Data Analysis in Python

Edouard Duchesnay
Edouard.duchesnay@gmail.com

Preliminaries

Important links

- Web page
- Github
- Latest pdf (simplified course with Python and statistics)
- Latest pdf (full course with machine learning)
- Official deposit for citation

Download sources and course

1. Go to github repository https://github.com/duchesnay/pystatsml
2. Download repository : <> Code / Download zip
3. Extract pystaml-master

Setup working environement

1. Open Visual Studio
2. File / open folder / pystatml-master/pystatml-master / Select Folder
3. Directories of interest:
    a. introdcution
    b. python_lang

Basic Python (two days)

Introduction

- Introduction to Python language
  - Python main features
  - Development process
- Python ecosystem for data-science
- Development with Integrated Development Environment (IDE) and JupyterLab
  - Visual Studio Code (VS Code)
  - Spyder
  - JupyterLab (Jupyter Notebook)
- Anaconda and Conda environments
  - Installation
  - Conda environments (*advanced*)
  - Miniconda (*advanced*)
  - Additional packages with pip (*advanced*)

Python language

- Import libraries
- Basic operations
- Data types
  - Lists
  - Tuples
  - Strings
  - Dictionaries
  - Sets
- Execution control statements
  - Conditional statements
  - Loops
  - Example, use loop, dictionary and set to count words in a sentence
- List comprehensions, iterators, etc.
  - List comprehensions
  - Set comprehension
  - Dictionary comprehension
  - Iterators `itertools` package
  - Exceptions handling
- Functions
  - Reference and copy
  - Example: function, and dictionary comprehension
- Regular expression (*advanced*)
- System programming (*advanced*)
  - Operating system interfaces (os)
  - File input/output
  - Explore, list directories
  - Command execution with subprocess (*advanced*)

    - Multiprocessing and multithreading (*advanced*)
- Scripts and argument parsing (*advanced*)
- Object Oriented Programming (OOP)
- Style guide for Python programming
- Documenting
- Modules and packages (*advanced*)
    - Package
    - Module
    - The search path
- Unit testing (*advanced*)
    - unittest: test your code
    - Doctest: add unit tests in docstring
- Exercises
    - Exercise 1: functions
    - Exercise 2: functions + list + loop
    - Exercise 3: File I/O
    - Exercise 4: OOP

Advanced Python (two days)

Introduction

- Introduction to Python language
  - Python main features
  - Development process
- Python ecosystem for data-science
- Development with Integrated Development Environment (IDE) and JupyterLab
  - Visual Studio Code (VS Code)
  - Spyder
  - JupyterLab (Jupyter Notebook)
- Anaconda and Conda environments
  - Installation
  - Conda environments (*advanced*)
  - Miniconda (*advanced*)
  - Additional packages with pip (*advanced*)

Python language

- Import libraries
- Basic operations
- Data types
  - Lists
  - Tuples
  - Strings
  - Dictionaries
  - Sets
- Execution control statements
  - Conditional statements
  - Loops
  - Example, use loop, dictionary and set to count words in a sentence
- List comprehensions, iterators, etc. (*basic and advanced*)
  - List comprehensions
  - Set comprehension
  - Dictionary comprehension
  - Iterators `itertools` package
  - Exceptions handling
- Functions
  - Reference and copy
  - Example: function, and dictionary comprehension
- Regular expression (*advanced*)
- System programming (*advanced*)
  - Operating system interfaces (os)
  - File input/output
  - Explore, list directories
  - Command execution with subprocess (*advanced*)

    - Multiprocessing and multithreading (*advanced*)
- Scripts and argument parsing (*advanced*)
- Object Oriented Programming (OOP)
- Style guide for Python programming
- Documenting
- Modules and packages (*advanced*)
    - Package
    - Module
    - The search path
- Unit testing (*advanced*)
    - unittest: test your code
    - Doctest: add unit tests in docstring
- Exercises
    - Exercise 1: functions
    - Exercise 2: functions + list + loop
    - Exercise 3: File I/O
    - Exercise 4: OOP

Scientific Python

- Numpy: arrays and matrices
    - Create arrays
    - Examining arrays
    - Reshaping
    - Summary on axis, reshaping/flattening and selection
    - Stack arrays
    - Selection
    - Vectorized operations
    - Broadcasting
    - Exercises
- Pandas: data manipulation
    - Create DataFrame
    - Combining DataFrames
    - Summarizing
    - Columns selection
    - Rows selection (basic)
    - Sorting
    - Rows iteration
    - Rows selection (filtering)
    - Sorting
    - Descriptive statistics
    - Quality check
    - Operation: multiplication
    - Renaming
    - Dealing with outliers
    - File I/O
    - Exercises

- Data visualization: matplotlib & seaborn
    - Basic plots
    - Scatter (2D) plots
    - Saving Figures
    - Multiple axis
    - Pairwise scatter plots
    - Time series

Statistics

- Univariate statistics
    - Libraries
    - Estimators of the main statistical measures
    - Main distributions
    - Hypothesis Testing
    - Testing pairwise associations
    - Pearson correlation test: test association between two quantitative variables
    - Two sample (Student) t-test: compare two means
    - ANOVA F-test (quantitative ~ categorial (>=2 levels))
    - Chi-square, chi2 (categorial ~ categorial)
    - Non-parametric test of pairwise associations
    - Linear model
    - Linear model with statsmodels
    - Multiple comparisons
- Lab: Brain volumes study
    - Manipulate data
    - Descriptive Statistics
    - Statistics
- Linear Mixed Models
    - Introduction
    - Random intercept
    - Random slope
    - Conclusion on modeling random effects
    - Theory of Linear Mixed Models
    - Checking model assumptions (Diagnostics)
    - References
- Multivariate statistics
    - Linear Algebra
    - Mean vector
    - Covariance matrix
    - Correlation matrix
    - Precision matrix
    - Mahalanobis distance
    - Multivariate normal distribution
    - Exercises
- Time series in python
    - Stationarity

- Pandas time series data structure
- Time series analysis of Google trends
- Read data
- Recode data
- Exploratory data analysis
- Resampling, smoothing, windowing, rolling average: trends
- First-order differencing: seasonal patterns
- Periodicity and correlation
- Autocorrelation
- Time series forecasting with Python using Autoregressive Moving Average (ARMA) models

CONTENTS

CHAPTER

ONE

INTRODUCTION

Important links:

- Web page
- Github
- Latest pdf
- Official deposit for citation.

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 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, Pytroch*) 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 wholes file:

```
python file.py
```

Interactive mode

1. `python` interpreter:

```
python
```

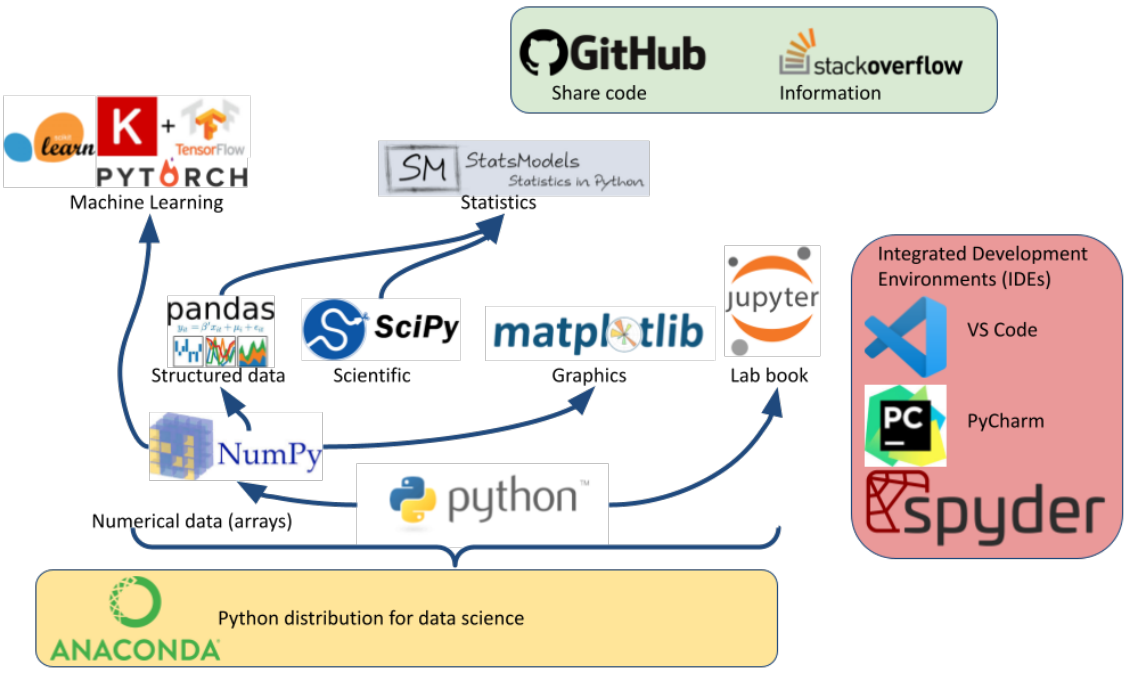
Quite with `CTL-D`

2. `ipython`: advanced interactive python interpreter:

```
ipython
```

Quite with `CTL-D`

1.2 Python ecosystem for data-science



Numpy: Basic numerical operation and matrix operation

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

1.3.1 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`

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`

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

1.3.2 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

1.3.3 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: inluding 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.

1.4 Anaconda and Conda environments

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

1.4.1 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"
```

1.4.2 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

List of all environments

```
conda env list
```

Creating an environment. Example, environment_student.yml:

```
name: pystatsml_student
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 environment_student.yml
```

Activate/deactivate an environment:

```
conda activate environment_student
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 environment.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 environment_student --all
```

1.4.3 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 sqlite
conda install -y spyder
conda install -y jupyter
```

1.4.4 Additional packages with 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
```

CHAPTER

TWO

PYTHON LANGUAGE

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

```
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())
```

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

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].upper())
```

```
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
```

2.4.3 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()

count = {word: 0 for word in set(words)}
for word in words:
    count[word] += 1

print(count)
```

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

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)
```

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

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 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 deuxtrois ['deux', 'trois', 2, 3]
```

Default arguments

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


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
```

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.

```
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', 'Lisa': 'daughter',
↪'Maggie': 'daughter', 'Mona': 'grandma', 'Abraham': 'grandpa'}
Ages: {'Homer': 45, 'Marge': 43, 'Bart': 11, 'Lisa': 10, 'Maggie': 1}
Join: [['Maggie', 'daughter', 1], ['Bart', 'son', 11], ['Homer', 'father', 45], [
↪'Lisa', 'daughter', 10], ['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 [tnrfv].
- `[^\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>`

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

```
/home/ed203246/git/pystatsml/python_lang/python_lang.py:1: SyntaxWarning: invalid␣
↪escape sequence '\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("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"))
```

(continues on next page)

(continued from previous page)

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

```
/home/ed203246/git/pystatsml/python_lang/python_lang.py:3: SyntaxWarning: invalid␣
↪escape sequence '\w'
  Source  `Kevin Markham <https://github.com/justmarkham/python-reference>`_
/home/ed203246/git/pystatsml/python_lang/python_lang.py:10: SyntaxWarning:␣
↪invalid escape sequence '\w'
  #
['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:2: SyntaxWarning: invalid␣
↪escape sequence '\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:2: SyntaxWarning: invalid␣
↪escape sequence '\s'

/home/ed203246/git/pystatsml/python_lang/python_lang.py:3: SyntaxWarning: invalid␣
↪escape sequence '\s'
  Source  `Kevin Markham <https://github.com/justmarkham/python-reference>`_
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:2: SyntaxWarning: invalid␣
↪escape sequence '\s'


'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', 'boot', 'cdrom']
```

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))
```

```
0.68
0.64
0.64
0.64
0.67
```

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: 0.61
Sequential execution, elapsed time: 1.5166292190551758
```

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()
```

(continues on next page)

(continued from previous page)

```
# 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: 0.8519494533538818
```

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 . 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: 0.68
Concurrent execution with processes, elapsed time: 0.9629409313201904
```

`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: 0.68
Concurrent execution with processes, elapsed time: 3.1754207611083984
```

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)
    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 0x73febfd78650>)
```

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
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
    ----------
    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
```

(continues on next page)

(continued from previous page)

```
@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 searches 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 asses to 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_<mymodule>.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.

Unitest 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.
```

(continues on next page)

(continued from previous page)

```
    ...
    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 ad 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_
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 * years_of_service`.
2. Create a subclass `Manager` which redefine `salary` method `2500 + 120 * 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 10.869 seconds)

CHAPTER

THREE

SCIENTIFIC PYTHON

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.

Sources:

- Kevin Markham: https://github.com/justmarkham

Computation time:

import numpy as np

l = [v for v in range(10 ** 8)] s = 0 %time for v in l: s += v

arr = np.arange(10 ** 8) %time arr.sum()

3.1.1 Create arrays

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

```
import numpy as np

data1 = [1, 2, 3, 4, 5]             # list
arr1 = np.array(data1)              # 1d array
data2 = [range(1, 5), range(5, 9)]  # list of lists
arr2 = np.array(data2)              # 2d array
arr2.tolist()                       # convert array back to list
```

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

create special arrays

```
np.zeros(10)
np.zeros((3, 6))
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
```

```
array([   1.,   10.,  100., 1000.])
```

arange is like range, except it returns an array (not a list)

```
int_array = np.arange(5)
float_array = int_array.astype(float)
```

3.1.2 Examining arrays

```
arr1.dtype      # float64
arr2.ndim       # 2
arr2.shape      # (2, 4) - axis 0 is rows, axis 1 is columns
arr2.size       # 8 - total number of elements
len(arr2)       # 2 - size of first dimension (aka axis)
```

```
2
```

3.1.3 Reshaping

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

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

Add an axis

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

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

Transpose

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

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

Flatten: always returns a flat copy of the orriginal array

```
arr_flt = arr.flatten()
arr_flt[0] = 33
print(arr_flt)
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_flt = arr.ravel()
arr_flt[0] = 33
print(arr_flt)
print(arr)
```

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

3.1.4 Summary on axis, 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)
```

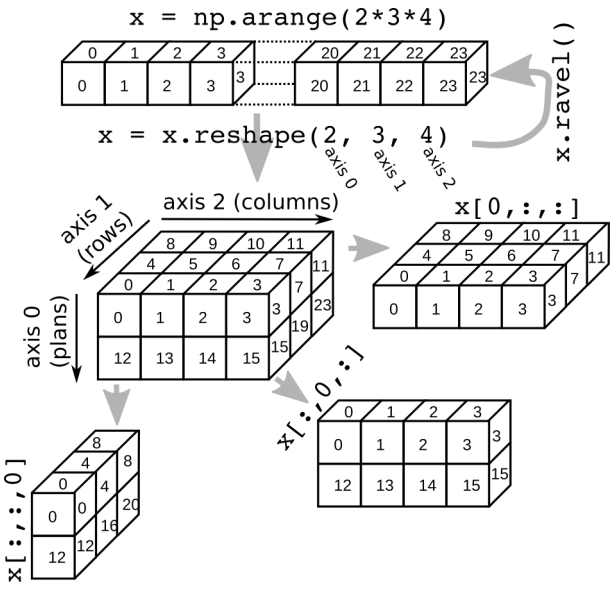
```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23]
```

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]
```

(continues on next page)



(continued from previous page)

```
  [ 8  9 10 11]]

 [[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]]
```

Simple example with 2 array

Exercise:

- Get second line

- Get third column

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

arr[1, :]
arr[:, 2]
```

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

array([2., 7.])
```

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 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]])
```

3.1.6 Selection

Single item

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

arr[0]         # 0th element (slices like a list)
arr[0, 3]      # row 0, column 3: returns 4
arr[0][3]      # alternative syntax
```

```
3.0
```

Slicing

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

```
arr[0, :]      # row 0: returns 1d array ([1, 2, 3, 4])
arr[:, 0]      # column 0: returns 1d array ([1, 5])
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)
```

```
[[1. 2. 3.]
 [6. 7. 8.]]
```

Slicing returns a view (not a copy) Modification

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

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

Row 0: reverse order

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

# The rule of thumb here can be: in the context of lvalue indexing (i.e. the␣
↪indices are placed in the left hand side value of an assignment), no view or␣
↪copy of the array is created (because there is no need to). However, with␣
↪regular values, the above rules for creating views does apply.
```

```
[ 4.  3.  2. 33.  0.]
```

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.  2.  3.]
 [ 6.  7.  8.]]
[[44.  2.  3.]
 [ 6.  7.  8.]]
[[ 0. 33.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]]
```

Boolean arrays indexing

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

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

```
[33.  6.  7.  8.  9.]
[44.  6.  7.  8.  9.]
[[ 0. 33.  2.  3.  4.]
 [ 5.  6.  7.  8.  9.]]
```

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)
```

```
[[0. 0. 2. 3. 4.]
 [5. 0. 0. 0. 0.]]
```

Boolean arrays indexing continues

```
names = np.array(['Bob', 'Joe', 'Will', 'Bob'])
names == 'Bob'                          # returns a boolean array
names[names != 'Bob']                   # logical selection
(names == 'Bob') | (names == 'Will')    # keywords "and/or" don't work with␣
↪boolean arrays
names[names != 'Bob'] = 'Joe'           # assign based on a logical selection
np.unique(names)                        # set function
```

```
array(['Bob', 'Joe'], dtype='<U4')
```

3.1.7 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.argmin()                # 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])
```

3.1.8 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.

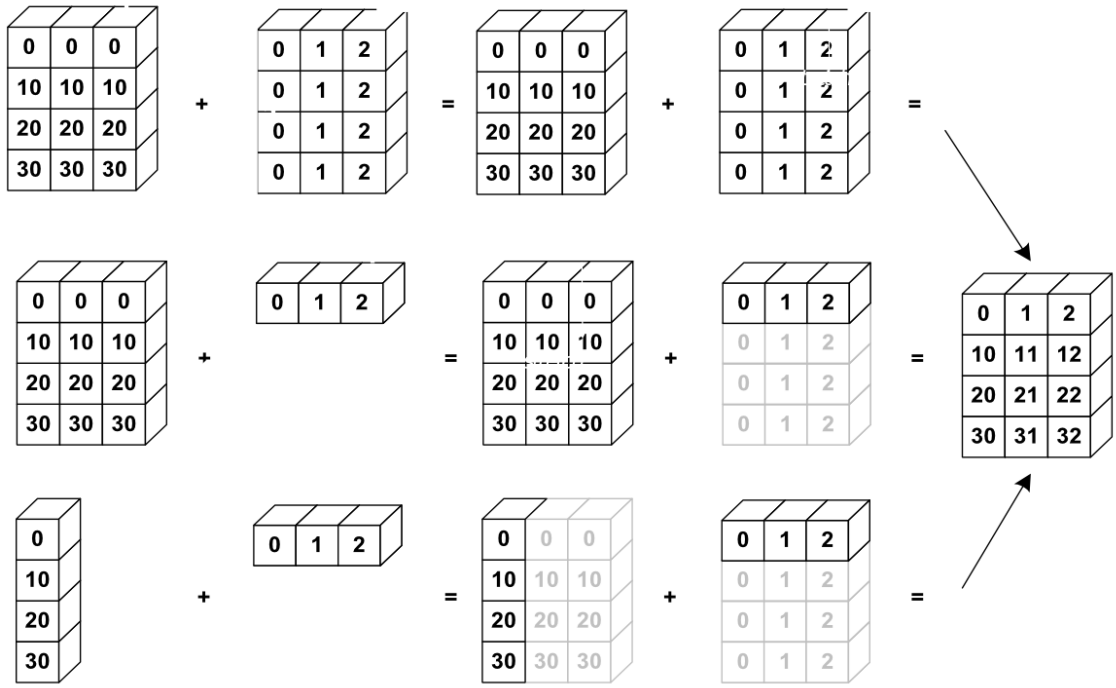


Fig. 1: Source: http://www.scipy-lectures.org

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

(continues on next page)

(continued from previous page)

```
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.],
       [  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
```

3.1.9 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 0.010 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 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'],
```

(continues on next page)

(continued from previous page)

```
                          job=['engineer', 'scientist']))

print(user3)
```

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

3.2.2 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   john   26      M     student   180.0
2   eric   22      M     student   175.0
3   paul   58      F     manager     NaN
4  peter   33      M    engineer     NaN
5  julie   44      F  scientist   171.0
```

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    john      age         26
2    eric      age         22
3    paul      age         58
```

(continues on next page)

(continued from previous page)

```
4   peter      age         33
5   julie      age         44
6   alice   gender          F
7    john   gender          M
8    eric   gender          M
9    paul   gender          F
10  peter   gender          M
11  julie   gender          F
12  alice      job    student
13   john      job    student
14   eric      job    student
15   paul      job    manager
16  peter      job   engineer
17  julie      job  scientist
18  alice   height      165.0
19   john   height      180.0
20   eric   height      175.0
21   paul   height        NaN
22  peter   height        NaN
23  julie   height      171.0
```

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

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

```
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
```

3.2.3 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
```

Descriptive statistics

```
users.describe(include="all")
```

Meta-information

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

```
(6, 5)
```

3.2.4 Columns selection

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

# 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
```

3.2.5 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, 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
3   paul   58      F     manager     NaN
5  julie   44      F  scientist   171.0
```

Get the two first rows using *iloc* (strictly integer position)

```
df.iloc[[0, 1], :]  # Ok, but watch the index: 0, 3
```

Use *loc*

```
try:
    df.loc[[0, 1], :]  # Failed
except KeyError as err:
    print(err)
```

```
'[1] not in index'
```

Reset index

```
df = df.reset_index(drop=True)  # Watch the index
print(df)
print(df.loc[[0, 1], :])
```

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

3.2.6 Sorting

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
john 26
```

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
john 26
```

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 categorial 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
4  peter   33      M   engineer     NaN
3   paul   58      F    manager     NaN
5  julie   44      F  scientist   171.0
0  alice   19      F    student   165.0
2   eric   22      M    student   175.0
1   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
```

(continues on next page)

(continued from previous page)

```
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
```

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
           name                                                      age  ...␣
↪gender height
          count unique    top freq mean  std  min  25%  50%  75%  max count  ... ␣
↪   max  count unique top freq        mean       std    min    25%    50%    75% ␣
↪   max
job                                                                          ...
engineer      1      1  peter    1  NaN  NaN  NaN  NaN  NaN  NaN  NaN   1.0  ... ␣
↪   NaN    0.0    NaN NaN  NaN         NaN       NaN    NaN    NaN    NaN    NaN ␣
↪   NaN
manager       1      1   paul    1  NaN  NaN  NaN  NaN  NaN  NaN  NaN   1.0  ... ␣
↪   NaN    0.0    NaN NaN  NaN         NaN       NaN    NaN    NaN    NaN    NaN ␣
↪   NaN
scientist     1      1  julie    1  NaN  NaN  NaN  NaN  NaN  NaN  NaN   1.0  ... ␣
↪   NaN    1.0    NaN NaN  NaN  171.000000       NaN  171.0  171.0  171.0  171.0 ␣
↪171.0
student       3      3  alice    1  NaN  NaN  NaN  NaN  NaN  NaN  NaN   3.0  ... ␣
↪   NaN    3.0    NaN NaN  NaN  173.333333  7.637626  165.0  170.0  175.0  177.5 ␣
↪180.0

[4 rows x 44 columns]
```

Groupby in a loop

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

```
engineer     name  age gender       job  height
```

(continues on next page)

(continued from previous page)

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

(continues on next page)

(continued from previous page)

```
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()

print(df)
```

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

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   john   26      M     student   180.0
2       2   eric   22      M     student   175.0
5       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())
```

```
248.48963819938044
```

Based on non-parametric statistics: use the median

Median absolute deviation (MAD), based on the median, is a robust non-parametric statistics. https://en.wikipedia.org/wiki/Median_absolute_deviation

```
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())
```

```
173.80000467192673 178.7023568870694
```

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

```
xls_filename = os.path.join(tmpdir, "users.xlsx")
users.to_excel(xls_filename, sheet_name='users', index=False)

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

# 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
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/neurospin/pystatsml/tree/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.

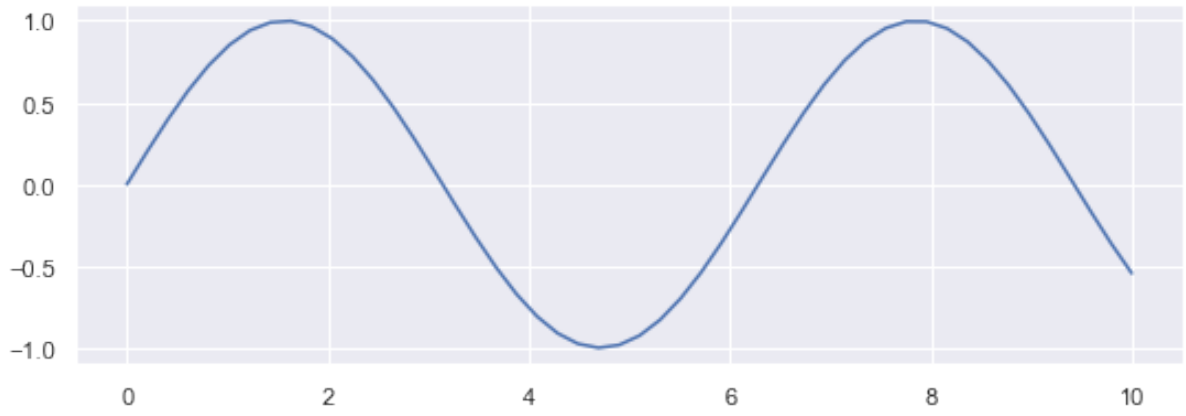
2. Save the original users and "imputed" frame in a single excel file "users.xlsx" with 2 sheets: original, imputed.

Total running time of the script: (0 minutes 0.890 seconds)

3.3 Data visualization: matplotlib & seaborn

3.3.1 Basic plots

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# inline plot (for jupyter)
%matplotlib inline

plt.figure(figsize=(9, 3))
x = np.linspace(0, 10, 50)
sinus = np.sin(x)

plt.plot(x, sinus)
plt.show()
```

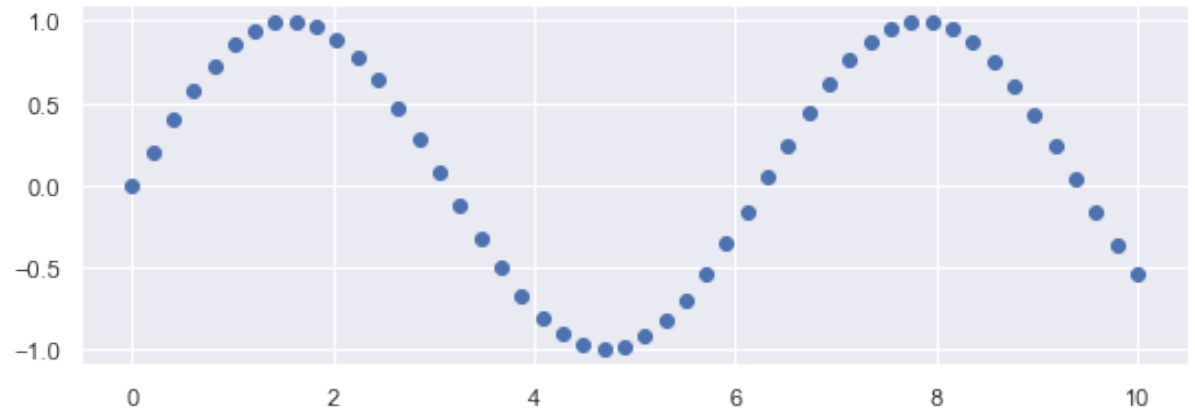



```
plt.figure(figsize=(9, 3))

plt.plot(x, sinus, "o")
```

