cv_list = \
    parallel(delayed(fit_predict)(estimator_name, estimator,
                                fold_num, X, y, train_idx, test_idx,
                                verbose=verbose)
             for estimator_name, estimator, fold_num, train_idx, test_idx
             in models_cv_iterator(models, cv_test))
if verbose > 0:
    print('Parallel execution, Elapsed time: \t%.3f sec' %
          (time.time() - start_time))

predictions_df = pd.concat([predictions_dataframe(r)
                           for r in fitted_models_cv_list])

return fitted_models_cv_list, predictions_df

```

4. Recompute scores from saved predictions

```

def classification_metrics(predictions_df, groupby=['model', 'fold']):
    models_cv_classif_scores = \
        [list(keys) +
         [
             metrics.balanced_accuracy_score(
                 y[df["test_idx"]], df["pred_labs_test"]),
             # take decision_function when exists else proba
             metrics.roc_auc_score(
                 y[df["test_idx"]],
                 np.where(~np.isnan(df['pred_decision_function_test']),
                         df['pred_decision_function_test'], df['pred_proba_test'])
             )
         ] for keys, df in predictions_df.groupby(groupby)
        ]

    scores_colnames = ['balanced_accuracy', 'roc_auc']
    models_cv_classif_scores = \
        pd.DataFrame(models_cv_classif_scores,
                     columns=groupby + scores_colnames)

    return models_cv_classif_scores

```


5. Add new model

```

estimators_ = [(model_name, model) for model_name, model in models.items()]
stacked_clf = StackingClassifier(
    estimators_, final_estimator=lm.LogisticRegression())
models["stacked_clf"] = stacked_clf

fitted_models_cv_list, predictions_df = \
    run_parallel(fit_predict_binary_classif, models, cv_test, memory,
                 clear_cache=False, verbose=10)

models_cv_classif_scores = classification_metrics(predictions_df,
                                                  groupby=['model', 'fold'])
print(models_cv_classif_scores.drop(
    columns=['fold']).groupby(['model']).mean())

```

5. Learning curve

```

# Create CV splitter using 10%, 33%, 55%, 78%, 100% of the training data
train_sizes = np.array([0.1, 0.33, 0.55, 0.78, 1.])

def sub_sample(idx, size):
    return idx[:round(len(idx) * size)]

# CV splitter key is made with: "fold-%i_size-%.2f"
cv_test_lc = {"fold-%i_size-%.2f" % (fold_num, size): (sub_sample(train_idx,
    ↪ size), test_idx) for
                fold_num, (train_idx, test_idx) in enumerate(cv_test.split(X, y))
    ↪ for size in train_sizes}

print("CV splitter using 10%, 33%, 55%, 78%, 100% of the training data")
print(cv_test_lc.keys())

# Fit the models over all folds x sample sizes
fitted_models_cv_lc_list, predictions_df = \
    run_parallel(fit_predict_binary_classif, models, cv_test_lc, memory,
                 clear_cache=False, verbose=10)

# Expands folds column into two columns 'fold' and "size"
predictions_df = expand_key_value_column(df=predictions_df, col='fold')

# Compute classification metrics for each 'model', 'fold', and "size":
models_cv_classif_scores = classification_metrics(predictions_df,
                                                  groupby=['model', 'fold', "size"
    ↪ ])

models_size_classif_score = models_cv_classif_scores.drop(
    columns=['fold']).groupby(['model', "size"]).mean().reset_index()

```

(continues on next page)

(continued from previous page)

```
print(models_size_classif_score)

sns.lineplot(data=models_size_classif_score, x="size", y="roc_auc", hue='model')
```

9.5.5 Fit/Predict and Compute Test Score (CV) with Cached Computation

```
memory = Memory(cachedir, verbose=0)
cross_validate_cached = memory.cache(cross_validate)

fitted_models_cv_scores = dict()
start_time_total = time.time()
for name, model in models.items():
    # name, model = "lrl2_cv", models["lrl2_cv"]
    start_time = time.time()
    fitted_model_cv_scores_ = cross_validate_cached(estimator=model, X=X, y=y,
    ↪cv=cv_test,
                                                    n_jobs=n_splits_test,
                                                    scoring=metrics_names,
                                                    return_estimator=True,
                                                    return_indices=True)
    print(name, 'Elapsed time: \t%.3f sec' % (time.time() - start_time))
    fitted_models_cv_scores[name] = fitted_model_cv_scores_

print('Total Elapsed time: \t%.3f sec' % (time.time() - start_time_total))
```

9.5.6 Average Test Scores (CV) and save it to a file

```
test_stat = [[name] + [res["test_" + metric].mean() for metric in metrics_names]
              for name, res in fitted_models_cv_scores.items()]

test_stat = pd.DataFrame(test_stat, columns=["model"]+metrics_names)
test_stat.to_csv(os.path.join(OUTPUT_DIR, "test_stat.csv"))
print(test_stat)

# #####
# # Retrieve Individuals Predictions
# # -----
# # %%
# # **1. Retrieve individuals predictions and save individuals predictions in csv_
# ↪file**

# def predict_fitted_models_cv(fitted_models_cv):
#     """Return predictions of dict of fitted models over folds

#     Parameters
#     -----
#     fitted_models_cv : dict of fitted model across folds, it has the form of:
```

(continues on next page)

(continued from previous page)

```

#         fitted_models_cv=\
#         {'model_name':{'estimator': [est_cv1, est_cv2, ],
#                                   'indices': {'test': [idx_cv1, idx_cv2, ],
#                                                'train': [idx_cv1, idx_cv2, ]}},
#         }
#         fitted_models_cv['name']['estimator'][0]
#         fitted_models_cv['name']['indices']['test'][0]
#         fitted_models_cv['name']['indices']['train'][0]

#     returns
#     -----
#     pd.DataFrame(model_name, fold, pred_vals_test, pred_labs_test)
#     """

#     # Iterate over models
#     predictions = pd.DataFrame()
#     for name, model in fitted_models_cv.items():
#         # name, model = "lrl2_cv", models_scores["lrl2_cv"]
#         # model_scores = models_scores["lrl2_cv"]

#         # Predicted values before threshold
#         pred_vals_test = np.full(y.shape, np.nan)
#         # Predicted values before threshold
#         pred_vals_train = np.full(y.shape, np.nan)
#         pred_labs_test = np.full(y.shape, np.nan) # Predicted labels
#         pred_labs_train = np.full(y.shape, np.nan) # Predicted labels
#         true_labs = np.full(y.shape, np.nan) # True labels
#         fold = np.full(y.shape, np.nan) # True labels

#         # Iterate over folds
#         for fold in range(len(model['estimator'])):
#             est = model['estimator'][fold]
#             test_idx = model['indices']['test'][fold]
#             train_idx = model['indices']['train'][fold]
#             X_test = X[test_idx]
#             X_train = X[train_idx]

#             # Predicted labels
#             pred_labs_test[test_idx] = est.predict(X_test)
#             pred_labs_train[train_idx] = est.predict(X_train)
#             fold[test_idx] = fold

#             # Predicted values before threshold
#             try:
#                 pred_vals_test[test_idx] = est.predict_proba(X_test)[:, 1]
#                 pred_vals_train[train_idx] = est.predict_proba(X_train)[:, 1]
#             except AttributeError:
#                 pred_vals_test[test_idx] = est.decision_function(X_test)
#                 pred_vals_train[train_idx] = est.decision_function(X_train)

```

(continues on next page)

(continued from previous page)

```

#         true_labs[test_idx] = y[test_idx]

#         predictions_ = pd.DataFrame(dict(model=name, fold=fold.astype(int),
#                                         pred_vals_test=pred_vals_test,
#                                         pred_labs_test=pred_labs_test.astype(
#                                             int),
#                                         true_labs=y))
#         assert np.all(true_labs == y)

#         predictions = pd.concat([predictions, predictions_])

#     return predictions

# predictions = predict_fitted_models_cv(fitted_models_cv_scores)
# print(predictions.head())
# predictions.to_csv(os.path.join(OUTPUT_DIR, "predictions.csv"))

# # %%
# # **2. Recompute scores from saved predictions**

# models_scores_cv = [[mod, fold,
#                       metrics.balanced_accuracy_score(
#                           df["true_labs"], df["pred_labs_test"]),
#                       metrics.roc_auc_score(df["true_labs"], df["pred_vals_test"]
# → ")]])
#                       for (mod, fold), df in predictions.groupby(["model", 'fold
# → '])]

# models_scores_cv = pd.DataFrame(models_scores_cv, columns=[
#                               "model", 'fold', 'balanced_accuracy', 'roc_auc
# → '])

# models_scores_bymodel = models_scores_cv.groupby("model").mean()
# models_scores_bymodel = models_scores_bymodel.drop('fold', axis=1)
# print(models_scores_bymodel)

```

Total running time of the script: (0 minutes 20.193 seconds)

DEEP LEARNING: INTRODUCTION

10.1 Backpropagation

Wikipedia: Backpropagation: In machine learning, backpropagation is a gradient estimation method commonly used for training a neural network to compute its parameter updates. Backpropagation applies gradient descent using chain rule.

Sources:

- [3Blue1Brown video: But what is a neural network? | Deep learning chapter 1](#)
- [3Blue1Brown video: Gradient descent, how neural networks learn | DL2](#)
- [3Blue1Brown video: Backpropagation, step-by-step | DL3](#)
- [3Blue1Brown video: Backpropagation calculus | DL4](#)
- [Pytorch examples](#)

10.1.1 Backpropagation and chain rule

We will set up a two layer network:

$$\mathbf{y} = \mathbf{W}^{(2)} \max(\mathbf{W}^{(1)} \mathbf{x}, 0)$$

A fully-connected ReLU network with one hidden layer and no biases, trained to predict y from x using Euclidean error.

1. Forward pass can be decomposed as:

$$x \rightarrow \boxed{z^{(1)} = x^\top w^{(1)}} \rightarrow \boxed{h^{(1)} = \max(z^{(1)}, 0)} \rightarrow \boxed{z^{(2)} = h^{(1)\top} w^{(2)}} \rightarrow \boxed{L(z^{(2)}, y) = (z^{(2)} - y)^2}$$

With **local** partial derivatives of output given inputs for each the four steps:

1. $z^{(1)} = x^\top w^{(1)}$
 - $\frac{\partial z^{(1)}}{\partial w^{(1)}} = x$
 - $\frac{\partial z^{(1)}}{\partial x} = w^{(1)}$
2. $h^{(1)} = \max(z^{(1)}, 0)$
 - $\frac{\partial h^{(1)}}{\partial z^{(1)}} = \begin{cases} 1 & \text{if } z^{(1)} > 0 \\ 0 & \text{else} \end{cases}$
3. $z^{(2)} = h^{(1)\top} w^{(2)}$
 - $\frac{\partial z^{(2)}}{\partial w^{(2)}} = h^{(1)}$

- $\frac{\partial z^{(2)}}{\partial h^{(1)}} = w^{(2)}$
4. $L(z^{(2)}, y) = (z^{(2)} - y)^2$
- $\frac{\partial L}{\partial z^{(2)}} = 2(z^{(2)} - y)$

2. Backward pass: compute gradient of the loss given each parameters vectors applying the chain rule from the loss downstream to the parameters:

For $w^{(2)}$:

$$\frac{\partial L}{\partial w^{(2)}} = \frac{\partial L}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial w^{(2)}} = 2(z^{(2)} - y)h^{(1)}$$

For $w^{(1)}$:

$$\frac{\partial L}{\partial w^{(1)}} = \frac{\partial L}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial h^{(1)}} \frac{\partial h^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial w^{(1)}} = 2(z^{(2)} - y)w^{(2)} \begin{cases} 1 & \text{if } z^{(1)} > 0 \\ 0 & \text{else} \end{cases} x$$

Recap: Vector derivatives

Given a function $z = x'w$ with z the output, x the input and w the coefficients.

Scalar to Scalar: $x \in \mathbb{R}, z \in \mathbb{R}, w \in \mathbb{R}$. Regular derivative:

$$\frac{\partial z}{\partial w} = x \in \mathbb{R}$$

If w changes by a small amount, how much will z change?

Vector to Scalar: $x \in \mathbb{R}^N, z \in \mathbb{R}, w \in \mathbb{R}^N$. The derivative is the **Gradient** of partial derivative:
 $\frac{\partial z}{\partial w} \in \mathbb{R}^N$

$$\frac{\partial z}{\partial w} = \nabla_w z = \begin{bmatrix} \frac{\partial z}{\partial w_1} \\ \vdots \\ \frac{\partial z}{\partial w_i} \\ \vdots \\ \frac{\partial z}{\partial w_N} \end{bmatrix}$$

For each element w_i of w , if it changes by a small amount then how much will z change?

Vector to Vector: $w \in \mathbb{R}^N, z \in \mathbb{R}^M$. The derivative is **Jacobian** of partial derivative:

TO BE COMPLETED

$$\frac{\partial z}{\partial w} \in \mathbb{R}^{N \times M}$$

Backpropagation summary

Backpropagation algorithm in a graph:

1. **Forward pass**, for each node compute local partial derivatives of output given inputs.
2. **Backward pass**: apply chain rule from the end to each parameters.
 - Update parameter with gradient descent using the current upstream gradient and the current local gradient.
 - Compute upstream gradient for the backward nodes.

Think locally and remember that at each node:

- For the loss the gradient is the error
- At each step, the upstream gradient is obtained by multiplying the upstream gradient (an error) with the current parameters (vector of matrix).
- At each step, the current local gradient equal the input, therefore the current update is the current upstream gradient time the input.

```
import numpy as np
import sklearn.model_selection

# Plot
import matplotlib.pyplot as plt
import seaborn as sns

# Plot parameters
plt.style.use('seaborn-v0_8-whitegrid')
fig_w, fig_h = plt.rcParams.get('figure.figsize')
plt.rcParams['figure.figsize'] = (fig_w, fig_h * .5)
```

10.1.2 Hands-on with Numpy and pytorch

Load iris data set

Goal: Predict $Y = [\text{petal\_length}, \text{petal\_width}] = f(X = [\text{sepal\_length}, \text{sepal\_width}])$

- Plot data with seaborn
- Remove setosa samples
- Recode 'versicolor':1, 'virginica':2
- Scale X and Y
- Split data in train/test 50%/50%

```
iris = sns.load_dataset("iris")
#g = sns.pairplot(iris, hue="species")
df = iris[iris.species != "setosa"]
g = sns.pairplot(df, hue="species")
df['species_n'] = iris.species.map({'versicolor':1, 'virginica':2})

# Y = 'petal_length', 'petal_width'; X = 'sepal_length', 'sepal_width'
X_iris = np.asarray(df.loc[:, ['sepal_length', 'sepal_width']], dtype=np.float32)
Y_iris = np.asarray(df.loc[:, ['petal_length', 'petal_width']], dtype=np.float32)
label_iris = np.asarray(df.species_n, dtype=int)

# Scale
from sklearn.preprocessing import StandardScaler
scalerx, scalery = StandardScaler(), StandardScaler()
X_iris = scalerx.fit_transform(X_iris)
Y_iris = StandardScaler().fit_transform(Y_iris)
```

(continues on next page)

(continued from previous page)

```
# Split train test
X_iris_tr, X_iris_val, Y_iris_tr, Y_iris_val, label_iris_tr, label_iris_val = \
    sklearn.model_selection.train_test_split(X_iris, Y_iris, label_iris,
                                             train_size=0.5, stratify=label_iris)
```

```
/tmp/ipykernel_181680/2720649438.py:5: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/
stable/user_guide/indexing.html#returning-a-view-versus-a-copy
df['species_n'] = iris.species.map({'versicolor':1, 'virginica':2})
```



Backpropagation with Numpy

This implementation uses Numpy to manually compute the forward pass, loss, and backward pass.

```

# X=X_iris_tr; Y=Y_iris_tr; X_val=X_iris_val; Y_val=Y_iris_val

def two_layer_regression_numpy_train(X, Y, X_val, Y_val, lr, nite):
    # N is batch size; D_in is input dimension;
    # H is hidden dimension; D_out is output dimension.
    # N, D_in, H, D_out = 64, 1000, 100, 10
    N, D_in, H, D_out = X.shape[0], X.shape[1], 100, Y.shape[1]

    W1 = np.random.randn(D_in, H)
    W2 = np.random.randn(H, D_out)

    losses_tr, losses_val = list(), list()

    learning_rate = lr
    for t in range(nite):
        # Forward pass: compute predicted y
        z1 = X.dot(W1)
        h1 = np.maximum(z1, 0)
        Y_pred = h1.dot(W2)

        # Compute and print loss
        loss = np.square(Y_pred - Y).sum()

        # Backprop to compute gradients of w1 and w2 with respect to loss
        grad_y_pred = 2.0 * (Y_pred - Y)
        grad_w2 = h1.T.dot(grad_y_pred)
        grad_h1 = grad_y_pred.dot(W2.T)
        grad_z1 = grad_h1.copy()
        grad_z1[z1 < 0] = 0
        grad_w1 = X.T.dot(grad_z1)

        # Update weights
        W1 -= learning_rate * grad_w1
        W2 -= learning_rate * grad_w2

        # Forward pass for validation set: compute predicted y
        z1 = X_val.dot(W1)
        h1 = np.maximum(z1, 0)
        y_pred_val = h1.dot(W2)
        loss_val = np.square(y_pred_val - Y_val).sum()

        losses_tr.append(loss)
        losses_val.append(loss_val)

        if t % 10 == 0:
            print(t, loss, loss_val)

    return W1, W2, losses_tr, losses_val

W1, W2, losses_tr, losses_val = \

```

(continues on next page)

(continued from previous page)

```

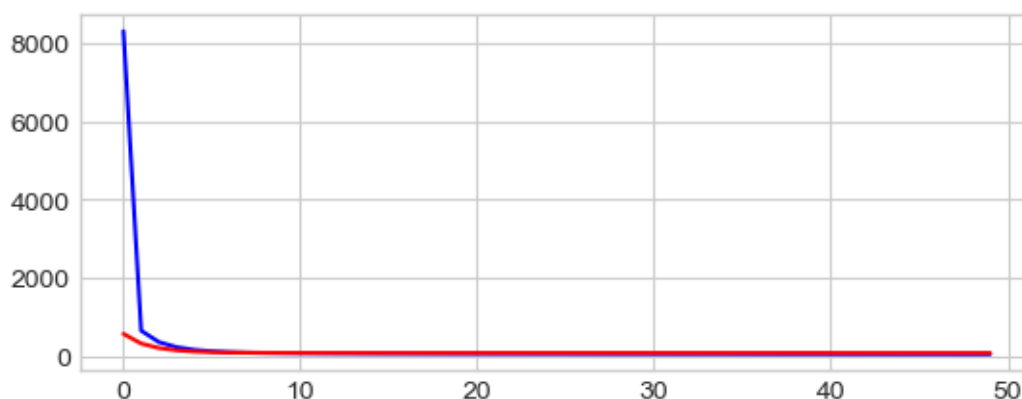
two_layer_regression_numpy_train(X=X_iris_tr, Y=Y_iris_tr,
                                X_val=X_iris_val, Y_val=Y_iris_val,
                                lr=1e-4, nite=50)
_ = plt.plot(np.arange(len(losses_tr)), losses_tr, "-b",
             np.arange(len(losses_val)), losses_val, "-r")

```

```

0 8320.86283567932 566.4003298191709
10 69.1144254443256 78.183681464354
20 55.983394365066715 77.33285806985458
30 50.57373709813517 77.19249121024394
40 47.19138471346897 76.77478593928662

```



Backpropagation with PyTorch Tensors

Learning PyTorch with Examples

Numpy is a great framework, but it cannot utilize GPUs to accelerate its numerical computations. For modern deep neural networks, GPUs often provide speedups of 50x or greater, so unfortunately numpy won't be enough for modern deep learning. Here we introduce the most fundamental PyTorch concept: the Tensor. A PyTorch Tensor is conceptually identical to a numpy array: a Tensor is an n-dimensional array, and PyTorch provides many functions for operating on these Tensors. Behind the scenes, Tensors can keep track of a computational graph and gradients, but they're also useful as a generic tool for scientific computing. Also unlike numpy, PyTorch Tensors can utilize GPUs to accelerate their numeric computations. To run a PyTorch Tensor on GPU, you simply need to cast it to a new datatype. Here we use PyTorch Tensors to fit a two-layer network to random data. Like the numpy example above we need to manually implement the forward and backward passes through the network:

```

import torch

# X=X_iris_tr; Y=Y_iris_tr; X_val=X_iris_val; Y_val=Y_iris_val

def two_layer_regression_tensor_train(X, Y, X_val, Y_val, lr, nite):

    dtype = torch.float
    device = torch.device("cpu")
    # device = torch.device("cuda:0") # Uncomment this to run on GPU

```

(continues on next page)

(continued from previous page)

```

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = X.shape[0], X.shape[1], 100, Y.shape[1]

# Create random input and output data
X = torch.from_numpy(X)
Y = torch.from_numpy(Y)
X_val = torch.from_numpy(X_val)
Y_val = torch.from_numpy(Y_val)

# Randomly initialize weights
W1 = torch.randn(D_in, H, device=device, dtype=dtype)
W2 = torch.randn(H, D_out, device=device, dtype=dtype)

losses_tr, losses_val = list(), list()

learning_rate = lr
for t in range(nite):
    # Forward pass: compute predicted y
    z1 = X.mm(W1)
    h1 = z1.clamp(min=0)
    y_pred = h1.mm(W2)

    # Compute and print loss
    loss = (y_pred - Y).pow(2).sum().item()

    # Backprop to compute gradients of w1 and w2 with respect to loss
    grad_y_pred = 2.0 * (y_pred - Y)
    grad_w2 = h1.t().mm(grad_y_pred)
    grad_h1 = grad_y_pred.mm(W2.t())
    grad_z1 = grad_h1.clone()
    grad_z1[z1 < 0] = 0
    grad_w1 = X.t().mm(grad_z1)

    # Update weights using gradient descent
    W1 -= learning_rate * grad_w1
    W2 -= learning_rate * grad_w2

    # Forward pass for validation set: compute predicted y
    z1 = X_val.mm(W1)
    h1 = z1.clamp(min=0)
    y_pred_val = h1.mm(W2)
    loss_val = (y_pred_val - Y_val).pow(2).sum().item()

    losses_tr.append(loss)
    losses_val.append(loss_val)

    if t % 10 == 0:

```

(continues on next page)

(continued from previous page)

```

        print(t, loss, loss_val)

    return W1, W2, losses_tr, losses_val

W1, W2, losses_tr, losses_val = \
    two_layer_regression_tensor_train(X=X_iris_tr, Y=Y_iris_tr, X_val=X_iris_val,
                                      Y_val=Y_iris_val,
                                      lr=1e-4, nite=50)

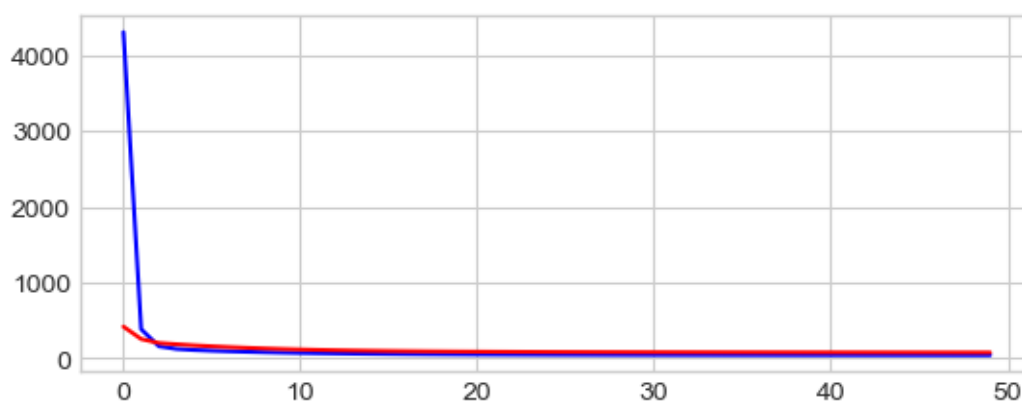
_ = plt.plot(np.arange(len(losses_tr)), losses_tr, "-b",
             np.arange(len(losses_val)), losses_val, "-r")

```

```

0 4303.93603515625 418.63006591796875
10 71.38607025146484 116.78504943847656
20 51.27054214477539 87.81427001953125
30 45.72554397583008 80.99897003173828
40 43.343406677246094 78.32192993164062

```



Backpropagation with PyTorch: Tensors and autograd

Learning PyTorch with Examples

A fully-connected ReLU network with one hidden layer and no biases, trained to predict y from x by minimizing squared Euclidean distance. This implementation computes the forward pass using operations on PyTorch Tensors, and uses PyTorch autograd to compute gradients. A PyTorch Tensor represents a node in a computational graph. If x is a Tensor that has `x.requires_grad=True` then `x.grad` is another Tensor holding the gradient of x with respect to some scalar value.

```

import torch

def two_layer_regression_autograd_train(X, Y, X_val, Y_val, lr, nite):

    dtype = torch.float
    device = torch.device("cpu")
    # device = torch.device("cuda:0") # Uncomment this to run on GPU

```

(continues on next page)

(continued from previous page)

```

# N is batch size; D_in is input dimension;
# H is hidden dimension; D_out is output dimension.
N, D_in, H, D_out = X.shape[0], X.shape[1], 100, Y.shape[1]

# Setting requires_grad=False indicates that we do not need to compute
# gradients with respect to these Tensors during the backward pass.
X = torch.from_numpy(X)
Y = torch.from_numpy(Y)
X_val = torch.from_numpy(X_val)
Y_val = torch.from_numpy(Y_val)

# Create random Tensors for weights.
# Setting requires_grad=True indicates that we want to compute gradients
# with respect to these Tensors during the backward pass.
W1 = torch.randn(D_in, H, device=device, dtype=dtype, requires_grad=True)
W2 = torch.randn(H, D_out, device=device, dtype=dtype, requires_grad=True)

losses_tr, losses_val = list(), list()

learning_rate = lr
for t in range(nite):
    # Forward pass: compute predicted y using operations on Tensors; these
    # are exactly the same operations we used to compute the forward pass
    # using Tensors, but we do not need to keep references to intermediate
    # values since we are not implementing the backward pass by hand.
    y_pred = X.mm(W1).clamp(min=0).mm(W2)

    # Compute and print loss using operations on Tensors.
    # Now loss is a Tensor of shape (1,)
    # loss.item() gets the scalar value held in the loss.
    loss = (y_pred - Y).pow(2).sum()

    # Use autograd to compute the backward pass. This call will compute the
    # gradient of loss with respect to all Tensors with requires_grad=True.
    # After this call w1.grad and w2.grad will be Tensors holding the
    # gradient of the loss with respect to w1 and w2 respectively.
    loss.backward()

    # Manually update weights using gradient descent. Wrap in torch.no_grad()
    # because weights have requires_grad=True, but we don't need to track this
    # in autograd.
    # An alternative way is to operate on weight.data and weight.grad.data.
    # Recall that tensor.data gives a tensor that shares the storage with
    # tensor, but doesn't track history.
    # You can also use torch.optim.SGD to achieve this.
    with torch.no_grad():
        W1 -= learning_rate * W1.grad
        W2 -= learning_rate * W2.grad

```

(continues on next page)

(continued from previous page)

```

# Manually zero the gradients after updating weights
W1.grad.zero_()
W2.grad.zero_()

y_pred = X_val.mm(W1).clamp(min=0).mm(W2)

# Compute and print loss using operations on Tensors.
# Now loss is a Tensor of shape (1,)
# loss.item() gets the scalar value held in the loss.
loss_val = (y_pred - Y).pow(2).sum()

if t % 10 == 0:
    print(t, loss.item(), loss_val.item())

losses_tr.append(loss.item())
losses_val.append(loss_val.item())

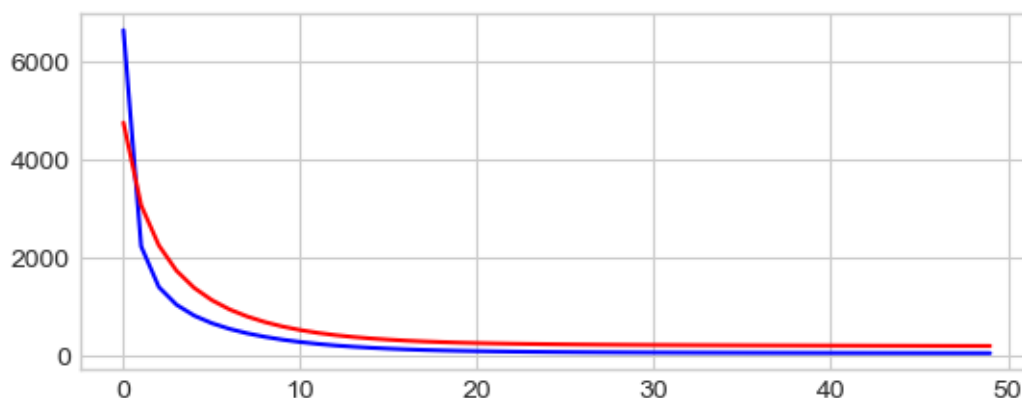
return W1, W2, losses_tr, losses_val
W1, W2, losses_tr, losses_val = \
    two_layer_regression_autograd_train(X=X_iris_tr, Y=Y_iris_tr,
                                         X_val=X_iris_val, Y_val=Y_iris_val,
                                         lr=1e-4, nite=50)
_ = plt.plot(np.arange(len(losses_tr)), losses_tr, "-b",
             np.arange(len(losses_val)), losses_val, "-r")

```

```

0 6619.669921875 4738.6064453125
10 278.9159851074219 521.75732421875
20 93.21061706542969 257.3363037109375
30 64.7365493774414 219.1724853515625
40 56.20269012451172 206.47886657714844

```



Backpropagation with PyTorch: nn

Learning PyTorch with Examples

This implementation uses the nn package from PyTorch to build the network. PyTorch autograd makes it easy to define computational graphs and take gradients, but raw autograd can be a bit too low-level for defining complex neural networks; this is where the nn package can help. The nn package defines a set of Modules, which you can think of as a neural network layer that has produces output from input and may have some trainable weights.

```
import torch

def two_layer_regression_nn_train(X, Y, X_val, Y_val, lr, nite):

    # N is batch size; D_in is input dimension;
    # H is hidden dimension; D_out is output dimension.
    N, D_in, H, D_out = X.shape[0], X.shape[1], 100, Y.shape[1]

    X = torch.from_numpy(X)
    Y = torch.from_numpy(Y)
    X_val = torch.from_numpy(X_val)
    Y_val = torch.from_numpy(Y_val)

    # Use the nn package to define our model as a sequence of layers.
    # nn.Sequential is a Module which contains other Modules, and applies
    # them in sequence to produce its output. Each Linear Module computes
    # output from input using a linear function, and holds internal Tensors
    # for its weight and bias.
    model = torch.nn.Sequential(
        torch.nn.Linear(D_in, H),
        torch.nn.ReLU(),
        torch.nn.Linear(H, D_out),
    )

    # The nn package also contains definitions of popular loss functions; in this
    # case we will use Mean Squared Error (MSE) as our loss function.
    loss_fn = torch.nn.MSELoss(reduction='sum')

    losses_tr, losses_val = list(), list()

    learning_rate = lr
    for t in range(nite):
        # Forward pass: compute predicted y by passing x to the model. Module
        # objects override the __call__ operator so you can call them like
        # functions. When doing so you pass a Tensor of input data to the Module
        # and it produces a Tensor of output data.
        y_pred = model(X)

        # Compute and print loss. We pass Tensors containing the predicted and
        # true values of y, and the loss function returns a Tensor containing the
        # loss.
```

(continues on next page)

(continued from previous page)

```

loss = loss_fn(y_pred, Y)

# Zero the gradients before running the backward pass.
model.zero_grad()

# Backward pass: compute gradient of the loss with respect to all the
# learnable parameters of the model. Internally, the parameters of each
# Module are stored in Tensors with requires_grad=True, so this call
# will compute gradients for all learnable parameters in the model.
loss.backward()

# Update the weights using gradient descent. Each parameter is a Tensor,
# so we can access its gradients like we did before.
with torch.no_grad():
    for param in model.parameters():
        param -= learning_rate * param.grad
    y_pred = model(X_val)
    loss_val = (y_pred - Y_val).pow(2).sum()

    if t % 10 == 0:
        print(t, loss.item(), loss_val.item())

    losses_tr.append(loss.item())
    losses_val.append(loss_val.item())

return model, losses_tr, losses_val

model, losses_tr, losses_val = \
    two_layer_regression_nn_train(X=X_iris_tr, Y=Y_iris_tr,
                                  X_val=X_iris_val, Y_val=Y_iris_val,
                                  lr=1e-4, nite=50)

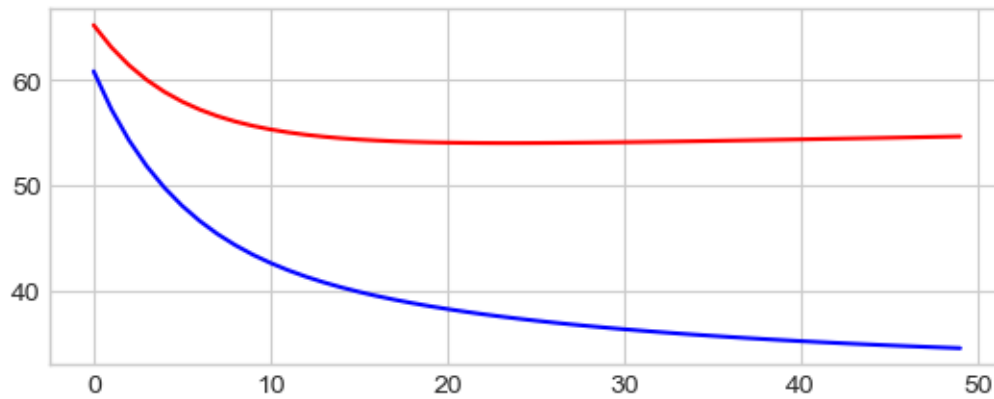
_ = plt.plot(np.arange(len(losses_tr)), losses_tr, "-b",
             np.arange(len(losses_val)), losses_val, "-r")

```

```

0 60.84171676635742 65.23538970947266
10 42.599857330322266 55.307090759277344
20 38.216976165771484 54.05012130737305
30 36.2943000793457 54.090911865234375
40 35.161956787109375 54.361236572265625

```



Backpropagation with PyTorch optim

This implementation uses the nn package from PyTorch to build the network. Rather than manually updating the weights of the model as we have been doing, we use the optim package to define an Optimizer that will update the weights for us. The optim package defines many optimization algorithms that are commonly used for deep learning, including SGD+momentum, RMSProp, Adam, etc.

```
import torch

def two_layer_regression_nn_optim_train(X, Y, X_val, Y_val, lr, nite):

    # N is batch size; D_in is input dimension;
    # H is hidden dimension; D_out is output dimension.
    N, D_in, H, D_out = X.shape[0], X.shape[1], 100, Y.shape[1]

    X = torch.from_numpy(X)
    Y = torch.from_numpy(Y)
    X_val = torch.from_numpy(X_val)
    Y_val = torch.from_numpy(Y_val)

    # Use the nn package to define our model and loss function.
    model = torch.nn.Sequential(
        torch.nn.Linear(D_in, H),
        torch.nn.ReLU(),
        torch.nn.Linear(H, D_out),
    )
    loss_fn = torch.nn.MSELoss(reduction='sum')

    losses_tr, losses_val = list(), list()

    # Use the optim package to define an Optimizer that will update the weights of
    # the model for us. Here we will use Adam; the optim package contains many
    # other optimization algorithm. The first argument to the Adam constructor
    # tells the optimizer which Tensors it should update.
    learning_rate = lr
    optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

(continues on next page)

(continued from previous page)

```

for t in range(nite):
    # Forward pass: compute predicted y by passing x to the model.
    y_pred = model(X)

    # Compute and print loss.
    loss = loss_fn(y_pred, Y)

    # Before the backward pass, use the optimizer object to zero all of the
    # gradients for the variables it will update (which are the learnable
    # weights of the model). This is because by default, gradients are
    # accumulated in buffers( i.e, not overwritten) whenever .backward()
    # is called. Checkout docs of torch.autograd.backward for more details.
    optimizer.zero_grad()

    # Backward pass: compute gradient of the loss with respect to model
    # parameters
    loss.backward()

    # Calling the step function on an Optimizer makes an update to its
    # parameters
    optimizer.step()

    with torch.no_grad():
        y_pred = model(X_val)
        loss_val = loss_fn(y_pred, Y_val)

    if t % 10 == 0:
        print(t, loss.item(), loss_val.item())

    losses_tr.append(loss.item())
    losses_val.append(loss_val.item())

return model, losses_tr, losses_val

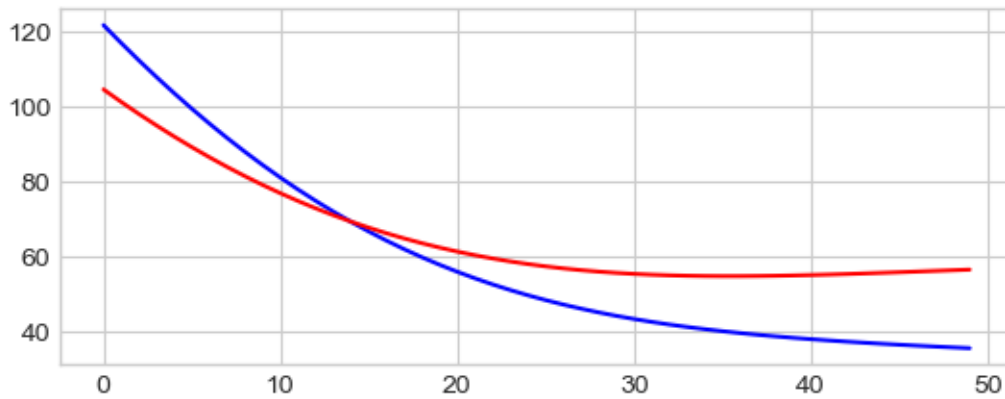
model, losses_tr, losses_val = \
    two_layer_regression_nn_optim_train(X=X_iris_tr, Y=Y_iris_tr,
                                       X_val=X_iris_val, Y_val=Y_iris_val,
                                       lr=1e-3, nite=50)
_ = plt.plot(np.arange(len(losses_tr)), losses_tr, "-b",
            np.arange(len(losses_val)), losses_val, "-r")

```

```

0 121.76207733154297 104.62301635742188
10 81.1495361328125 76.97682189941406
20 56.0490837097168 61.42247772216797
30 43.4493293762207 55.462921142578125
40 38.075889587402344 55.15611267089844

```



10.2 Multilayer Perceptron (MLP)

Sources:

Sources:

- 3Blue1Brown video: But what is a neural network? | Deep learning chapter 1
- Stanford cs231n: Deep learning
- Pytorch: WWW tutorials
- Pytorch: github tutorials
- Pytorch: github examples
- Pytorch examples
- MNIST/pytorch nextjournal.com/gkoehler/pytorch-mnist
- Pytorch: [github/pytorch/examples](https://github.com/pytorch/examples)
- kaggle: MNIST/pytorch

```
import os
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.optim import lr_scheduler
# import torchvision
# from torchvision import transforms
# from torchvision import datasets
# from torchvision import models
#
from pathlib import Path
# Plot
import matplotlib.pyplot as plt
import seaborn as sns

# Plot parameters
```

(continues on next page)

(continued from previous page)

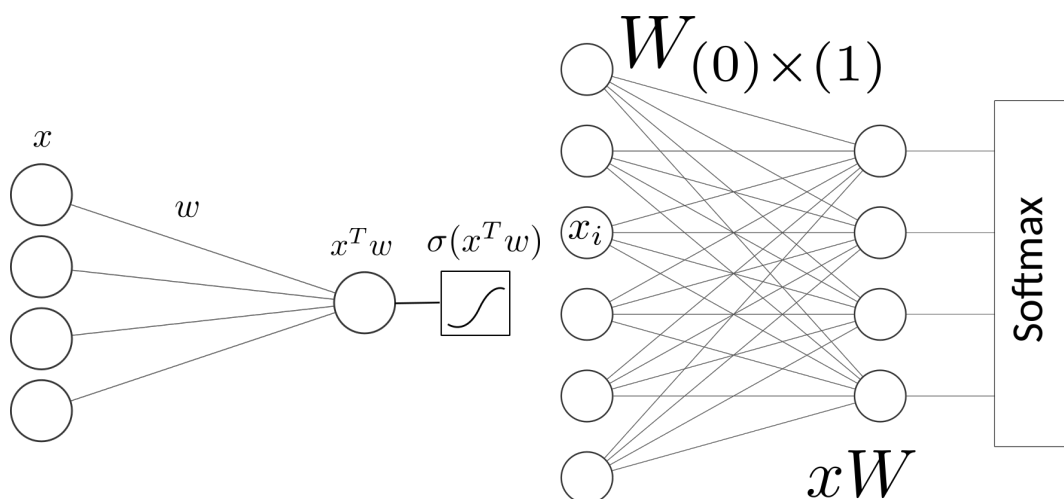
```
plt.style.use('seaborn-v0_8-whitegrid')
fig_w, fig_h = plt.rcParams.get('figure.figsize')
plt.rcParams['figure.figsize'] = (fig_w, fig_h * .5)

# Device configuration
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
# device = 'cpu' # Force CPU
print(device)
```

cpu

10.2.1 Single Layer Softmax Classifier (Multinomial Logistic Regression)

Recall of Binary logistic regression



Input layer of size (0) Output layer of size (1) Input layer of size (0) Output layer of size (1)

One neuron as output layer

$$f(x) = \sigma(x^T w + b)$$

Where

- Input: x : a vector of dimension (p) (layer 0).
- Parameters: w : a vector of dimension (p) (layer 1). b is the scalar bias.
- Output: $f(x)$ a vector of dimension 1.

With multinomial logistic regression we have k possible labels to predict. If we consider the MNIST Handwritten Digit Recognition, the inputs is a $28 \times 28 = 784$ image and the output is a vector of $k = 10$ labels or probabilities.

$$f(x) = \text{softmax}(x^T W + b)$$

- Input: x : a vector of dimension $(p = 784)$ (layer 0).
- Parameters: W : the matrix of coefficients of dimension $(p \times k)$ (layer 1). b is a (k) -dimensional vector of bias.

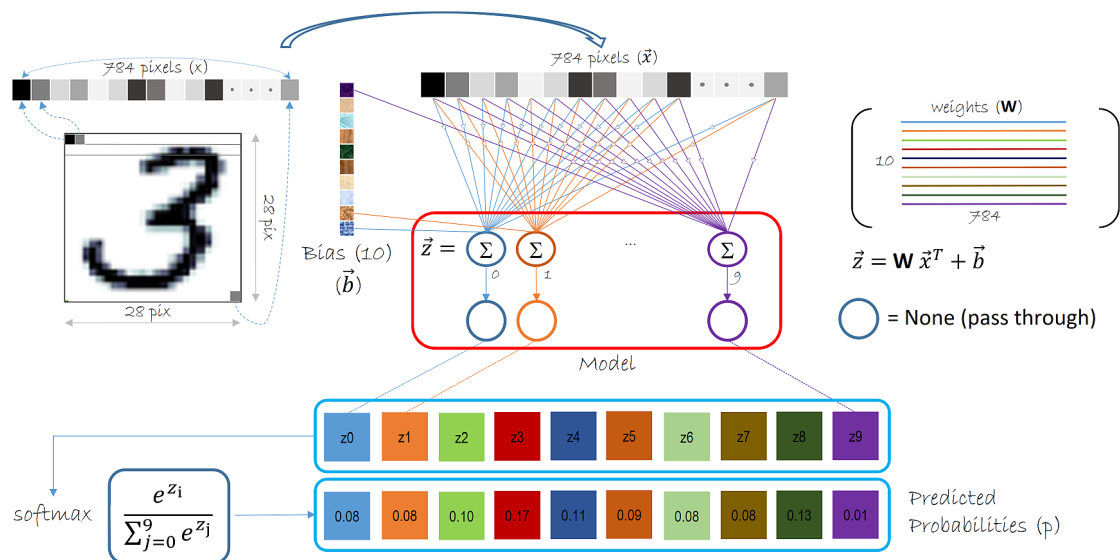


Fig. 1: Multinomial Logistic Regression on MNIST

- Output: $f(x)$ a vector of dimension ($k = 10$) possible labels

The softmax function is a crucial component in many machine learning and deep learning models, particularly in the context of classification tasks. It is used to convert a vector of raw scores (logits) into a probability distribution. Here's a detailed explanation of the softmax function: The softmax function takes a vector of real numbers as input and outputs a vector of probabilities that sum to 1. The formula for the softmax function is:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

where: - z_i is the i -th element of the input vector z . - e is the base of the natural logarithm. - The sum in the denominator is over all elements of the input vector.