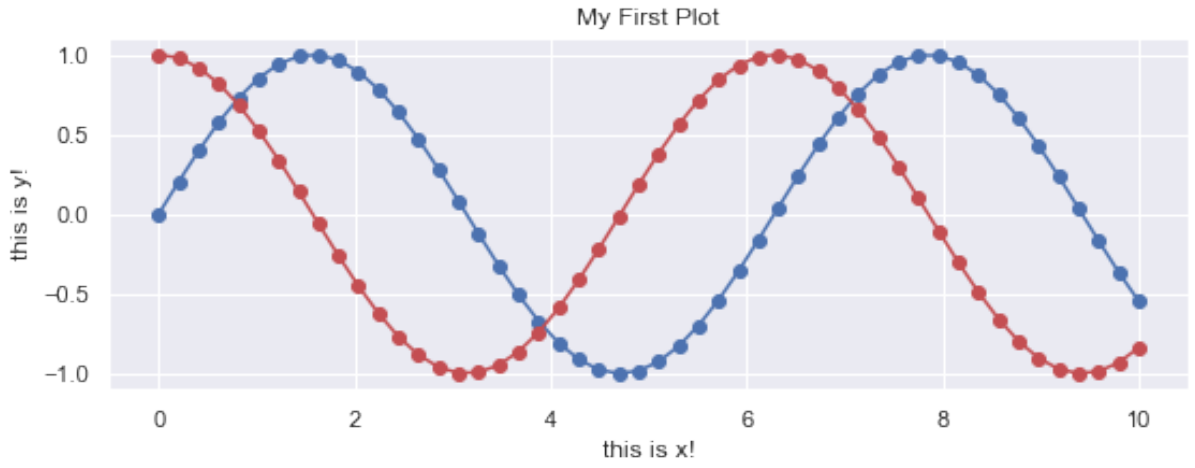
(continues on next page)

(continued from previous page)

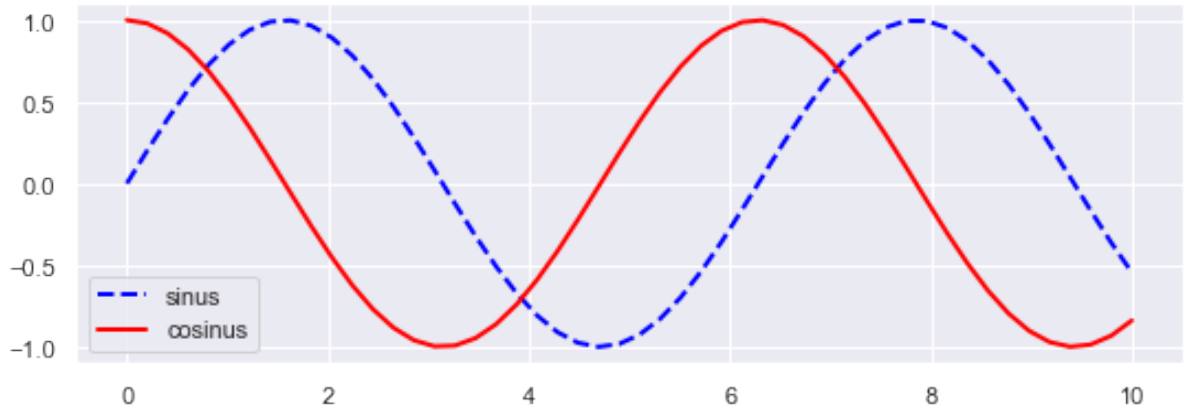
```
plt.show()
# use plt.plot to get color / marker abbreviations
```




```
# Rapid multiplot

plt.figure(figsize=(9, 3))
cosinus = np.cos(x)
plt.plot(x, sinus, "-b", x, sinus, "ob", x, cosinus, "-r", x, cosinus, "or")
plt.xlabel('this is x!')
plt.ylabel('this is y!')
plt.title('My First Plot')
plt.show()
```




```
# Step by step

plt.figure(figsize=(9, 3))
plt.plot(x, sinus, label='sinus', color='blue', linestyle='--', linewidth=2)
plt.plot(x, cosinus, label='cosinus', color='red', linestyle='-', linewidth=2)
plt.legend()
plt.show()
```



3.3.2 Scatter (2D) plots

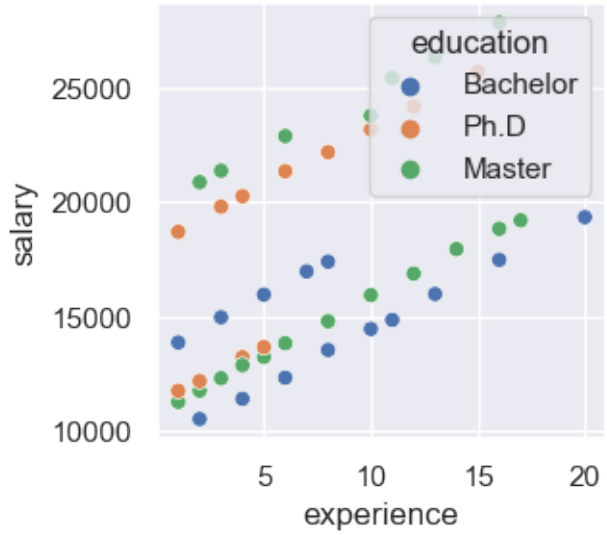
Load dataset

```
import pandas as pd
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)

df = salary
print(df.head())
```

```
   salary  experience education management
0   13876           1  Bachelor          Y
1   11608           1      Ph.D          N
2   18701           1      Ph.D          Y
3   11283           1    Master          N
4   11767           1      Ph.D          N
```

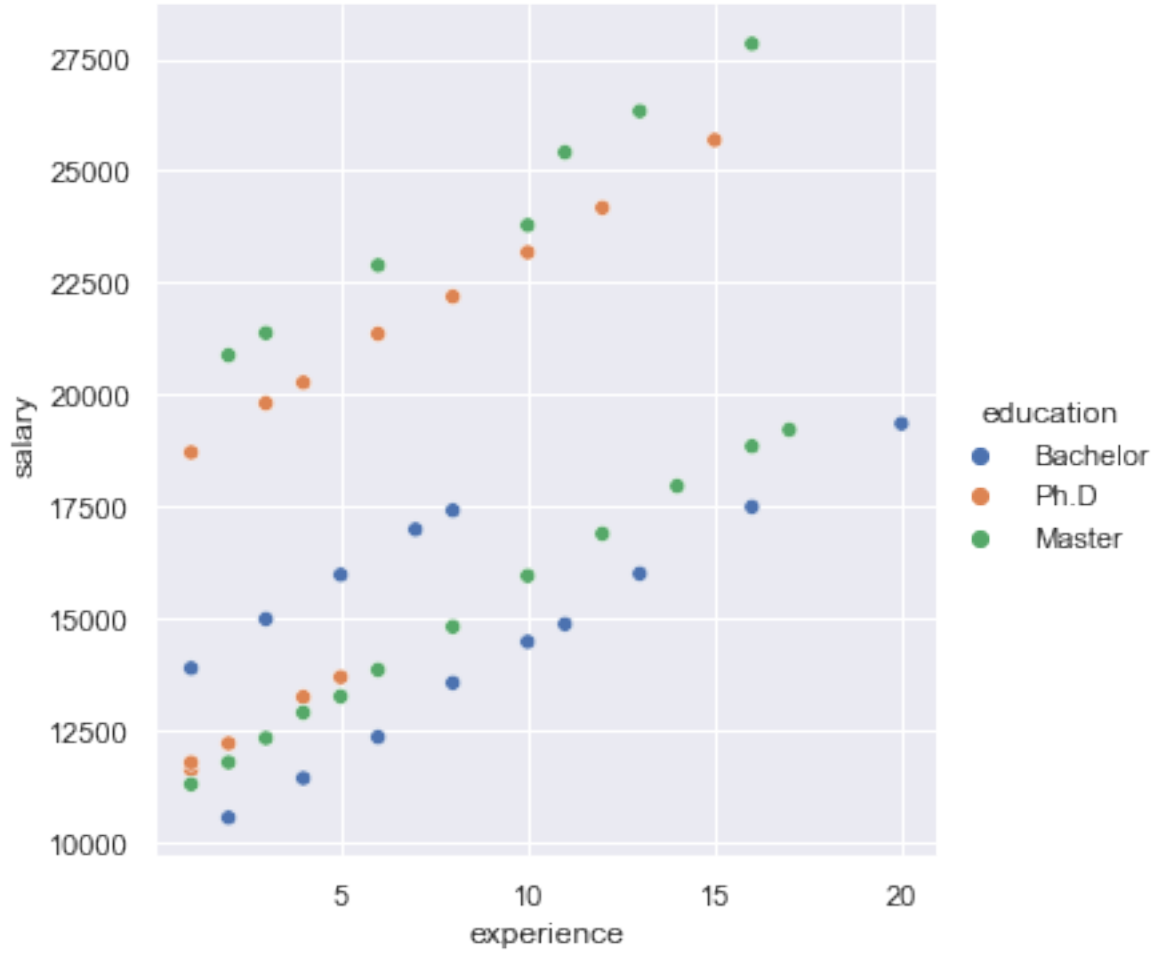
Simple scatter with colors

```
plt.figure(figsize=(3, 3), dpi=100)
_ = sns.scatterplot(x="experience", y="salary", hue="education", data=salary)
```



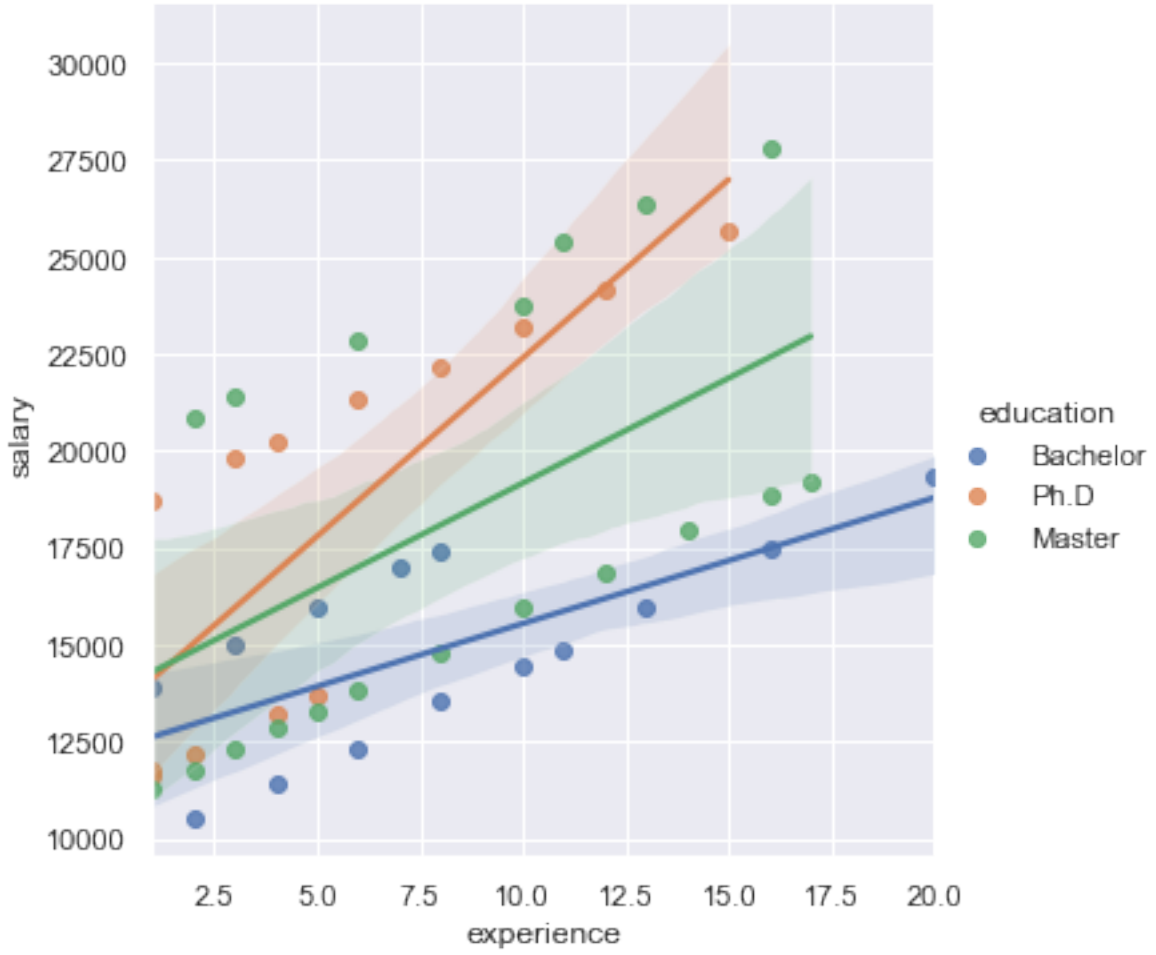
Legend outside

```
ax = sns.relplot(x="experience", y="salary", hue="education", data=salary)
```



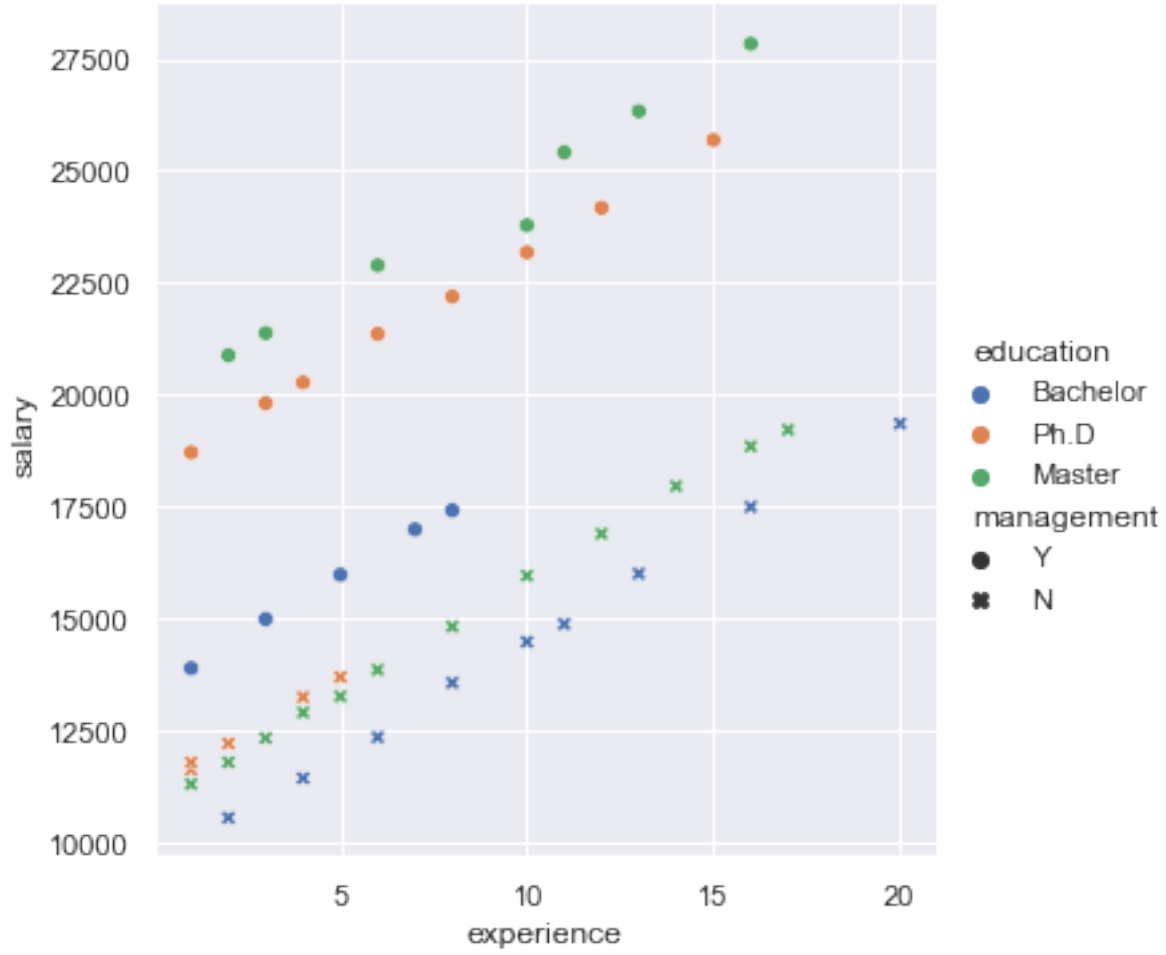
Linear model

```
ax = sns.lmplot(x="experience", y="salary", hue="education", data=salary)
```



Scatter plot with colors and symbols

```
ax = sns.relplot(x="experience", y="salary", hue="education", style='management',␣
↪data=salary)
```



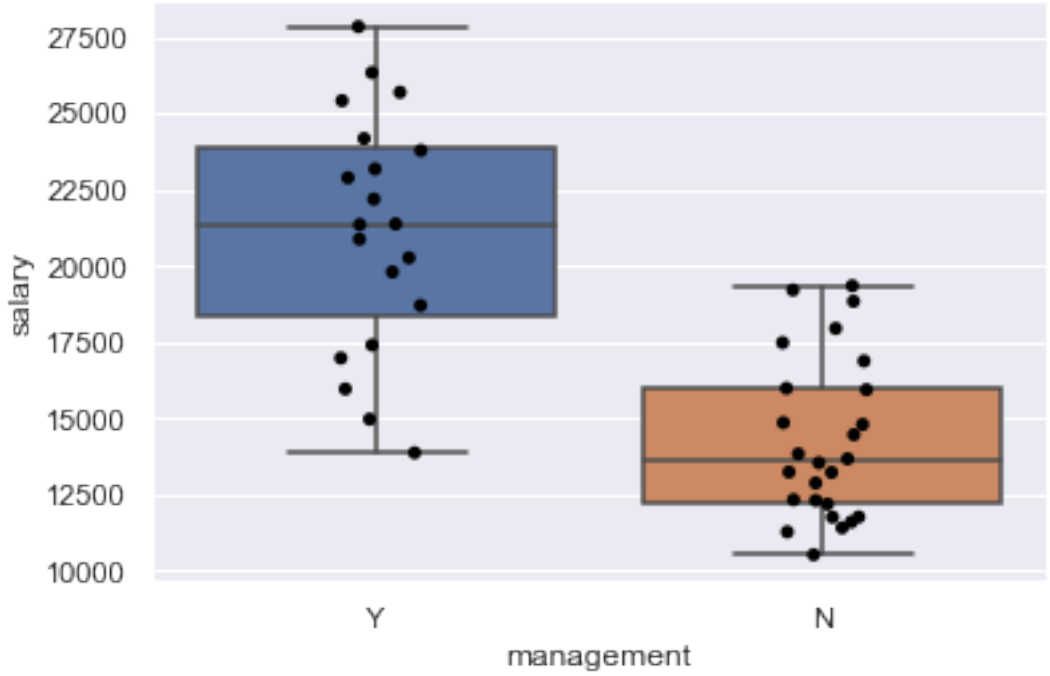
3.3.3 Saving Figures

```
### bitmap format
plt.plot(x, sinus)
plt.savefig("sinus.png")
plt.close()

# Prefer vectorial format (SVG: Scalable Vector Graphics) can be edited with
# Inkscape, Adobe Illustrator, Blender, etc.
plt.plot(x, sinus)
plt.savefig("sinus.svg")
plt.close()

# Or pdf
plt.plot(x, sinus)
plt.savefig("sinus.pdf")
plt.close()
```

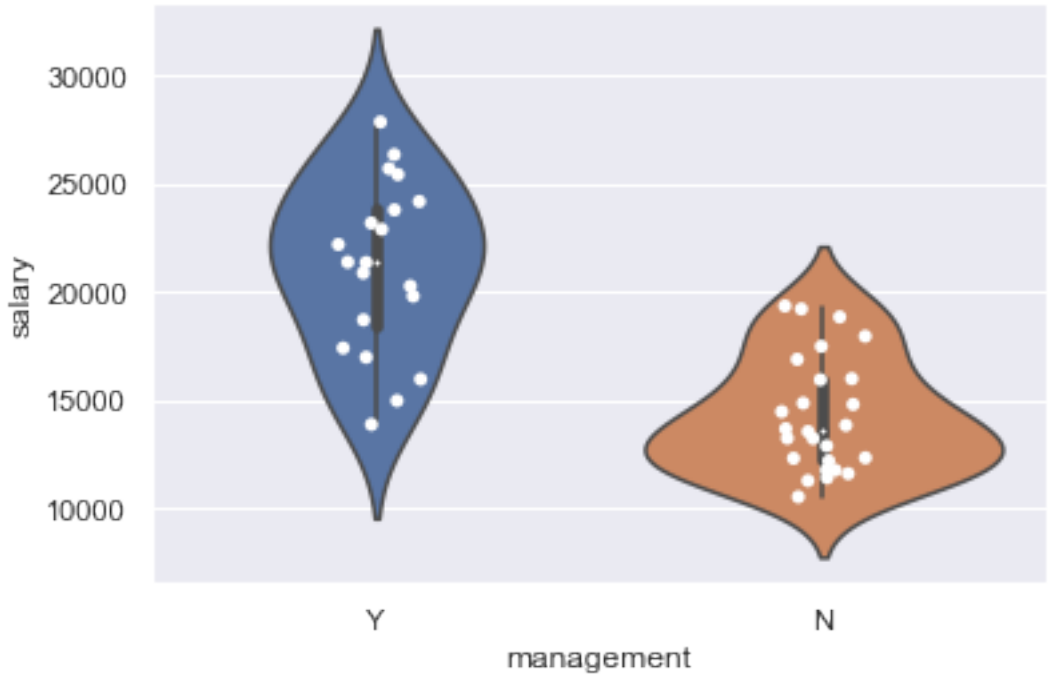
Boxplot and violin plot: one factor

Box plots are non-parametric: they display variation in samples of a statistical population without making any assumptions of the underlying statistical distribution.

```
ax = sns.boxplot(x="management", y="salary", data=salary)
ax = sns.stripplot(x="management", y="salary", data=salary, jitter=True, color=
↪"black")
```

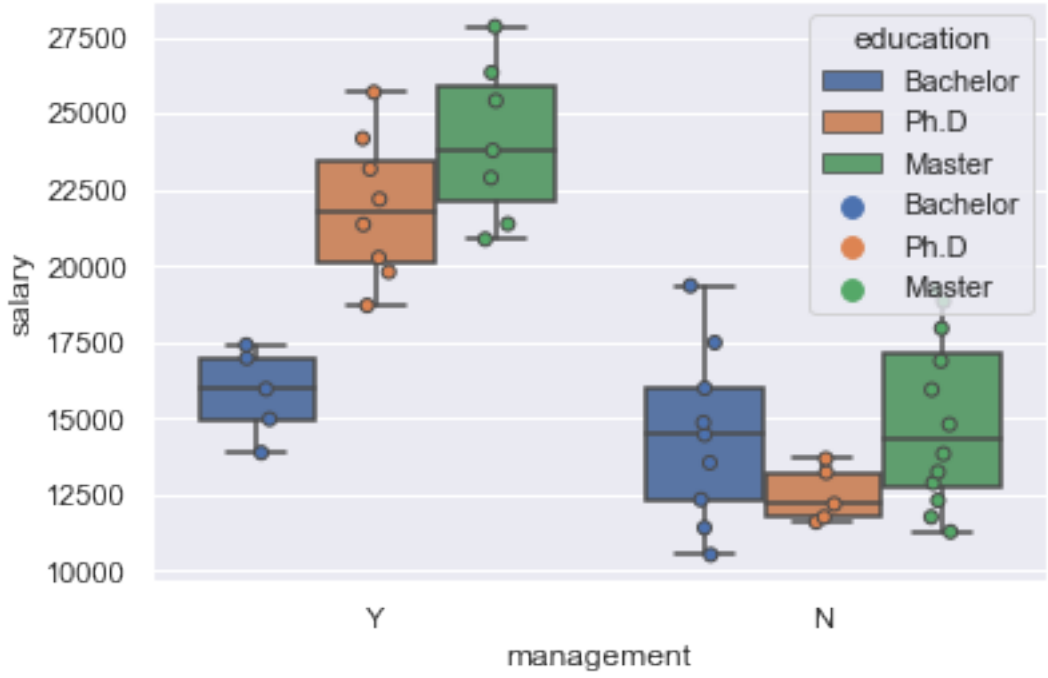



```
ax = sns.violinplot(x="management", y="salary", data=salary)
ax = sns.stripplot(x="management", y="salary", data=salary, jitter=True, color=
↪"white")
```

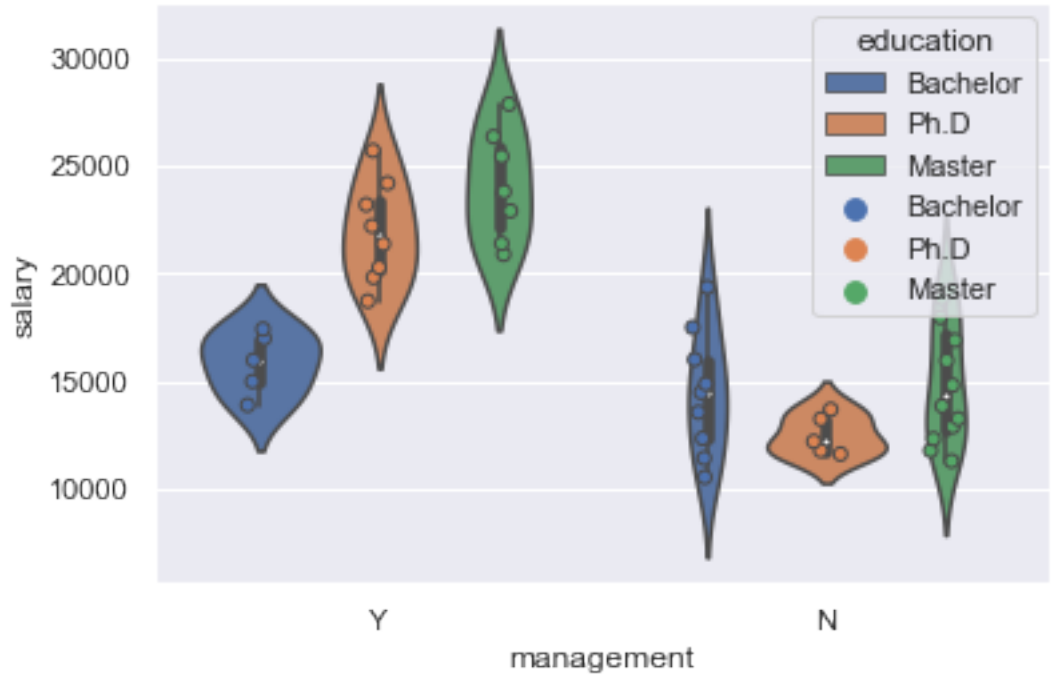


Boxplot and violin plot: two factors

```
ax = sns.boxplot(x="management", y="salary", hue="education", data=salary)
ax = sns.stripplot(x="management", y="salary", hue="education", data=salary,␣
↪jitter=True, dodge=True, linewidth=1)
```