Softmax Properties

1. **Probability Distribution:** The output of the softmax function is a probability distribution, meaning that all the outputs are non-negative and sum to 1.
2. **Exponential Function:** The use of the exponential function ensures that the outputs are positive and that larger input values correspond to larger probabilities.
3. **Normalization:** The softmax function normalizes the input values by dividing by the sum of the exponentials of all input values, ensuring that the outputs sum to 1

MNIST classification using multinomial logistic

source: [Logistic regression MNIST](#)

Here we fit a multinomial logistic regression with L2 penalty on a subset of the MNIST digits classification task.

source: [scikit-learn.org](#)

Hyperparameters

10.2.2 Dataset: MNIST Handwritten Digit Recognition

MNIST Loader

```
from pystatsml.datasets import load_mnist_pytorch

dataloaders, WD = load_mnist_pytorch(
    batch_size_train=64, batch_size_test=10000)
os.makedirs(os.path.join(WD, "models"), exist_ok=True)

# Info about the dataset
D_in = np.prod(dataloaders["train"].dataset.data.shape[1:])
D_out = len(dataloaders["train"].dataset.targets.unique())
print("Datasets shapes:", {
    x: dataloaders[x].dataset.data.shape for x in ['train', 'test']})
print("N input features:", D_in, "Output classes:", D_out)
```

```
/home/ed203246/data/pystatml/dl_mnist_pytorch
Datasets shapes: {'train': torch.Size([60000, 28, 28]), 'test': torch.Size([10000,
↪ 28, 28])}
N input features: 784 Output classes: 10
```

Now let's take a look at some mini-batches examples.

```
batch_idx, (example_data, example_targets) = next(
    enumerate(dataloaders["train"]))
print("Train batch:", example_data.shape, example_targets.shape)
batch_idx, (example_data, example_targets) = next(
    enumerate(dataloaders["test"]))
print("Val batch:", example_data.shape, example_targets.shape)
```

```
Train batch: torch.Size([64, 1, 28, 28]) torch.Size([64])
Val batch: torch.Size([10000, 1, 28, 28]) torch.Size([10000])
```

So one test data batch is a tensor of shape: . This means we have 1000 examples of 28x28 pixels in grayscale (i.e. no rgb channels, hence the one). We can plot some of them using matplotlib.

```
def show_data_label_prediction(data, y_true, y_pred=None, shape=(2, 3)):
    y_pred = [None] * len(y_true) if y_pred is None else y_pred
    fig = plt.figure()
    for i in range(np.prod(shape)):
        plt.subplot(*shape, i+1)
        plt.tight_layout()
        plt.imshow(data[i][0], cmap='gray', interpolation='none')
        plt.title("True: {} Pred: {}".format(y_true[i], y_pred[i]))
        plt.xticks([])
        plt.yticks([])

show_data_label_prediction(
    data=example_data, y_true=example_targets, y_pred=None, shape=(2, 3))
```


True: 4 Pred: None



True: 5 Pred: None



True: 0 Pred: None



True: 0 Pred: None



True: 3 Pred: None



True: 8 Pred: None



```
X_train = dataloaders["train"].dataset.data.numpy()
X_train = X_train.reshape((X_train.shape[0], -1))
y_train = dataloaders["train"].dataset.targets.numpy()

X_test = dataloaders["test"].dataset.data.numpy()
X_test = X_test.reshape((X_test.shape[0], -1))
y_test = dataloaders["test"].dataset.targets.numpy()

print(X_train.shape, y_train.shape)
```

```
(60000, 784) (60000,)
```

```
import matplotlib.pyplot as plt
import numpy as np

from sklearn.linear_model import LogisticRegression
# from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.utils import check_random_state

scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Turn up tolerance for faster convergence
clf = LogisticRegression(C=50., solver='sag', tol=0.1)
clf.fit(X_train, y_train)
# sparsity = np.mean(clf.coef_ == 0) * 100
score = clf.score(X_test, y_test)

print("Test score with penalty: %.4f" % score)
```

```
Test score with penalty: 0.8997
```

```
coef = clf.coef_.copy()
plt.figure(figsize=(10, 5))
scale = np.abs(coef).max()
```

(continues on next page)

(continued from previous page)

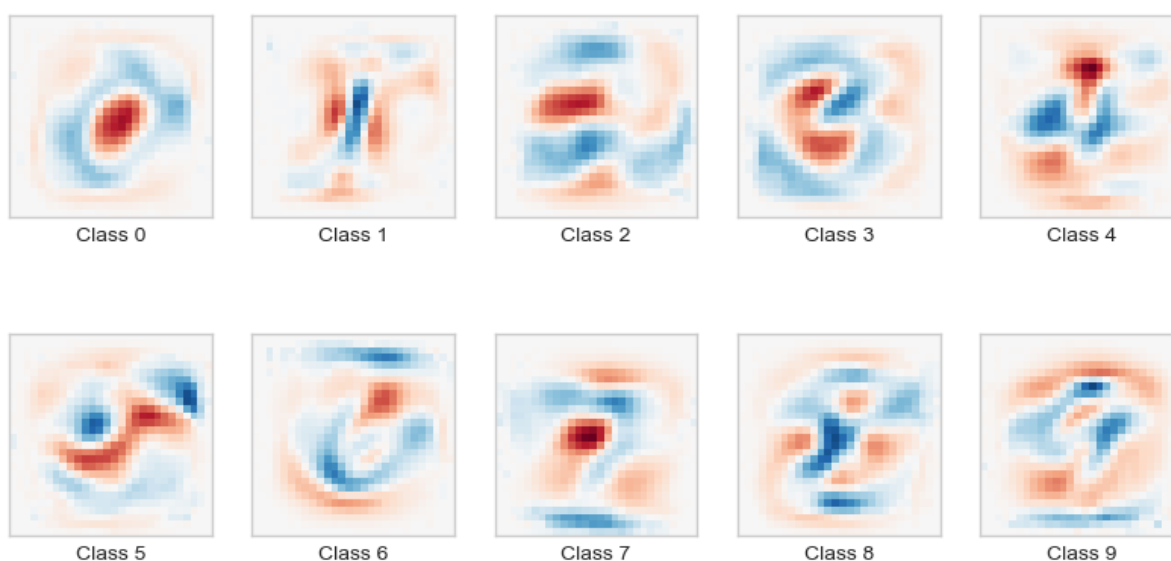
```

for i in range(10):
    l1_plot = plt.subplot(2, 5, i + 1)
    l1_plot.imshow(coef[i].reshape(28, 28), interpolation='nearest',
                  cmap=plt.cm.RdBu, vmin=-scale, vmax=scale)
    l1_plot.set_xticks(())
    l1_plot.set_yticks(())
    l1_plot.set_xlabel('Class %i' % i)
plt.suptitle('Classification vector for...')

plt.show()

```

Classification vector for...



10.2.3 Model: Two Layer MLP

MLP with Scikit-learn

```

from sklearn.neural_network import MLPClassifier

mlp = MLPClassifier(hidden_layer_sizes=(100, ), max_iter=10, alpha=1e-4,
                    solver='sgd', verbose=10, tol=1e-4, random_state=1,
                    learning_rate_init=0.01, batch_size=64)

mlp.fit(X_train, y_train)
print("Training set score: %f" % mlp.score(X_train, y_train))
print("Test set score: %f" % mlp.score(X_test, y_test))

print("Coef shape=", len(mlp.coefs_))

fig, axes = plt.subplots(4, 4)
# use global min / max to ensure all weights are shown on the same scale
vmin, vmax = mlp.coefs_[0].min(), mlp.coefs_[0].max()

```

(continues on next page)

(continued from previous page)

```

for coef, ax in zip(mlp.coefs_[0].T, axes.ravel()):
    ax.matshow(coef.reshape(28, 28), cmap=plt.cm.gray, vmin=.5 * vmin,
               vmax=.5 * vmax)
    ax.set_xticks(())
    ax.set_yticks(())

plt.show()

```

```

Iteration 1, loss = 0.28828673
Iteration 2, loss = 0.13388073
Iteration 3, loss = 0.09366379
Iteration 4, loss = 0.07317648
Iteration 5, loss = 0.05340251
Iteration 6, loss = 0.04468092
Iteration 7, loss = 0.03548097
Iteration 8, loss = 0.02862098
Iteration 9, loss = 0.02449230
Iteration 10, loss = 0.01874513

```

```

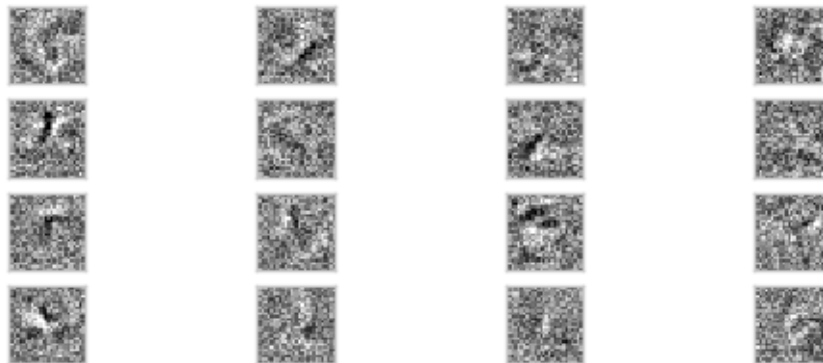
/home/ed203246/git/pystatsml/.pixi/envs/default/lib/python3.12/site-packages/
↳ sklearn/neural_network/_multilayer_perceptron.py:690: ConvergenceWarning:
↳ Stochastic Optimizer: Maximum iterations (10) reached and the optimization hasn
↳ 't converged yet.
warnings.warn(

```

```

Training set score: 0.997800
Test set score: 0.974900
Coef shape= 2

```



MLP with pytorch

```

class TwoLayerMLP(nn.Module):

    def __init__(self, d_in, d_hidden, d_out):
        super(TwoLayerMLP, self).__init__()
        self.d_in = d_in

```

(continues on next page)

(continued from previous page)

```
self.linear1 = nn.Linear(d_in, d_hidden)
self.linear2 = nn.Linear(d_hidden, d_out)

def forward(self, X):
    X = X.view(-1, self.d_in)
    X = self.linear1(X)
    return F.log_softmax(self.linear2(X), dim=1)
```

Train the Model

- First we want to make sure our network is in training mode.
- Iterate over epochs
- Alternate train and validation dataset
- Iterate over all training/val data once per epoch. Loading the individual batches is handled by the DataLoader.
- Set the gradients to zero using `optimizer.zero_grad()` since PyTorch by default accumulates gradients.
- Forward pass:
 - `model(inputs)`: Produce the output of our network.
 - `torch.max(outputs, 1)`: softmax predictions.
 - `criterion(outputs, labels)`: loss between the output and the ground truth label.
- In training mode, backward pass `backward()`: collect a new set of gradients which we propagate back into each of the network's parameters using `optimizer.step()`.
- We'll also keep track of the progress with some printouts. In order to create a nice training curve later on we also create two lists for saving training and testing losses. On the x-axis we want to display the number of training examples the network has seen during training.
- Save model state: Neural network modules as well as optimizers have the ability to save and load their internal state using `.state_dict()`. With this we can continue training from previously saved state dicts if needed - we'd just need to call `.load_state_dict(state_dict)`.

See `train_val_model` function.

```
from pystatsml.dl_utils import train_val_model
```

Save and reload PyTorch model

PyTorch doc: Save and reload PyTorch model: *Note “If you only plan to keep the best performing model (according to the acquired validation loss), don't forget that `best_model_state = model.state_dict()` returns a reference to the state and not its copy! You must serialize `best_model_state` or use `best_model_state = deepcopy(model.state_dict())` otherwise your `best_model_state` will keep getting updated by the subsequent training iterations. As a result, the final model state will be the state of the overfitted model.”*

Save/Load `state_dict` (Recommended) Save:

```
import torch
from copy import deepcopy
torch.save(deepcopy(model.state_dict()), PATH)
```

Load:

```
model = TheModelClass(*args, **kwargs)
model.load_state_dict(torch.load(PATH, weights_only=True))
model.eval()
```

Run one epoch and save the model

```
from copy import deepcopy

model = TwoLayerMLP(D_in, 50, D_out).to(device)
print(next(model.parameters()).is_cuda)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
criterion = nn.NLLLoss()

# Explore the model
for parameter in model.parameters():
    print(parameter.shape)

print("Total number of parameters =", np.sum(
    [np.prod(parameter.shape) for parameter in model.parameters()]))

model, losses, accuracies = \
    train_val_model(model, criterion, optimizer, dataloaders,
                    num_epochs=1, log_interval=1)

print(next(model.parameters()).is_cuda)
torch.save(deepcopy(model.state_dict()),
            os.path.join(WD, 'models/mod-%s.pth' % model.__class__.__name__))
```

```
False
torch.Size([50, 784])
torch.Size([50])
torch.Size([10, 50])
torch.Size([10])
Total number of parameters = 39760
Epoch 0/0
-----
train Loss: 0.4500 Acc: 87.61%
test Loss: 0.3059 Acc: 91.05%

Training complete in 0m 5s
Best val Acc: 91.05%
False
```

Reload model

```
model_ = TwoLayerMLP(D_in, 50, D_out)
model_.load_state_dict(torch.load(os.path.join(
    WD, 'models/mod-%s.pth' % model.__class__.__name__), weights_only=True))
model_.to(device)
```

```
TwoLayerMLP(
  (linear1): Linear(in_features=784, out_features=50, bias=True)
  (linear2): Linear(in_features=50, out_features=10, bias=True)
)
```

Use the model to make new predictions. Consider the device, ie, load data on device `example_data.to(device)` from prediction, then move back to cpu `example_data.cpu()`.

```
batch_idx, (example_data, example_targets) = next(
    enumerate(data loaders["test"]))
example_data = example_data.to(device)

with torch.no_grad():
    output = model(example_data).cpu()

example_data = example_data.cpu()

# Softmax predictions
preds = output.argmax(dim=1)

print("Output shape=", output.shape, "label shape=", preds.shape)
print("Accuracy = {:.2f}%".format(
    (example_targets == preds).sum().item() * 100. / len(example_targets)))

show_data_label_prediction(
    data=example_data, y_true=example_targets, y_pred=preds, shape=(3, 4))
```

```
Output shape= torch.Size([10000, 10]) label shape= torch.Size([10000])
Accuracy = 91.05%
```

True: 5 Pred: 5 	True: 5 Pred: 5 	True: 6 Pred: 6 	True: 3 Pred: 3 
True: 7 Pred: 7 	True: 7 Pred: 7 	True: 8 Pred: 8 	True: 5 Pred: 5 
True: 7 Pred: 7 	True: 4 Pred: 4 	True: 1 Pred: 1 	True: 1 Pred: 1 

Plot missclassified samples

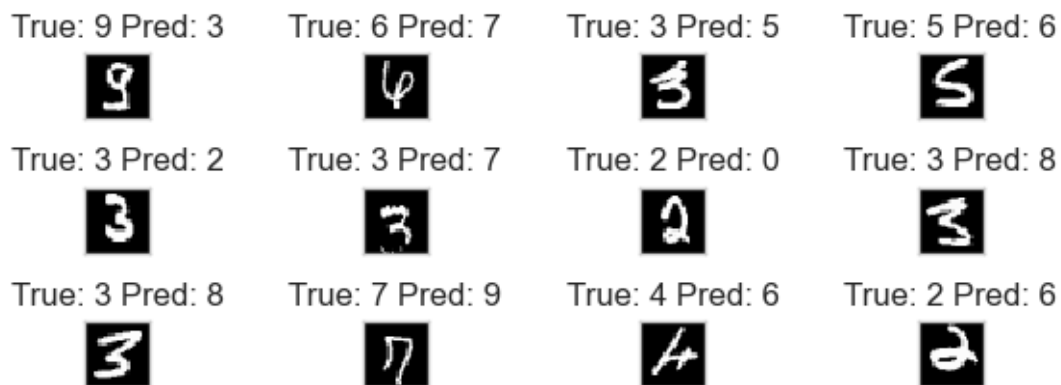
```
errors = example_targets != preds
```

(continues on next page)

(continued from previous page)

```
# print(errors, np.where(errors))
print("Nb errors = {}, (Error rate = {:.2f}%)".format(
    errors.sum(), 100 * errors.sum().item() / len(errors)))
err_idx = np.where(errors)[0]
show_data_label_prediction(data=example_data[err_idx],
                           y_true=example_targets[err_idx],
                           y_pred=preds[err_idx], shape=(3, 4))
```

Nb errors = 895, (Error rate = 8.95%)



Continue training from checkpoints: reload the model and run 10 more epochs

```
model = TwoLayerMLP(D_in, 50, D_out)
model.load_state_dict(torch.load(os.path.join(
    WD, 'models/mod-%s.pth' % model.__class__.__name__), weights_only=False))
model.to(device)

optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
criterion = nn.NLLLoss()

model, losses, accuracies = \
    train_val_model(model, criterion, optimizer, dataloaders,
                    num_epochs=10, log_interval=2)

_ = plt.plot(losses['train'], '-b', losses['test'], '--r')
```

```
Epoch 0/9
-----
train Loss: 0.3097 Acc: 91.11%
test Loss: 0.2904 Acc: 91.91%

Epoch 2/9
-----
train Loss: 0.2844 Acc: 91.94%
test Loss: 0.2809 Acc: 92.10%
```

(continues on next page)

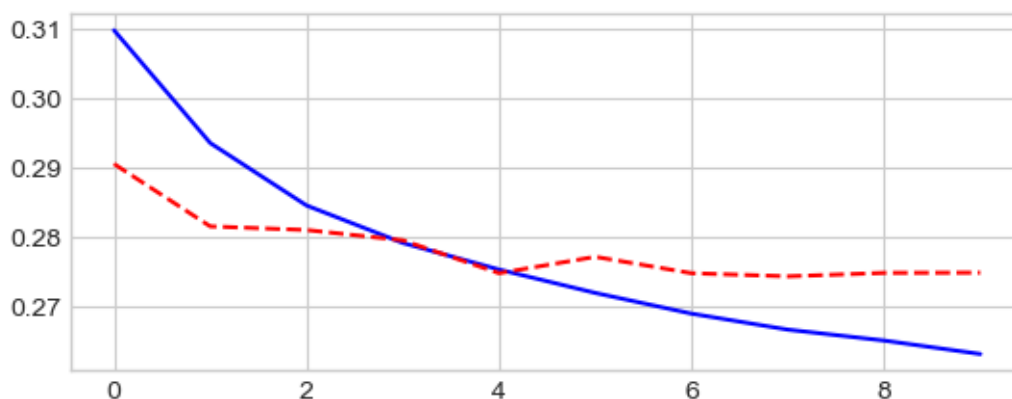
(continued from previous page)

```
Epoch 4/9
-----
train Loss: 0.2752 Acc: 92.21%
test Loss: 0.2747 Acc: 92.23%
```

```
Epoch 6/9
-----
train Loss: 0.2688 Acc: 92.45%
test Loss: 0.2747 Acc: 92.17%
```

```
Epoch 8/9
-----
train Loss: 0.2650 Acc: 92.64%
test Loss: 0.2747 Acc: 92.32%
```

```
Training complete in 0m 40s
Best val Acc: 92.32%
```



10.2.4 Test several MLP architectures

- Define a `MultiLayerMLP([D_in, 512, 256, 128, 64, D_out])` class that take the size of the layers as parameters of the constructor.
- Add some non-linearity with `relu` activation function

```
class MLP(nn.Module):

    def __init__(self, d_layer):
        super(MLP, self).__init__()
        self.d_layer = d_layer
        # Add linear layers
        layer_list = [nn.Linear(d_layer[l], d_layer[l+1])
                      for l in range(len(d_layer) - 1)]
        self.linears = nn.ModuleList(layer_list)

    def forward(self, X):
        X = X.view(-1, self.d_layer[0])
        # relu(Wl x) for all hidden layer
```

(continues on next page)

(continued from previous page)

```

    for layer in self.linears[:-1]:
        X = F.relu(layer(X))
    # softmax(Wl x) for output layer
    return F.log_softmax(self.linears[-1](X), dim=1)

```

```

model = MLP([D_in, 512, 256, 128, 64, D_out]).to(device)

optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
criterion = nn.NLLLoss()

model, losses, accuracies = \
    train_val_model(model, criterion, optimizer, dataloaders,
                    num_epochs=10, log_interval=2)

_ = plt.plot(losses['train'], '-b', losses['test'], '--r')

```

```

Epoch 0/9
-----
train Loss: 1.1449 Acc: 64.01%
test Loss: 0.3366 Acc: 89.96%

Epoch 2/9
-----
train Loss: 0.1693 Acc: 95.00%
test Loss: 0.1361 Acc: 96.11%

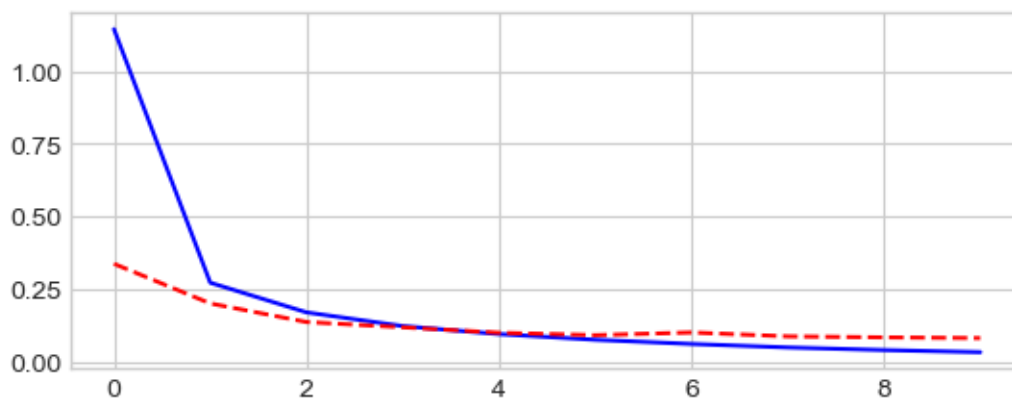
Epoch 4/9
-----
train Loss: 0.0946 Acc: 97.21%
test Loss: 0.0992 Acc: 96.94%

Epoch 6/9
-----
train Loss: 0.0606 Acc: 98.20%
test Loss: 0.1002 Acc: 96.97%

Epoch 8/9
-----
train Loss: 0.0395 Acc: 98.87%
test Loss: 0.0833 Acc: 97.42%

Training complete in 1m 11s
Best val Acc: 97.60%