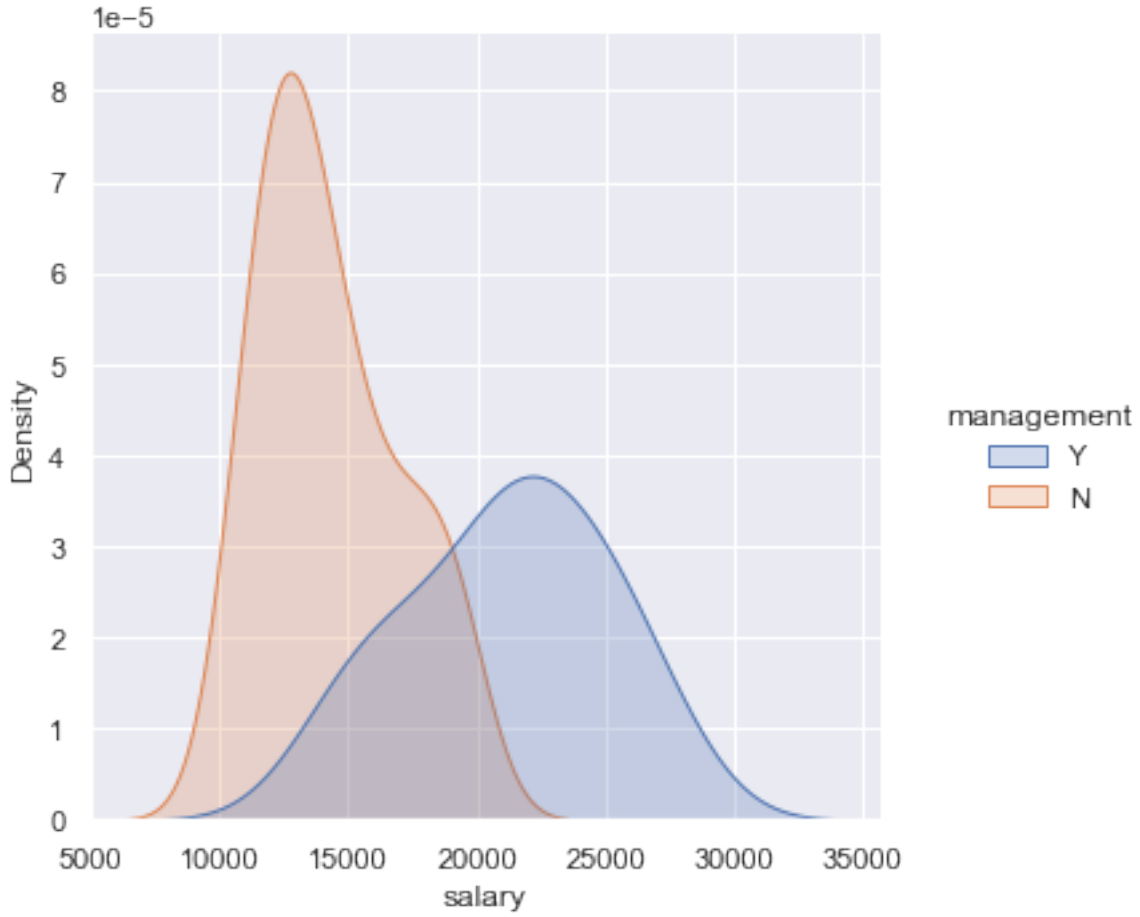

```
ax = sns.violinplot(x="management", y="salary", hue="education", data=salary)
ax = sns.stripplot(x="management", y="salary", hue="education", data=salary,␣
↪jitter=True, dodge=True, linewidth=1)
```



Distributions and density plot

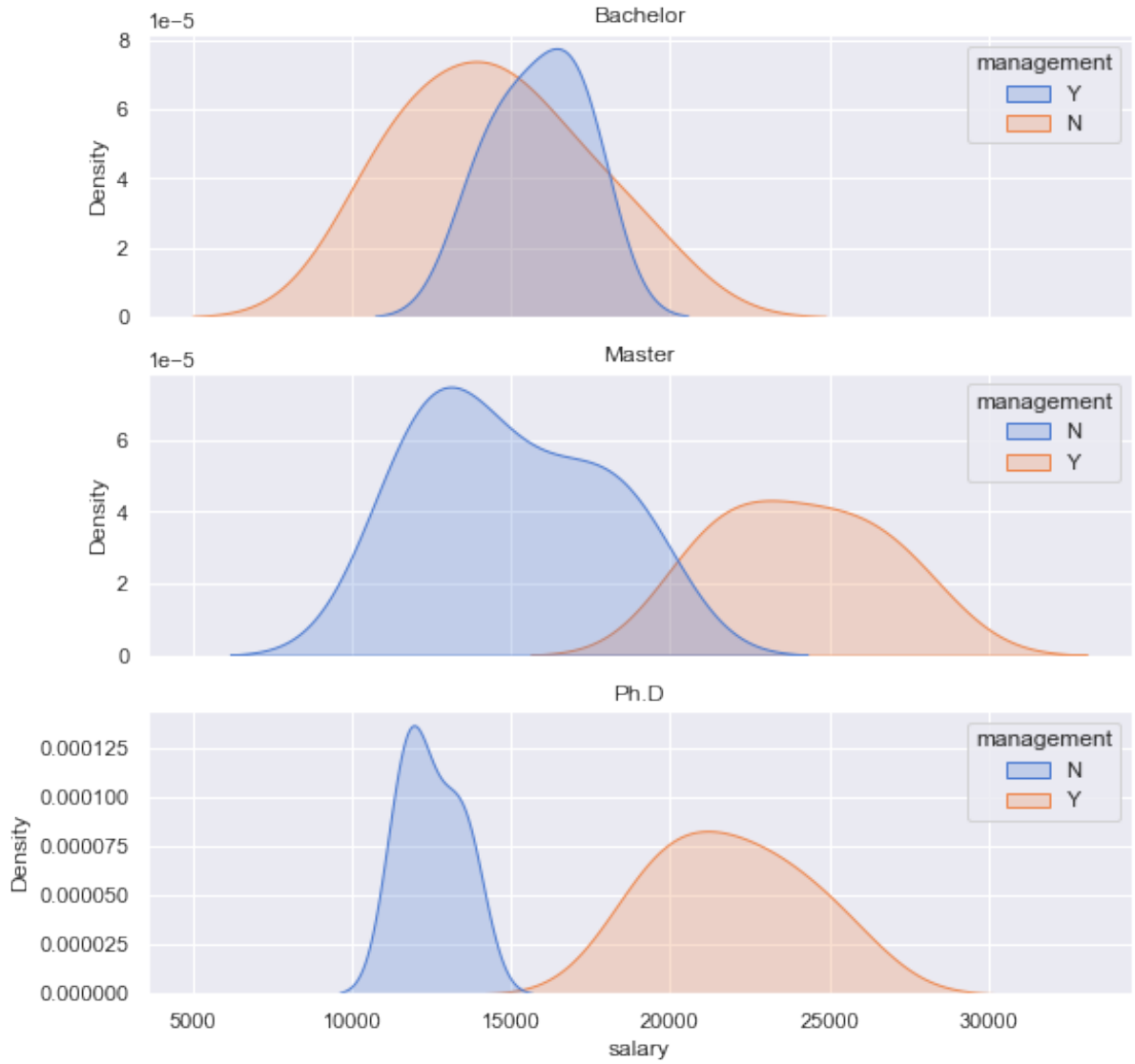
Distributions with seaborn

```
ax = sns.displot(x="salary", hue="management", kind="kde", data=salary, fill=True)
```



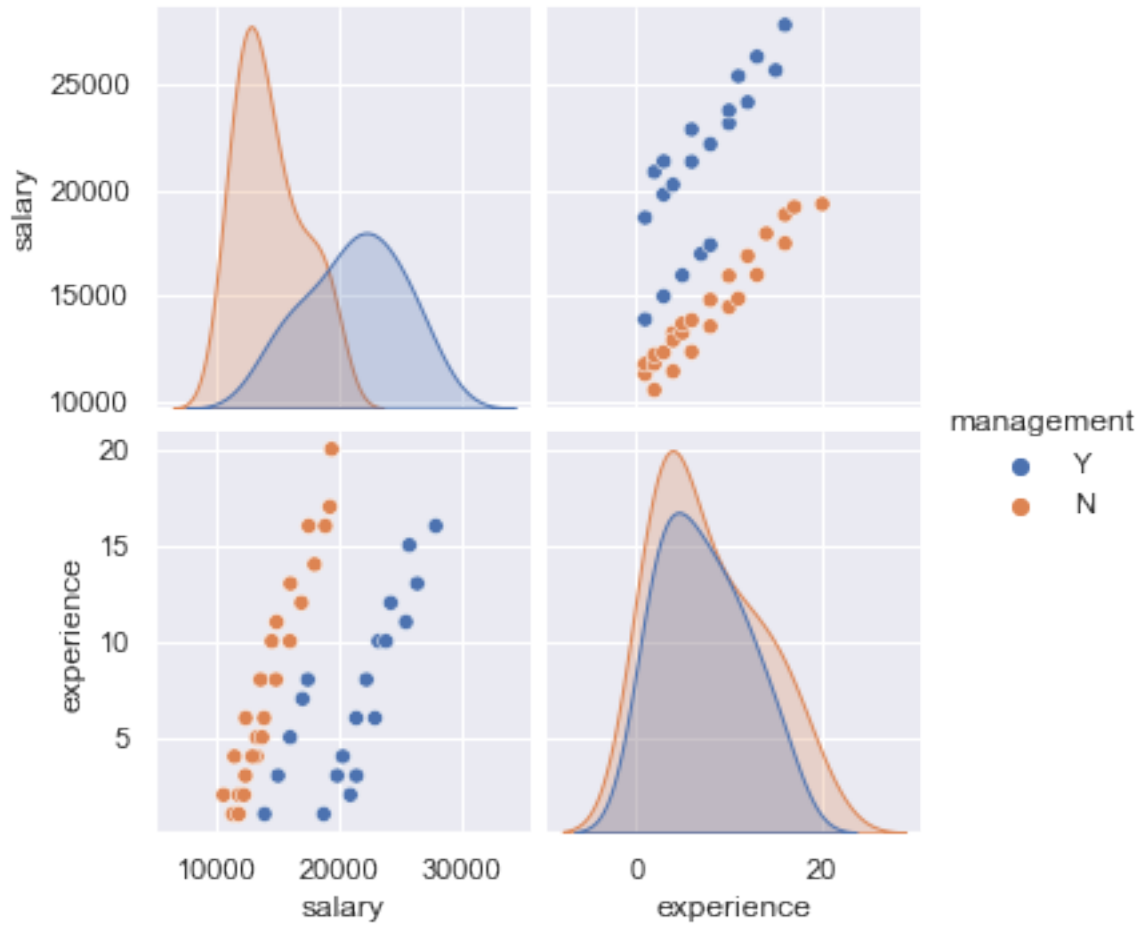
3.3.4 Multiple axis

```
fig, axes = plt.subplots(3, 1, figsize=(9, 9), sharex=True)

i = 0
for edu, d in salary.groupby(['education']):
    sns.kdeplot(x="salary", hue="management", data=d, fill=True, ax=axes[i],␣
↪palette="muted")
    axes[i].set_title(edu)
    i += 1
```



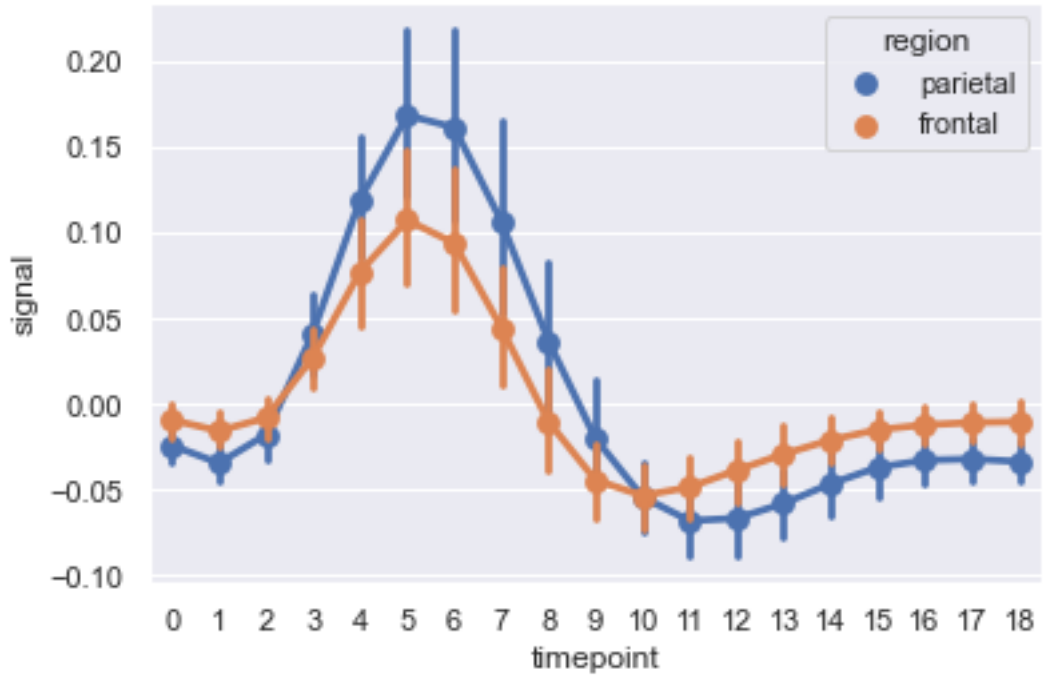
3.3.5 Pairwise scatter plots

```
ax = sns.pairplot(salary, hue="management")
```



3.3.6 Time series

```
import seaborn as sns
sns.set(style="darkgrid")

# Load an example dataset with long-form data
fmri = sns.load_dataset("fmri")

# Plot the responses for different events and regions
ax = sns.pointplot(x="timepoint", y="signal",
             hue="region", style="event",
             data=fmri)
```


region
parietal
frontal
signal
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
timepoint

CHAPTER

FOUR

STATISTICS

4.1 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.

4.1.1 Libraries

Data

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

Plots

```
import matplotlib.pyplot as plt
import seaborn as sns
```

Statistics

- Basic: scipy.stats
- Advanced: statsmodels. statsmodels API:
  - `statsmodels.api`: Cross-sectional models and methods. Canonically 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`.

```
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
```

```
%matplotlib inline
```

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 datset
iris = sm.datasets.get_rdataset("iris").data
iris.columns = [s.replace('.', '') for s in iris.columns]
```

4.1.2 Estimators of the main statistical measures

Mean

Properties of the expected value operator $\mathrm{E}(\cdot)$ of a random variable X

$$E(X + c) = E(X) + c \tag{4.1}$$
$$E(X + Y) = E(X) + E(Y) \tag{4.2}$$
$$E(aX) = aE(X) \tag{4.3}$$

The estimator $\bar{x}$ on a sample of size n: $x = x_1, ..., x_n$ is given by

$$\bar{x} = \frac{1}{n}\sum_i x_i$$

$\bar{x}$ is itself a random variable with properties:

- $E(\bar{x}) = \bar{x}$,
- $\mathrm{Var}(\bar{x}) = \frac{\mathrm{Var}(X)}{n}$.

Variance

$$\mathrm{Var}(X) = E((X - E(X))^2) = E(X^2) - (E(X))^2$$

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

$$\mathrm{Std}(X) = \sqrt{\mathrm{Var}(X)}$$

The estimator is simply $\sigma_x = \sqrt{\sigma_x^2}$.

Covariance

$$\mathrm{Cov}(X, Y) = E((X - E(X))(Y - E(Y))) = E(XY) - E(X)E(Y).$$

Properties:

$$\mathrm{Cov}(X, X) = \mathrm{Var}(X) \tag{4.4}$$
$$\mathrm{Cov}(X, Y) = \mathrm{Cov}(Y, X) \tag{4.5}$$
$$\mathrm{Cov}(cX, Y) = c\,\mathrm{Cov}(X, Y) \tag{4.6}$$
$$\mathrm{Cov}(X + c, Y) = \mathrm{Cov}(X, Y) \tag{4.7}$$
$$\tag{4.8}$$

The estimator with $df = 1$ is

$$\sigma_{xy} = \frac{1}{n-1}\sum_i (x_i - \bar{x})(y_i - \bar{y}).$$

Correlation

$$\mathrm{Cor}(X, Y) = \frac{\mathrm{Cov}(X, Y)}{\mathrm{Std}(X)\,\mathrm{Std}(Y)}$$

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:

$$\mathrm{SE}(X) = \frac{\mathrm{Std}(X)}{\sqrt{n}}.$$

It is most commonly considered for the mean with the estimator

$$\mathrm{SE}(X) = \mathrm{Std}(X) = \sigma_{\bar{x}} \tag{4.9}$$

$$= \frac{\sigma_x}{\sqrt{n}}. \tag{4.10}$$

Descriptives 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
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)

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))
```

```
[[ 0.01025944 -0.00661557]
 [-0.00661557  0.0167    ]]
```

Descriptives statistics on Iris dataset

With Pandas

Columns' means

```
iris.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.std()
```

```
SepalLength    0.828066
SepalWidth     0.435866
PetalLength    1.765298
PetalWidth     0.762238
dtype: float64
```

With Numpy

```
X = iris[['SepalLength', 'SepalWidth', 'PetalLength', 'PetalWidth']].values
iris.columns
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])
```

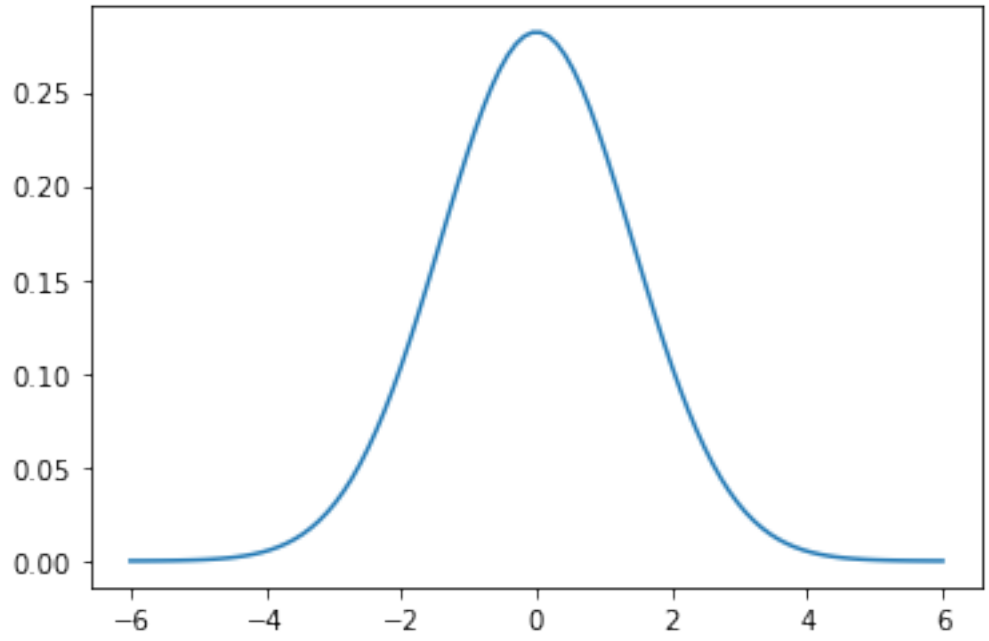
4.1.3 Main distributions

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.

```
mu = 0 # mean
variance = 2 #variance
sigma = np.sqrt(variance) #standard deviation",
x = np.linspace(mu - 3 * variance, mu + 3 * variance, 100)
_ = plt.plot(x, scipy.stats.norm.pdf(x, mu, sigma))
```



The Chi-Square distribution

The chi-square or χ^2_n 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 $Z^2 \sim \chi^2_1$ (one df).
- **The distribution of sum of squares** of n normal random variables: $\sum_i^n Z_i^2 \sim \chi^2_n$

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**.

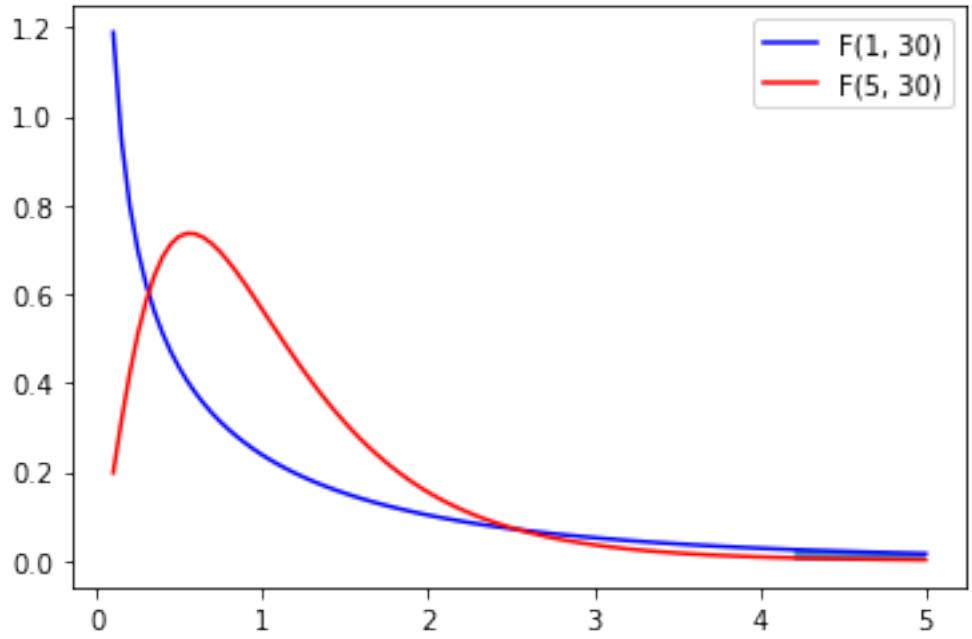
The Fisher's F-distribution

The F-distribution, $F_{n,p}$, with n and p degrees of freedom is the ratio of two independent χ^2 variables. Let $X \sim \chi^2_n$ and $Y \sim \chi^2_p$ 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 ?**.

```
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()

# cdf(x, df1, df2): Cumulative distribution function of F.
# ie.
proba_at_f_inf_3 = scipy.stats.f.cdf(3, 1, 30) # P(F(1,30) < 3)

# ppf(q, df1, df2): Percent point function (inverse of cdf) at q of F.
f_at_proba_inf_95 = scipy.stats.f.ppf(.95, 1, 30) # q such P(F(1,30) < .95)
assert scipy.stats.f.cdf(f_at_proba_inf_95, 1, 30) == .95

# sf(x, df1, df2): Survival function (1 - cdf) at x of F.
proba_at_f_sup_3 = scipy.stats.f.sf(3, 1, 30) # P(F(1,30) > 3)
assert  proba_at_f_inf_3 + proba_at_f_sup_3 == 1

# p-value: P(F(1, 30)) < 0.05
low_proba_fvalues = fvalues[fvalues > f_at_proba_inf_95]
plt.fill_between(low_proba_fvalues, 0, scipy.stats.f.pdf(low_proba_fvalues, 1,␣
↪30),
                 alpha=.8, label="P < 0.05")
plt.show()
```



The Student's t-distribution

Let $M \sim \mathcal{N}(0, 1)$ and $V \sim \chi^2_n$. 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. **Is this parameters different from a given value?**

4.1.4 Hypothesis Testing

Examples

- Test a proportion: Biased coin ? 200 heads have been found over 300 flips, is it coins biased ?
- Test the association between two variables.
  - Exemple height and sex: In a sample of 25 individuals (15 females, 10 males), is female height is different from male height ?
  - Exemple age and arterial hypertension: In a sample of 25 individuals is age height correlated with arterial hypertension ?

Steps

1. Model the data
2. Fit: estimate the model parameters (frequency, mean, correlation, regression coeficient)
3. Compute a test statistic from model the parameters.
4. Formulate the null hypothesis: What would be the (distribution of the) test statistic if the observations are the result of pure chance.
5. Compute the probability (p-value) to obtain a larger value for the test statistic by chance (under the null hypothesis).

Flip coin: Simplified example

Biased coin ? 2 heads have been found over 3 flips, is it coins biased ?

1. Model the data: number of heads follow a Binomial disctribution.
2. Compute model parameters: N=3, P = the frequency of number of heads over the number of flip: 2/3.
3. Compute a test statistic, same as frequency.
4. Under the null hypothesis the distribution of the number of tail is:

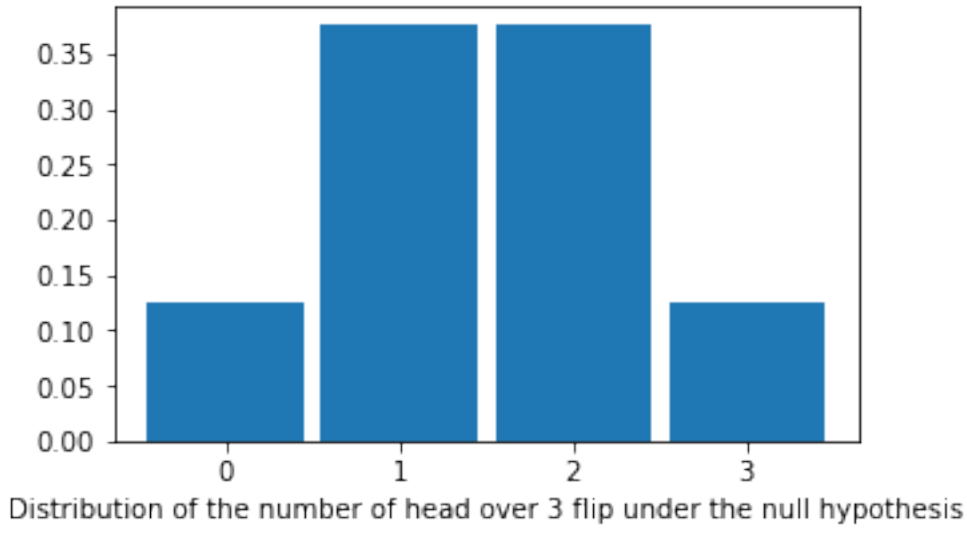
1	2	3	count #heads
			0
H			1
	H		1
		H	1
H	H		2
H		H	2
	H	H	2
H	H	H	3

8 possibles configurations, probabilities of differents values for p are: x measure the number of success.