```



10.2.5 Reduce the size of training dataset

Reduce the size of the training dataset by considering only 10 minibatches for size16.

```
train_size = 10 * 16

# Stratified sub-sampling
targets = dataloaders["train"].dataset.targets.numpy()
nclasses = len(set(targets))

indices = np.concatenate([np.random.choice(np.where(targets == lab)[0],
                                           int(train_size / nclasses),
                                           replace=False)
                          for lab in set(targets)])
np.random.shuffle(indices)

train_loader = \
    torch.utils.data.DataLoader(dataloaders["train"].dataset,
                               batch_size=16,
                               sampler=torch.utils.data.SubsetRandomSampler(indices))

# Check train subsampling
train_labels = np.concatenate([labels.numpy()
                              for inputs, labels in train_loader])
print("Train size=", len(train_labels), " Train label count=",
      {lab: np.sum(train_labels == lab) for lab in set(train_labels)})
print("Batch sizes=", [inputs.size(0) for inputs, labels in train_loader])

# Put together train and val
dataloaders = dict(train=train_loader, test=dataloaders["test"])

# Info about the dataset
D_in = np.prod(dataloaders["train"].dataset.data.shape[1:])
D_out = len(dataloaders["train"].dataset.targets.unique())
print("Datasets shape", {x: dataloaders[x].dataset.data.shape
                        for x in dataloaders.keys()})
print("N input features", D_in, "N output", D_out)
```

```

Train size= 160 Train label count= {np.int64(0): np.int64(16), np.int64(1): np.
↪int64(16), np.int64(2): np.int64(16), np.int64(3): np.int64(16), np.int64(4): ↪
↪np.int64(16), np.int64(5): np.int64(16), np.int64(6): np.int64(16), np.
↪int64(7): np.int64(16), np.int64(8): np.int64(16), np.int64(9): np.int64(16)}
Batch sizes= [16, 16, 16, 16, 16, 16, 16, 16, 16, 16]
Datasets shape {'train': torch.Size([60000, 28, 28]), 'test': torch.Size([10000, ↪
↪28, 28])}
N input features 784 N output 10

```

```

model = MLP([D_in, 512, 256, 128, 64, D_out]).to(device)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
criterion = nn.NLLLoss()

model, losses, accuracies = \
    train_val_model(model, criterion, optimizer, dataloaders,
                    num_epochs=100, log_interval=20)

_ = plt.plot(losses['train'], '-b', losses['test'], '--r')

```

```

Epoch 0/99
-----
train Loss: 2.3042 Acc: 10.00%
test Loss: 2.3013 Acc: 9.80%

Epoch 20/99
-----
train Loss: 2.0282 Acc: 30.63%
test Loss: 2.0468 Acc: 24.58%

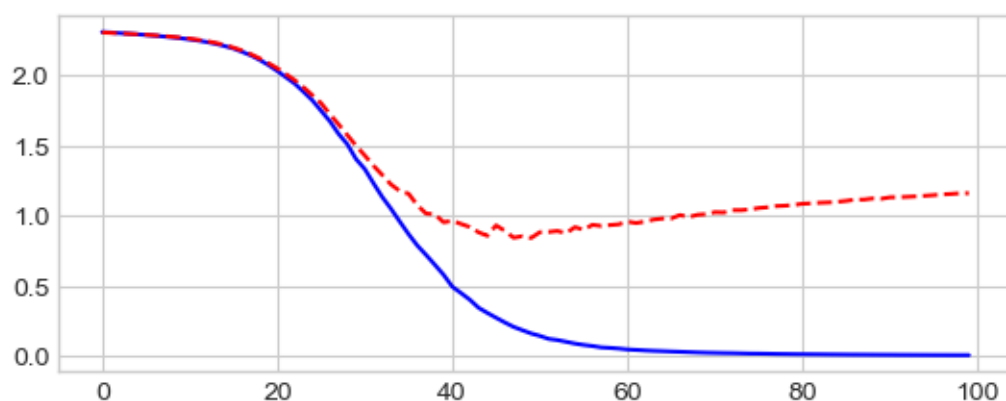
Epoch 40/99
-----
train Loss: 0.4945 Acc: 88.75%
test Loss: 0.9630 Acc: 66.42%

Epoch 60/99
-----
train Loss: 0.0483 Acc: 100.00%
test Loss: 0.9570 Acc: 74.40%

Epoch 80/99
-----
train Loss: 0.0145 Acc: 100.00%
test Loss: 1.0840 Acc: 74.40%

Training complete in 1m 7s
Best val Acc: 75.32%

```



Use an optimizer with an adaptative learning rate: Adam

```
model = MLP([D_in, 512, 256, 128, 64, D_out]).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
criterion = nn.NLLLoss()

model, losses, accuracies = \
    train_val_model(model, criterion, optimizer, dataloaders,
                    num_epochs=100, log_interval=20)

_ = plt.plot(losses['train'], '-b', losses['test'], '--r')
```

Epoch 0/99

train Loss: 2.2763 Acc: 15.00%

test Loss: 2.1595 Acc: 45.78%

Epoch 20/99

train Loss: 0.0010 Acc: 100.00%

test Loss: 1.1346 Acc: 77.06%

Epoch 40/99

train Loss: 0.0003 Acc: 100.00%

test Loss: 1.2461 Acc: 76.97%

Epoch 60/99

train Loss: 0.0001 Acc: 100.00%

test Loss: 1.3149 Acc: 77.00%

Epoch 80/99

train Loss: 0.0001 Acc: 100.00%

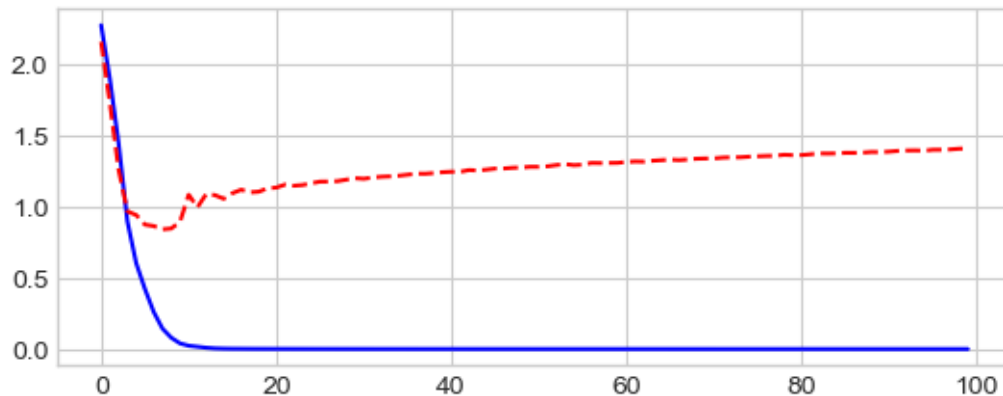
test Loss: 1.3633 Acc: 77.07%

Training complete in 1m 6s

(continues on next page)

(continued from previous page)

Best val Acc: 77.54%



10.2.6 Run MLP on CIFAR-10 dataset

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class. The ten classes are: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck

Load CIFAR-10 dataset [CIFAR-10 Loader](#)

```
from pystatsml.datasets import load_cifar10_pytorch

dataloaders, _ = load_cifar10_pytorch(
    batch_size_train=100, batch_size_test=100)

# Info about the dataset
D_in = np.prod(dataloaders["train"].dataset.data.shape[1:])
D_out = len(set(dataloaders["train"].dataset.targets))
print("Datasets shape:", {
    x: dataloaders[x].dataset.data.shape for x in dataloaders.keys()})
print("N input features:", D_in, "N output:", D_out)
```

```
Files already downloaded and verified
Files already downloaded and verified
Datasets shape: {'train': (50000, 32, 32, 3), 'test': (10000, 32, 32, 3)}
N input features: 3072 N output: 10
```

Run MLP Classifier with hidden layers of sizes: 512, 256, 128, and 64:

```
model = MLP([D_in, 512, 256, 128, 64, D_out]).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
criterion = nn.NLLLoss()
```

(continues on next page)

(continued from previous page)

```
model, losses, accuracies = \
    train_val_model(model, criterion, optimizer, dataloaders,
                    num_epochs=20, log_interval=10)

_ = plt.plot(losses['train'], '-b', losses['test'], '--r')
```

Epoch 0/19

train Loss: 1.6872 Acc: 39.50%

test Loss: 1.5318 Acc: 45.31%

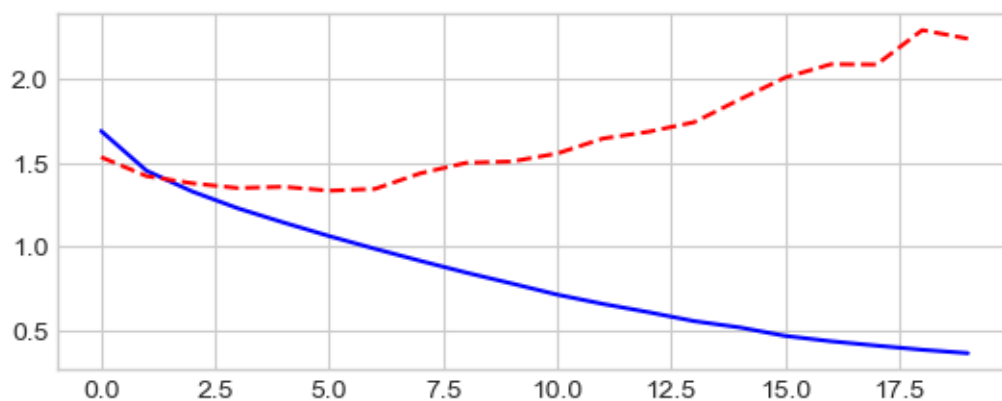
Epoch 10/19

train Loss: 0.7136 Acc: 74.50%

test Loss: 1.5536 Acc: 54.77%

Training complete in 4m 3s

Best val Acc: 54.84%



DEEP LEARNING FOR COMPUTER VISION

11.1 Convolutional Neural Networks (CNNs)

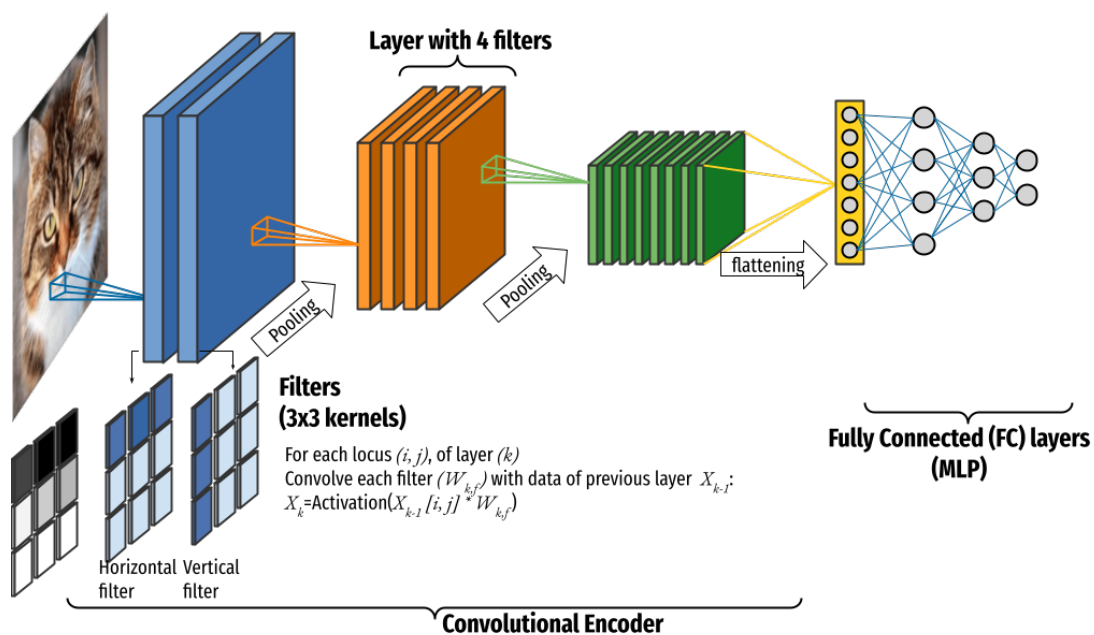


Fig. 1: Principles of CNNs

Sources:

- [3Blue1Brown video: Convolutions in Image Processing](#)
- [far1din video: Convolutional Neural Networks from Scratch](#)
- [What is a Convolutional Neural Network?.](#)
- [CNN Stanford cs231n](#)
- [Deep learning Stanford cs231n](#)
- [Pytorch](#)
 - [WWW tutorials](#)
 - [github tutorials](#)
 - [github examples](#)
- [MNIST and pytorch:](#)

- [MNIST nextjournal.com/gkoehler/pytorch-mnist](https://nextjournal.com/gkoehler/pytorch-mnist)
- [MNIST github/pytorch/examples](https://github.com/pytorch/examples)
- [MNIST kaggle](https://www.kaggle.com/c/mnist)

11.1.1 Introduction to CNNs

CNNs are deep learning architectures designed for processing grid-like data such as images. Inspired by the biological visual cortex, they learn hierarchical feature representations, making them effective for tasks like image classification, object detection, and segmentation.

Key Principles of CNNs:

- **Convolutional Layers** are the core building block of a CNN, which applies a convolution operation to the input, passing the result to the next layer: it perform feature extraction using learnable filters (kernels), allowing CNNs to detect local patterns such as edges and textures.
- **Activation Functions** introduce non-linearity into the model, enabling the network to learn complex patterns. ReLU (Rectified Linear Unit) is the most commonly used activation function, improving training speed and mitigating vanishing gradients. Possible function are Tanh or Sigmoid and most commonly used the **ReLu(Rectified Linear Unit)** function. ReLu accelerate the training because the derivative of sigmoid becomes very small in the saturating region and therefore the updates to the weights almost vanish. This is called **vanishing gradient problem**..
- **Pooling Layers** reduces the spatial dimensions (height and width) of the input feature maps by downsampling the input feature maps summarizing the presence of features in patches of the feature map. Max pooling and average pooling are the most common functions.
- **Fully Connected Layers** flatten extracted features and connects to a classifier, typically a softmax layer for classification tasks.
- **Dropout**: reduces the over-fitting by using a Dropout layer after every FC layer. Dropout layer has a probability, p , associated with it and is applied at every neuron of the response map separately. It randomly switches off the activation with the probability p .
- **Batch Normalization** normalizes the inputs of each layer to have a mean of zero and a variance of one, which improve network stability. This normalization is performed for each mini-batch during training.

11.1.2 CNN Architectures: Evolution from LeNet to ResNet

LeNet-5 (1998)

First successful CNN for handwritten digit recognition.

AlexNet (2012)

Revolutionized deep learning by winning the ImageNet competition. Introduced ReLU activation, dropout, and GPU acceleration. Featured Convolutional Layers stacked on top of each other (previously it was common to only have a single CONV layer always immediately followed by a POOL layer).

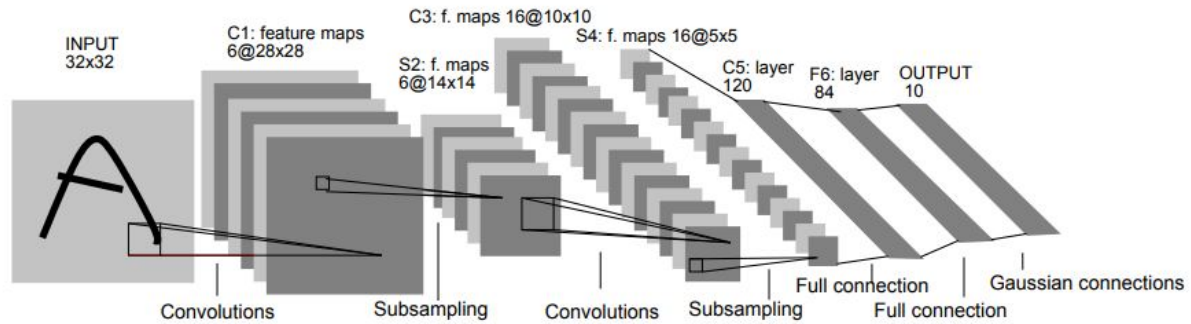
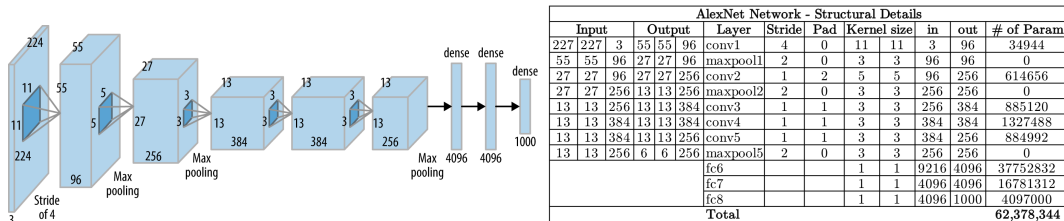
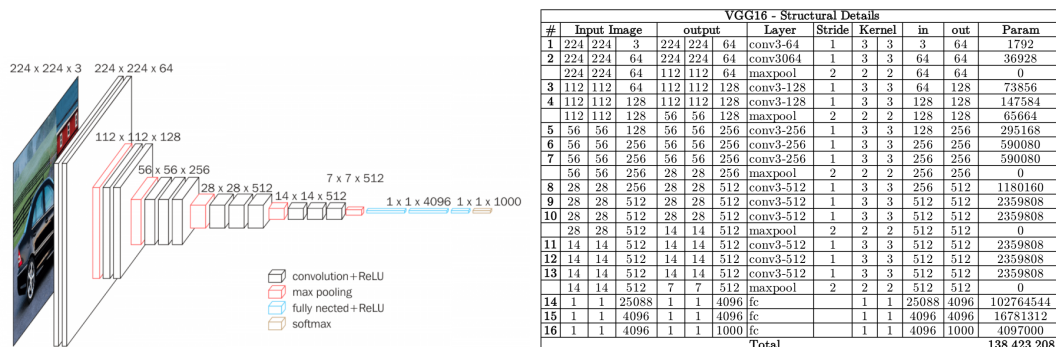


Fig. 2: LeNet



VGG (2014)

Introduced a simple yet deep architecture with 3×3 convolutions.



GoogLeNet (Inception) (2014)

Introduced the **Inception module**, using multiple kernel sizes in parallel.

ResNet (2015)

Introduced **skip connections**, allowing training of very deep networks.

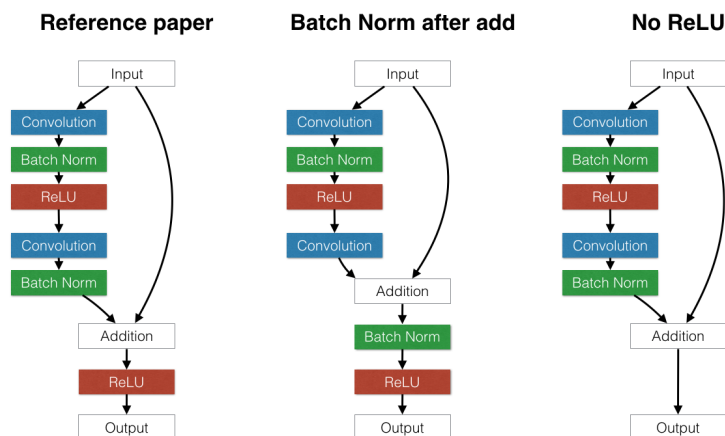


Fig. 3: ResNet block

ResNet18 - Structural Details													
#	Input Image			output			Layer	Stride	Pad	Kernel	in	out	Param
1	227	227	3	112	112	64	conv1	2	1	7 7	3	64	9472
	112	112	64	56	56	64	maxpool	2	0.5	3 3	64	64	0
2	56	56	64	56	56	64	conv2-1	1	1	3 3	64	64	36928
3	56	56	64	56	56	64	conv2-2	1	1	3 3	64	64	36928
4	56	56	64	56	56	64	conv2-3	1	1	3 3	64	64	36928
5	56	56	64	56	56	64	conv2-4	1	1	3 3	64	64	36928
6	56	56	64	28	28	128	conv3-1	2	0.5	3 3	64	128	73856
7	28	28	128	28	28	128	conv3-2	1	1	3 3	128	128	147584
8	28	28	128	28	28	128	conv3-3	1	1	3 3	128	128	147584
9	28	28	128	28	28	128	conv3-4	1	1	3 3	128	128	147584
10	28	28	128	14	14	256	conv4-1	2	0.5	3 3	128	256	295168
11	14	14	256	14	14	256	conv4-2	1	1	3 3	256	256	590080
12	14	14	256	14	14	256	conv4-3	1	1	3 3	256	256	590080
13	14	14	256	14	14	256	conv4-4	1	1	3 3	256	256	590080
14	14	14	256	7	7	512	conv5-1	2	0.5	3 3	256	512	1180160
15	7	7	512	7	7	512	conv5-2	1	1	3 3	512	512	2359808
16	7	7	512	7	7	512	conv5-3	1	1	3 3	512	512	2359808
17	7	7	512	7	7	512	conv5-4	1	1	3 3	512	512	2359808
	7	7	512	1	1	512	avg pool	7	0	7 7	512	512	0
18	1	1	512	1	1	1000	fc				512	1000	513000
							Total						11,511,784

image 32*32

3x3 conv, 64

3x3 conv, 64

3x3 conv, 64

3x3 conv, 64

3x3 conv, 128

3x3 conv, 128

3x3 conv, 128

3x3 conv, 256

3x3 conv, 256

3x3 conv, 512

3x3 conv, 512

3x3 conv, 512

avg pool

fc, 10

32*32

6*16

8*8

4*4

1*1

Architectures general guidelines

- ConvNets stack CONV, POOL, FC layers
- Trend towards smaller filters and deeper architectures: stack 3x3, instead of 5x5
- Trend towards getting rid of POOL/FC layers (just CONV)
- Historically architectures looked like $[(\text{CONV-RELU}) \times N \text{ POOL?}] \times M (\text{FC-RELU}) \times K, \text{SOFTMAX}$ where N is usually up to ~ 5 , M is large, $0 \leq K \leq 2$.
- But recent advances such as ResNet/GoogLeNet have challenged this paradigm

Conclusion and Further Topics

- **Recent architectures:** EfficientNet, Vision Transformers (ViTs), MobileNet for edge devices.
- **Advanced topics:** Transfer learning, object detection (YOLO, Faster R-CNN), segmentation (U-Net).
- **Hands-on implementation:** Implement CNNs using TensorFlow/PyTorch for real-world applications.