- $P(x = 0) = 1/8$
- $P(x = 1) = 3/8$
- $P(x = 2) = 3/8$
- $P(x = 3) = 1/8$

```
plt.figure(figsize=(5, 3))
plt.bar([0, 1, 2, 3], [1/8, 3/8, 3/8, 1/8], width=0.9)
_ = plt.xticks([0, 1, 2, 3], [0, 1, 2, 3])
plt.xlabel("Distribution of the number of head over 3 flip under the null␣
↪hypothesis")
```

```
Text(0.5, 0, 'Distribution of the number of head over 3 flip under the null␣
↪hypothesis')
```



3. Compute the probability (p-value) to observe a value larger or equal that 2 under the null

hypothesis ? This probability is the p-value:

$$P(x \geq 2|H_0) = P(x = 2) + P(x = 3) = 3/8 + 1/8 = 4/8 = 1/2$$

Flip coin: Real Example

Biased coin ? 60 heads have been found over 100 flips, is it coins biased ?

1. Model the data: number of heads follow a Binomial disctribution.
2. Compute model parameters: N=100, P=60/100.
3. Compute a test statistic, same as frequency.
4. Compute a test statistic: 60/100.
5. Under the null hypothesis the distribution of the number of tail (k) follow the **binomial distribution** of parameters N=100, **P=0.5**:

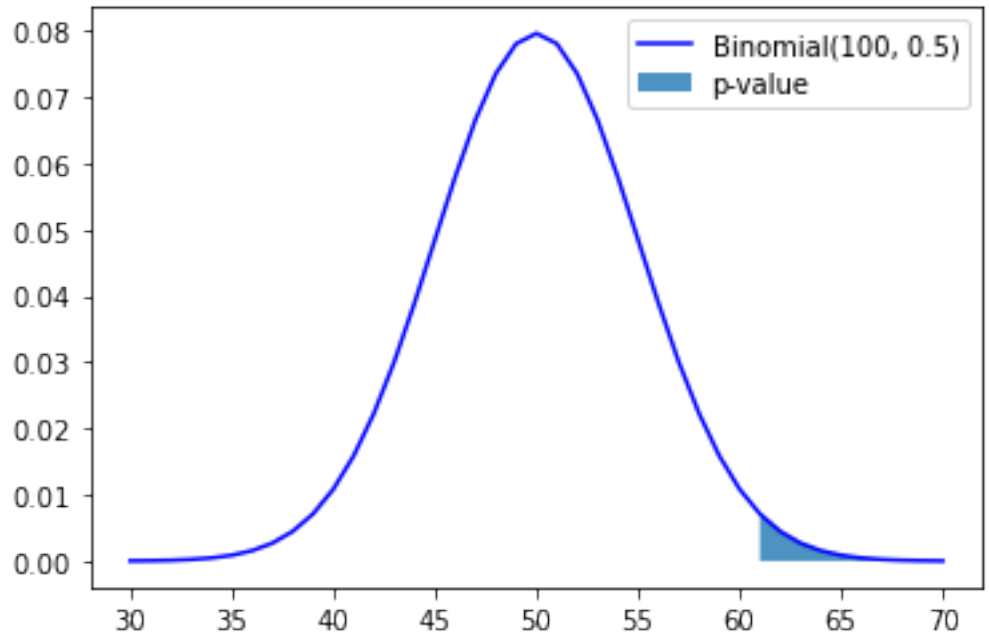
$$Pr(X = k|H_0) = Pr(X = k|n = 100, p = 0.5) = \binom{100}{k} 0.5^k (1 - 0.5)^{(100-k)}.$$

$$\begin{aligned} P(X = k \geq 60|H_0) &= \sum_{k=60}^{100} \binom{100}{k} 0.5^k (1 - 0.5)^{(100-k)} \\ &= 1 - \sum_{k=1}^{60} \binom{100}{k} 0.5^k (1 - 0.5)^{(100-k)}, \text{the cumulative distribution function.} \end{aligned}$$

Use tabulated binomial distribution

```
succes = np.linspace(30, 70, 41)
plt.plot(succes, scipy.stats.binom.pmf(succes, 100, 0.5),
         'b-', label="Binomial(100, 0.5)")
upper_succes_tvalues = succes[succes > 60]
plt.fill_between(upper_succes_tvalues, 0,
                 scipy.stats.binom.pmf(upper_succes_tvalues, 100, 0.5),
                 alpha=.8, label="p-value")
_ = plt.legend()


pval = 1 - scipy.stats.binom.cdf(60, 100, 0.5)
print(pval)
```

```
0.01760010010885238
```



Random sampling of the Binomial distribution under the null hypothesis

```
sccess_h0 = scipy.stats.binom.rvs(100, 0.5, size=10000, random_state=4)
print(sccess_h0)

pval_rnd = np.sum(sccess_h0 >= 60) / (len(sccess_h0) + 1)
print("P-value using monte-carlo sampling of the Binomial distribution under H0=",
      pval_rnd)
```

```
[60 52 51 ... 45 51 44]
P-value using monte-carlo sampling of the Binomial distribution under H0= 0.
↪025897410258974102
```

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 $1.75\ m$.

Assumptions

1. Independence of **residuals** (ε_i). This assumptions **must** be satisfied.
2. Normality of residuals. Approximately normally distributed can be accepted.

Remarks: Although the parent population does not need to be normally distributed, the distribution of the population of sample means, $\overline{x}$, is assumed to be normal. By the central limit theorem, if the sampling of the parent population is independent then the sample means will be approximately normal.

1 Model the data

Assume that height is normally distributed: $X \sim \mathcal{N}(\mu, \sigma)$, ie:

$$\text{height}_i = \text{average height over the population} + \text{error}_i \tag{4.11}$$
$$x_i = \bar{x} + \varepsilon_i \tag{4.12}$$

The ε_i are called the residuals

2 Fit: estimate the model parameters

$\bar{x}, s_x$ are the estimators of μ, σ.

3 Compute a test statistic

In testing the null hypothesis that the population mean is equal to a specified value $\mu_0 = 1.75$, one uses the statistic:

$$t = \frac{\text{difference of means}}{\text{std-dev of noise}}\sqrt{n} \tag{4.13}$$
$$t = \text{effect size}\sqrt{n} \tag{4.14}$$
$$t = \frac{\bar{x} - \mu_0}{s_x}\sqrt{n} \tag{4.15}$$

4 Compute the probability of the test statistic under the null hypotheis. This require to have the distribution of the t statistic under H_0.

Example

Given the following samples, we will test whether its true mean is 1.75.

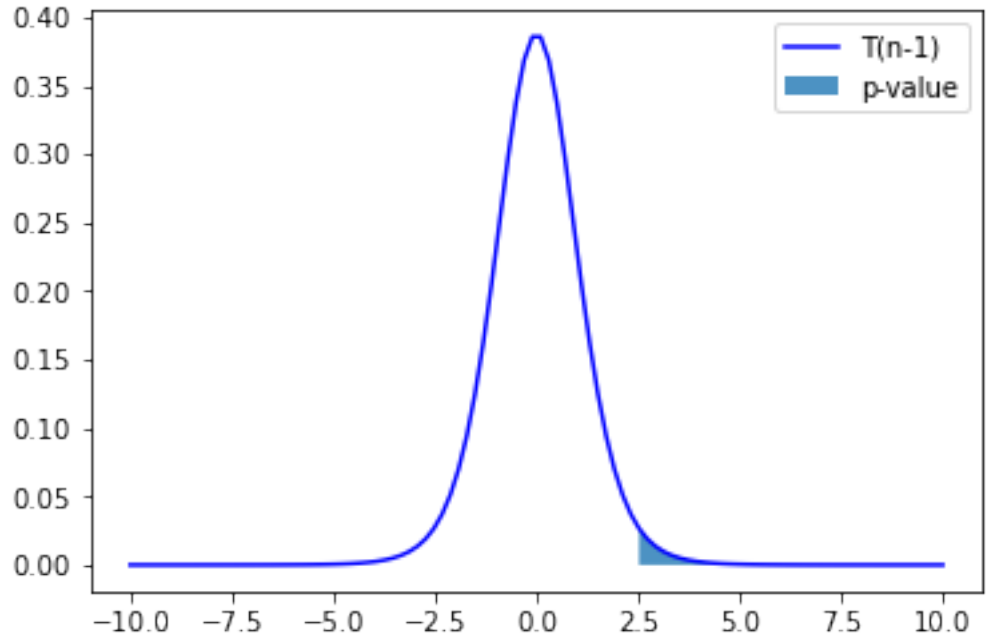
Warning, when computing the std or the variance, set `ddof=1`. The default value, `ddof=0`, leads to the biased estimator of the variance.

```
x = [1.83,  1.83,  1.73,  1.82,  1.83,  1.73,  1.99,  1.85,  1.68,  1.87]

xbar = np.mean(x)  # sample mean
mu0 = 1.75  # hypothesized value
s = np.std(x, ddof=1)  # sample standard deviation
n = len(x)  # sample size

print(xbar)

tobs = (xbar - mu0) / (s / np.sqrt(n))
print(tobs)
```

```
1.816
2.3968766311585883
```

The **:math:`p`-value** is the probability to observe a value t more extreme than the observed one t_{obs} under the null hypothesis H_0: $P(t > t_{obs}|H_0)$

```
tvalues = np.linspace(-10, 10, 100)
plt.plot(tvalues, scipy.stats.t.pdf(tvalues, n-1), 'b-', label="T(n-1)")
upper_tval_tvalues = tvalues[tvalues > tobs]
plt.fill_between(upper_tval_tvalues, 0, scipy.stats.t.pdf(upper_tval_tvalues, n-
↪1),
                 alpha=.8, label="p-value")
_ = plt.legend()
```



4.1.5 Testing pairwise associations

Univariate statistical analysis: explore association betweens 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 possibles 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?

See: http://www.ats.ucla.edu/stat/mult_pkg/whatstat/

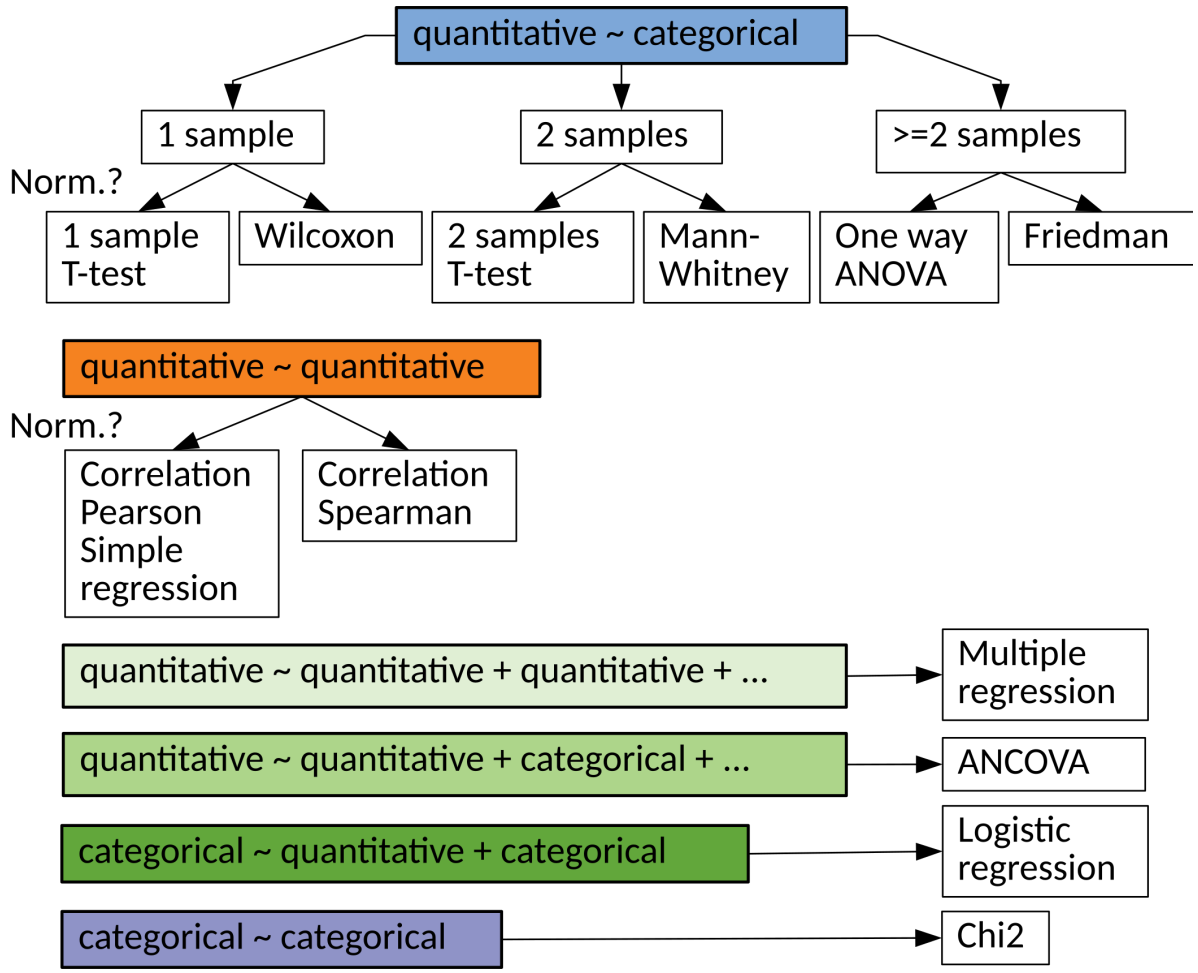


Fig. 1: Statistical tests

4.1.6 Pearson correlation test: 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 obeserved. The linear correlation coeficient 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.

```
n = 50
x = np.random.normal(size=n)
y = 2 * x + np.random.normal(size=n)

# Compute with scipy
```

(continues on next page)

(continued from previous page)

```
cor, pval = scipy.stats.pearsonr(x, y)
print(cor, pval)
```

```
0.8297883544365898 9.497428029783463e-14
```

4.1.7 Two sample (Student) t-test: compare two means

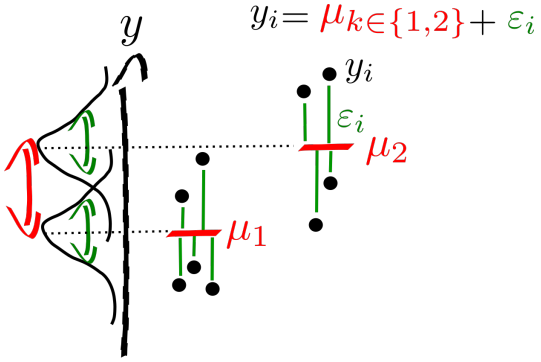


Fig. 2: Two-sample model

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).

Assumptions

1. Independence of **residuals** (ε_i). This assumptions **must** be satisfied.
2. Normality of residuals. Approximately normally distributed can be accepted.
3. Homosedasticity use T-test, Heterosedasticity use Welch t-test.

1. Model the data

Assume that the two random variables are normally distributed: $y_1 \sim \mathcal{N}(\mu_1, \sigma_1), y_2 \sim \mathcal{N}(\mu_2, \sigma_2)$.

2. Fit: estimate the model parameters

Estimate means and variances: $\bar{y_1}, s^2_{y_1}, \bar{y_2}, s^2_{y_2}$.

3. t-test

The general principle is

$$t = \frac{\text{difference of means}}{\text{standard dev of error}} \tag{4.16}$$

$$= \frac{\text{difference of means}}{\text{its standard error}} \tag{4.17}$$

$$= \frac{\bar{y_1} - \bar{y_2}}{\sqrt{\sum \varepsilon^2}}\sqrt{n-2} \tag{4.18}$$

$$= \frac{\bar{y_1} - \bar{y_2}}{s_{\bar{y_1}-\bar{y_2}}} \tag{4.19}$$

Since y_1 and y_2 are independant:

$$s^2_{\bar{y_1}-\bar{y_2}} = s^2_{\bar{y_1}} + s^2_{\bar{y_2}} = \frac{s^2_{y_1}}{n_1} + \frac{s^2_{y_2}}{n_2} \tag{4.20}$$

$$\text{thus} \tag{4.21}$$

$$s_{\bar{y_1}-\bar{y_2}} = \sqrt{\frac{s^2_{y_1}}{n_1} + \frac{s^2_{y_2}}{n_2}} \tag{4.22}$$

Equal or unequal sample sizes, unequal variances (Welch's t-test)

Welch's t-test defines the t statistic as

$$t = \frac{\bar{y_1} - \bar{y_2}}{\sqrt{\frac{s^2_{y_1}}{n_1} + \frac{s^2_{y_2}}{n_2}}}.$$

To compute the p-value one needs the degrees of freedom associated with this variance estimate. It is approximated using the Welch–Satterthwaite equation:

$$\nu \approx \frac{\left(\frac{s^2_{y_1}}{n_1} + \frac{s^2_{y_2}}{n_2}\right)^2}{\frac{s^4_{y_1}}{n_1^2(n_1-1)} + \frac{s^4_{y_2}}{n_2^2(n_2-1)}}.$$

Equal or unequal sample sizes, equal variances

If we assume equal variance (ie, $s^2_{y_1} = s^2_{y_1} = s^2$), where s^2 is an estimator of the common variance of the two samples:

$$s^2 = \frac{s^2_{y_1}(n_1 - 1) + s^2_{y_2}(n_2 - 1)}{n_1 + n_2 - 2} \tag{4.23}$$

$$= \frac{\sum_i^{n_1}(y_{1i} - \bar{y_1})^2 + \sum_j^{n_2}(y_{2j} - \bar{y_2})^2}{(n_1 - 1) + (n_2 - 1)} \tag{4.24}$$

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 $n_1 = n_2 = n$ we get

$$t = \frac{\bar{y_1} - \bar{y_2}}{s\sqrt{2}} \cdot \sqrt{n} \tag{4.25}$$

$$\approx \text{effect size} \cdot \sqrt{n} \tag{4.26}$$

$$\approx \frac{\text{difference of means}}{\text{standard deviation of the noise}} \cdot \sqrt{n} \tag{4.27}$$

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
scipy.stats.ttest_ind(height[grp == "M"], height[grp == "F"], equal_var=True)
```

```
Ttest_indResult(statistic=3.5511519888466885, pvalue=0.00228208937112721)
```

4.1.8 ANOVA F-test (quantitative ~ categorial (>=2 levels))

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.

Wikipedia:

- 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.

Assumptions

1. The samples are randomly selected in an independent manner from the k populations.
2. All k populations have distributions that are approximately normal. Check by plotting groups distribution.
3. The k population variances are equal. Check by plotting groups distribution.

1. Model the data

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: $\bar{y}_i, \sigma_i, \quad \forall i \in \{1, 2, 3\}$.

3. F-test

The formula for the one-way ANOVA F-test statistic is

$$F = \frac{\text{Explained variance}}{\text{Unexplained variance}} \tag{4.28}$$

$$= \frac{\text{Between-group variability}}{\text{Within-group variability}} = \frac{s_B^2}{s_W^2}. \tag{4.29}$$

The "explained variance", or "between-group variability" is

$$s_B^2 = \sum_i n_i(\bar{y}_{i\cdot} - \bar{y})^2/(K-1),$$

where $\bar{y}_{i\cdot}$ denotes the sample mean in the ith group, n_i is the number of observations in the ith 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

$$s_W^2 = \sum_{ij}(y_{ij} - \bar{y}_{i\cdot})^2/(N-K),$$

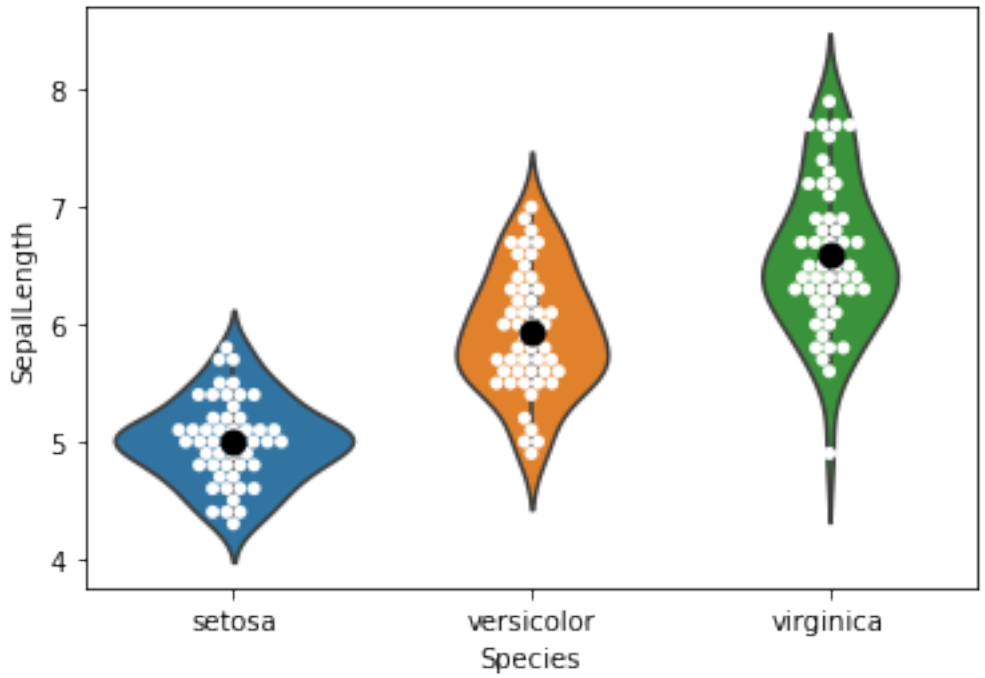
where y_{ij} is the jth observation in the ith out of K groups and N is the overall sample size. This F-statistic follows the F-distribution with $K-1$ and $N-K$ degrees of freedom under the null hypothesis. The statistic will be large if the between-group variability is large relative to the within-group 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.

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
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
```



4.1.9 Chi-square, χ^2 (categorial ~ categorial)

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:")
```

(continues on next page)

(continued from previous page)

```
print("---------------")
print(crosstab)

chi2, pval, dof, expected = scipy.stats.chi2_contingency(crosstab)
print("Statistics:")
print("-----------")
print("Chi2 = %f, pval = %f" % (chi2, pval))
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]
```

(continues on next page)

(continued from previous page)

```
 [0.25 0.25]]
Expected cross-table (frequencies * N):
[[5. 5.]
 [5. 5.]]
```

4.1.10 Non-parametric test of pairwise associations

Spearman rank-order correlation (quantitative ~ quantitative)

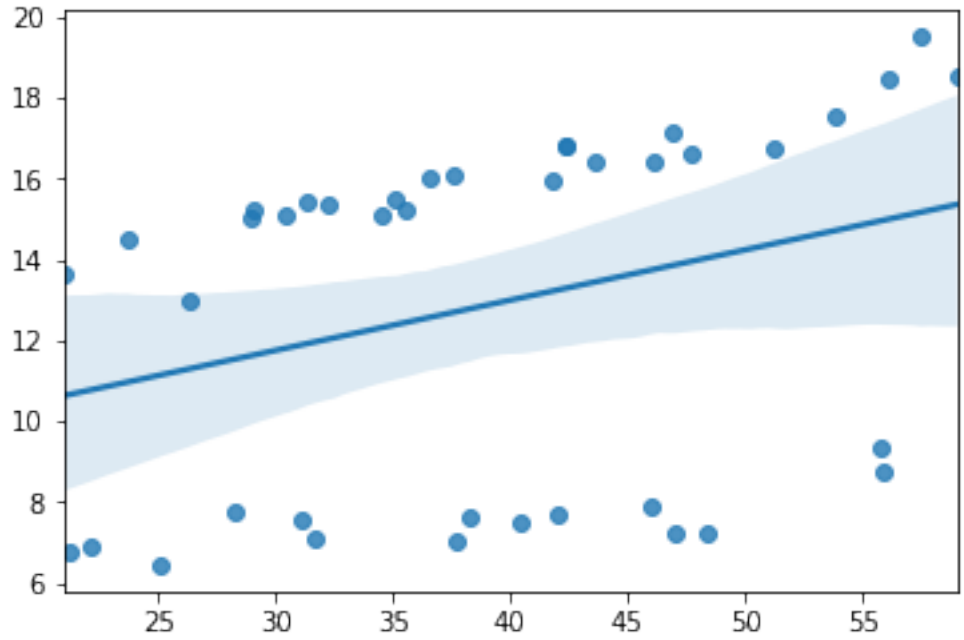
The Spearman correlation is a non-parametric measure of the monotonicity of the relationship between two datasets.

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 presure, 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 ~ cte)

Source: https://en.wikipedia.org/wiki/Wilcoxon_signed-rank_test

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))
```

(continues on next page)

(continued from previous page)

```
# Wilcoxon
print(scipy.stats.wilcoxon(bv0, bv1))
```

```
Ttest_relResult(statistic=0.7766377807752968, pvalue=0.44693401731548044)
WilcoxonResult(statistic=23.0, pvalue=0.001209259033203125)
```

Mann–Whitney U test (quantitative ~ categorial (2 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
# Buismess Volume group 0
bv0 = np.random.normal(loc=1, scale=.1, size=n)

# Buismess Volume group 1
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))
```

```
Ttest_indResult(statistic=0.6104564820307219, pvalue=0.5451934484051324)
MannwhitneyuResult(statistic=41.0, pvalue=9.037238869417781e-06)
```

4.1.11 Linear model

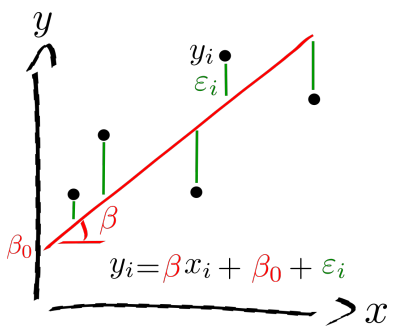


Fig. 3: Linear model

Given n random samples $(y_i, x_{1i}, \ldots, x_{pi})$, $i = 1, \ldots, n$, the linear regression models the relation between the observations y_i and the independent variables x_i^p is formulated as

$$y_i = \beta_0 + \beta_1 x_{1i} + \cdots + \beta_p x_{pi} + \varepsilon_i \qquad i = 1, \ldots, n$$

- The β's are the model parameters, ie, the regression coeficients.
- 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 independant 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 ith 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 it so 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:

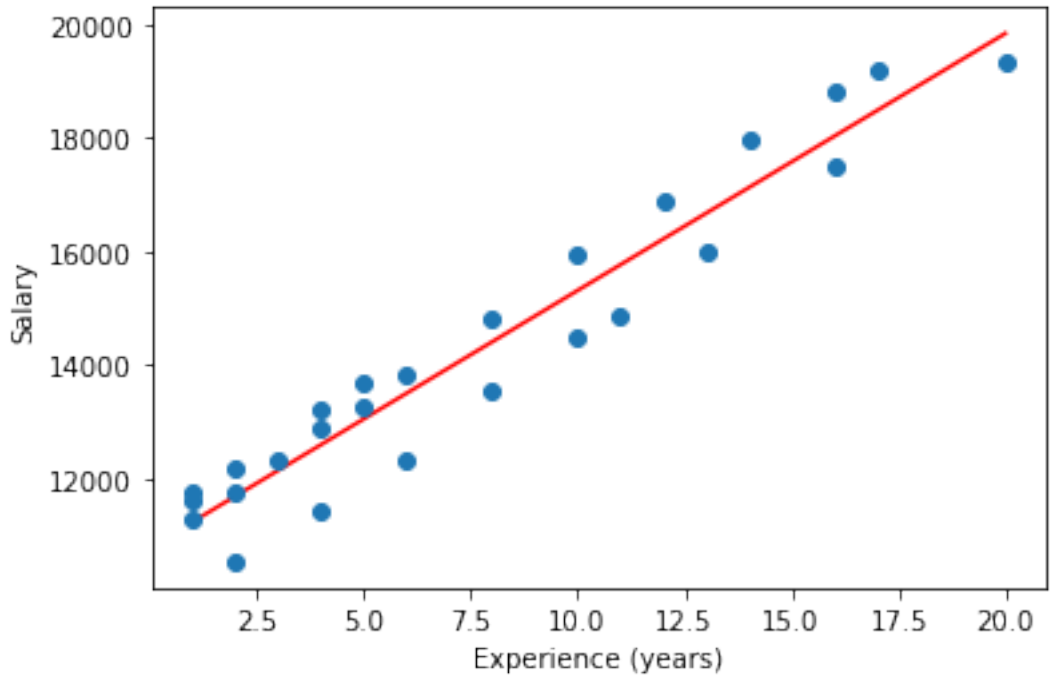
$$\begin{aligned} \sum_i x_i (y_i - \beta\, x_i - \bar{y} + \beta \bar{x}) &= 0 \\ \sum_i x_i y_i - \bar{y} \sum_i x_i &= \beta \sum_i (x_i - \bar{x}) \end{aligned}$$

Divide both sides by n:

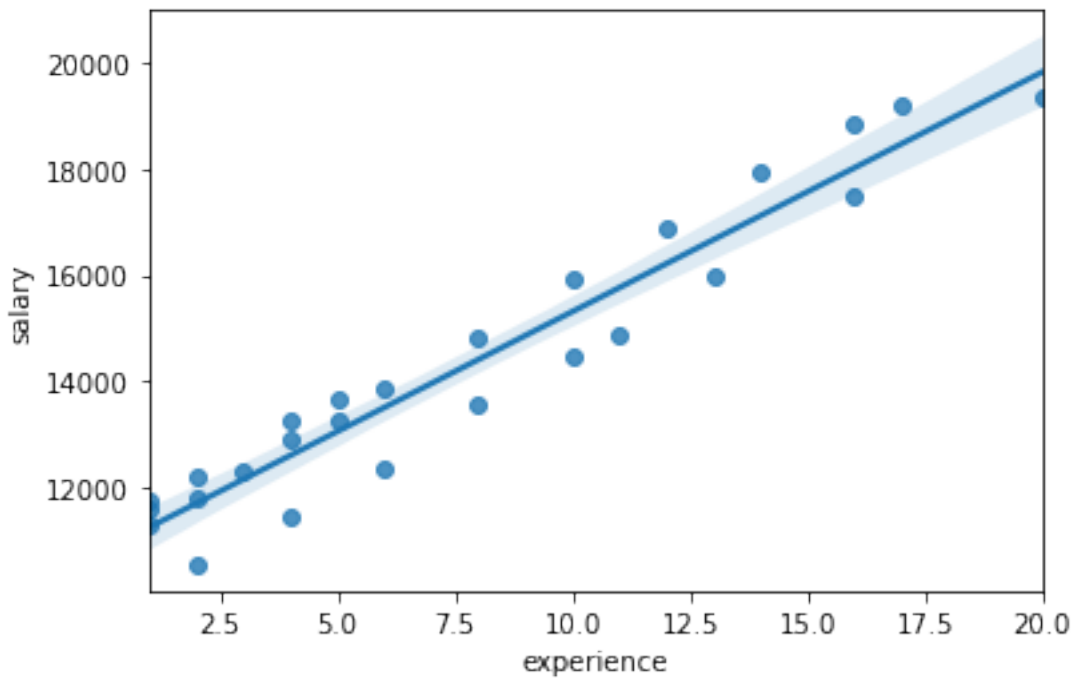
$$\begin{aligned} \frac{1}{n} \sum_i x_i y_i - \bar{y}\bar{x} &= \frac{1}{n} \beta \sum_i (x_i - \bar{x}) \\ \beta &= \frac{\frac{1}{n} \sum_i x_i y_i - \bar{y}\bar{x}}{\frac{1}{n} \sum_i (x_i - \bar{x})} = \frac{Cov(x, y)}{Var(x)}. \end{aligned}$$

```
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,\np-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()

print("Using seaborn")
ax = sns.regplot(x="experience", y="salary", data=df)
```

```
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
```




```
Using seaborn
```



Multiple regression

Theory

Muliple Linear Regression is the most basic supervised learning algorithm.

Given: a set of training data $\{x_1, ..., x_N\}$ with corresponding targets $\{y_1, ..., y_N\}$.

In linear regression, we assume that the model that generates the data involves only a linear combination of the input variables, i.e.

$$y_i = \beta_0 + \beta_1 x_{i1} + ... + \beta_P x_{iP} + \varepsilon_i,$$

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:

$$y_i = \mathbf{x}_i^T \beta + \varepsilon_i,$$

where $\beta \in R^{P+1}$ is a vector of weights that define the $P + 1$ parameters of the model. From now we have P regressors + the intercept.

Using the matrix notation:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & \dots & x_{1P} \\ 1 & x_{21} & \dots & x_{2P} \\ 1 & x_{31} & \dots & x_{3P} \\ 1 & x_{41} & \dots & x_{4P} \\ 1 & x_5 & \dots & x_5 \end{bmatrix} \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_P \end{bmatrix} + \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \epsilon_3 \\ \epsilon_4 \\ \epsilon_5 \end{bmatrix}$$

Let $X = [x_0^T, ..., x_N^T]$ be the $(N \times P + 1)$ **design matrix** of N samples of P input features with one column of one and let be $y = [y_1, ..., y_N]$ be a vector of the N targets.

$$y = X\beta + \varepsilon$$

Minimize the Mean Squared Error MSE loss:

$$MSE(\beta) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \mathbf{x}_i^T\beta)^2$$

Using the matrix notation, the **mean squared error (MSE) loss can be rewritten**:

$$MSE(\beta) = \frac{1}{N}||y - X\beta||_2^2.$$

The β that minimises the MSE can be found by:

$$\nabla_\beta \left(\frac{1}{N}||y - X\beta||_2^2\right) = 0 \tag{4.30}$$

$$\frac{1}{N}\nabla_\beta (y - X\beta)^T(y - X\beta) = 0 \tag{4.31}$$

$$\frac{1}{N}\nabla_\beta (y^T y - 2\beta^T X^T y + \beta^T X^T X\beta) = 0 \tag{4.32}$$

$$-2X^T y + 2X^T X\beta = 0 \tag{4.33}$$

$$X^T X\beta = X^T y \tag{4.34}$$

$$\beta = (X^T X)^{-1} X^T y, \tag{4.35}$$

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.pinv2(X)
betahat = np.dot(Xpinv, y)
print("Estimated beta:\n", betahat)
```

```
Estimated beta:
 [10.14742501  0.57938106  0.51654653  0.17862194]
```

4.1.12 Linear model with statsmodels

Sources: http://statsmodels.sourceforge.net/devel/examples/

Multiple regression

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:                Fri, 08 Jan 2021   Prob (F-statistic):           0.000106
Time:                        15:05:47   Log-Likelihood:                -71.271
No. Observations:                  50   AIC:                             150.5
Df Residuals:                      46   BIC:                             158.2
Df Model:                           3
Covariance Type:            nonrobust
```

(continues on next page)

(continued from previous page)

```
==============================================================================
                 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
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

Statsmodels with 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 ~ 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:                Fri, 08 Jan 2021   Prob (F-statistic):           0.000106
Time:                        15:05:47   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      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
```

(continues on next page)

(continued from previous page)

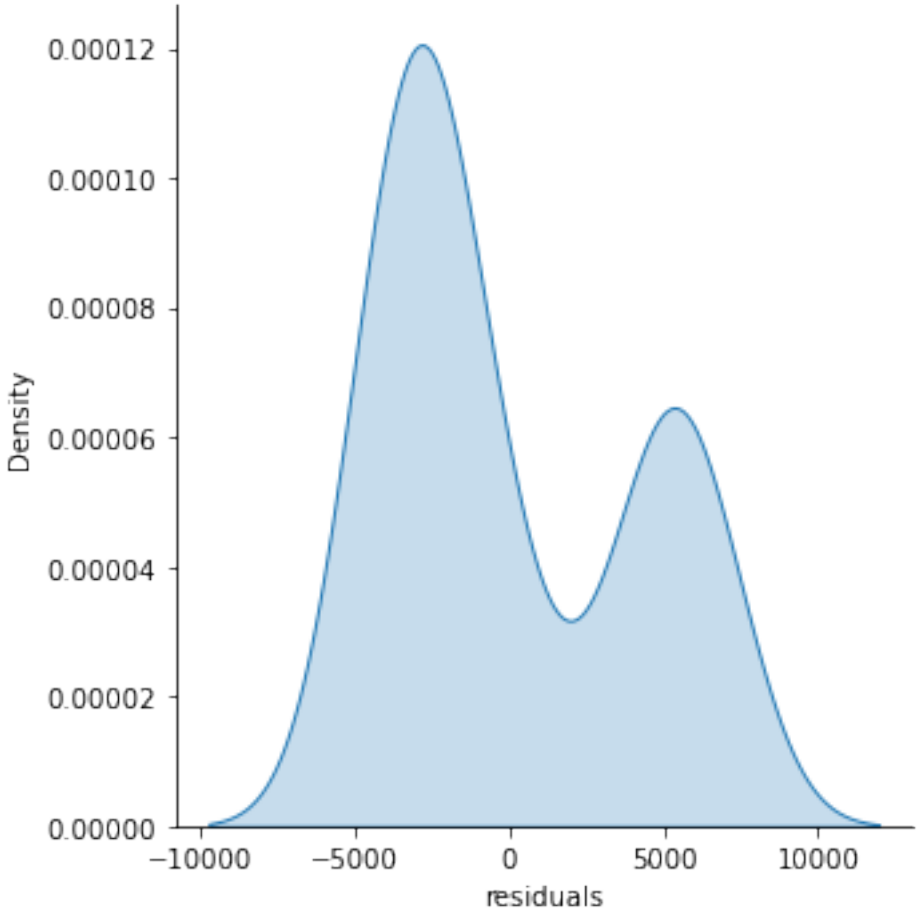
```
x3             0.1786      0.144      1.240      0.221      -0.111       0.469
==============================================================================
Omnibus:                        2.493   Durbin-Watson:                   2.369
Prob(Omnibus):                  0.288   Jarque-Bera (JB):                1.544
Skew:                           0.330   Prob(JB):                        0.462
Kurtosis:                       3.554   Cond. No.                         1.27
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
```

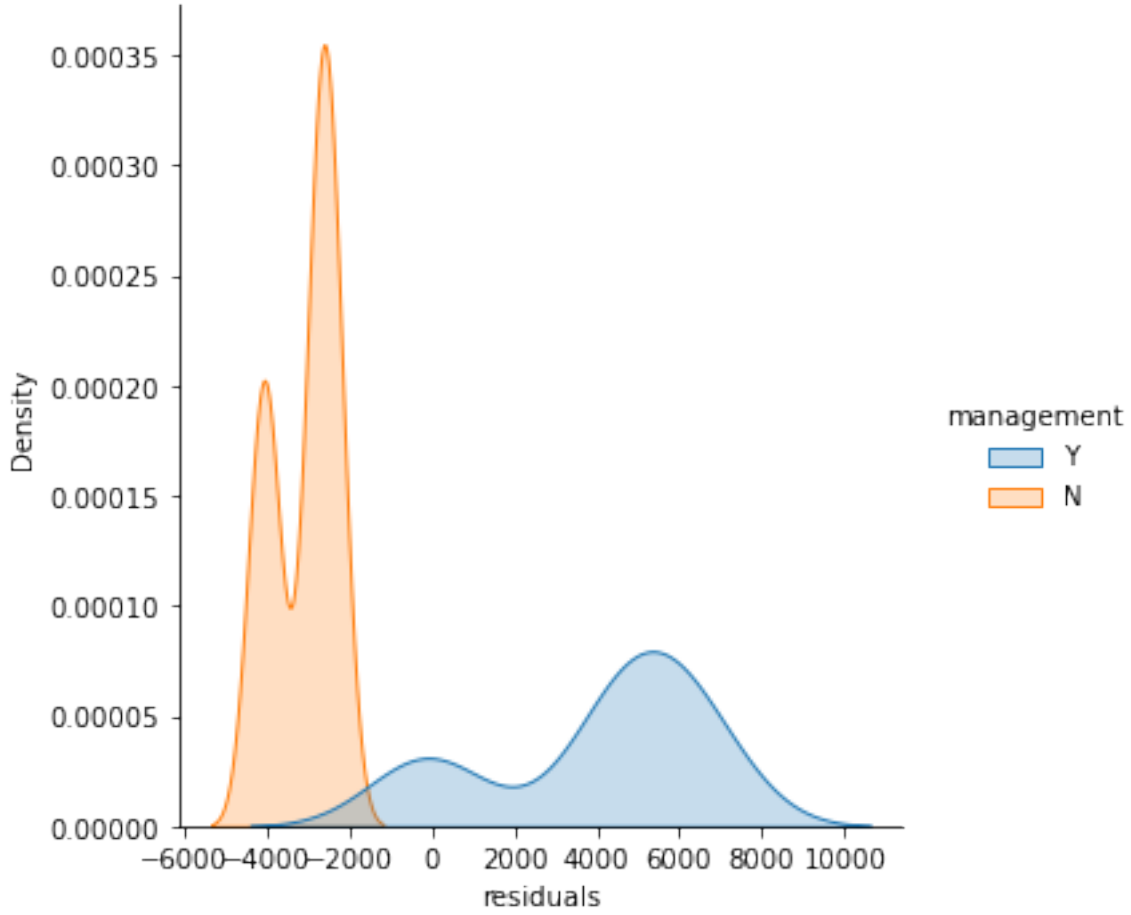
Multiple regression with categorical independent variables or factors: Analysis of covariance (ANCOVA)

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)
ax = sns.displot(df, x='residuals', kind="kde", hue='management', fill=True)
```

```
Jarque-Bera normality test p-value 0.04374
```


0.00012
0.00010
0.00008
0.00006
0.00004
0.00002
0.00000
Density
−10000
−5000
0
5000
10000
residuals



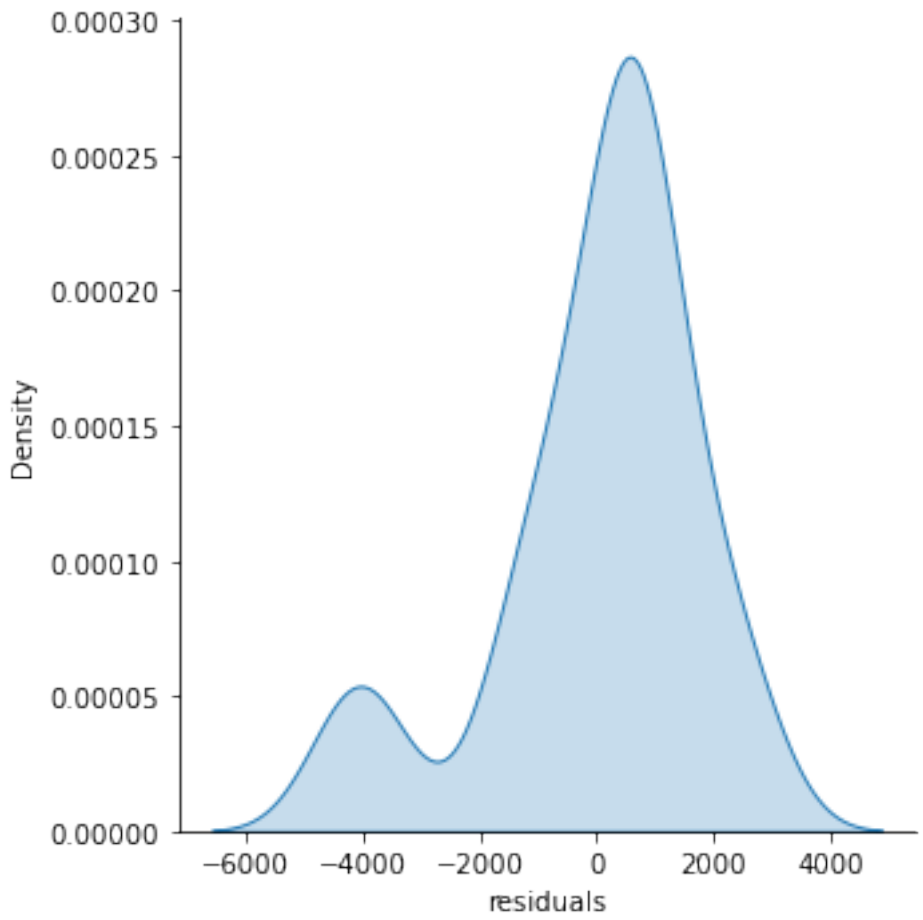
Normality assumption of the residuals can be rejected (p-value < 0.05). There is an efect of the "management" factor, take it into account.

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)
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
```



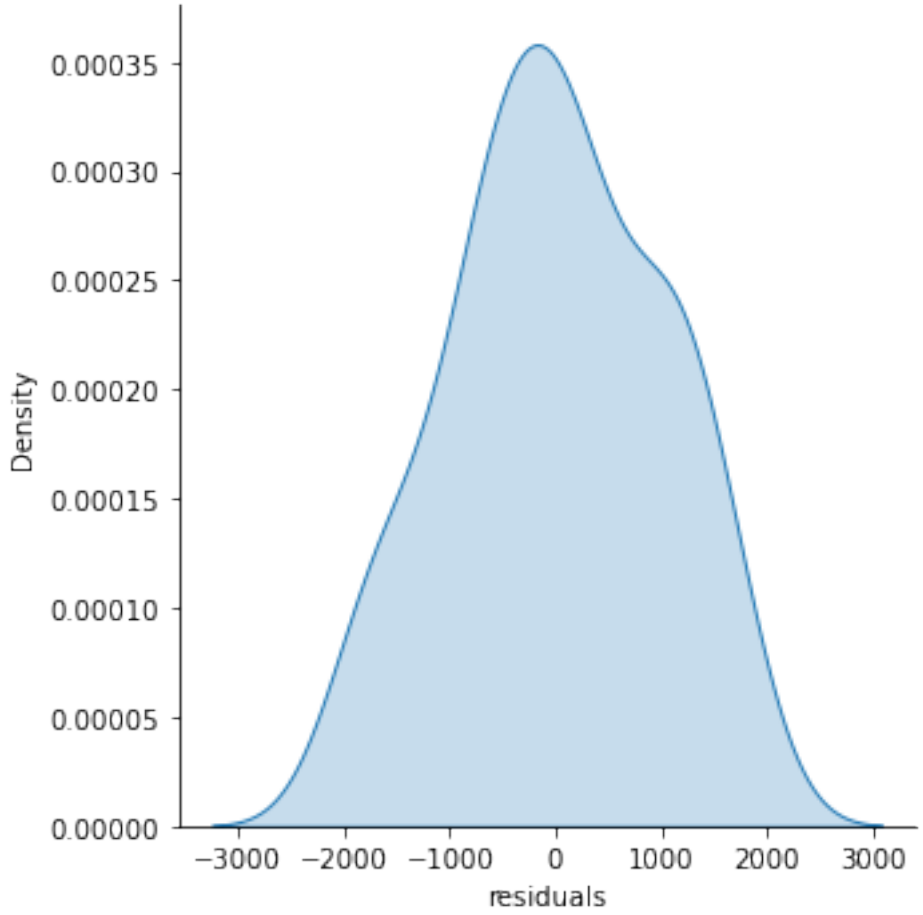
Distribution of residuals is still not normal but closer to normality. Both management and experience are significantly associated with salary.

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)
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 assumtion 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
```

```
(43.35158945918107, 7.672449570495418e-11, 2.0)
```

twoway is significantly better than one way

Factor coding

See http://statsmodels.sourceforge.net/devel/contrasts.html

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.]
 [1. 0. 1. 0. 2.]
 [1. 1. 0. 0. 3.]]
```

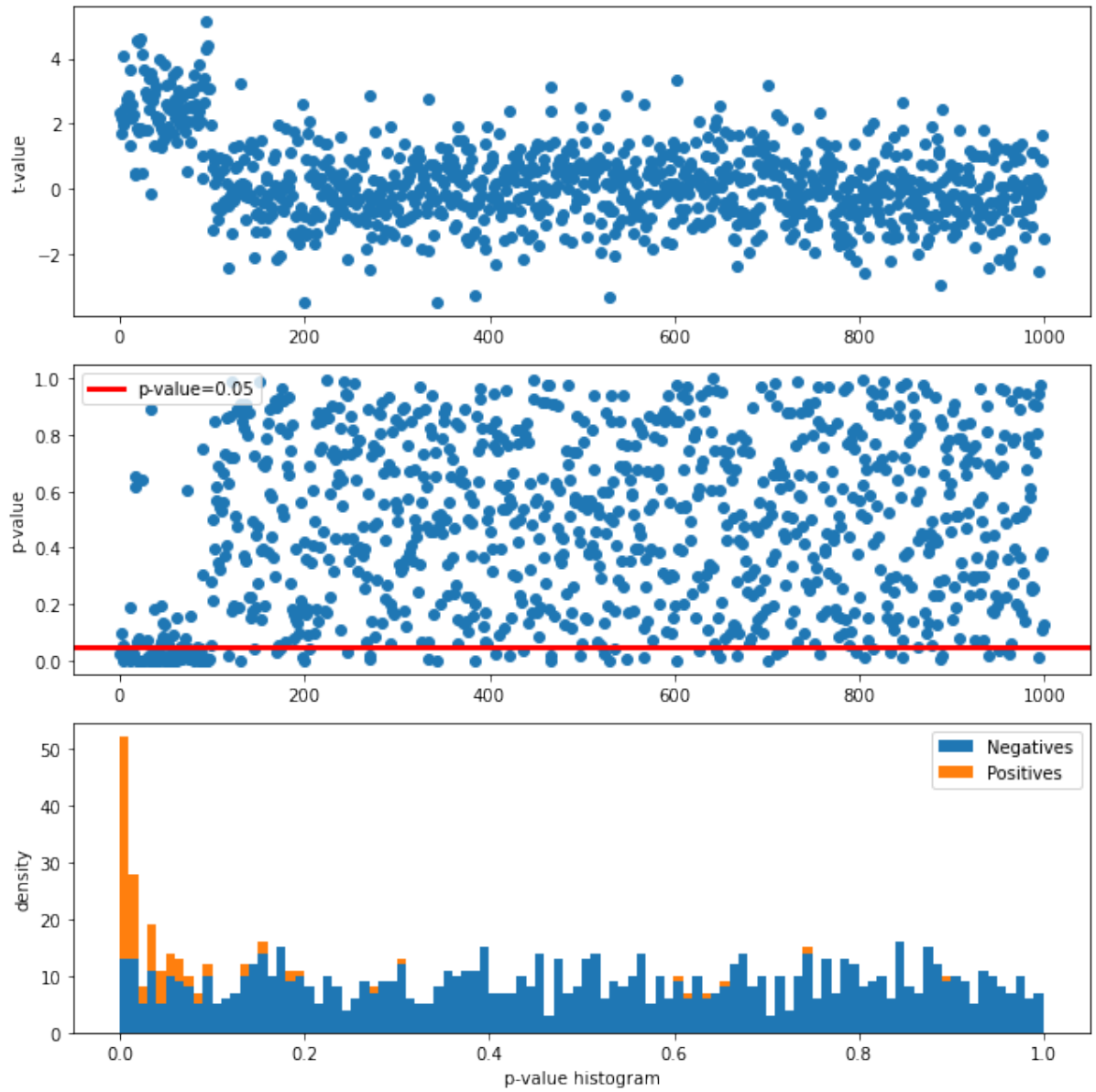
Contrasts and post-hoc tests

```
# 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
==============================================================================
```

4.1.13 Multiple comparisons

```
np.random.seed(seed=42)  # make example reproducible

# Dataset
n_samples, n_features = 100, 1000
n_info = int(n_features/10)  # number of features with information
n1, n2 = int(n_samples/2), n_samples - int(n_samples/2)
snr = .5
Y = np.random.randn(n_samples, n_features)
grp = np.array(["g1"] * n1 + ["g2"] * n2)

# Add some group effect for Pinfo features
Y[grp=="g1", :n_info] += snr

#
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):
    tvals[j], pvals[j] = stats.ttest_ind(Y[grp=="g1", j], Y[grp=="g2", j],
                                         equal_var=True)

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_info:], pvals[:n_info]],
    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-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))
```

```
No correction, FP: 47 (expected: 45.00), TP: 71
```

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 familywise 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))
```

```
FWER correction, FP: 0, TP: 6
```

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))
```

```
FDR correction, FP: 3, TP: 20
```

4.2 Lab: 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.

4.2.1 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.

```
import os
import os.path
import pandas as pd
import tempfile
import urllib.request

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)
```

4.2.2 Descriptive Statistics

Load excel file *brain_vol.xlsx*

```
import os
import pandas as pd
import seaborn as sns
import statsmodels.formula.api as smfrmla
import statsmodels.api as sm

brain_vol = pd.read_excel(os.path.join(WD, "data", "brain_vol.xlsx"),
                          sheet_name='data')
```

(continues on next page)

(continued from previous page)

```
# 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
```

(continues on next page)

(continued from previous page)

```
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
```

4.2.3 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.