11.1.3 Training function

```
%matplotlib inline

import os
import numpy as np
from pathlib import Path

# ML
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import torchvision
import torchvision.transforms as transforms
from torchvision import models

# Device configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
# device = 'cpu' # Force CPU
# print(device)

# Plot
import matplotlib.pyplot as plt
import seaborn as sns

# Plot parameters
plt.style.use('seaborn-v0_8-whitegrid')
fig_w, fig_h = plt.rcParams.get('figure.figsize')
plt.rcParams['figure.figsize'] = (fig_w, fig_h * .5)
%matplotlib inline
```

See `train_val_model` function.

```
from pystatsml.dl_utils import train_val_model
```

11.1.4 CNN in PyTorch

LeNet-5

Here we implement LeNet-5 with relu activation. Sources:

- Github: LeNet-5-PyTorch,
- Kaggle: lenet with pytorch.

```
import torch.nn as nn
import torch.nn.functional as F

class LeNet5(nn.Module):
    """
    layers: (nb channels in input layer,
            nb channels in 1rst conv,
```

(continues on next page)

(continued from previous page)

```

        nb channels in 2nd conv,
        nb neurons for 1rst FC: TO BE TUNED,
        nb neurons for 2nd FC,
        nb neurons for 3rd FC,
        nb neurons output FC TO BE TUNED)

"""
def __init__(self, layers = (1, 6, 16, 1024, 120, 84, 10), debug=False):
    super(LeNet5, self).__init__()
    self.layers = layers
    self.debug = debug
    self.conv1 = nn.Conv2d(layers[0], layers[1], 5, padding=2)
    self.conv2 = nn.Conv2d(layers[1], layers[2], 5)
    self.fc1 = nn.Linear(layers[3], layers[4])
    self.fc2 = nn.Linear(layers[4], layers[5])
    self.fc3 = nn.Linear(layers[5], layers[6])

def forward(self, x):
    x = F.max_pool2d(F.relu(self.conv1(x)), 2) # same shape / 2
    x = F.max_pool2d(F.relu(self.conv2(x)), 2) # -4 / 2
    if self.debug:
        print("### DEBUG: Shape of last convnet=",
              x.shape[1:], ". FC size=", np.prod(x.shape[1:]))
    x = x.view(-1, self.layers[3])
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return F.log_softmax(x, dim=1)

```

VGGNet like: conv-relu blocks

```

# Defining the network (LeNet-5)
import torch.nn as nn
import torch.nn.functional as F

class MiniVGGNet(torch.nn.Module):

    def __init__(self, layers=(1, 16, 32, 1024, 120, 84, 10), debug=False):
        super(MiniVGGNet, self).__init__()
        self.layers = layers
        self.debug = debug

        # Conv block 1
        self.conv11 = nn.Conv2d(in_channels=layers[0], out_channels=layers[1],
                                kernel_size=3, stride=1, padding=0, bias=True)
        self.conv12 = nn.Conv2d(in_channels=layers[1], out_channels=layers[1],
                                kernel_size=3, stride=1, padding=0, bias=True)

        # Conv block 2
        self.conv21 = nn.Conv2d(in_channels=layers[1], out_channels=layers[2],

```

(continues on next page)

(continued from previous page)

```

        kernel_size=3, stride=1, padding=0, bias=True)
self.conv22 = nn.Conv2d(in_channels=layers[2], out_channels=layers[2],
                        kernel_size=3, stride=1, padding=1, bias=True)

# Fully connected layer
self.fc1 = nn.Linear(layers[3], layers[4])
self.fc2 = nn.Linear(layers[4], layers[5])
self.fc3 = nn.Linear(layers[5], layers[6])

def forward(self, x):
    x = F.relu(self.conv11(x))
    x = F.relu(self.conv12(x))
    x = F.max_pool2d(x, 2)

    x = F.relu(self.conv21(x))
    x = F.relu(self.conv22(x))
    x = F.max_pool2d(x, 2)

    if self.debug:
        print("### DEBUG: Shape of last convnet=", x.shape[1:],
              ". FC size=", np.prod(x.shape[1:]))
    x = x.view(-1, self.layers[3])
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)

    return F.log_softmax(x, dim=1)

```

ResNet-like Model

Stack multiple resnet blocks

```

# ----- #
# An implementation of https://arxiv.org/pdf/1512.03385.pdf #
# See section 4.2 for the model architecture on CIFAR-10 #
# Some part of the code was referenced from below #
# https://github.com/pytorch/vision/blob/master/torchvision/models/resnet.py #
# ----- #
import torch.nn as nn

# 3x3 convolution
def conv3x3(in_channels, out_channels, stride=1):
    return nn.Conv2d(in_channels, out_channels, kernel_size=3,
                    stride=stride, padding=1, bias=False)

# Residual block
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super(ResidualBlock, self).__init__()

```

(continues on next page)

(continued from previous page)

```

        self.conv1 = conv3x3(in_channels, out_channels, stride)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(out_channels, out_channels)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = downsample

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out)
        out = self.bn2(out)
        if self.downsample:
            residual = self.downsample(x)
        out += residual
        out = self.relu(out)
        return out

# ResNet
class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes=10):
        super(ResNet, self).__init__()
        self.in_channels = 16
        self.conv = conv3x3(3, 16)
        self.bn = nn.BatchNorm2d(16)
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self.make_layer(block, 16, layers[0])
        self.layer2 = self.make_layer(block, 32, layers[1], 2)
        self.layer3 = self.make_layer(block, 64, layers[2], 2)
        self.avg_pool = nn.AvgPool2d(8)
        self.fc = nn.Linear(64, num_classes)

    def make_layer(self, block, out_channels, blocks, stride=1):
        downsample = None
        if (stride != 1) or (self.in_channels != out_channels):
            downsample = nn.Sequential(
                conv3x3(self.in_channels, out_channels, stride=stride),
                nn.BatchNorm2d(out_channels))
        layers = []
        layers.append(block(self.in_channels, out_channels, stride, downsample))
        self.in_channels = out_channels
        for i in range(1, blocks):
            layers.append(block(out_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        out = self.conv(x)

```

(continues on next page)

(continued from previous page)

```

out = self.bn(out)
out = self.relu(out)
out = self.layer1(out)
out = self.layer2(out)
out = self.layer3(out)
out = self.avg_pool(out)
out = out.view(out.size(0), -1)
out = self.fc(out)
return F.log_softmax(out, dim=1)
#return out

```

ResNet9

Sources:

- DAWNBench on cifar10
- ResNet9: train to 94% CIFAR10 accuracy in 100 seconds

11.1.5 Classification: MNIST digits

MNIST Loader

```

from pystatsml.datasets import load_mnist_pytorch

dataloaders, WD = load_mnist_pytorch(
    batch_size_train=64, batch_size_test=10000)
os.makedirs(os.path.join(WD, "models"), exist_ok=True)

# Info about the dataset
D_in = np.prod(dataloaders["train"].dataset.data.shape[1:])
D_out = len(dataloaders["train"].dataset.targets.unique())
print("Datasets shapes:", {
    x: dataloaders[x].dataset.data.shape for x in ['train', 'test']})
print("N input features:", D_in, "Output classes:", D_out)

```

```
/home/ed203246/data/pystatml/dl_mnist_pytorch
```

NameError

Traceback (most recent call last)

Cell In[12], line 8

```

5 os.makedirs(os.path.join(WD, "models"), exist_ok=True)
7 # Info about the dataset
----> 8 D_in = np.prod(dataloaders["train"].dataset.data.shape[1:])
9 D_out = len(dataloaders["train"].dataset.targets.unique())
10 print("Datasets shapes:", {
11     x: dataloaders[x].dataset.data.shape for x in ['train', 'test']})

```

(continues on next page)

(continued from previous page)

```
NameError: name 'np' is not defined
```

LeNet

Dry run in debug mode to get the shape of the last convnet layer.

```
model = LeNet5((1, 6, 16, 1, 120, 84, 10), debug=True)
batch_idx, (data_example, target_example) = next(
    enumerate(dataloaders["train"]))
print(model)
_ = model(data_example)
```

```
LeNet5(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=1, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
### DEBUG: Shape of last convnet= torch.Size([16, 5, 5]) . FC size= 400
```

Set First FC layer to 400

```
model = LeNet5((1, 6, 16, 400, 120, 84, 10)).to(device)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
criterion = nn.NLLLoss()

# Explore the model
for parameter in model.parameters():
    print(parameter.shape)

print("Total number of parameters =", np.sum([np.prod(parameter.shape) for
                                              parameter in model.parameters()]))

model, losses, accuracies = train_val_model(model, criterion, optimizer,
                                             dataloaders,
                                             num_epochs=5, log_interval=2)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')
```

```
torch.Size([6, 1, 5, 5])
torch.Size([6])
torch.Size([16, 6, 5, 5])
torch.Size([16])
torch.Size([120, 400])
torch.Size([120])
torch.Size([84, 120])
torch.Size([84])
```

(continues on next page)

(continued from previous page)

```

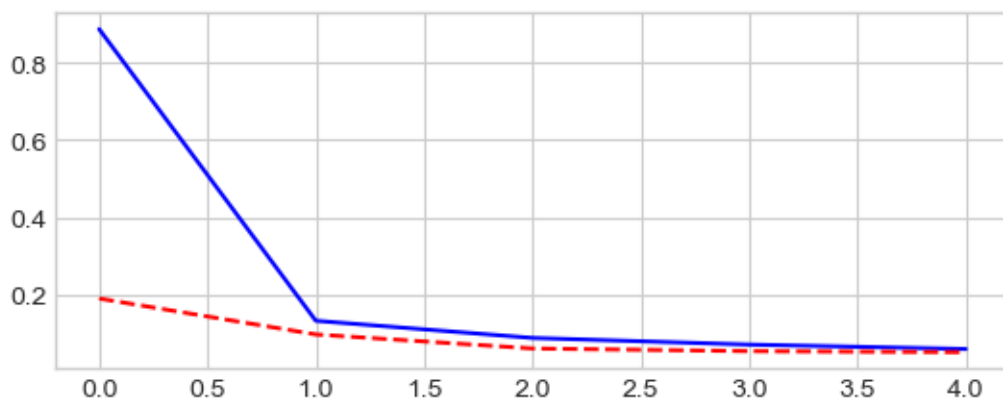
torch.Size([10, 84])
torch.Size([10])
Total number of parameters = 61706
Epoch 0/4
-----
train Loss: 0.8882 Acc: 72.55%
val Loss: 0.1889 Acc: 94.00%

Epoch 2/4
-----
train Loss: 0.0865 Acc: 97.30%
val Loss: 0.0592 Acc: 98.07%

Epoch 4/4
-----
train Loss: 0.0578 Acc: 98.22%
val Loss: 0.0496 Acc: 98.45%

Training complete in 0m 55s
Best val Acc: 98.45%

```



MiniVGGNet

```

model = MiniVGGNet(layers=(1, 16, 32, 1, 120, 84, 10), debug=True)

print(model)
_ = model(data_example)

```

```

MiniVGGNet(
  (conv11): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1))
  (conv12): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1))
  (conv21): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
  (conv22): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (fc1): Linear(in_features=1, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)

```

(continues on next page)

(continued from previous page)

```
)
### DEBUG: Shape of last convnet= torch.Size([32, 5, 5]) . FC size= 800
```

Set First FC layer to 800

```
model = MiniVGGNet((1, 16, 32, 800, 120, 84, 10)).to(device)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
criterion = nn.NLLLoss()

# Explore the model
for parameter in model.parameters():
    print(parameter.shape)

print("Total number of parameters =",
      np.sum([np.prod(parameter.shape)
              for parameter in model.parameters()]))

model, losses, accuracies = train_val_model(model, criterion, optimizer,
                                             dataloaders,
                                             num_epochs=5, log_interval=2)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')
```

```
torch.Size([16, 1, 3, 3])
torch.Size([16])
torch.Size([16, 16, 3, 3])
torch.Size([16])
torch.Size([32, 16, 3, 3])
torch.Size([32])
torch.Size([32, 32, 3, 3])
torch.Size([32])
torch.Size([120, 800])
torch.Size([120])
torch.Size([84, 120])
torch.Size([84])
torch.Size([10, 84])
torch.Size([10])
Total number of parameters = 123502
Epoch 0/4
-----
train Loss: 1.2111 Acc: 58.85%
val Loss: 0.1599 Acc: 94.67%

Epoch 2/4
-----
train Loss: 0.0781 Acc: 97.58%
val Loss: 0.0696 Acc: 97.75%

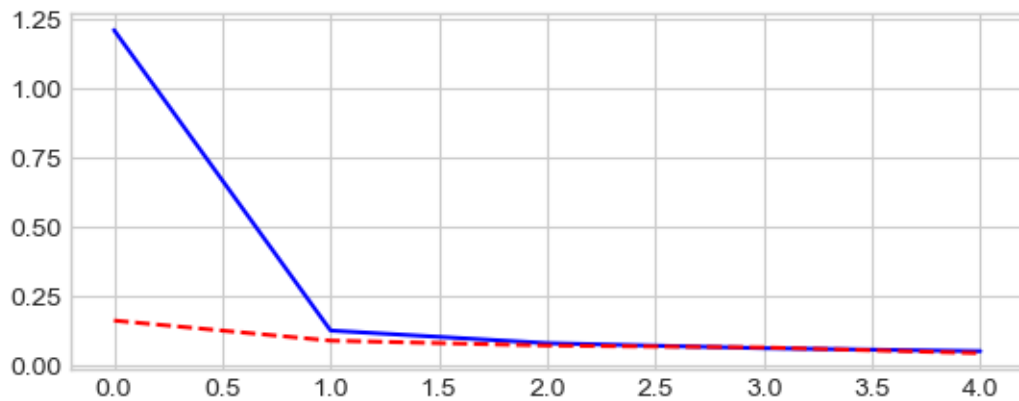
Epoch 4/4
```

(continues on next page)

(continued from previous page)

```
-----
train Loss: 0.0493 Acc: 98.48%
val Loss: 0.0420 Acc: 98.62%
```

```
Training complete in 2m 9s
Best val Acc: 98.62%
```



Reduce the size of training dataset

Reduce the size of the training dataset by considering only 10 minibatches for size16.

```
train_loader, val_loader = dataloaders["train"], dataloaders["test"]

train_size = 10 * 16

# Stratified sub-sampling
targets = train_loader.dataset.targets.numpy()
nclasses = len(set(targets))

indices = np.concatenate([np.random.choice(np.where(targets == lab)[0],
                                           int(train_size / nclasses),
                                           replace=False)
                          for lab in set(targets)])
np.random.shuffle(indices)

train_loader = torch.utils.data.DataLoader(train_loader.dataset, batch_size=16,
                                           sampler=torch.utils.data.SubsetRandomSampler(indices))

# Check train subsampling
train_labels = np.concatenate([labels.numpy()
                              for inputs, labels in train_loader])
print("Train size=", len(train_labels), " Train label count=",
      {lab: np.sum(train_labels == lab) for lab in set(train_labels)})
print("Batch sizes=", [inputs.size(0) for inputs, labels in train_loader])

# Put together train and val
dataloaders = dict(train=train_loader, val=val_loader)
```

(continues on next page)

(continued from previous page)

```
# Info about the dataset
data_shape = dataloaders["train"].dataset.data.shape[1:]
D_in = np.prod(data_shape)
D_out = len(dataloaders["train"].dataset.targets.unique())
print("Datasets shape", {x: dataloaders[x].dataset.data.shape
                        for x in ['train', 'val']})
print("N input features", D_in, "N output", D_out)
```

```
Train size= 160 Train label count= {np.int64(0): np.int64(16), np.int64(1): np.
↪int64(16), np.int64(2): np.int64(16), np.int64(3): np.int64(16), np.int64(4): ↪
↪np.int64(16), np.int64(5): np.int64(16), np.int64(6): np.int64(16), np.
↪int64(7): np.int64(16), np.int64(8): np.int64(16), np.int64(9): np.int64(16)}
Batch sizes= [16, 16, 16, 16, 16, 16, 16, 16]
Datasets shape {'train': torch.Size([60000, 28, 28]), 'val': torch.Size([10000, ↪
↪28, 28])}
N input features 784 N output 10
```

LeNet5

```
model = LeNet5((1, 6, 16, 400, 120, 84, D_out)).to(device)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
criterion = nn.NLLLoss()

model, losses, accuracies = train_val_model(model, criterion, optimizer,
                                           dataloaders,
                                           num_epochs=100, log_interval=20)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')
```

```
Epoch 0/99
-----
train Loss: 2.3072 Acc: 7.50%
val Loss: 2.3001 Acc: 8.89%

Epoch 20/99
-----
train Loss: 0.4810 Acc: 83.75%
val Loss: 0.7552 Acc: 72.66%

Epoch 40/99
-----
train Loss: 0.1285 Acc: 95.62%
val Loss: 0.6663 Acc: 81.72%

Epoch 60/99
-----
train Loss: 0.0065 Acc: 100.00%
val Loss: 0.6982 Acc: 84.26%
```

(continues on next page)

(continued from previous page)

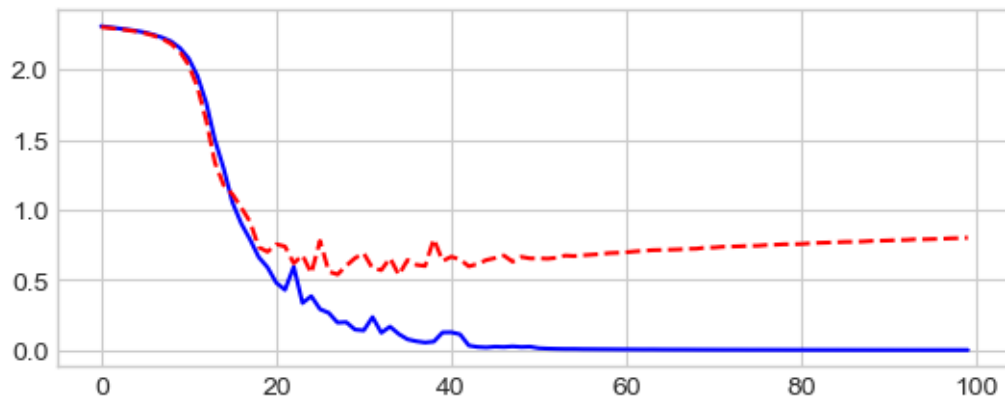
Epoch 80/99

train Loss: 0.0032 Acc: 100.00%

val Loss: 0.7571 Acc: 84.26%

Training complete in 1m 37s

Best val Acc: 84.34%



MiniVGGNet

```

model = MiniVGGNet((1, 16, 32, 800, 120, 84, 10)).to(device)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
criterion = nn.NLLLoss()

model, losses, accuracies = train_val_model(model, criterion, optimizer,
                                             dataloaders,
                                             num_epochs=100, log_interval=20)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')

```

Epoch 0/99

train Loss: 2.3048 Acc: 10.00%

val Loss: 2.3026 Acc: 10.28%

Epoch 20/99

train Loss: 2.2865 Acc: 26.25%

val Loss: 2.2861 Acc: 23.22%

Epoch 40/99

train Loss: 0.3847 Acc: 85.00%

val Loss: 0.8042 Acc: 75.76%

Epoch 60/99

(continues on next page)

(continued from previous page)

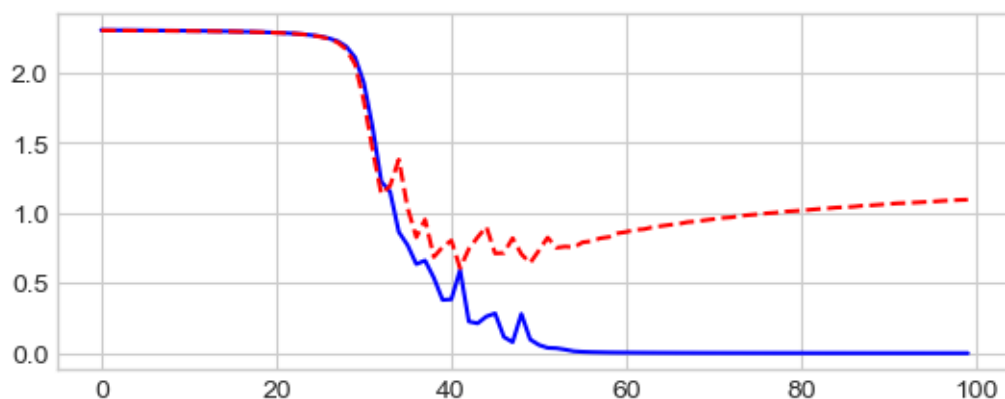
```
-----
train Loss: 0.0047 Acc: 100.00%
val Loss: 0.8659 Acc: 83.57%
```

```
Epoch 80/99
-----
```

```
train Loss: 0.0013 Acc: 100.00%
val Loss: 1.0183 Acc: 83.39%
```

```
Training complete in 4m 39s
```

```
Best val Acc: 84.01%
```



11.1.6 Classification: CIFAR-10 dataset with 10 classes

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class.

Source [Yunjey Choi Github pytorch tutorial](#)

Load CIFAR-10 dataset [CIFAR-10 Loader](#)

```
from pystatsml.datasets import load_cifar10_pytorch

dataloaders, _ = load_cifar10_pytorch(
    batch_size_train=100, batch_size_test=100)

# Info about the dataset
D_in = np.prod(dataloaders["train"].dataset.data.shape[1:])
D_out = len(set(dataloaders["train"].dataset.targets))
print("Datasets shape:", {
    x: dataloaders[x].dataset.data.shape for x in dataloaders.keys()})
print("N input features:", D_in, "N output:", D_out)
```

LeNet

```
model = LeNet5((3, 6, 16, 1, 120, 84, D_out), debug=True)
batch_idx, (data_example, target_example) = next(enumerate(train_loader))
```

(continues on next page)

(continued from previous page)

```
print(model)
_ = model(data_example)
```

```
LeNet5(
  (conv1): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=1, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
### DEBUG: Shape of last convnet= torch.Size([16, 6, 6]) . FC size= 576
```

Set 576 neurons to the first FC layer

SGD with momentum lr=0.001, momentum=0.5

```
model = LeNet5((3, 6, 16, 576, 120, 84, D_out)).to(device)
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.5)
criterion = nn.NLLLoss()

# Explore the model
for parameter in model.parameters():
    print(parameter.shape)

print("Total number of parameters =",
      np.sum([np.prod(parameter.shape)
              for parameter in model.parameters()]))

model, losses, accuracies = train_val_model(model, criterion, optimizer,
                                             dataloaders,
                                             num_epochs=25, log_interval=5)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')
```

```
torch.Size([6, 3, 5, 5])
torch.Size([6])
torch.Size([16, 6, 5, 5])
torch.Size([16])
torch.Size([120, 576])
torch.Size([120])
torch.Size([84, 120])
torch.Size([84])
torch.Size([10, 84])
torch.Size([10])
Total number of parameters = 83126
Epoch 0/24
-----
train Loss: 2.3037 Acc: 10.06%
val Loss: 2.3032 Acc: 10.05%
```

(continues on next page)

(continued from previous page)

Epoch 5/24

train Loss: 2.3005 Acc: 10.72%

val Loss: 2.2998 Acc: 10.61%

Epoch 10/24

train Loss: 2.2931 Acc: 11.90%

val Loss: 2.2903 Acc: 11.27%

Epoch 15/24

train Loss: 2.2355 Acc: 16.46%

val Loss: 2.2134 Acc: 17.75%

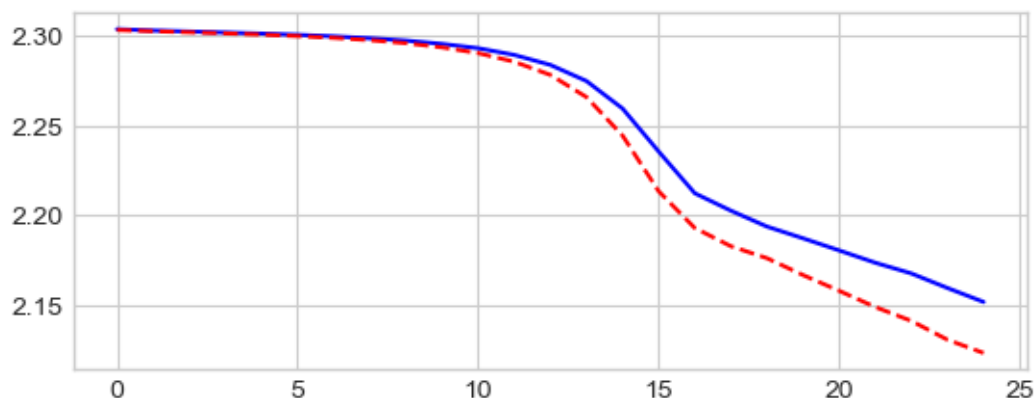
Epoch 20/24

train Loss: 2.1804 Acc: 19.07%

val Loss: 2.1579 Acc: 20.26%

Training complete in 5m 13s

Best val Acc: 23.19%

Increase learning rate and momentum $lr=0.01$, $momentum=0.9$

```

model = LeNet5((3, 6, 16, 576, 120, 84, D_out)).to(device)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
criterion = nn.NLLLoss()

model, losses, accuracies = train_val_model(model, criterion, optimizer,
                                             dataloaders,
                                             num_epochs=25, log_interval=5)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')

```

Epoch 0/24

(continues on next page)

(continued from previous page)

```
train Loss: 2.1798 Acc: 17.53%
val Loss: 1.9141 Acc: 31.27%
```

```
Epoch 5/24
```

```
-----
train Loss: 1.3804 Acc: 49.93%
val Loss: 1.3098 Acc: 53.23%
```

```
Epoch 10/24
```

```
-----
train Loss: 1.2019 Acc: 56.79%
val Loss: 1.0886 Acc: 60.91%
```

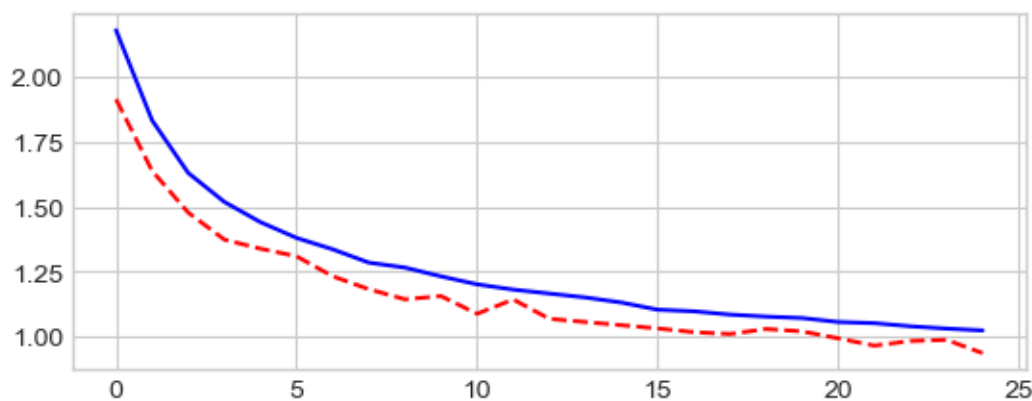
```
Epoch 15/24
```

```
-----
train Loss: 1.1043 Acc: 60.61%
val Loss: 1.0321 Acc: 63.26%
```

```
Epoch 20/24
```

```
-----
train Loss: 1.0569 Acc: 62.31%
val Loss: 0.9942 Acc: 65.55%
```

```
Training complete in 5m 15s
Best val Acc: 67.18%
```



Adaptative learning rate: Adam

```
model = LeNet5((3, 6, 16, 576, 120, 84, D_out)).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
criterion = nn.NLLLoss()

model, losses, accuracies = train_val_model(model, criterion, optimizer,
                                             dataloaders,
                                             num_epochs=25, log_interval=5)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')
```

Epoch 0/24

```
-----
train Loss: 1.8866 Acc: 29.71%
val Loss: 1.6111 Acc: 40.21%
```

Epoch 5/24

```
-----
train Loss: 1.3877 Acc: 49.62%
val Loss: 1.3016 Acc: 53.23%
```

Epoch 10/24

```
-----
train Loss: 1.2274 Acc: 55.93%
val Loss: 1.1575 Acc: 58.78%
```

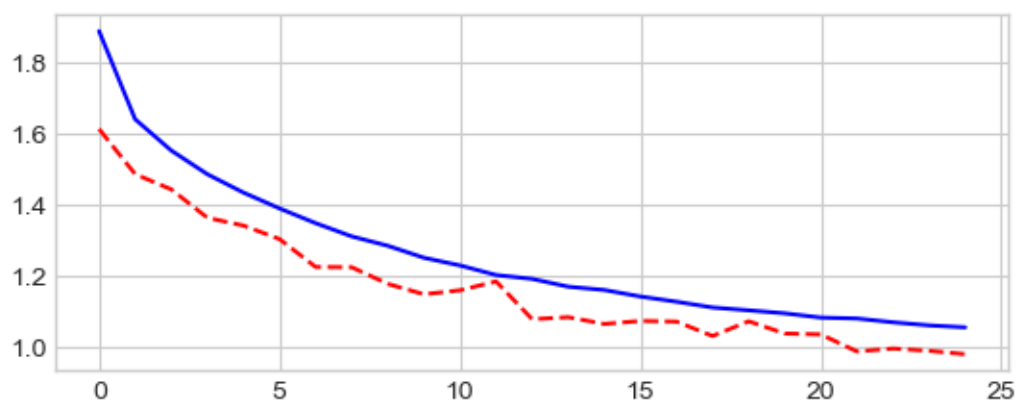
Epoch 15/24

```
-----
train Loss: 1.1399 Acc: 59.28%
val Loss: 1.0712 Acc: 61.84%
```

Epoch 20/24

```
-----
train Loss: 1.0806 Acc: 61.62%
val Loss: 1.0334 Acc: 62.69%
```

```
Training complete in 5m 25s
Best val Acc: 65.14%
```



MiniVGGNet

```
model = MiniVGGNet(layers=(3, 16, 32, 1, 120, 84, D_out), debug=True)
print(model)
_ = model(data_example)
```

```
MiniVGGNet(
  (conv11): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1))
  (conv12): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1))
```

(continues on next page)

(continued from previous page)

```
(conv21): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
(conv22): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(fc1): Linear(in_features=1, out_features=120, bias=True)
(fc2): Linear(in_features=120, out_features=84, bias=True)
(fc3): Linear(in_features=84, out_features=10, bias=True)
)
### DEBUG: Shape of last convnet= torch.Size([32, 6, 6]) . FC size= 1152
```

Set 1152 neurons to the first FC layer

SGD with large momentum and learning rate

```
model = MiniVGGNet((3, 16, 32, 1152, 120, 84, D_out)).to(device)
optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)
criterion = nn.NLLLoss()

model, losses, accuracies = train_val_model(model, criterion, optimizer,
                                             dataloaders,
                                             num_epochs=25, log_interval=5)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')
```

Epoch 0/24

train Loss: 2.2581 Acc: 13.96%
val Loss: 2.0322 Acc: 25.49%

Epoch 5/24

train Loss: 1.4107 Acc: 48.84%
val Loss: 1.3065 Acc: 52.92%

Epoch 10/24

train Loss: 1.0621 Acc: 62.12%
val Loss: 1.0013 Acc: 64.64%

Epoch 15/24

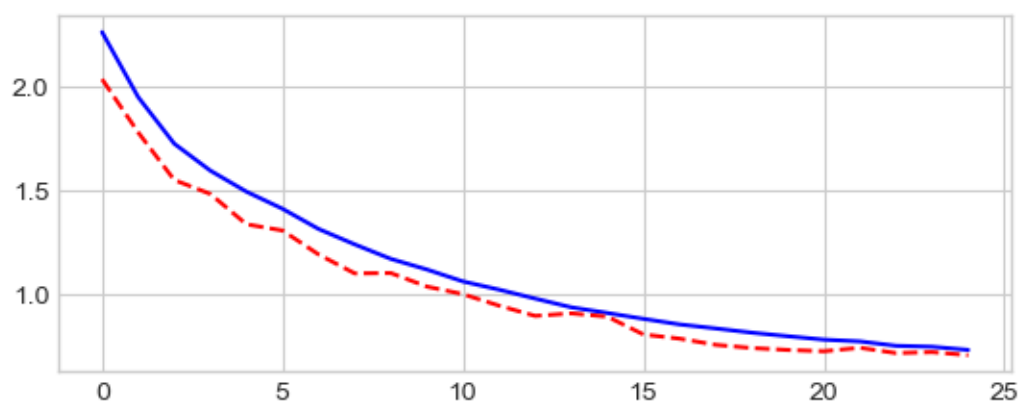
train Loss: 0.8828 Acc: 68.70%
val Loss: 0.8078 Acc: 72.08%

Epoch 20/24

train Loss: 0.7830 Acc: 72.52%
val Loss: 0.7273 Acc: 74.83%

Training complete in 11m 44s

Best val Acc: 75.50%



Adam

```
model = MiniVGNet((3, 16, 32, 1152, 120, 84, D_out)).to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
criterion = nn.NLLLoss()

model, losses, accuracies = \
    train_val_model(model, criterion, optimizer, dataloaders,
                    num_epochs=25, log_interval=5)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')
```

Epoch 0/24

```
-----
train Loss: 1.8556 Acc: 30.40%
val Loss: 1.5847 Acc: 40.66%
```

Epoch 5/24

```
-----
train Loss: 1.2417 Acc: 55.39%
val Loss: 1.0908 Acc: 61.45%
```

Epoch 10/24

```
-----
train Loss: 1.0203 Acc: 63.66%
val Loss: 0.9503 Acc: 66.19%
```

Epoch 15/24

```
-----
train Loss: 0.9051 Acc: 67.98%
val Loss: 0.8536 Acc: 70.10%
```

Epoch 20/24

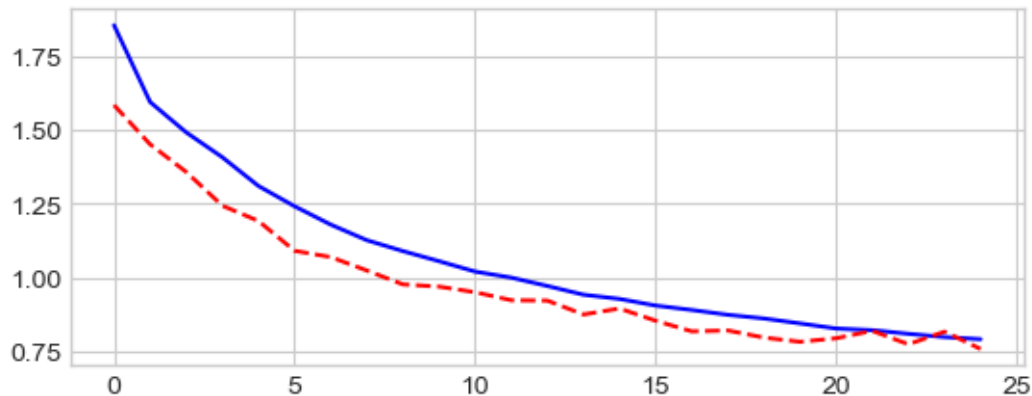
```
-----
train Loss: 0.8273 Acc: 70.74%
val Loss: 0.7942 Acc: 72.55%
```

Training complete in 11m 60s

(continues on next page)

(continued from previous page)

Best val Acc: 74.00%



ResNet

```

model = ResNet(ResidualBlock, [2, 2, 2], num_classes=D_out).to(device)
# 195738 parameters
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
criterion = nn.NLLLoss()

model, losses, accuracies = \
    train_val_model(model, criterion, optimizer, dataloaders,
                    num_epochs=25, log_interval=5)

_ = plt.plot(losses['train'], '-b', losses['val'], '--r')

```

Epoch 0/24

train Loss: 1.4107 Acc: 48.21%

val Loss: 1.2645 Acc: 54.80%

Epoch 5/24

train Loss: 0.6440 Acc: 77.60%

val Loss: 0.8178 Acc: 72.40%

Epoch 10/24

train Loss: 0.4914 Acc: 82.89%

val Loss: 0.6432 Acc: 78.16%

Epoch 15/24

train Loss: 0.4024 Acc: 86.27%

val Loss: 0.5026 Acc: 83.43%

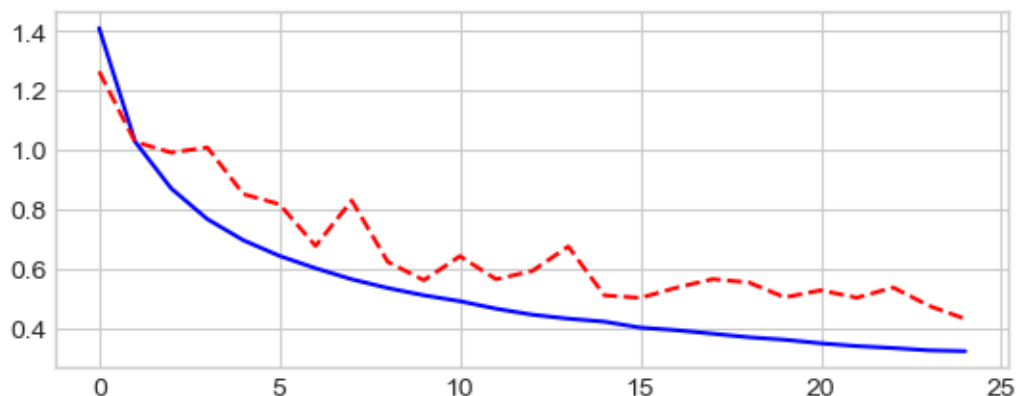
Epoch 20/24

(continues on next page)

(continued from previous page)

```
-----
train Loss: 0.3496 Acc: 87.86%
val Loss: 0.5282 Acc: 82.18%
```

```
Training complete in 58m 9s
Best val Acc: 85.61%
```



11.1.7 Segmentation with U-Net

Source [Segmentation Models](#):

U-Net is a fully convolutional neural network architecture designed for semantic image segmentation. It consists of two main parts:

- An encoder (downsampling path) that extracts increasingly abstract features
- A decoder (upsampling path) that gradually recovers spatial details

The key is the use of skip connections between corresponding encoder and decoder layers. These connections allow the decoder to access fine-grained details from earlier encoder layers, which helps produce more precise segmentation masks.

The skip connections work by concatenating feature maps from the encoder directly into the decoder at corresponding resolutions. This helps preserve important spatial information that would otherwise be lost during the encoding process.

Example: Image Segmentation with U-Net using PyTorch

Below is an example of how to implement image segmentation using the U-Net architecture with PyTorch on a real dataset. We will use the Oxford-IIIT Pet Dataset for this example.

Step1: Load the Dataset

We will use the Oxford-IIIT Pet Dataset, which can be downloaded from [here](#). For simplicity, we will assume the dataset is already downloaded and structured as follows:

Step 2: Define the U-Net Model

Here is the implementation of the U-Net model in PyTorch:

```
import torch
import torch.nn as nn
```

(continues on next page)

(continued from previous page)

```

class UNet(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(UNet, self).__init__()

        def conv_block(in_channels, out_channels):
            return nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
                nn.ReLU(inplace=True),
                nn.Conv2d(out_channels, out_channels,
                        kernel_size=3, padding=1),
                nn.ReLU(inplace=True)
            )

        def up_conv(in_channels, out_channels):
            return nn.ConvTranspose2d(in_channels, out_channels, kernel_size=2,
                                     stride=2)

        self.enc1 = conv_block(in_channels, 64)
        self.enc2 = conv_block(64, 128)
        self.enc3 = conv_block(128, 256)
        self.enc4 = conv_block(256, 512)

        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)

        self.bottleneck = conv_block(512, 1024)

        self.upconv4 = up_conv(1024, 512)
        self.dec4 = conv_block(1024, 512)
        self.upconv3 = up_conv(512, 256)
        self.dec3 = conv_block(512, 256)
        self.upconv2 = up_conv(256, 128)
        self.dec2 = conv_block(256, 128)
        self.upconv1 = up_conv(128, 64)
        self.dec1 = conv_block(128, 64)

        self.conv_final = nn.Conv2d(64, out_channels, kernel_size=1)

    def forward(self, x):
        enc1 = self.enc1(x)
        enc2 = self.enc2(self.pool(enc1))
        enc3 = self.enc3(self.pool(enc2))
        enc4 = self.enc4(self.pool(enc3))

        bottleneck = self.bottleneck(self.pool(enc4))

        dec4 = self.upconv4(bottleneck)
        dec4 = torch.cat((dec4, enc4), dim=1)

```

(continues on next page)

(continued from previous page)

```

dec4 = self.dec4(dec4)
dec3 = self.upconv3(dec4)
dec3 = torch.cat((dec3, enc3), dim=1)
dec3 = self.dec3(dec3)
dec2 = self.upconv2(dec3)
dec2 = torch.cat((dec2, enc2), dim=1)
dec2 = self.dec2(dec2)
dec1 = self.upconv1(dec2)
dec1 = torch.cat((dec1, enc1), dim=1)
dec1 = self.dec1(dec1)

return self.conv_final(dec1)

```

Step 3: Load and Preprocess the Dataset

We use the torchvision library to load and preprocess the dataset:

```

from torchvision import transforms
from torch.utils.data import DataLoader, Dataset
from PIL import Image
import os
import os.path
from pathlib import Path

# Directory
DIR = os.path.join(Path.home(), "data", "pystatml", "dl_Oxford-IIITPet")
# <Directory>/images: input images
# <Directory>/annotations: corresponding masks

class PetDataset(Dataset):
    def __init__(self, image_dir, mask_dir, transform=None):
        self.image_dir = image_dir
        self.mask_dir = mask_dir
        self.transform = transform
        self.image_filenames = os.listdir(image_dir)

    def __len__(self):
        return len(self.image_filenames)

    def __getitem__(self, idx):
        img_path = os.path.join(self.image_dir, self.image_filenames[idx])
        mask_path = os.path.join(self.mask_dir,
                                  self.image_filenames[idx].replace('.jpg', '.png'))
        image = Image.open(img_path).convert('RGB')
        mask = Image.open(mask_path).convert('L')

        if self.transform:
            image = self.transform(image)
            mask = self.transform(mask)

```

(continues on next page)

(continued from previous page)

```

    return image, mask

transform = transforms.Compose([
    transforms.Resize((128, 128)),
    transforms.ToTensor()
])

dataset = PetDataset(os.path.join(DIR, 'images'),
                    os.path.join(DIR, 'annotations'), transform=transform)
dataloader = DataLoader(dataset, batch_size=4, shuffle=True)

```

Step 4: Train the U-Net Model

Finally, we will train the U-Net model:

```

import torch.optim as optim

model = UNet(in_channels=3, out_channels=1)
criterion = nn.BCEWithLogitsLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

def train(model, dataloader, criterion, optimizer, num_epochs=1):
    model.train()
    for epoch in range(num_epochs):
        for images, masks in dataloader:
            optimizer.zero_grad()
            outputs = model(images)
            loss = criterion(outputs, masks)
            loss.backward()
            optimizer.step()
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.4f}')

# Do not executer (takes 2H)
# train(model, dataloader, criterion, optimizer)

```

```

Epoch [1/10], Loss: 0.0414
Epoch [2/10], Loss: 0.0438
Epoch [3/10], Loss: 0.0430
Epoch [4/10], Loss: 0.0402
Epoch [5/10], Loss: 0.0449
Epoch [6/10], Loss: 0.0430
Epoch [7/10], Loss: 0.0438
Epoch [8/10], Loss: 0.0433
Epoch [9/10], Loss: 0.0440
Epoch [10/10], Loss: 0.0453

```

Save the model and reload the model

```
model_dirname = os.path.join(DIR, "models")
model_filename = os.path.join(model_dirname, "unet.pt")
os.makedirs(model_dirname, exist_ok=True)

torch.save(model.state_dict(), model_filename)
model_ = UNet(in_channels=3, out_channels=1)
model_.load_state_dict(torch.load(model_filename, weights_only=True))
_ = model_.eval()
```

Visualize the results

```
# Visualize the results
def visualize_results(model, dataloader, num_images=3):
    model.eval()
    images, masks = next(iter(dataloader))
    with torch.no_grad():
        outputs = model(images)
        outputs = torch.sigmoid(outputs)
        outputs = outputs.cpu().numpy()

    images = images.cpu().numpy()
    masks = masks.cpu().numpy()

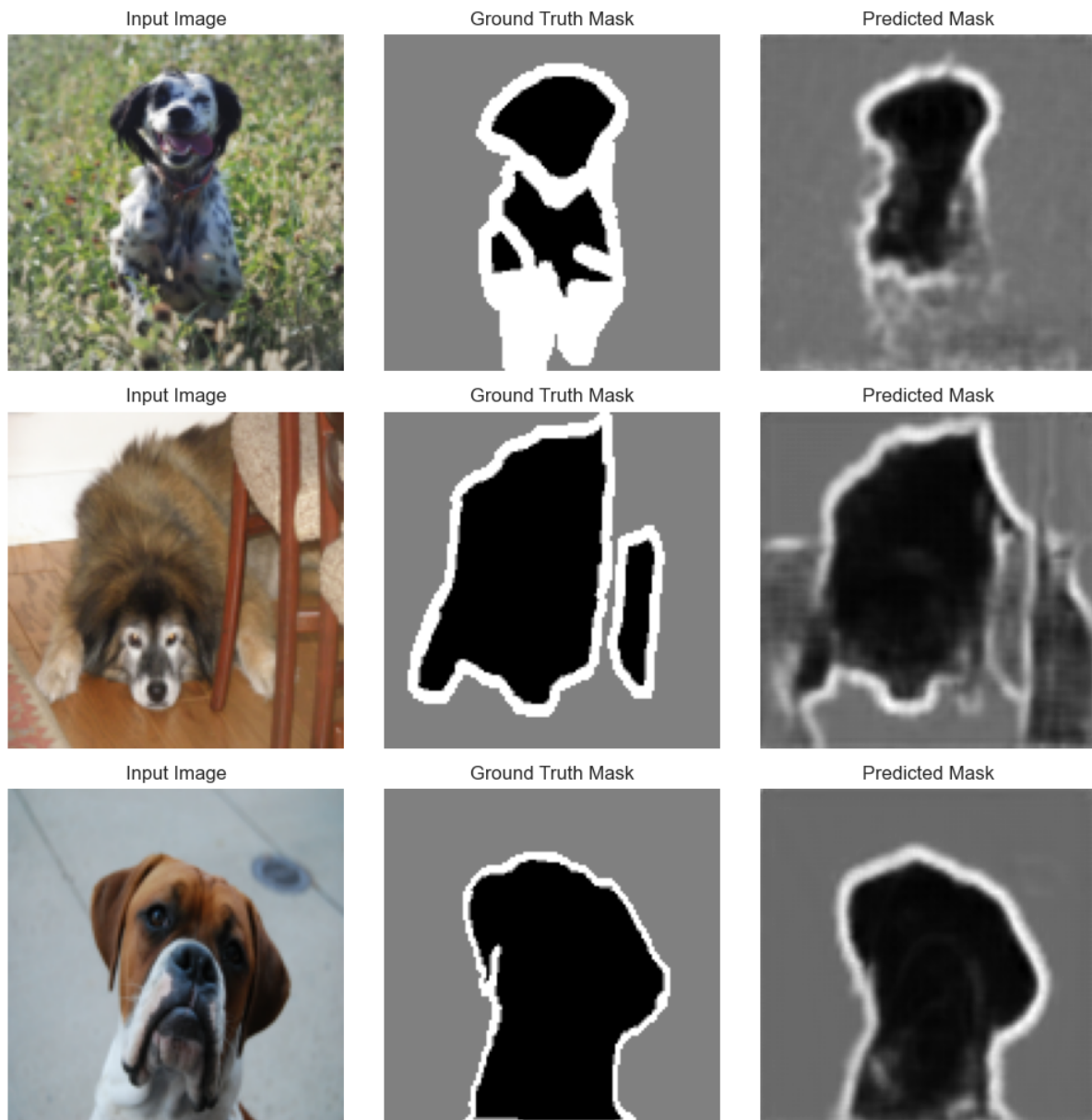
    fig, axes = plt.subplots(num_images, 3, figsize=(10, 10))
    for i in range(num_images):
        axes[i, 0].imshow(np.transpose(images[i], (1, 2, 0)))
        axes[i, 0].set_title('Input Image')
        axes[i, 0].axis('off')

        axes[i, 1].imshow(masks[i].squeeze(), cmap='gray')
        axes[i, 1].set_title('Ground Truth Mask')
        axes[i, 1].axis('off')

        axes[i, 2].imshow(outputs[i].squeeze(), cmap='gray')
        axes[i, 2].set_title('Predicted Mask')
        axes[i, 2].axis('off')

    plt.tight_layout()
    plt.show()

visualize_results(model, dataloader)
```