```
import statsmodels.api as sm
import statsmodels.formula.api as smfrmla
import scipy.stats
import seaborn as sns
```

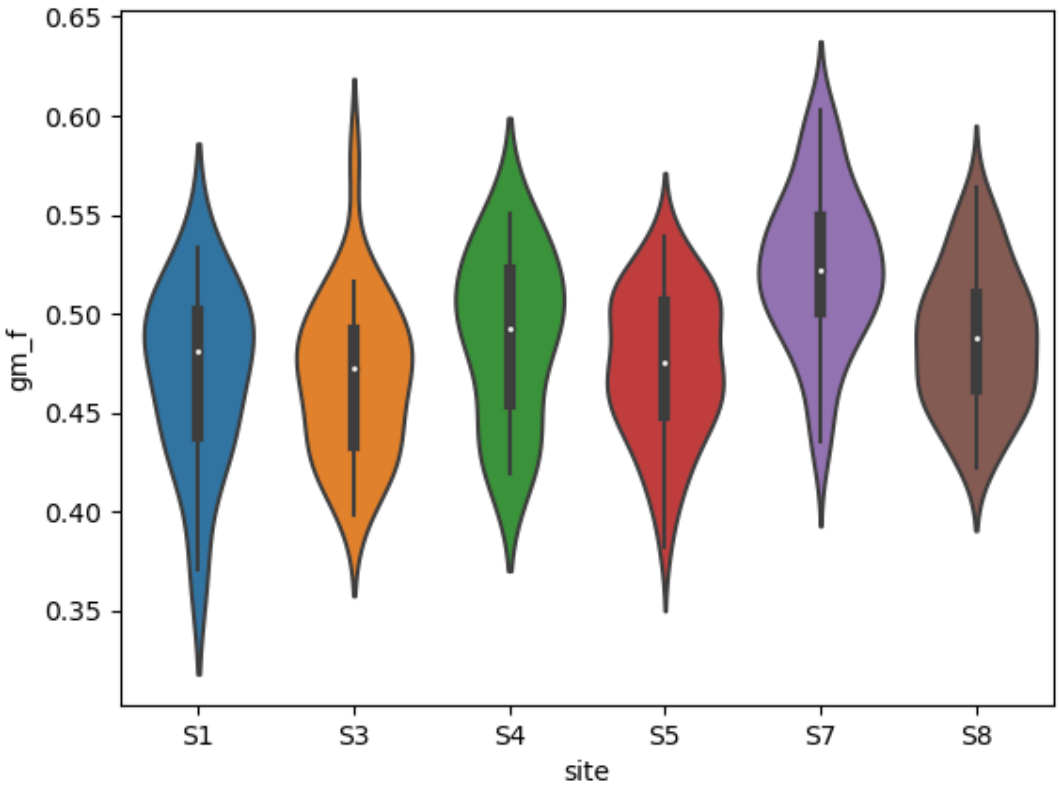
1 Site effect on Grey Matter atrophy

The model is Oneway Anova gm_f ~ 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)
```




```
<Axes: xlabel='site', ylabel='gm_f'>
```

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

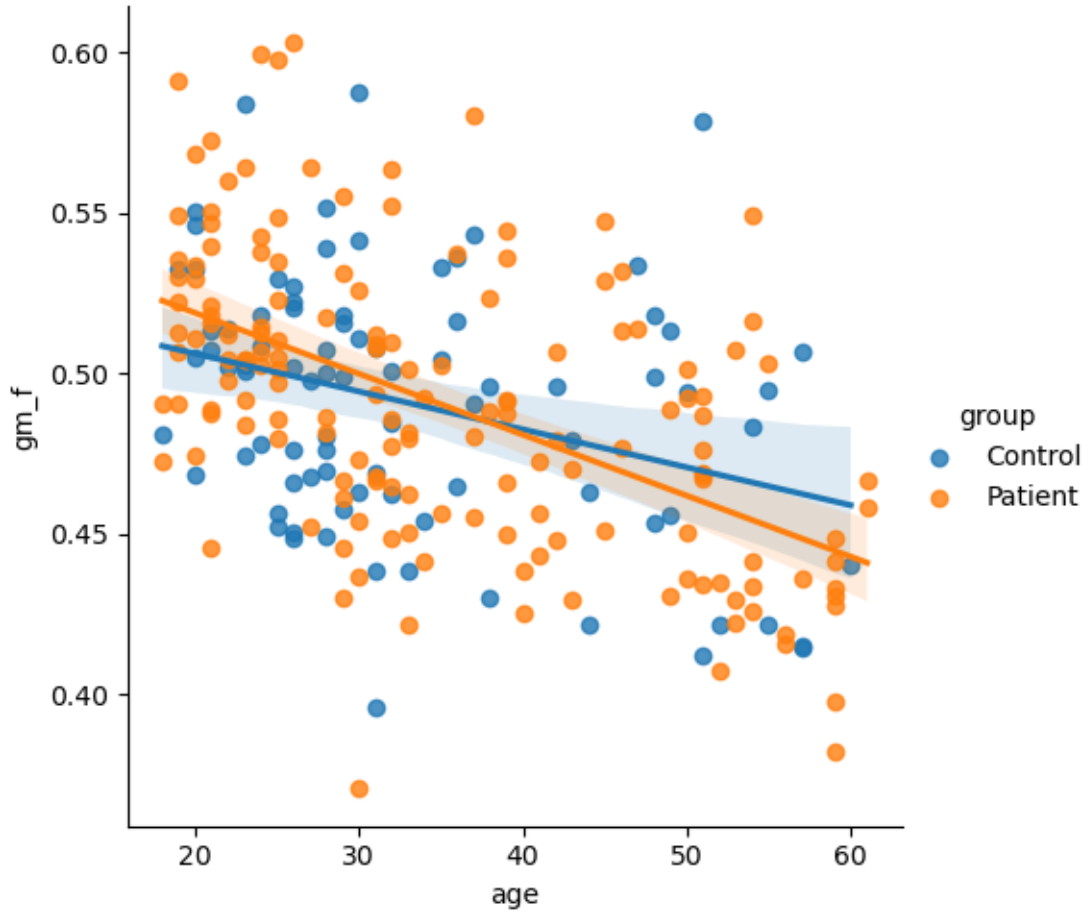
```
anova = smfrmla.ols("gm_f ~ site", data=brain_vol1).fit()
# print(anova.summary())
print("Site explains %.2f%% of the grey matter fraction variance" %
      (anova.rsquared * 100))

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"]
```




```
/home/ed203246/anaconda3/lib/python3.11/site-packages/seaborn/axisgrid.py:118:␣
↪UserWarning: The figure layout has changed to tight
  self._figure.tight_layout(*args, **kwargs)
```

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
--- 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 = smfrmla.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 = smfrmla.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., 28 mars 2024   Prob (F-statistic):            0.00226
Time:                        13:40:17   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
```

(continues on next page)

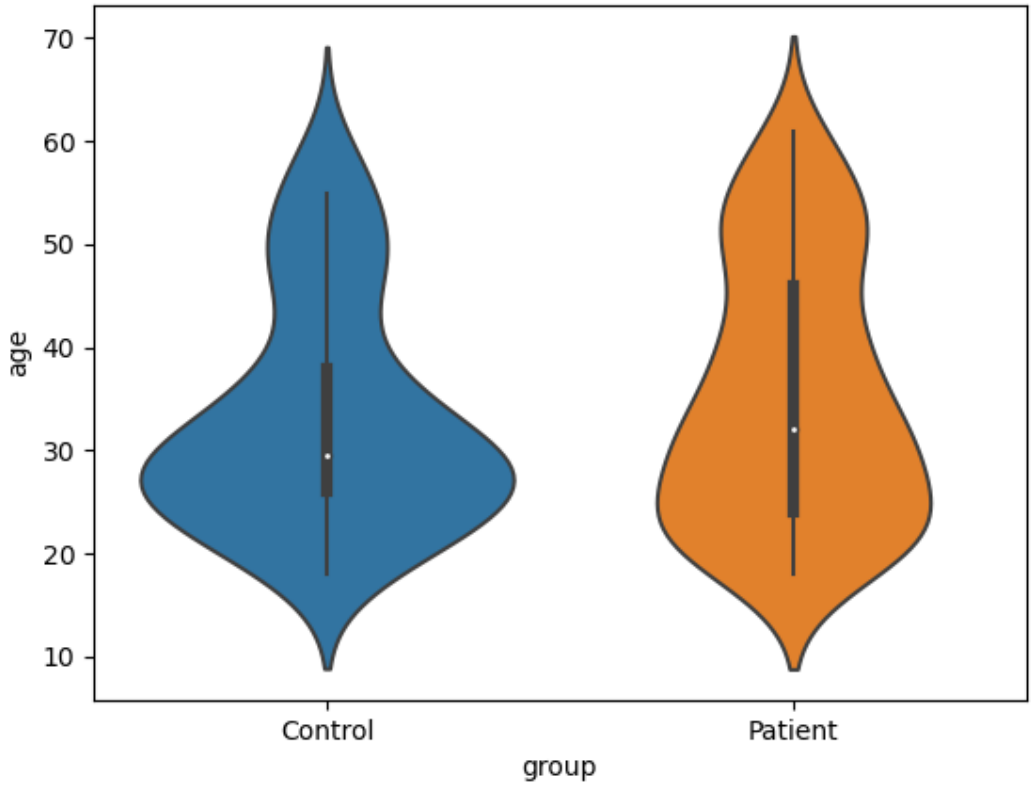
(continued from previous page)

```
--- In patient population ---
                            OLS Regression Results
==============================================================================
Dep. Variable:                    gm_f   R-squared:                       0.280
Model:                            OLS   Adj. R-squared:                  0.275
Method:                 Least Squares   F-statistic:                     60.16
Date:                jeu., 28 mars 2024   Prob (F-statistic):           1.09e-12
Time:                        13:40:17   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 ans 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=-1.2155557697674162, pvalue=0.225343592508479, df=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., 28 mars 2024   Prob (F-statistic):              0.225
Time:                        13:40:17   Log-Likelihood:                -949.69
No. Observations:                 243   AIC:                             1903.
Df Residuals:                     241   BIC:                             1910.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
```

(continues on next page)

(continued from previous page)

```
                      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("Obeserved contingency table")
print(crosstab)

chi2, pval, dof, expected = scipy.stats.chi2_contingency(crosstab)

print("Chi2 = %f, pval = %f" % (chi2, pval))
print("No significant difference in sex between patients and controls")
```

```
Obeserved 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))

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 2.724 seconds)

4.3 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 brillant book from Brady T. West, Kathleen B. Welch and Andrzej T. Galecki, see [Brady et al. 2014] in the references section below.

4.3.1 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 independents**. 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 is 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**
3. creates **clusters** of indivuduals 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.

4.3.2 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 strutured by the sampling of classroom, see Fig below. `score` from the same classroom are are not indendant 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**.

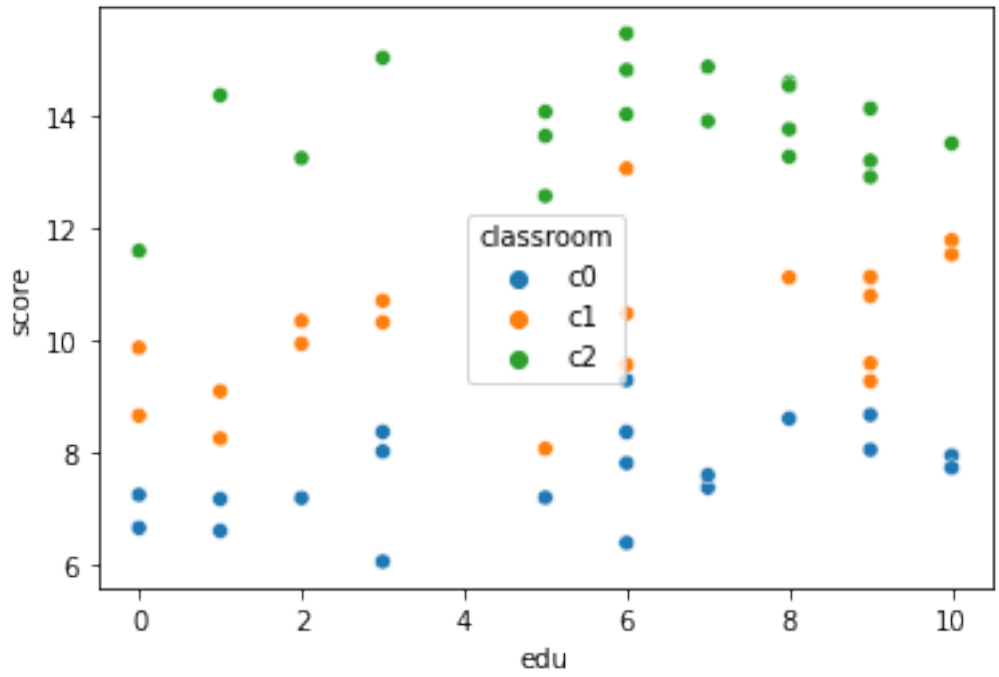
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import statsmodels.api as sm
```

(continues on next page)

(continued from previous page)

```
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 $y =$ score on the dependant variable $x =$ 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 classeroom, j index is not taken into account by the model and could be removed from the equation.

The general R formula is: `y ~ x` which in this case is `score ~ edu`. This model is:

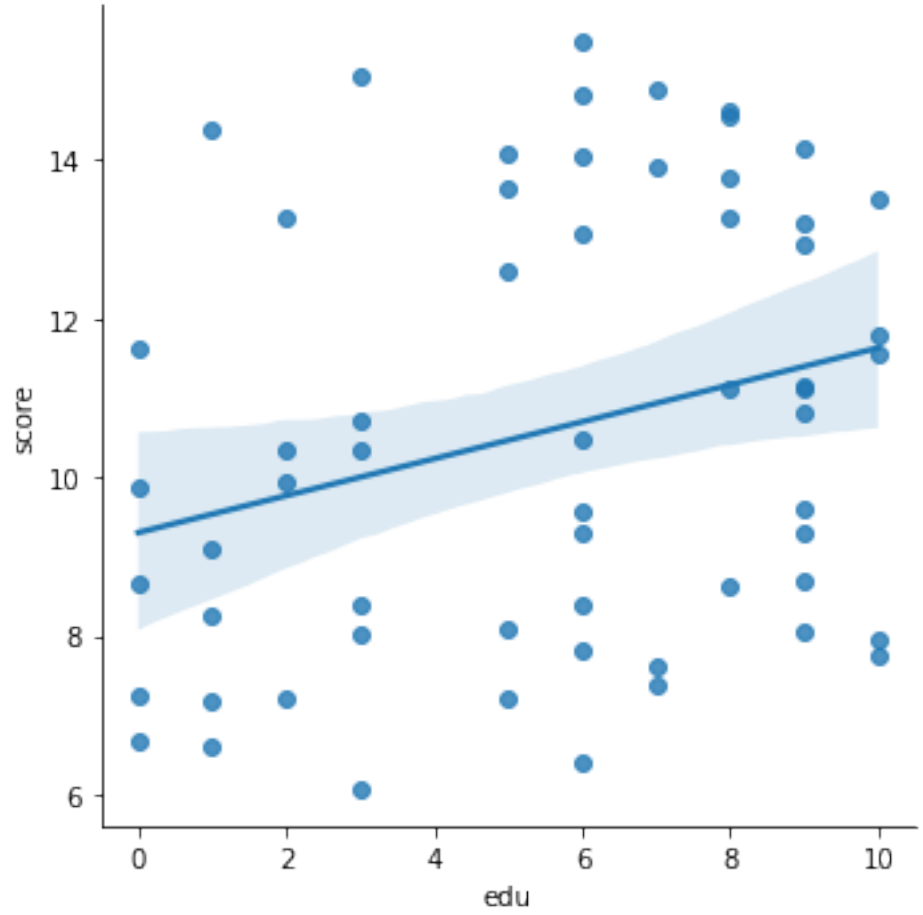
- **Not sensitive** since it does not model the classroom effect (high standard error).
- **Wrong** because, residuals are not normals, and it considers samples from the same classroom to be indenpendant.

```
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]
------------------------------------------------------------------------------
c0             0.2328      0.109      2.139      0.037       0.015       0.451
==============================================================================
MSE=7.262
```

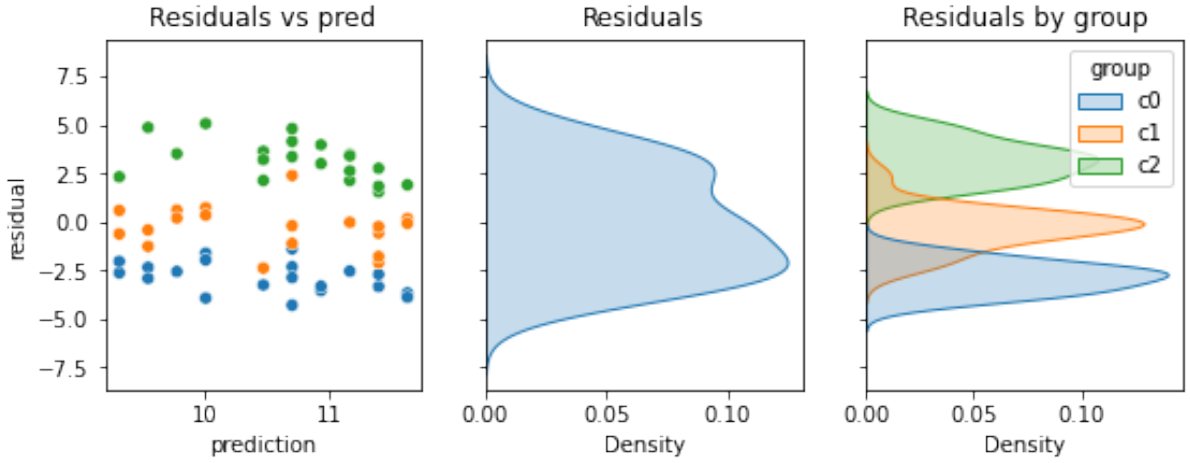
Plot

```
_ = 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 $z =$ `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 ~ x + z` which in this case is `score ~ edu + classroom`.

This model is:

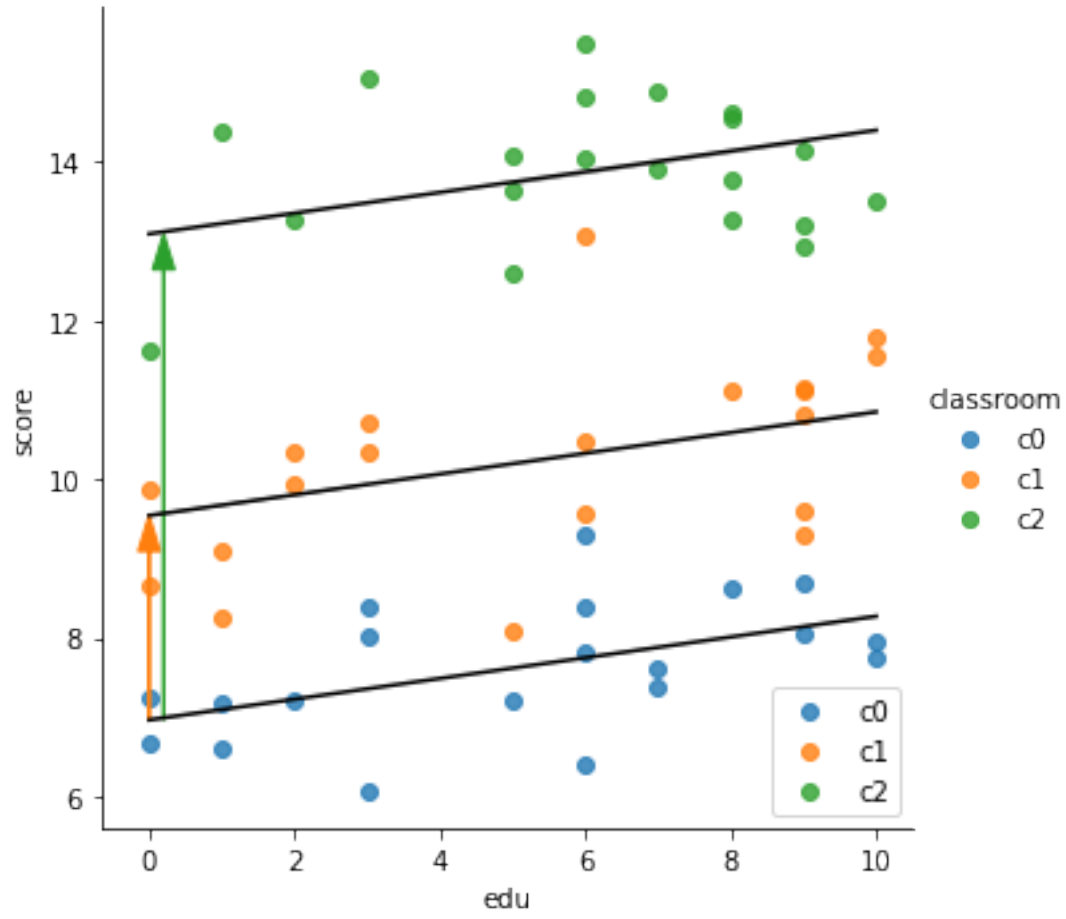
- **Sensitive** since it does not model the classroom effect (lower standard error). But,
- **questionable** because it considers the classroom to have a fixed constant effect without any uncertainty. However, those classrooms have been sampled from a larger samples of classrooms within the country.

```
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 (independant 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']
```

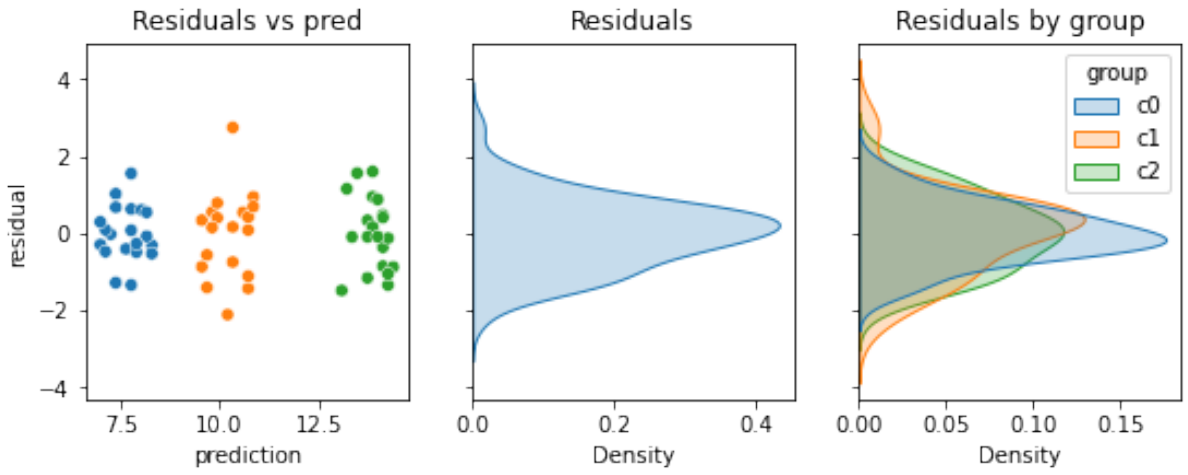
(continues on next page)

(continued from previous page)

```
[[ 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 0.869277616553041
## 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 coeficient 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 coeficient 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 conrespond 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 independant. Measures between top level units are independant.

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 classrom:

$$x_j = \text{mean}_i(x_{ij}), y_j = \text{mean}_i(y_{ij}), \text{for } j \in \{1, 2, 3\}.$$

2. **Level 2 (between independant units)** Regress averaged `score` on a averaged `edu`:

$$y_j = \beta_0 + \beta_1 x_j + \varepsilon_j$$

. The general R formula is: `y ~ x` which in this case is `score ~ 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.

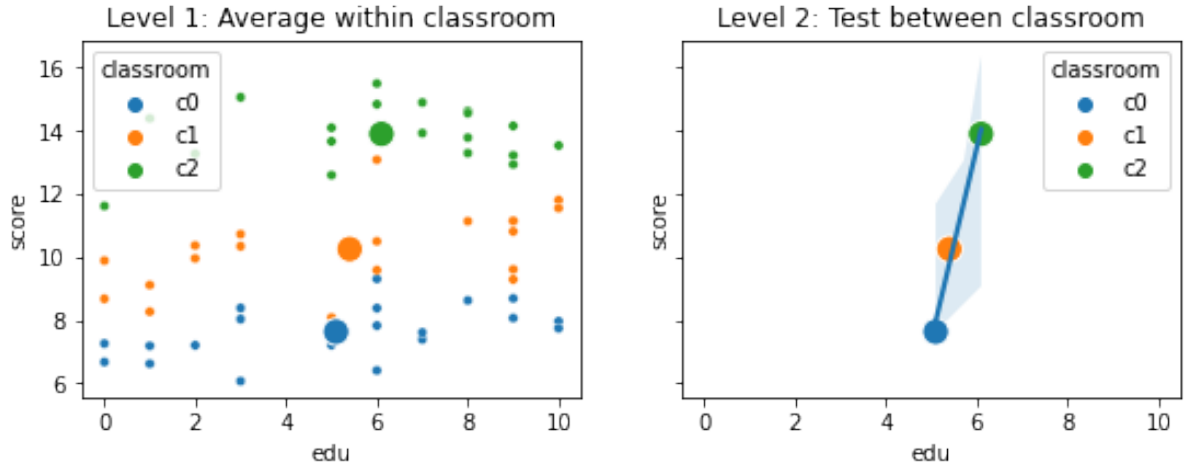
```
agregate = df.groupby('classroom').mean()
lm_agregate = smf.ols('score ~ edu', agregate).fit()
#print(lm_agregate.summary())
print(lm_agregate.t_test('edu'))

print("MSE=%.3f" % lm_agregate.mse_resid)
results.loc[len(results)] = ["Aggregation"] +\
    list(rmse_coef_tstat_pval(mod=lm_agregate, 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

```
agregate = agregate.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=agregate, ax=axes[0], s=150)
axes[0].set_title("Level 1: Average within classroom")

sns.regplot(x="edu", y="score", data=agregate, ax=axes[1])
sns.scatterplot(x='edu', y='score', hue='classroom',
                data=agregate, 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 paramteres 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 ~ x` which in this case is `score ~ edu` within classrooms.

2. Test across the classrooms if is the $\text{mean}_j(\beta_{1j}) = \beta_0 \neq 0$:

$$\beta_{1j} = \beta_0 + \varepsilon_j$$

   The general R formula is: `y ~ 1` which in this case is `beta_edu ~ 1`.

This model is:

- **Correct** because the invidividual estimated parameters are independent.