U-Net: Training Image Segmentation Models in PyTorch

A simple pytorch implementation of U-net

- The model
- Train with predefined dataset and dataloader

PyTorch - Lung Segmentation using pretrained U-net

- UNet architecture with pre-trained ResNet34 from [segmentation_models.pytorch](#) library which has many inbuilt segmentation architectures with different backbones.
- Identify “Pneumothorax” or a collapsed lung from chest x-rays.
- Data Augmentation
- Train-val Dataset and DataLoader

- User defined Loss

11.2 Pretraining and Transfer Learning

Sources [Transfer Learning cs231n @ Stanford](#): *In practice, very few people train an entire Convolutional Network from scratch (with random initialization), because it is relatively rare to have a dataset of sufficient size. Instead, it is common to pretrain a ConvNet on a very large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then use the ConvNet either as an initialization or a fixed feature extractor for the task of interest.*

These two major transfer learning scenarios look as follows:

1. CNN as fixed feature extractor:

- Take a CNN pretrained on ImageNet
- Remove the last fully-connected layer (this layer's outputs are the 1000 class scores for a different task like ImageNet).
- Treat the rest of the CNN as a fixed feature extractor for the new dataset.
- This last fully connected layer is replaced with a new one with random weights and only this layer is trained:
- Freeze the weights for all of the network except that of the final fully connected layer.

2. Fine-tuning all the layers of the CNN:

- Same procedure, but do not freeze the weights of the CNN, by continuing the back-propagation on the new task.

```
from torch.optim import lr_scheduler
import torch.optim as optim
import torch.nn as nn
import torch
import os
import numpy as np

# Plot
import matplotlib.pyplot as plt
import seaborn as sns

# Plot parameters
plt.style.use('seaborn-v0_8-whitegrid')
fig_w, fig_h = plt.rcParams.get('figure.figsize')
plt.rcParams['figure.figsize'] = (fig_w, fig_h * .5)

# Device configuration
device = torch.device('cuda:0' if torch.cuda.is_available() else 'cpu')
# device = 'cpu' # Force CPU
print(device)
```

```
cpu
```

11.2.1 Training function

See `train_val_model` function.

```
from pystatsml.dl_utils import train_val_model
```

11.2.2 Classification: CIFAR-10 dataset with 10 classes

Load CIFAR-10 dataset [CIFAR-10 Loader](#)

```
from pystatsml.datasets import load_cifar10_pytorch

dataloaders, _ = load_cifar10_pytorch(
    batch_size_train=100, batch_size_test=100)

# Info about the dataset
D_in = np.prod(dataloaders["train"].dataset.data.shape[1:])
D_out = len(set(dataloaders["train"].dataset.targets))
print("Datasets shape:", {
    x: dataloaders[x].dataset.data.shape for x in dataloaders.keys()})
print("N input features:", D_in, "N output:", D_out)
```

```
Files already downloaded and verified
Files already downloaded and verified
Datasets shape: {'train': (50000, 32, 32, 3), 'test': (10000, 32, 32, 3)}
N input features: 3072 N output: 10
```

Finetuning the convnet

- Load a pretrained model and reset final fully connected layer.
- SGD optimizer.

```
from torchvision.models import resnet18, ResNet18_Weights

model_ft = resnet18(weights=ResNet18_Weights.DEFAULT)
num_ftrs = model_ft.fc.in_features
# Here the size of each output sample is set to 10.
model_ft.fc = nn.Linear(num_ftrs, D_out)

model_ft = model_ft.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that all parameters are being optimized
optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.001, momentum=0.9)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma=0.1)

model, losses, accuracies = \
    train_val_model(model_ft, criterion, optimizer_ft,
```

(continues on next page)

(continued from previous page)

```

        dataloaders, scheduler=exp_lr_scheduler, num_epochs=5,
        log_interval=5)

epochs = np.arange(len(losses['train']))
_ = plt.plot(epochs, losses['train'], '-b', epochs, losses['test'], '--r')

```

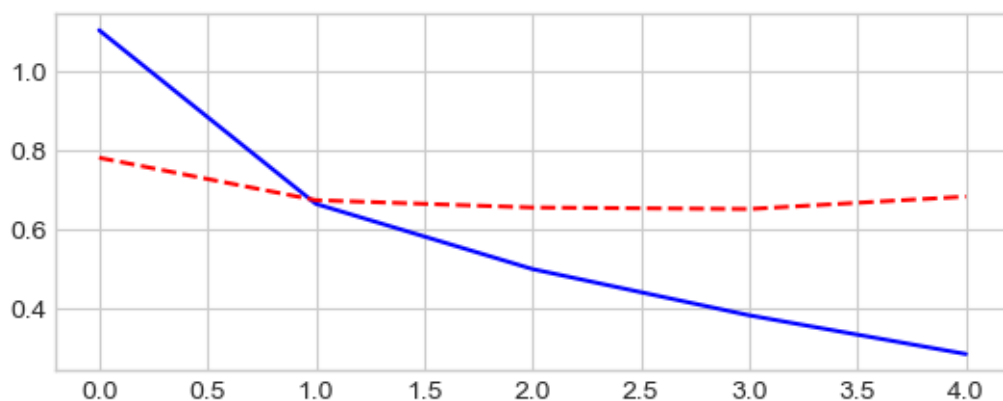
Epoch 0/4

train Loss: 1.1057 Acc: 61.23%

test Loss: 0.7816 Acc: 72.62%

Training complete in 31m 43s

Best val Acc: 78.90%



Adam optimizer

```

model_ft = resnet18(weights=ResNet18_Weights.DEFAULT)
# model_ft = models.resnet18(pretrained=True)
num_ftrs = model_ft.fc.in_features
# Here the size of each output sample is set to 10.
model_ft.fc = nn.Linear(num_ftrs, D_out)

model_ft = model_ft.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that all parameters are being optimized
optimizer_ft = torch.optim.Adam(model_ft.parameters(), lr=0.001)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7, gamma=0.1)

model, losses, accuracies = \
    train_val_model(model_ft, criterion, optimizer_ft,
                    dataloaders, scheduler=exp_lr_scheduler, num_epochs=5,
                    log_interval=5)

```

(continues on next page)

(continued from previous page)

```
epochs = np.arange(len(losses['train']))
_ = plt.plot(epochs, losses['train'], '-b', epochs, losses['test'], '--r')
```

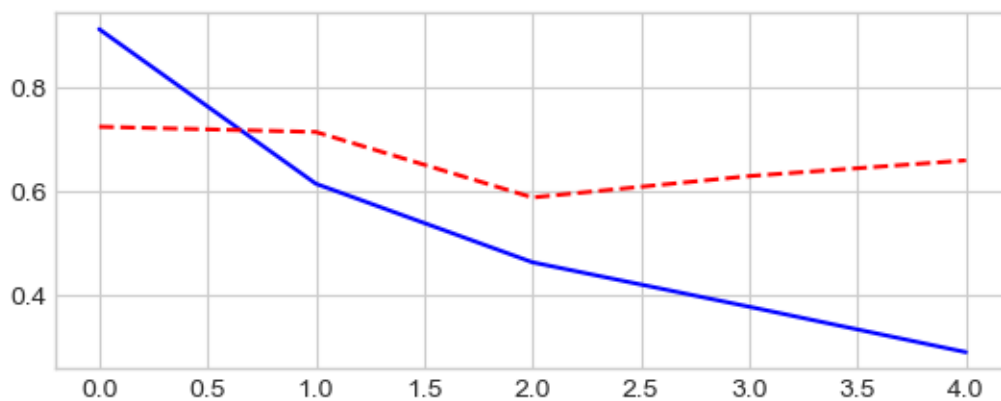
Epoch 0/4

train Loss: 0.9112 Acc: 69.17%

test Loss: 0.7230 Acc: 75.18%

Training complete in 31m 9s

Best val Acc: 80.49%



ResNet as a feature extractor

Freeze all the network except the final layer: `requires_grad == False` to freeze the parameters so that the gradients are not computed in `backward()`.

```
model_conv = resnet18(weights=ResNet18_Weights.DEFAULT)
# model_conv = torchvision.models.resnet18(pretrained=True)
for param in model_conv.parameters():
    param.requires_grad = False

# Parameters of newly constructed modules have requires_grad=True by default
num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, D_out)

model_conv = model_conv.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that only parameters of final layer are being optimized as
# opposed to before.
optimizer_conv = optim.SGD(model_conv.fc.parameters(), lr=0.001, momentum=0.9)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=7, gamma=0.1)
model, losses, accuracies = \
    train_val_model(model_conv, criterion, optimizer_conv,
```

(continues on next page)

(continued from previous page)

```

        dataloaders, scheduler=exp_lr_scheduler, num_epochs=5,
        log_interval=5)

epochs = np.arange(len(losses['train']))
_ = plt.plot(epochs, losses['train'], '-b', epochs, losses['test'], '--r')

```

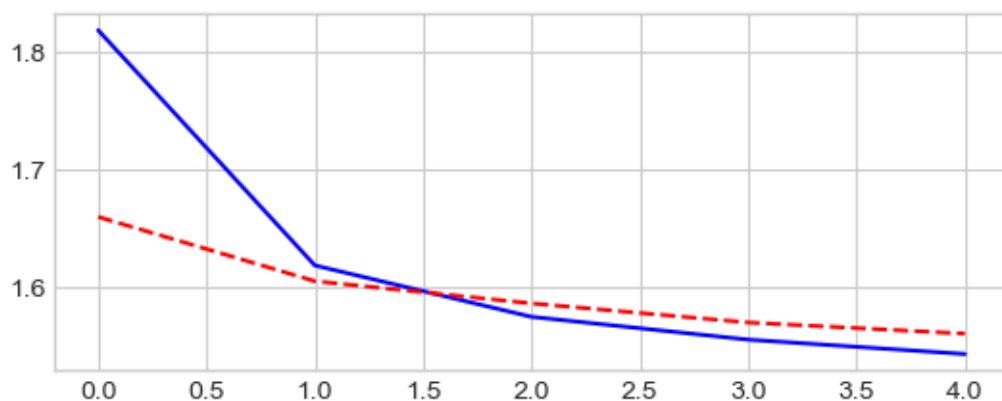
Epoch 0/4

train Loss: 1.8177 Acc: 36.64%

test Loss: 1.6591 Acc: 42.88%

Training complete in 8m 6s

Best val Acc: 46.44%



Adam optimizer

```

model_conv = resnet18(weights=ResNet18_Weights.DEFAULT)
# model_conv = torchvision.models.resnet18(pretrained=True)
for param in model_conv.parameters():
    param.requires_grad = False

# Parameters of newly constructed modules have requires_grad=True by default
num_ftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftrs, D_out)

model_conv = model_conv.to(device)

criterion = nn.CrossEntropyLoss()

# Observe that only parameters of final layer are being optimized as
# opposed to before.
optimizer_conv = optim.Adam(model_conv.fc.parameters(), lr=0.001)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=7, gamma=0.1)

model, losses, accuracies = \

```

(continues on next page)

(continued from previous page)

```
train_val_model(model_conv, criterion, optimizer_conv,
                 dataloaders, scheduler=exp_lr_scheduler, num_epochs=5,
                 log_interval=5)

epochs = np.arange(len(losses['train']))
_ = plt.plot(epochs, losses['train'], '-b', epochs, losses['test'], '--r')
```

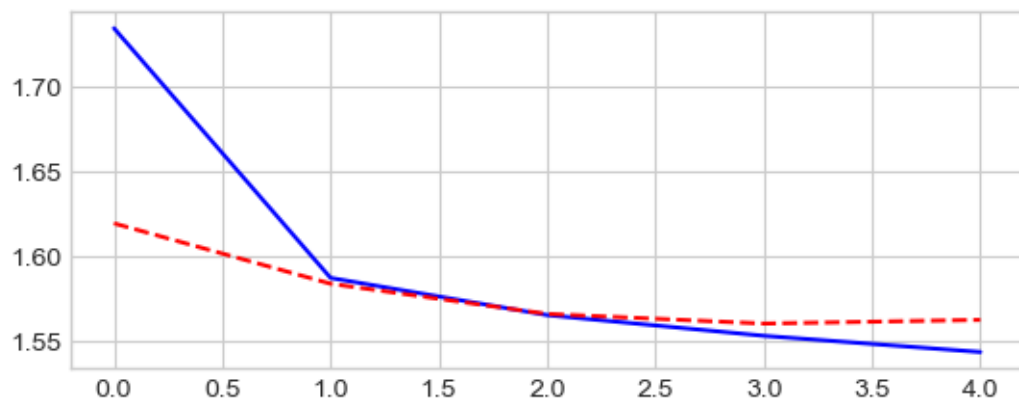
Epoch 0/4

train Loss: 1.7337 Acc: 39.62%

test Loss: 1.6193 Acc: 44.09%

Training complete in 7m 59s

Best val Acc: 46.43%



NATURAL LANGUAGE PROCESSING

12.1 Bag-of-Words Models

Bag-of-Words Model from Wikipedia: The bag-of-words model is a model of text which uses a representation of text that is based on an **unordered collection** (or “bag”) of words. [...] It **disregards word order** [...] but **captures multiplicity**.

12.1.1 Introduction

1. Preparing text data (pre-processing)
 - Standardization: removing irrelevant information, such as punctuation, special characters, lower-upper case, and stopwords.
 - Tokenization (text splitting)
 - Stemming/Lemmatization
2. Encode texts into a numerical vectors (features extraction)
 - Bag of Words Vectorization-based Models: consider phrases as **sets** of words. Words are encoded as vectors independently of the context in which they appear in corpus.
 - Embedding: phrases are **sequences** of words. Words are encoded as vectors integrating their context of appearance in corpus.
3. Predictive analysis
 - Text classification: “What’s the topic of this text?”
 - Content filtering: “Does this text contain abuse?”, spam detection,
 - **Sentiment analysis**: Does this text sound positive or negative?
4. Generate new text
 - Translation
 - Chatbot/summarization

12.1.2 Preparing text data

Standardization and Tokenization

```
# Example usage
```

```
text = """Check out the new http://example.com website! It's awesome.
```

(continues on next page)

(continued from previous page)

```
Hé, it is for programmers that like to program with programming language.
"""
```

The Do It Yourself way

Basic standardization consist of: - Lower case words - Remove numbers - Remove punctuation

```
# import regex
import re

# Convert to lower case
lower_string = text.lower()

# Remove numbers
no_number_string = re.sub(r'\d+', '', lower_string)

# Remove all punctuation except words and space
no_punc_string = re.sub(r'^\w\s', '', no_number_string)

# Remove white spaces
no_wspace_string = no_punc_string.strip()

# Tokenization
print(no_wspace_string.split())
```

NLTK to perform more sophisticated standardization, including:

Basic standardization consist of: - Lower case words - Remove URLs - Remove strip accents - **stop words** are commonly used words that are often removed from text during preprocessing to focus on the more informative words. These words typically include articles, prepositions, conjunctions, and pronouns such as “the,” “is,” “in,” “and,” “but,” “on,” etc. The rationale behind removing stop words is that they occur very frequently in the language and generally do not contribute significant meaning to the analysis or understanding of the text. By eliminating stop words, NLP models can reduce the dimensionality of the data and improve computational efficiency without losing important information.

```
import nltk
import re
import string
import unicodedata
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer, WordNetLemmatizer

# Download necessary NLTK data
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')
nltk.download('omw-1.4')

def strip_accents(text):
```

(continues on next page)

(continued from previous page)

```

# Normalize the text to NFKD form and strip accents
text = unicodedata.normalize('NFKD', text)
text = ''.join([c for c in text if not unicodedata.combining(c)])
return text

def standardize_tokenize(text, stemming=False, lemmatization=False):
    # Convert to lowercase
    text = text.lower()

    # Remove URLs
    text = re.sub(r'http\S+|www\S+|https\S+', '', text, flags=re.MULTILINE)

    # Remove numbers
    text = re.sub(r'\d+', '', text)

    # Remove punctuation
    # string.punctuation provides a string of all punctuation characters.
    # str.maketrans() creates a translation table that maps each punctuation
    # character to None.
    # text.translate(translator) uses this translation table to remove all
    # punctuation characters from the input string.
    text = text.translate(str.maketrans('', '', string.punctuation))

    # Strip accents
    text = strip_accents(text)

    # Tokenize the text
    words = word_tokenize(text)

    # Remove stop words
    stop_words = set(stopwords.words('english'))
    words = [word for word in words if word not in stop_words]

    # Remove repeated words
    words = list(dict.fromkeys(words))

    # Initialize stemmer and lemmatizer
    stemmer = PorterStemmer()
    lemmatizer = WordNetLemmatizer()

    # Apply stemming and lemmatization

    words = [stemmer.stem(word) for word in words] if stemming \
        else words

    words = [lemmatizer.lemmatize(word) for word in words] if lemmatization \
        else words

    return words

```

(continues on next page)

(continued from previous page)

```
# Create callable with default values
import functools
standardize_tokenize_stemming = \
    functools.partial(standardize_tokenize, stemming=True)
standardize_tokenize_lemmatization = \
    functools.partial(standardize_tokenize, lemmatization=True)
standardize_tokenize_stemming_lemmatization = \
    functools.partial(standardize_tokenize, stemming=True, lemmatization=True)
```

```
standardize_tokenize(text)
```

Stemming and lemmatization

Stemming and lemmatization are techniques used to reduce words to their base or root form, which helps in standardizing text and improving the performance of various NLP tasks.

Stemming is the process of reducing a word to its base or root form, often by removing suffixes or prefixes. The resulting stem may not be a valid word but is intended to capture the word's core meaning. Stemming algorithms, such as the Porter Stemmer or Snowball Stemmer, use heuristic rules to chop off common morphological endings from words.

Example: The words “running,” “runner,” and “ran” might all be reduced to “run.”

```
# standardize_tokenize(text, stemming=True)
standardize_tokenize_stemming(text)
```

Lemmatization is the process of reducing a word to its lemma, which is its canonical or dictionary form. Unlike stemming, lemmatization considers the word's part of speech and uses a more comprehensive approach to ensure that the transformed word is a valid word in the language. Lemmatization typically requires more linguistic knowledge and is implemented using libraries like WordNet.

Example: The words “running” and “ran” would both be reduced to “run,” while “better” would be reduced to “good.”

```
# standardize_tokenize(text, lemmatization=True)
standardize_tokenize_lemmatization(text)
```

While both stemming and lemmatization aim to reduce words to a common form, lemmatization is generally more accurate and produces words that are meaningful in the context of the language. However, stemming is faster and simpler to implement. The choice between the two depends on the specific requirements and constraints of the NLP task at hand.

```
# standardize_tokenize(text, stemming=True, lemmatization=True)
standardize_tokenize_stemming_lemmatization(text)
```

Scikit-learn analyzer is simple and will be sufficient most of the time.

```
from sklearn.feature_extraction.text import CountVectorizer
```

(continues on next page)

(continued from previous page)

```
analyzer = CountVectorizer(strip_accents='unicode', stop_words='english').build_
↪ analyzer()
analyzer(text)
```

12.1.3 Bag of Words (BOWs) Encoding

Source: text feature extraction with scikit-learn

Simple Count Vectorization

CountVectorizer: “Convert a collection of text documents to a matrix of token counts. Note that ‘CountVectorizer’ preforms the standardization and the tokenization.”

It creates one feature (column) for each tokens (words) in the corpus, and returns one line per sentence, counting the occurrence of each tokens.

```
corpus = [
    'This is the first document. This DOCUMENT is in english.',
    'in French, some letters have accents, like é.',
    'Is this document in French?',
]

from sklearn.feature_extraction.text import CountVectorizer

vectorizer = CountVectorizer(strip_accents='unicode', stop_words='english')
X = vectorizer.fit_transform(corpus)
print(vectorizer.get_feature_names_out())

# Note that the shape of the array is:
# number of sentences by number of existing token
print(X.toarray())
```

Word n-grams are contiguous sequences of ‘n’ words from a given text. They are used to **capture the context** and structure of language by considering the relationships between words within these sequences. The value of ‘n’ determines the length of the word sequence:

- Unigram (1-gram): A single word (e.g., “natural”).
- Bigram (2-gram): A sequence of two words (e.g., “natural language”).
- Trigram (3-gram): A sequence of three words (e.g., “natural language processing”).

```
vectorizer2 = CountVectorizer(analyzer='word', ngram_range=(2, 2),
                             strip_accents='unicode', stop_words='english')
X2 = vectorizer2.fit_transform(corpus)
print(vectorizer2.get_feature_names_out())
print(X2.toarray())
```

TF-IDF Vectorization approach:

TF-IDF (Term Frequency-Inverse Document Frequency) feature extraction:

“TF-IDF (Term Frequency-Inverse Document Frequency) integrates two metrics: Term Frequency (TF) and Inverse Document Frequency (IDF). This method is employed when working with multiple documents, operating on the principle that rare words provide more insight into a document’s content than frequently occurring words across the entire document set.”

“A challenge with relying solely on word frequency is that commonly used words may overshadow the document, despite offering less”informational content” compared to rarer, potentially domain-specific terms. To address this, one can adjust the frequency of words by considering their prevalence across all documents, thereby reducing the scores of frequently used words that are common across the corpus.”

Term Frequency: Provide large weight to frequent words. Given a token t (term, word), a document d

$$TF(t, d) = \frac{\text{number of times } t \text{ appears in } d}{\text{total number of term in } d}$$

Inverse Document Frequency: Give more importance to rare “meaningfull” words a appear in few documents.

If N is the total number of documents, and df is the number of documents with token t , then:

$$IDF(t) = \frac{N}{1 + df}$$

$IDF(t) \approx 1$ if t appears in all documents, while $IDF(t) \approx N$ if t is a rare meaningful word that appears in only one document.

Finally:

$$TF-IDF(t, d) = TF(t, d) * IDF(t)$$

TfidfVectorizer:

Convert a collection of raw documents to a matrix of TF-IDF (Term Frequency-Inverse Document Frequency)

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer(strip_accents='unicode', stop_words='english')
X = vectorizer.fit_transform(corpus)
print(vectorizer.get_feature_names_out())
print(X.toarray().round(3))
print(X.shape)
```

Lab 1: Sentiment Analysis of Financial data

Sources: [Sentiment Analysis of Financial data](#)

The data is intended for advancing financial sentiment analysis research. It’s two datasets (FiQA, Financial PhraseBank) combined into one easy-to-use CSV file. It provides financial sentences with sentiment labels. Citations Malo, Pekka, et al. “Good debt or bad debt: Detecting semantic orientations in economic texts.” *Journal of the Association for Information Science and Technology* 65.4 (2014): 782-796.

Import libraries


```

import numpy as np
import pandas as pd

# Plot
import matplotlib.pyplot as plt
%matplotlib inline
from wordcloud import WordCloud

# ML
from sklearn import metrics
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import GradientBoostingClassifier

from sklearn.feature_extraction.text import CountVectorizer

```

Load the Dataset

```

data = pd.read_csv('../datasets/FinancialSentimentAnalysis.csv')

print("Shape:", data.shape, "columns:", data.columns)
print(data.describe())
data.head()

```

Target variable

```

y = data['Sentiment']
y.value_counts(), y.value_counts(normalize=True).round(2)

```

Input data: BOWs encoding

Choose tokenizer

```

text = 'Tesla to recall 2,700 Model X SUVs over seat issue https://t.co/
↪OdPraN59Xq $TSLA https://t.co/xvn4b1Iwpy https://t.co/ThfvWTnRPs'
vectorizer = CountVectorizer(stop_words='english', strip_accents='unicode')

tokenizer_sklearn = vectorizer.build_analyzer()
print(" ".join(tokenizer_sklearn(text)))
print("Shape: ", CountVectorizer(tokenizer=tokenizer_sklearn).fit_transform(data[
↪ 'Sentence']).shape)

print(" ".join(standardize_tokenize(text)))
print("Shape: ", CountVectorizer(tokenizer=standardize_tokenize).fit_
↪ transform(data['Sentence']).shape)

print(" ".join(standardize_tokenize_stemming(text)))
print("Shape: ", CountVectorizer(tokenizer=standardize_tokenize_stemming).fit_
↪ transform(data['Sentence']).shape)

print(" ".join(standardize_tokenize_lemmatization(text)))

```

(continues on next page)

(continued from previous page)

```
print("Shape: ", CountVectorizer(tokenizer=standardize_tokenize_lemmatization).
    ↪ fit_transform(data['Sentence']).shape)

print(" ".join(standardize_tokenize_stemming_lemmatization(text)))
print("Shape: ", CountVectorizer(tokenizer=standardize_tokenize_stemming_
    ↪ lemmatization).fit_transform(data['Sentence']).shape)
```

```
# vectorizer = CountVectorizer(stop_words='english', strip_accents='unicode')
# vectorizer = CountVectorizer(tokenizer=standardize_tokenize)
# vectorizer = CountVectorizer(tokenizer=standardize_tokenize_stemming)
# vectorizer = CountVectorizer(tokenizer=standardize_tokenize_lemmatization)
vectorizer = CountVectorizer(tokenizer=standardize_tokenize_stemming_
    ↪ lemmatization)
# vectorizer = TfidfVectorizer(stop_words='english', strip_accents='unicode')
# vectorizer = TfidfVectorizer(tokenizer=standardize_tokenize_stemming_
    ↪ lemmatization)

# Retrieve the analyzer to store transformed sentences in dataframe
tokenizer = vectorizer.build_analyzer()
data['Sentence_stdz'] = [" ".join(tokenizer(s)) for s in data['Sentence']]

X = vectorizer.fit_transform(data['Sentence'])
# print("Tokens:", vectorizer.get_feature_names_out())
print("Nb of tokens:", len(vectorizer.get_feature_names_out()))
print("Dimension of input data", X.shape)
```

Classification with scikit-learn models

```
# clf = LogisticRegression(class_weight='balanced', max_iter=3000)
# clf = GradientBoostingClassifier()
clf = MultinomialNB()

from sklearn.model_selection import train_test_split
idx = np.arange(y.shape[0])
X_train, X_test, x_str_train, x_str_test, y_train, y_test, idx_train, idx_test = \
    train_test_split(X, data['Sentence'], y, idx, test_size=0.25, random_state=5,
    ↪ stratify=y)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
```

Display prediction performances

```
print(metrics.balanced_accuracy_score(y_test, y_pred))
print(metrics.classification_report(y_test, y_pred))
cm = metrics.confusion_matrix(y_test, y_pred, normalize='true')
cm_ = metrics.ConfusionMatrixDisplay(cm, display_labels=clf.classes_)

cm_.plot()
plt.show()
```

Print some samples

```

probas = pd.DataFrame(clf.predict_proba(X), columns=clf.classes_)
df = pd.concat([data, probas], axis=1)
df['SentimentPred'] = clf.predict(X)

df.to_excel("/tmp/test.xlsx")

# Keep only test data, correctly classified, ordered by
df = df.iloc[idx_test]
df = df[df['SentimentPred'] == df['Sentiment']]

```

Positive sentences

```

sentence_positive = df[df['Sentiment'] == 'positive'].sort_values(by='positive',
↪ascending=False)['Sentence_stdz']
print("Most positive sentence", sentence_positive[:5])

plt.figure(figsize = (20,20))
wc = WordCloud(max_words = 1000 , width = 1600 , height = 800,
collocations=False).generate(" ".join(sentence_positive))
plt.imshow(wc)

```

Negative sentences

```

sentence_negative = df[df['Sentiment'] == 'negative'].sort_values(by='negative',
↪ascending=False)['Sentence_stdz']
print("Most negative sentence", sentence_negative[:5])

plt.figure(figsize = (20,20))
wc = WordCloud(max_words = 1000 , width = 1600 , height = 800,
collocations=False).generate(" ".join(sentence_negative))
plt.imshow(wc)

```

Lab 2: Twitter Sentiment Analysis

- Source [Twitter Sentiment Analysis Using Python | Introduction & Techniques](#)
- Dataset [Sentiment140 dataset with 1.6 million twe](#)

Step-1: Import the Necessary Dependencies

Install some packages:

```

conda install wordcloud
conda install nltk

```

```

# utilities
import re
import numpy as np
import pandas as pd
# plotting
import seaborn as sns

```

(continues on next page)

(continued from previous page)

```

from wordcloud import WordCloud
import matplotlib.pyplot as plt
# nltk
from nltk.stem import WordNetLemmatizer
# sklearn
from sklearn.svm import LinearSVC
from sklearn.naive_bayes import BernoulliNB
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import confusion_matrix, classification_report

```

Step-2: Read and Load the Dataset

Download the dataset from Kaggle

```

# Importing the dataset
DATASET_COLUMNS=['target','ids','date','flag','user','text']
DATASET_ENCODING = "ISO-8859-1"
df = pd.read_csv('~/.data/NLP/training.1600000.processed.noemoticon.csv',
                  encoding=DATASET_ENCODING, names=DATASET_COLUMNS)
df.sample(5)

```

Step-3: Exploratory Data Analysis

```

print("Columns names:", df.columns)
print("Shape of data:", df.shape)
print("type of data:\n", df.dtypes)
df.head()

```

Step-4: Data Visualization of Target Variables

- Selecting the text and Target column for our further analysis
- Replacing the values to ease understanding. (Assigning 1 to Positive sentiment 4)

```

data = df[['text','target']]
data['target'] = data['target'].replace(4,1)
print(data['target'].unique())

import seaborn as sns
sns.countplot(x='target', data=data)

print("Count and proportion of target")
data.target.value_counts(), data.target.value_counts(normalize=True).round(2)

```

Step-5: Data Preprocessing

5.4: Separating positive and negative tweets 5.5: Taking 20000 positive and negatives sample from the data so we can run it on our machine easily 5.6: Combining positive and negative tweets

```
data_pos = data[data['target'] == 1]
data_neg = data[data['target'] == 0]
data_pos = data_pos.iloc[:20000]
data_neg = data_neg.iloc[:20000]
dataset = pd.concat([data_pos, data_neg])
```

5.7: Text pre-processing

```
def standardize_stemming_lemmatization(text):
    out = " ".join(standardize_tokenize_stemming_lemmatization(text))
    return out

dataset['text_stdz'] = dataset['text'].apply(lambda x: standardize_stemming_
↪ lemmatization(x))
```

QC, check for empty standardized strings

```
rm = dataset['text_stdz'].isnull() | (dataset['text_stdz'].str.len() == 0)

print(rm.sum(), "row are empty of null, to be removed")
dataset = dataset[~rm]
print(dataset.shape)

# Save dataset to excel file to explore
dataset.to_excel('/tmp/test.xlsx', sheet_name='data', index=False)
```

5.18: Plot a cloud of words for negative tweets

```
data_neg = dataset.loc[dataset.target == 0, 'text_stdz']
plt.figure(figsize = (20,20))
wc = WordCloud(max_words = 1000 , width = 1600 , height = 800,
               collocations=False).generate(" ".join(data_neg))
plt.imshow(wc)
```

5.18: Plot a cloud of words for positive tweets

```
data_pos = dataset.loc[dataset.target == 1, 'text_stdz']
plt.figure(figsize = (20,20))
wc = WordCloud(max_words = 1000 , width = 1600 , height = 800,
               collocations=False).generate(" ".join(data_pos))
plt.imshow(wc)
```

Step-6: Splitting Our Data Into Train and Test Subsets

```
X, y = dataset.text_stdz, dataset.target
# Separating the 95% data for training data and 5% for testing data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.05, random_
↪ state=26105111)
```

Step-7: Transforming the Dataset Using TF-IDF Vectorizer

```
vectoriser = TfidfVectorizer(ngram_range=(1,2), max_features=500000)
vectoriser.fit(X_train)
#print('No. of feature_words: ', len(vectoriser.get_feature_names()))

X_train = vectoriser.transform(X_train)
X_test = vectoriser.transform(X_test)
```

Step-8: Function for Model Evaluation

After training the model, we then apply the evaluation measures to check how the model is performing. Accordingly, we use the following evaluation parameters to check the performance of the models respectively:

- Accuracy Score
- Confusion Matrix with Plot
- ROC-AUC Curve

```
def model_Evaluate(model):
    # Predict values for Test dataset
    y_pred = model.predict(X_test)
    # Print the evaluation metrics for the dataset.
    print(classification_report(y_test, y_pred))
    # Compute and plot the Confusion matrix
    cf_matrix = confusion_matrix(y_test, y_pred)
    categories = ['Negative', 'Positive']
    group_names = ['True Neg', 'False Pos', 'False Neg', 'True Pos']
    group_percentages = ['{0:.2%}'.format(value) for value in cf_matrix.flatten()
    ↪ / np.sum(cf_matrix)]
    labels = [f'{v1}n{v2}' for v1, v2 in zip(group_names, group_percentages)]
    labels = np.asarray(labels).reshape(2,2)
    sns.heatmap(cf_matrix, annot = labels, cmap = 'Blues', fmt = '',
    xticklabels = categories, yticklabels = categories)
    plt.xlabel("Predicted values", fontdict = {'size':14}, labelpad = 10)
    plt.ylabel("Actual values", fontdict = {'size':14}, labelpad = 10)
    plt.title ("Confusion Matrix", fontdict = {'size':18}, pad = 20)
```

Step-9: Model Building

In the problem statement, we have used three different models respectively :

- Bernoulli Naive Bayes Classifier
- SVM (Support Vector Machine)
- Logistic Regression

The idea behind choosing these models is that we want to try all the classifiers on the dataset ranging from simple ones to complex models, and then try to find out the one which gives the best performance among them.

```
BNBmodel = BernoulliNB()
BNBmodel.fit(X_train, y_train)
```

(continues on next page)

(continued from previous page)

```
model_Evaluate(BNBmodel)
y_pred1 = BNBmodel.predict(X_test)
```

8.2: Plot the ROC-AUC Curve for model-1

```
from sklearn.metrics import roc_curve, auc
fpr, tpr, thresholds = roc_curve(y_test, y_pred1)
roc_auc = auc(fpr, tpr)
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=1, label='ROC curve (area = %0.2f)' %
    ↪roc_auc)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC CURVE')
plt.legend(loc="lower right")
plt.show()
```


MATHEMATICAL DETAILS AND DEMONSTRATIONS

13.1 Negative Log-Likelihood (NLL) for Binary Classification with Sigmoid Activation

13.1.1 Demonstration of Negative Log-Likelihood (NLL)

Setup

- Inputs: $\{(x_i, y_i)\}_{i=1}^n$, with $y_i \in \{0, 1\}$
- Model:

$$\hat{p}_i = \sigma(\mathbf{w}^\top \mathbf{x}_i) = \frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}_i}}$$

- Objective:

$$\mathcal{L}_{\text{NLL}} = - \sum_{i=1}^n \log P(y_i \mid \mathbf{x}_i; \mathbf{w})$$

Since $y_i \in \{0, 1\}$, we model the likelihood as:

$$P(y_i \mid \mathbf{x}_i; \mathbf{w}) = \hat{p}_i^{y_i} (1 - \hat{p}_i)^{1-y_i}$$

Step-by-step Expansion

$$\mathcal{L}_{\text{NLL}} = - \sum_{i=1}^n \log (\hat{p}_i^{y_i} (1 - \hat{p}_i)^{1-y_i})$$

Apply log properties:

$$= - \sum_{i=1}^n [y_i \log(\hat{p}_i) + (1 - y_i) \log(1 - \hat{p}_i)]$$

Now substitute $\hat{p}_i = \sigma(z_i) = \frac{1}{1+e^{-z_i}}$ where $z_i = \mathbf{w}^\top \mathbf{x}_i$:

- Use:

$$\log(\sigma(z)) = -\log(1 + e^{-z}), \quad \log(1 - \sigma(z)) = -z - \log(1 + e^{-z})$$

So the per-example loss becomes:

$$\ell_i(\mathbf{w}) = - [y_i \log \sigma(z_i) + (1 - y_i) \log(1 - \sigma(z_i))]$$

$$= - [y_i(-\log(1 + e^{-z_i})) + (1 - y_i)(-z_i - \log(1 + e^{-z_i}))]$$

Simplify:

$$\ell_i(\mathbf{w}) = \log(1 + e^{-z_i}) + (1 - y_i)z_i$$

Therefore, the total loss over n examples is:

Final Simplified Expression

$$\mathcal{L}_{\text{NLL}}(\mathbf{w}) = \sum_{i=1}^n [\log(1 + e^{-z_i}) + (1 - y_i)z_i] \quad \text{with } y_i \in \{0, 1\},$$

simplifies to

$$\mathcal{L}_{\text{NLL}}(\mathbf{w}) = \sum_{i=1}^n \log(1 + e^{-y_i z_i}) \quad \text{with } y_i \in \{-1, +1\}.$$

This final form is particularly elegant and often used in optimization routines.

13.1.2 Gradient of Negative Log-Likelihood (NLL)

Recap: The Model and Loss

We have:

- Input-label pairs: $\{(x_i, y_i)\}_{i=1}^n$, where $y_i \in \{0, 1\}$
- Linear logit: $z_i = \mathbf{w}^\top \mathbf{x}_i$
- Sigmoid output:

$$\hat{p}_i = \sigma(z_i) = \frac{1}{1 + e^{-z_i}}$$

- NLL loss:

$$\mathcal{L}(\mathbf{w}) = - \sum_{i=1}^n [y_i \log(\hat{p}_i) + (1 - y_i) \log(1 - \hat{p}_i)]$$

We aim to compute the gradient $\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w})$

Step 1: Loss per Sample

Define per-sample loss:

$$\ell_i(\mathbf{w}) = - [y_i \log(\hat{p}_i) + (1 - y_i) \log(1 - \hat{p}_i)]$$

Take derivative w.r.t. $\mathbf{w}$. Using the chain rule:

$$\nabla_{\mathbf{w}} \ell_i = \frac{d\ell_i}{d\hat{p}_i} \cdot \frac{d\hat{p}_i}{dz_i} \cdot \frac{dz_i}{d\mathbf{w}}$$

Step 2: Compute Gradients

- Derivative of the loss w.r.t. $\hat{p}_i$:

$$\frac{d\ell_i}{d\hat{p}_i} = - \left(\frac{y_i}{\hat{p}_i} - \frac{1 - y_i}{1 - \hat{p}_i} \right)$$

- Derivative of sigmoid:

$$\frac{d\hat{p}_i}{dz_i} = \hat{p}_i(1 - \hat{p}_i)$$

- Derivative of $z_i = \mathbf{w}^\top \mathbf{x}_i$:

$$\frac{dz_i}{d\mathbf{w}} = \mathbf{x}_i$$

Putting it all together:

$$\nabla_{\mathbf{w}} \ell_i = [\hat{p}_i - y_i] \mathbf{x}_i$$

Step 3: Final Gradient over Dataset

Sum over all samples:

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \sum_{i=1}^n (\hat{p}_i - y_i) \mathbf{x}_i$$

Or in matrix form, if $\mathbf{X} \in \mathbb{R}^{n \times p}$, $\hat{\mathbf{p}} \in \mathbb{R}^n$, and $\mathbf{y} \in \mathbb{R}^n$:

$$\nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}) = \mathbf{X}^\top (\hat{\mathbf{p}} - \mathbf{y})$$

Summary

- Gradient of binary NLL with sigmoid:

$$\nabla_{\mathbf{w}} \mathcal{L} = \sum_{i=1}^n (\sigma(\mathbf{w}^\top \mathbf{x}_i) - y_i) \mathbf{x}_i$$

- In matrix form:

$$\nabla_{\mathbf{w}} \mathcal{L} = \mathbf{X}^\top (\hat{\mathbf{p}} - \mathbf{y})$$

This form is used in logistic regression and binary classifiers trained via gradient descent.

13.1.3 Hessian matrix (i.e., the matrix of second derivatives) for Negative Log-Likelihood (NLL)

Recap: Setup

We are given: - Dataset: $\{(x_i, y_i)\}_{i=1}^n$, with $y_i \in \{0, 1\}$ - Model:

$$\hat{p}_i = \sigma(z_i) = \frac{1}{1 + e^{-z_i}}, \quad \text{where } z_i = \mathbf{w}^\top \mathbf{x}_i$$

- Loss function:

$$\mathcal{L}(\mathbf{w}) = - \sum_{i=1}^n [y_i \log(\hat{p}_i) + (1 - y_i) \log(1 - \hat{p}_i)]$$

We already know the gradient is:

$$\nabla_{\mathbf{w}} \mathcal{L} = \sum_{i=1}^n (\hat{p}_i - y_i) \mathbf{x}_i$$

Goal: Hessian :math:`\nabla^2\_{\mathbf{w}} \mathcal{L}`

We now compute the second derivative of $\mathcal{L}$, i.e., the **Hessian matrix** $\mathbf{H} \in \mathbb{R}^{p \times p}$, where each entry is:

$$\mathbf{H}_{jk} = \frac{\partial^2 \mathcal{L}}{\partial w_j \partial w_k}$$

Step-by-Step Derivation

Recall:

$$\nabla_{\mathbf{w}} \mathcal{L} = \sum_{i=1}^n (\hat{p}_i - y_i) \mathbf{x}_i$$

But note that $\hat{p}_i = \sigma(z_i) = \sigma(\mathbf{w}^\top \mathbf{x}_i)$, so $\hat{p}_i$ depends on $\mathbf{w}$ too.

We differentiate the gradient:

$$\nabla_{\mathbf{w}}^2 \mathcal{L} = \sum_{i=1}^n \nabla_{\mathbf{w}} [(\hat{p}_i - y_i) \mathbf{x}_i]$$

The only term depending on $\mathbf{w}$ is $\hat{p}_i$. We apply the chain rule:

$$\nabla_{\mathbf{w}} \hat{p}_i = \sigma'(z_i) \cdot \nabla_{\mathbf{w}} z_i = \hat{p}_i(1 - \hat{p}_i) \mathbf{x}_i$$

So the outer derivative becomes:

$$\nabla_{\mathbf{w}} [(\hat{p}_i - y_i) \mathbf{x}_i] = \hat{p}_i(1 - \hat{p}_i) \mathbf{x}_i \mathbf{x}_i^\top$$

Hence:

$$\nabla_{\mathbf{w}}^2 \mathcal{L} = \sum_{i=1}^n \hat{p}_i(1 - \hat{p}_i) \mathbf{x}_i \mathbf{x}_i^\top$$

This is a **weighted sum of outer products** of input vectors.

Matrix Form

Let: - $\mathbf{X} \in \mathbb{R}^{n \times p}$: input matrix (rows = $\mathbf{x}_i^\top$) - $\hat{\mathbf{p}} \in \mathbb{R}^n$: predicted probabilities - Define $\mathbf{S} = \text{diag}(\hat{p}_i(1 - \hat{p}_i)) \in \mathbb{R}^{n \times n}$

Then the Hessian is:

$$\nabla_{\mathbf{w}}^2 \mathcal{L} = \mathbf{X}^\top \mathbf{S} \mathbf{X}$$

Summary

- The Hessian of the NLL loss with sigmoid output is:

$$\nabla_{\mathbf{w}}^2 \mathcal{L} = \sum_{i=1}^n \hat{p}_i(1 - \hat{p}_i) \mathbf{x}_i \mathbf{x}_i^\top$$

- In matrix form:

$$\nabla_{\mathbf{w}}^2 \mathcal{L} = \mathbf{X}^\top \mathbf{S} \mathbf{X}$$

- This is **positive semi-definite**, hence the NLL is convex for logistic regression.

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`