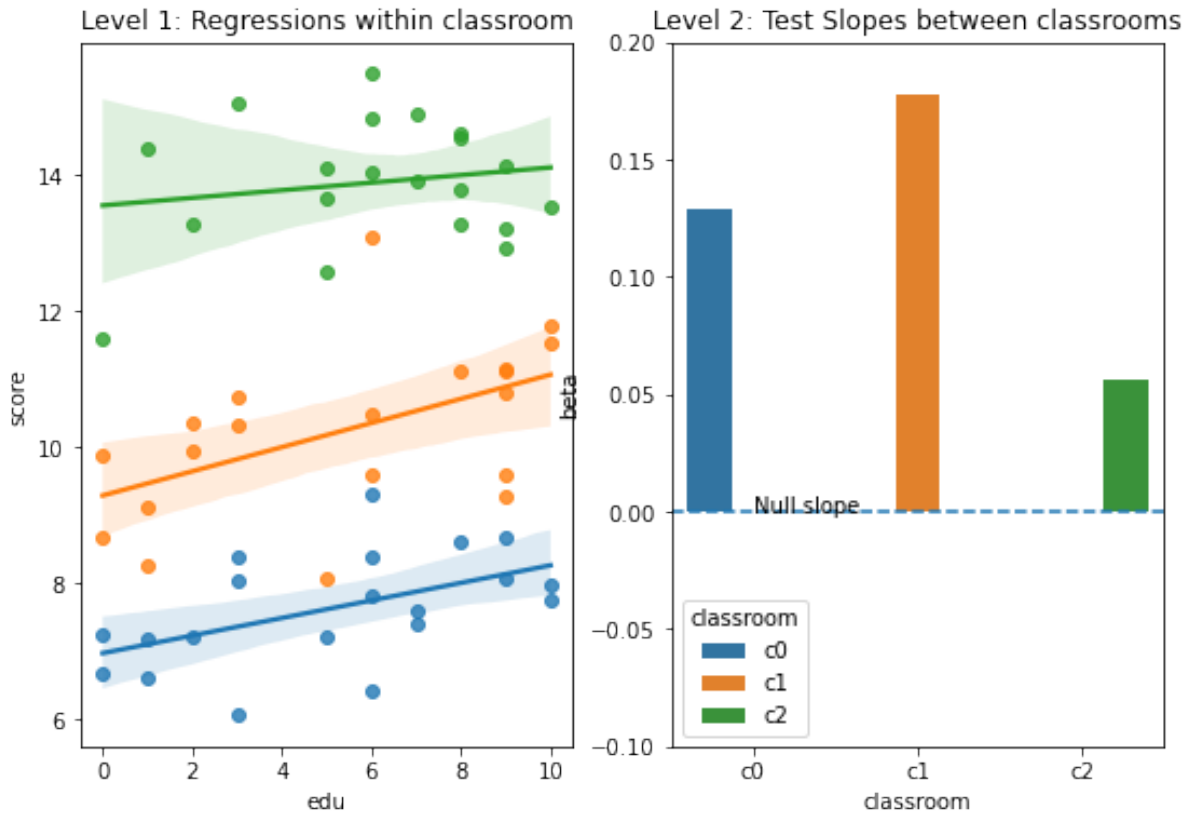
- **sensitive** since it allows to model differents slope for each classroom (see fixed interaction or random slope below). But it is but **not optimally designed** since there are many models, and each one does not take advantage of the information in data from other classroom. This can also make the 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 = `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 ~ x + (1|z)` which in this case it is `score ~ edu + (1|classroom)`. For python statmodels, the grouping factor `|classroom` is omited an 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
=====================================================
```

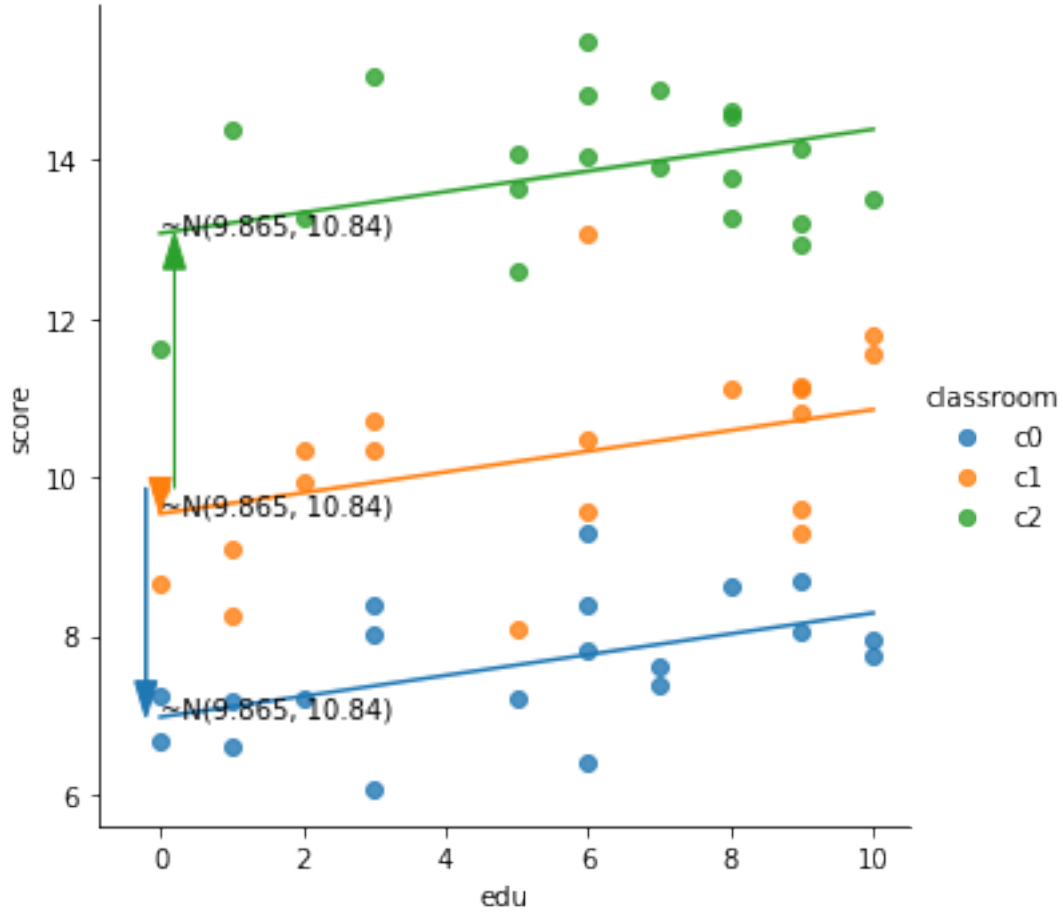
Explore model

```
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)
```



4.3.3 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 = `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) specitic 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 mor simply `score ~ edu * classroom` that denotes the full model with the additive contribution of each regressor and all their interactions.

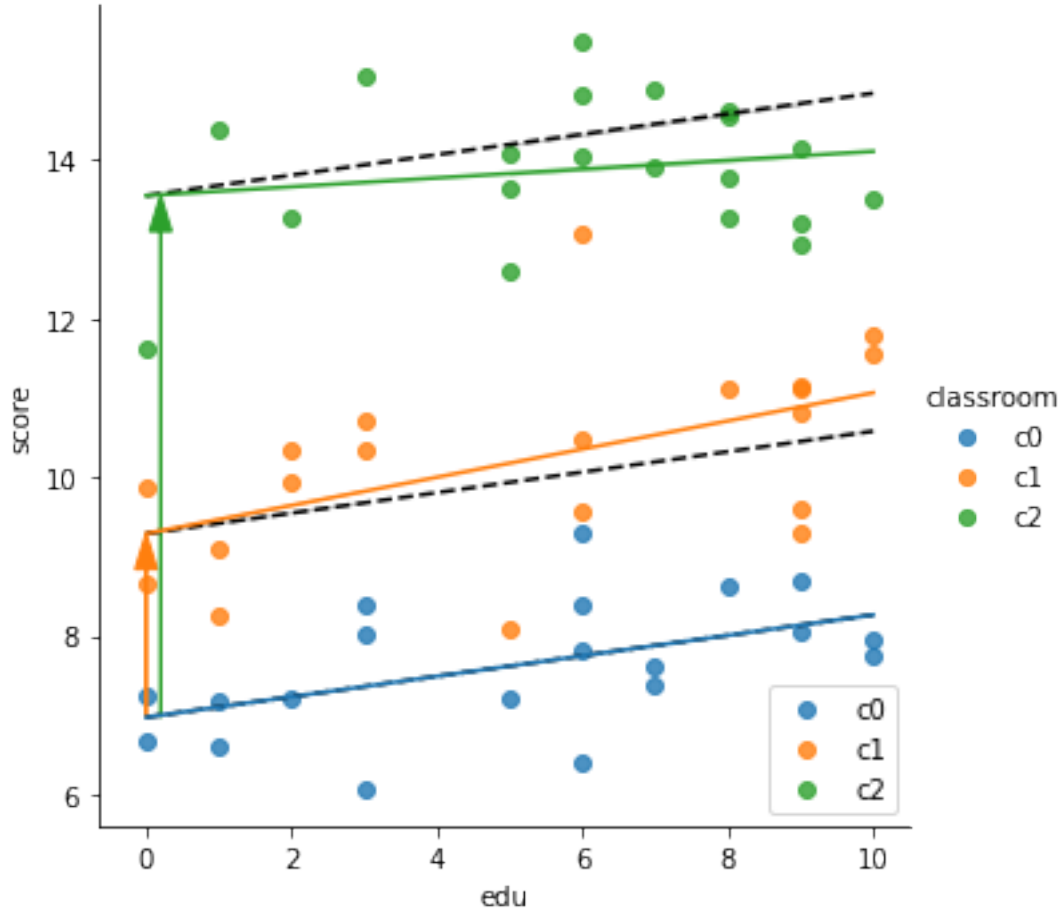
```
ancova_full = smf.ols('score ~ edu + classroom + edu:classroom', df).fit()
# Full model (including interaction) can use this notation:
# ancova_full = smf.ols('score ~ edu * classroom', df).fit()

# 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 interpcept, 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)

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, $u_j^1 \sim \mathcal{N}(\mathbf{0}, \sigma^{\mathbf{1}})$.
- 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 $(\sigma^1, \sigma^\alpha)$ should be estimated together with fixed effect $(\beta_0 + \beta_1)$ and random effect $(u^1, u_j^\alpha$, 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   12.279
Group x edu Cov -0.121    0.318
edu Var          0.001    0.012
============================================================
```

```
/home/ed203246/anaconda3/lib/python3.8/site-packages/statsmodels/base/model.
↪py:566: ConvergenceWarning: Maximum Likelihood optimization failed to converge.␣
↪Check mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
/home/ed203246/anaconda3/lib/python3.8/site-packages/statsmodels/regression/mixed_
↪linear_model.py:2200: ConvergenceWarning: Retrying MixedLM optimization with␣
↪lbfgs
  warnings.warn(
/home/ed203246/anaconda3/lib/python3.8/site-packages/statsmodels/regression/mixed_
↪linear_model.py:1634: UserWarning: Random effects covariance is singular
  warnings.warn(msg)
/home/ed203246/anaconda3/lib/python3.8/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 considere to remove random slopes.

4.3.4 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  0.127269  2.756917  0.005835
```

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 looses a lot of degrees of freedom (df = 2 = 3 classroom - 1 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).

4.3.5 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 :

$$\begin{bmatrix}\text{score}\\ 7.2\\ 7.9\\ 9.1\\ 11.1\\ 14.6\\ 14.0\end{bmatrix} = \begin{bmatrix}\text{Inter} & \text{Edu}\\ 1 & 2\\ 1 & 10\\ 1 & 1\\ 1 & 9\\ 1 & 8\\ 1 & 5\end{bmatrix} \begin{bmatrix}\text{Fix}\\ \beta_0\\ \beta_1\end{bmatrix} + \begin{bmatrix}\text{c1} & \text{c2} & \text{c3}\\ 1 & 0 & 0\\ 1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\\ 0 & 0 & 1\end{bmatrix} \begin{bmatrix}\text{Rand}\\ u_1\\ u_2\\ u_3\end{bmatrix} + \begin{bmatrix}\text{Err}\\ \epsilon_1\\ \epsilon_2\\ \epsilon_3\\ \epsilon_4\\ \epsilon_5\\ \epsilon_6\end{bmatrix}$$

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 $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{R})$, 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 $\mathbf{D}$ is plays a central role of the covariance structures associated with the mixed effect.

Covariance structures of the residuals covariance matrix: :math:`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: $\mathbf{R} = \sigma \mathbf{I}_N$. 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 prartice 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 distritution 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 :math:`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-independance 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 constist 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 reweigtht the residuals according to the non-independance 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 $(\sigma, \sigma_k, \ldots)$ 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".

4.3.6 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.

4.3.7 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

4.4 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 independent samples column **vectors** $[\mathbf{x}_1, \ldots, \mathbf{x}_i, \ldots, \mathbf{x}_N]$ of length P

$$\mathbf{X} = \begin{bmatrix} -\mathbf{x}_1^T- \\ \vdots \\ -\mathbf{x}_i^T- \\ \vdots \\ -\mathbf{x}_P^T- \end{bmatrix} = \begin{bmatrix} x_{11} & \cdots & x_{1j} & \cdots & x_{1P} \\ \vdots & & \vdots & & \vdots \\ x_{i1} & \cdots & x_{ij} & \cdots & x_{iP} \\ \vdots & & \vdots & & \vdots \\ x_{N1} & \cdots & x_{Nj} & \cdots & x_{NP} \end{bmatrix} = \begin{bmatrix} x_{11} & \ldots & x_{1P} \\ \vdots & & \vdots \\ & \mathbf{X} & \\ \vdots & & \vdots \\ x_{N1} & \ldots & x_{NP} \end{bmatrix}_{N \times P}.$$

4.4.1 Linear Algebra

Euclidean norm and distance

The Euclidean norm of a vector $\mathbf{a} \in \mathbb{R}^P$ is denoted

$$\|\mathbf{a}\|_2 = \sqrt{\sum_i^P {a_i}^2}$$

The Euclidean distance between two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^P$ is

$$\|\mathbf{a} - \mathbf{b}\|_2 = \sqrt{\sum_i^P (a_i - b_i)^2}$$

Dot product and projection

Source: Wikipedia

Algebraic definition

The dot product, denoted "$\cdot$" of two P-dimensional vectors $\mathbf{a} = [a_1, a_2, ..., a_P]$ and $\mathbf{a} = [b_1, b_2, ..., b_P]$ is defined as

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T\mathbf{b} = \sum_i a_i b_i = \begin{bmatrix} a_1 & \ldots & \mathbf{a}^T & \ldots & a_P \end{bmatrix} \begin{bmatrix} b_1 \\ \vdots \\ \mathbf{b} \\ \vdots \\ b_P \end{bmatrix}.$$

The Euclidean norm of a vector can be computed using the dot product, as

$$\|\mathbf{a}\|_2 = \sqrt{\mathbf{a} \cdot \mathbf{a}}.$$

Geometric definition: projection

In Euclidean space, a Euclidean vector is a geometrical object that possesses both a 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 magnitude of a vector $\mathbf{a}$ is denoted by $\|\mathbf{a}\|_2$. The dot product of two Euclidean vectors $\mathbf{a}$ and $\mathbf{b}$ is defined by

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|_2 \, \|\mathbf{b}\|_2 \cos\theta,$$

where θ is the angle between $\mathbf{a}$ and $\mathbf{b}$.

In particular, if $\mathbf{a}$ and $\mathbf{b}$ are orthogonal, then the angle between them is 90° and

$$\mathbf{a} \cdot \mathbf{b} = 0.$$

At the other extreme, if they are codirectional, then the angle between them is 0° and

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|_2 \, \|\mathbf{b}\|_2$$

This implies that the dot product of a vector $\mathbf{a}$ by itself is

$$\mathbf{a} \cdot \mathbf{a} = \|\mathbf{a}\|_2^2.$$

The scalar projection (or scalar component) of a Euclidean vector $\mathbf{a}$ in the direction of a Euclidean vector $\mathbf{b}$ is given by

$$a_b = \|\mathbf{a}\|_2 \cos\theta,$$

where θ is the angle between $\mathbf{a}$ and $\mathbf{b}$.

In terms of the geometric definition of the dot product, this can be rewritten

$$a_b = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{b}\|_2},$$

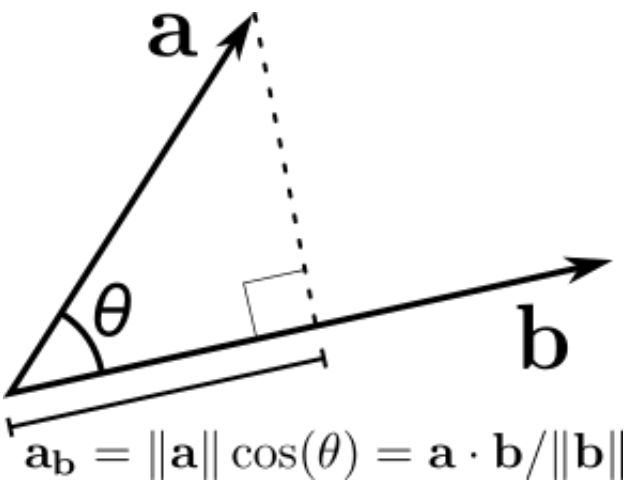


Fig. 4: Projection.

```
import numpy as np
np.random.seed(42)

a = np.random.randn(10)
b = np.random.randn(10)

np.dot(a, b)
```

```
-4.085788532659924
```

4.4.2 Mean vector

The mean $(P \times 1)$ column-vector μ whose estimator is

$$\bar{\mathbf{x}} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x_i} = \frac{1}{N}\sum_{i=1}^{N}\begin{bmatrix} x_{i1} \\ \vdots \\ x_{ij} \\ \vdots \\ x_{iP} \end{bmatrix} = \begin{bmatrix} \bar{x}_1 \\ \vdots \\ \bar{x}_j \\ \vdots \\ \bar{x}_P \end{bmatrix}.$$

4.4.3 Covariance matrix

- The covariance matrix $\mathbf{\Sigma_{XX}}$ 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:

$$\mathbf{\Sigma_{XX}} = E(\mathbf{X} - E(\mathbf{X}))^T E(\mathbf{X} - E(\mathbf{X})),$$

whose estimator $\mathbf{S_{XX}}$ is a $P \times P$ matrix given by

$$\mathbf{S_{XX}} = \frac{1}{N-1}(\mathbf{X} - \mathbf{1}\bar{\mathbf{x}}^T)^T(\mathbf{X} - \mathbf{1}\bar{\mathbf{x}}^T).$$

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

$$\mathbf{S_{XX}} = \frac{1}{N-1}\mathbf{X}^T\mathbf{X} = \frac{1}{N-1}\begin{bmatrix} x_{11} & \cdots & x_{N1} \\ x_{1j} & \cdots & x_{Nj} \\ \vdots & & \vdots \\ x_{1P} & \cdots & x_{NP} \end{bmatrix}\begin{bmatrix} x_{11} & \cdots & x_{1k} & x_{1P} \\ \vdots & & \vdots & \vdots \\ x_{N1} & \cdots & x_{Nk} & x_{NP} \end{bmatrix} = \begin{bmatrix} s_1 & \cdots & s_{1k} & s_{1P} \\ & \ddots & s_{jk} & \vdots \\ & & s_k & s_{kP} \\ & & & s_P \end{bmatrix},$$

where

$$s_{jk} = s_{kj} = \frac{1}{N-1}\mathbf{x_j}^T\mathbf{x_k} = \frac{1}{N-1}\sum_{i=1}^{N} x_{ij}x_{ik}$$

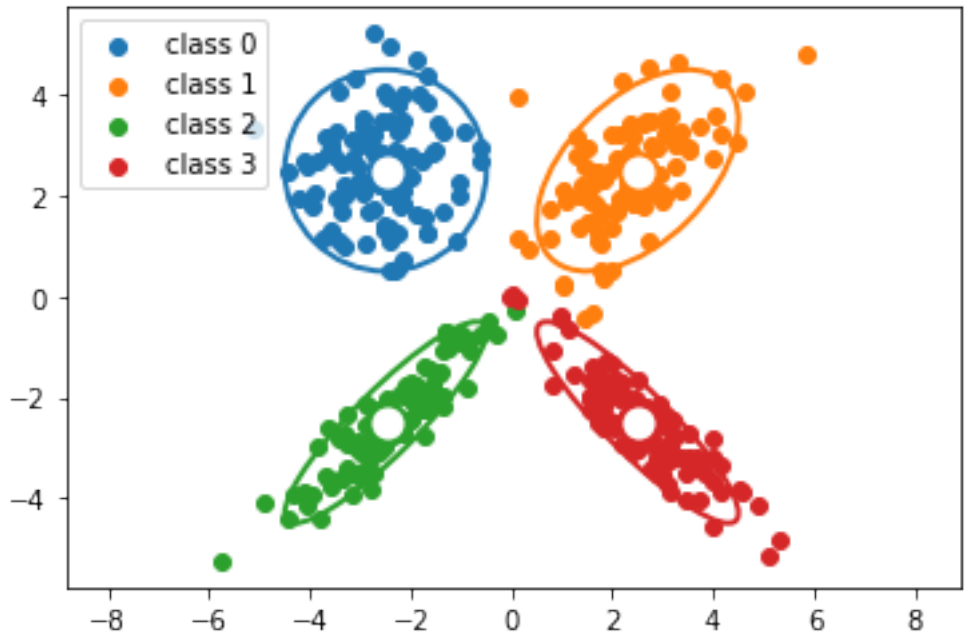
is an estimator of the covariance between the j^{th} and k^{th} variables.

```
## Avoid warnings and force inline plot
%matplotlib inline
import warnings
warnings.filterwarnings("ignore")
##
import numpy as np
import scipy
```

(continues on next page)

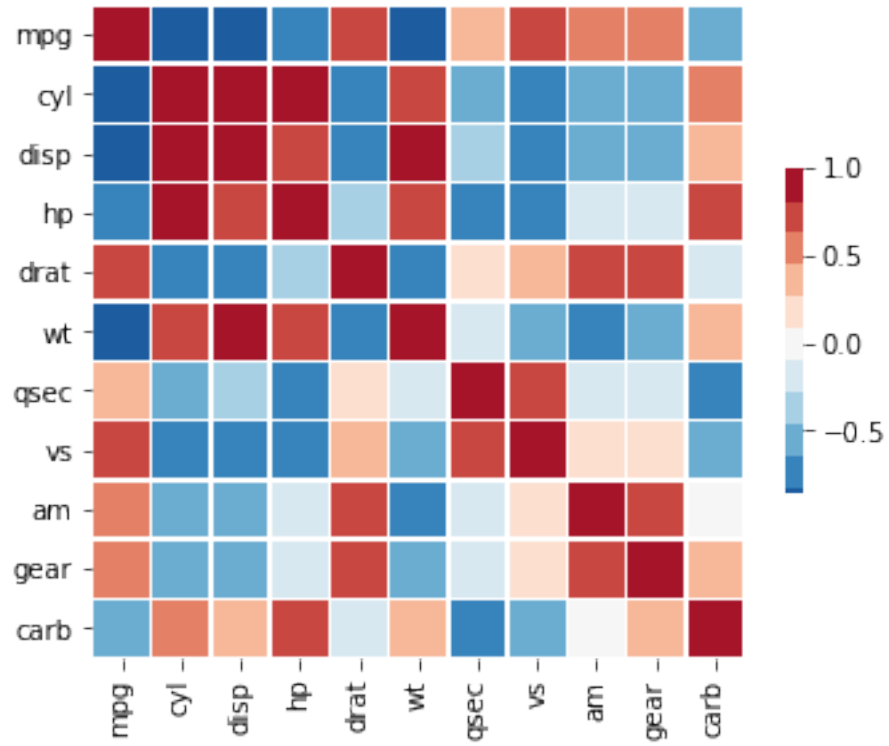
(continued from previous page)

```
import matplotlib.pyplot as plt
import seaborn as sns
import pystatsml.plot_utils
import seaborn as sns  # nice color

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
    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')
```



4.4.4 Correlation matrix

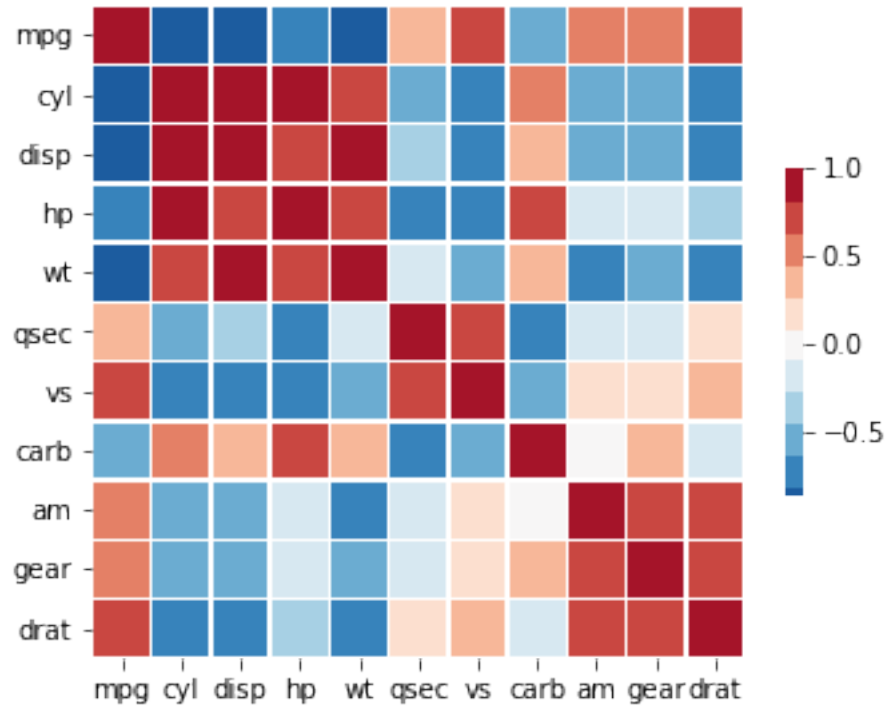
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

url = 'https://python-graph-gallery.com/wp-content/uploads/mtcars.csv'
df = pd.read_csv(url)

# 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
d = 2 * (1 - np.abs(corr))

from sklearn.cluster import AgglomerativeClustering
clustering = AgglomerativeClustering(n_clusters=3, linkage='single', affinity=
↪"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
↪']]
```



4.4.5 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]]
```

4.4.6 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

$$D_M(\mathbf{x}, \mu) = \sqrt{(\mathbf{x} - \mu)^T \mathbf{\Sigma}^{-1} (\mathbf{x} - \mu)}.$$

Intuitions

- Distances along the principal directions of dispersion are contracted since they correspond to likely dispersion of points.
- Distances othogonal to the principal directions of dispersion are dilated since they correspond to unlikely dispersion of points.

For example

$$D_M(\mathbf{1}) = \sqrt{\mathbf{1}^T \mathbf{\Sigma}^{-1} \mathbf{1}}.$$

```
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.        ]
```

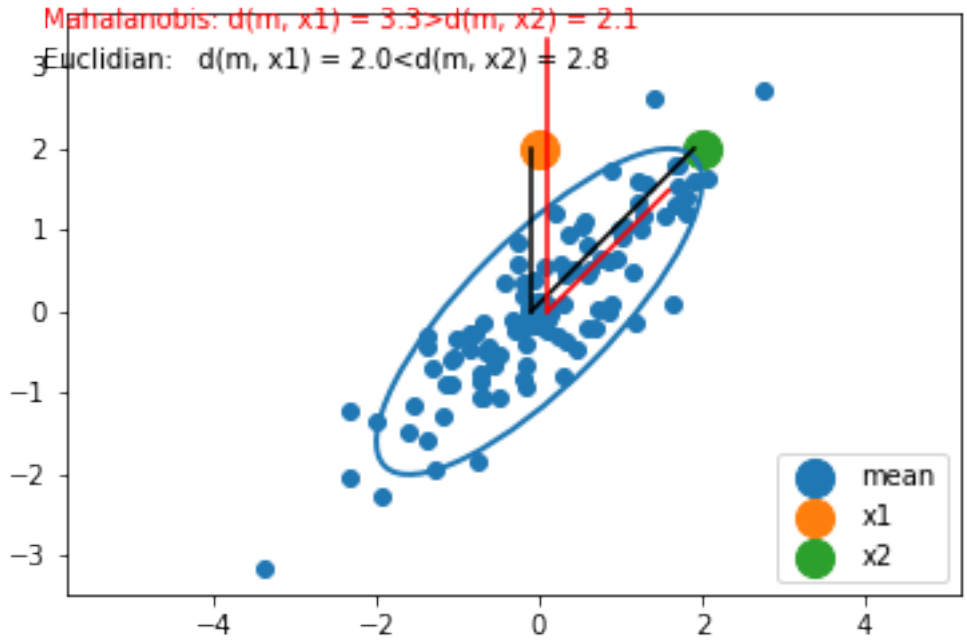
```
import numpy as np
import scipy
import matplotlib.pyplot as plt
import seaborn as sns
import pystatsml.plot_utils
%matplotlib inline
np.random.seed(40)
colors = sns.color_palette()

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,
```

(continues on next page)

(continued from previous page)

```
        'Euclidian:   d(m, x1) = %.1f<d(m, x2) = %.1f' % (d2_m_x1, d2_m_x2),␣
↪color='k')
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 $\mathcal{N}(\mathbf{x}|\mu, \mathbf{\Sigma})$. 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.

4.4.7 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 $\mathbf{x} = [x_1, x_2, \ldots, x_P]^T$ is

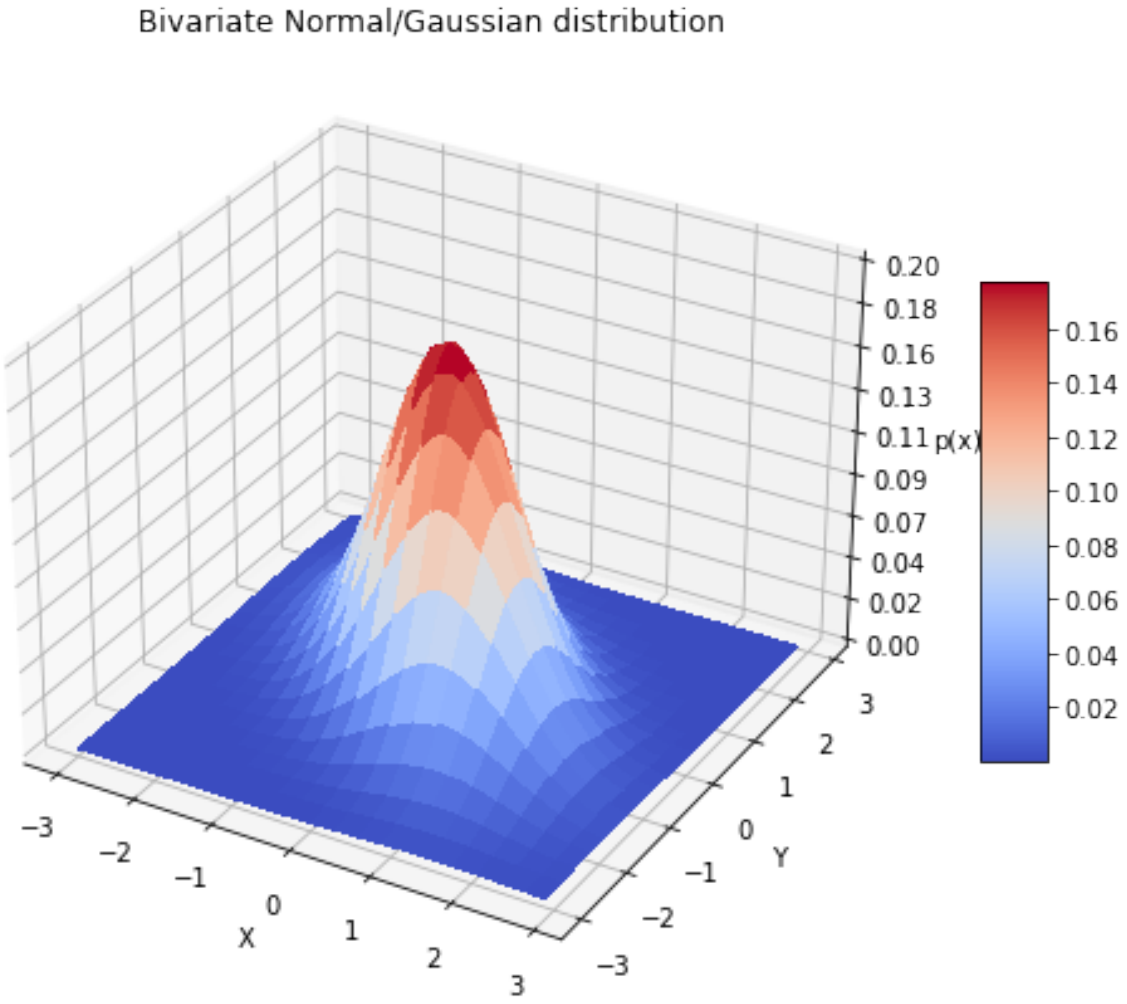
$$\mathcal{N}(\mathbf{x}|\mu, \mathbf{\Sigma}) = \frac{1}{(2\pi)^{P/2}|\mathbf{\Sigma}|^{1/2}} \exp\{-\frac{1}{2}(\mathbf{x}-\mu)^T\mathbf{\Sigma}^{-1}(\mathbf{x}-\mu)\}.$$

```
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_scpy = multivariate_normal(mu, sigma).pdf(np.stack((x, y), axis=2))
assert np.allclose(norm, norm_scpy)

# Plot
fig = plt.figure(figsize=(10, 7))
ax = fig.gca(projection='3d')
surf = ax.plot_surface(x, y, norm, rstride=3,
        cstride=3, cmap=plt.cm.coolwarm,
        linewidth=1, antialiased=False
    )
```

(continues on next page)

(continued from previous page)

```
ax.set_zlim(0, 0.2)
ax.zaxis.set_major_locator(plt.LinearLocator(10))
ax.zaxis.set_major_formatter(plt.FormatStrFormatter('%.02f'))

ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('p(x)')

plt.title('Bivariate Normal/Gaussian distribution')
fig.colorbar(surf, shrink=0.5, aspect=7, cmap=plt.cm.coolwarm)
plt.show()
```



4.4.8 Exercises

Dot product and Euclidean norm

Given $\mathbf{a} = [2, 1]^T$ and $\mathbf{b} = [1, 1]^T$

1. Write a function `euclidean(x)` that computes the Euclidean norm of vector, $\mathbf{x}$.
2. Compute the Euclidean norm of $\mathbf{a}$.
3. Compute the Euclidean distance of $\|\mathbf{a} - \mathbf{b}\|_2$.
4. Compute the projection of $\mathbf{b}$ in the direction of vector $\mathbf{a}$: b_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 $\mathbf{a}$.

Covariance matrix and Mahalanobis norm

1. Sample a dataset $\mathbf{X}$ of $N = 100$ samples of 2-dimensional vectors from the bivariate normal distribution $\mathcal{N}(\mu, \mathbf{\Sigma})$ where $\mu = [1, 1]^T$ and $\mathbf{\Sigma} = \begin{bmatrix} 1 & 0.8 \\ 0.8, 1 \end{bmatrix}$.
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 $\mathbf{S}$. Compare the estimated covariance matrix $\mathbf{S}$ to the true covariance matrix, $\mathbf{\Sigma}$.
4. Compute $\mathbf{S}^{-1}$ (`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 $\mathbf{x}$ to the mean, $\bar{\mathbf{x}}$.
6. Compute the Mahalanobis and Euclidean distances of each sample $\mathbf{x}_i$ to the mean $\bar{\mathbf{x}}$. Store the results in a 100×2 dataframe.

4.5 Time series in python

Two libraries:

- Pandas: https://pandas.pydata.org/pandas-docs/stable/timeseries.html
- scipy http://www.statsmodels.org/devel/tsa.html

4.5.1 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-stationaruty 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.

4.5.2 Pandas time series data structure

A Series is similar to a list or an array in Python. It represents a series of values (numeric or otherwise) such as a column of data. It provides additional functionality, methods, and operators, which make it a more powerful version of a list.

```
%matplotlib inline

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

```
# Create a Series from a list
ser = pd.Series([1, 3])
print(ser)

# String as index
prices = {'apple': 4.99,
          'banana': 1.99,
          'orange': 3.99}
ser = pd.Series(prices)
print(ser)

x = pd.Series(np.arange(1,3), index=[x for x in 'ab'])
print(x)
print(x['b'])
```

```
0    1
1    3
dtype: int64
apple     4.99
```

(continues on next page)

(continued from previous page)

```
banana    1.99
orange    3.99
dtype: float64
a    1
b    2
dtype: int64
2
```

4.5.3 Time series analysis of Google trends

source: https://www.datacamp.com/community/tutorials/time-series-analysis-tutorial

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

4.5.4 Read data

```
try:
    url = "https://raw.githubusercontent.com/datacamp/datacamp_facebook_live_ny_
↪resolution/master/datasets/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
```

(continues on next page)

(continued from previous page)

```
3  2004-04                 70                 22                 48
4  2004-05                 72                 22                 43
              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
```

4.5.5 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 inplace 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)

print(df.head())
```

```
            diet  gym  finance
month
2004-01-01   100   31       48
2004-02-01    75   26       49
2004-03-01    67   24       47
2004-04-01    70   22       48
2004-05-01    72   22       43
```

4.5.6 Exploratory data analysis

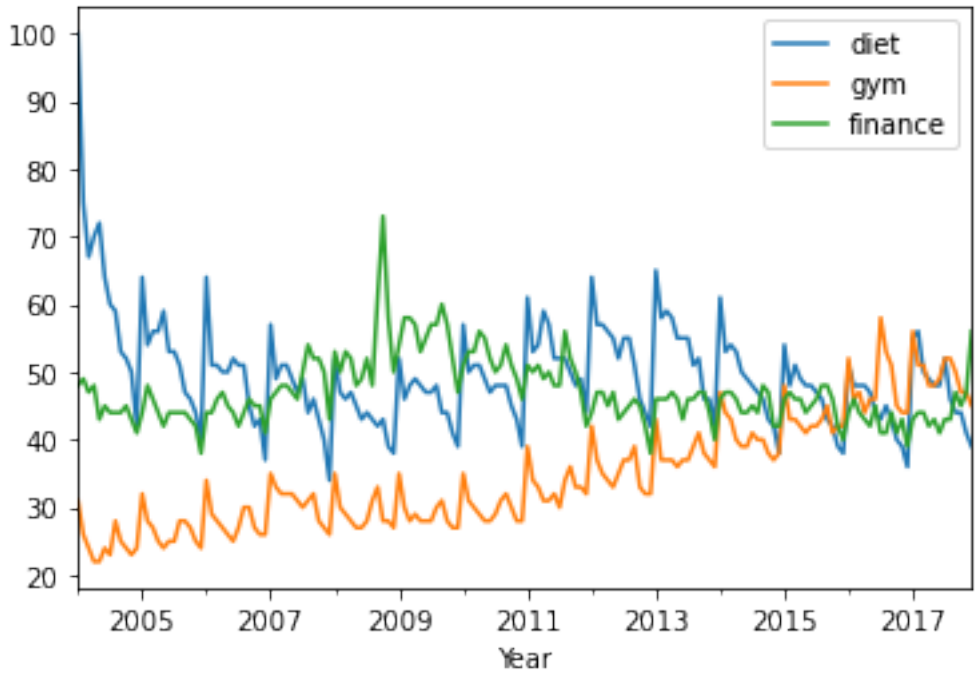
You can use a built-in pandas visualization method .plot() to plot your data as 3 line plots on a single figure (one for each column, namely, 'diet', 'gym', and 'finance').

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

# change figure parameters
```

(continues on next page)

(continued from previous page)

```
# df.plot(figsize=(20,10), linewidth=5, fontsize=20)

# Plot single column
# df[['diet']].plot(figsize=(20,10), linewidth=5, fontsize=20)
# plt.xlabel('Year', fontsize=20);
```



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.

4.5.7 Resampling, smoothing, windowing, rolling average: trends

Rolling average, 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.

Remove Seasonality with pandas Series.

See: http://pandas.pydata.org/pandas-docs/stable/timeseries.html A: 'year end frequency' year frequency

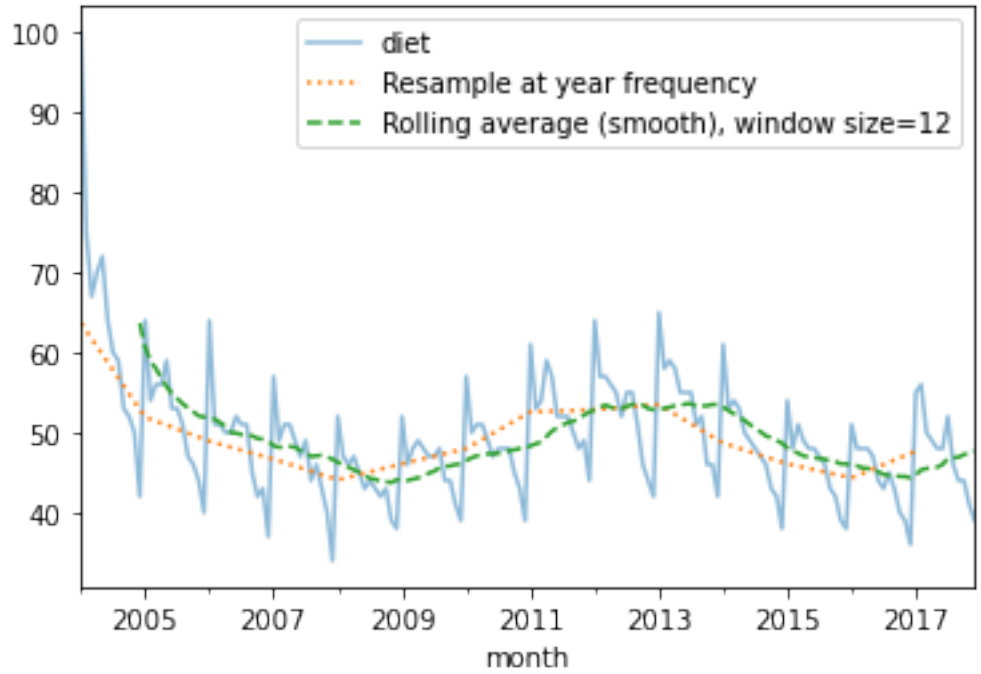
```
diet = df['diet']

diet_resamp_yr = diet.resample('A').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)
```

(continues on next page)

(continued from previous page)

```
diet_roll_yr.plot(style='--', label='Rolling average (smooth), window size=12',␣
↪ax=ax)
ax.legend()
```

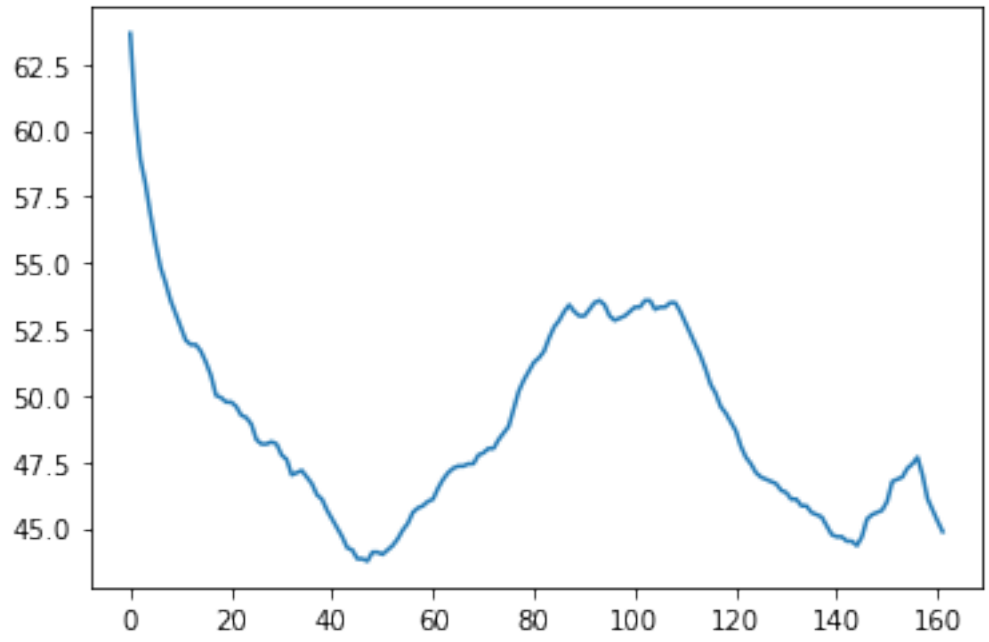
```
<matplotlib.legend.Legend at 0x7f670db34a10>
```



Rolling average (smoothing) with Numpy

```
x = np.asarray(df[['diet']])
win = 12
win_half = int(win / 2)
# print([((idx-win_half), (idx+win_half)) for idx in np.arange(win_half, len(x))])

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)
```

```
[<matplotlib.lines.Line2D at 0x7f670cbbb1d0>]
```

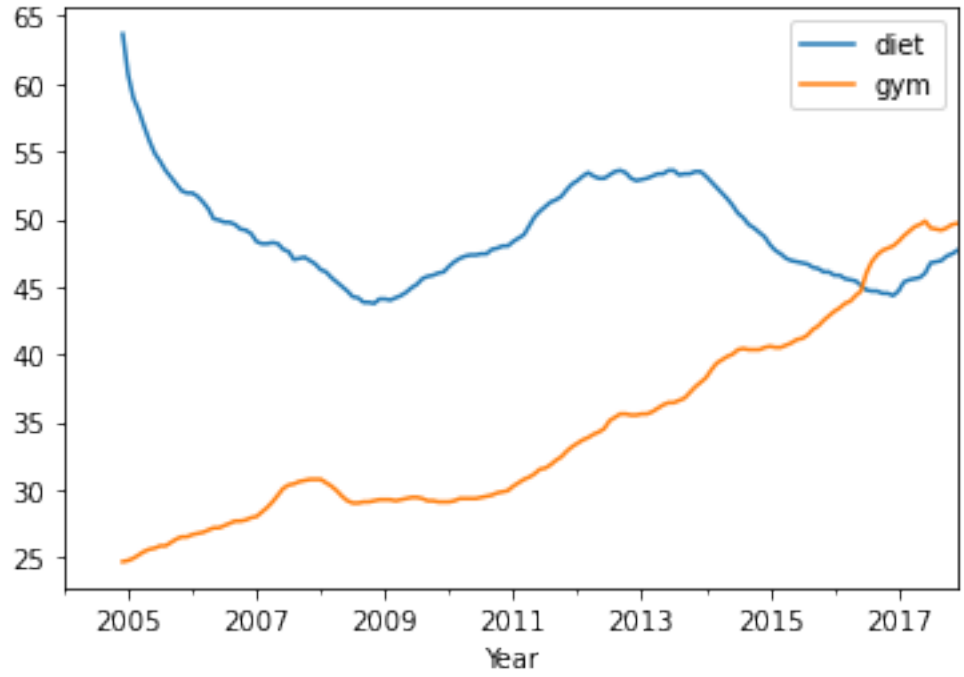


Trends Plot Diet and Gym

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

```
gym = df['gym']

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

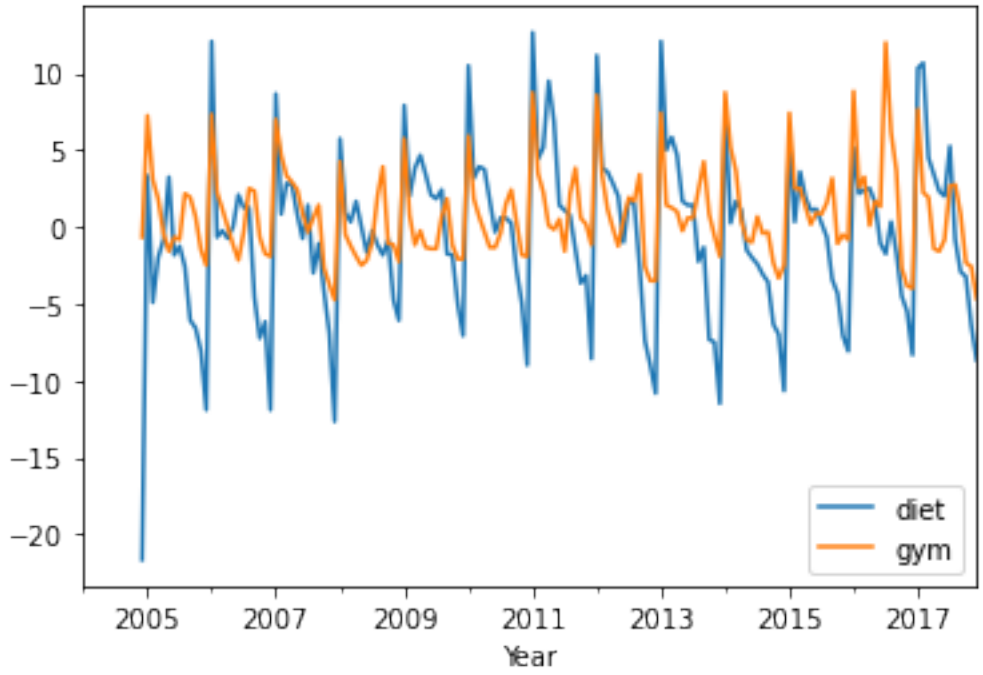
```
Text(0.5, 0, 'Year')
```



Detrending

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

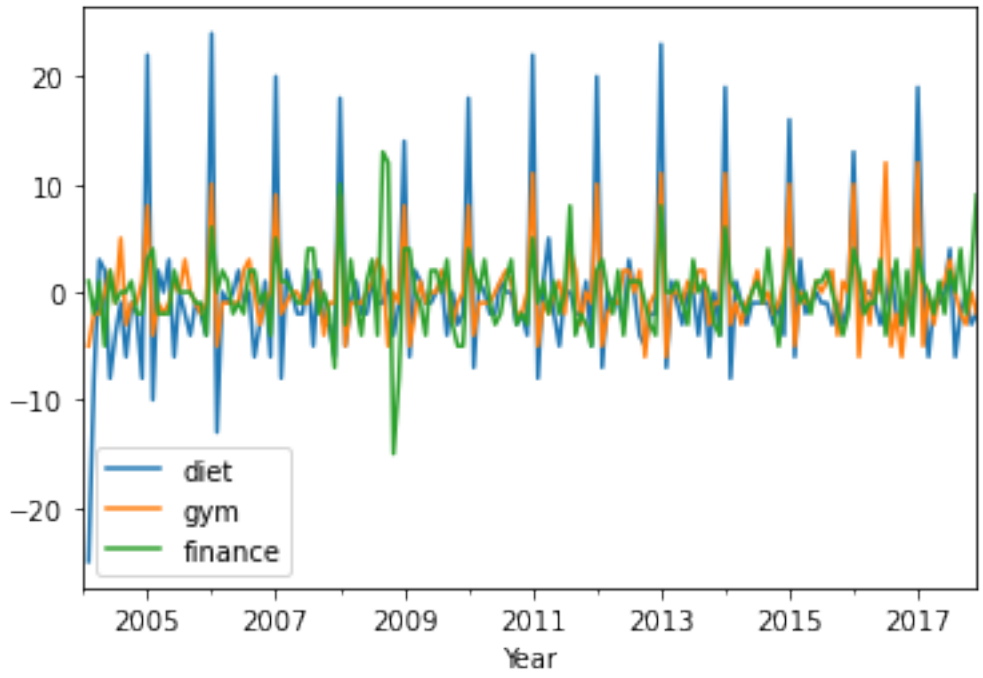
```
Text(0.5, 0, 'Year')
```



4.5.8 First-order differencing: seasonal patterns

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

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

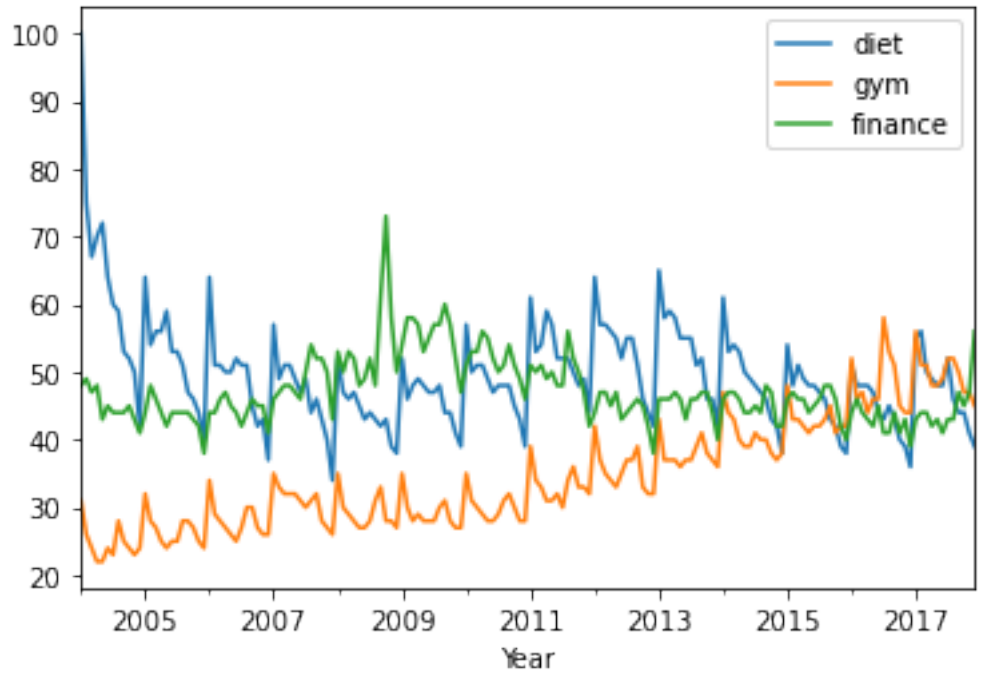
```
Text(0.5, 0, 'Year')
```



4.5.9 Periodicity and correlation

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

```
             diet       gym   finance
diet     1.000000 -0.100764 -0.034639
gym     -0.100764  1.000000 -0.284279
finance -0.034639 -0.284279  1.000000
```



Plot correlation matrix

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

```
             diet       gym   finance
diet     1.000000 -0.100764 -0.034639
gym     -0.100764  1.000000 -0.284279
finance -0.034639 -0.284279  1.000000
```

'diet' and 'gym' are negatively correlated! Remember that you have a seasonal and a trend component. From the correlation coefficient, 'diet' and 'gym' are negatively correlated:

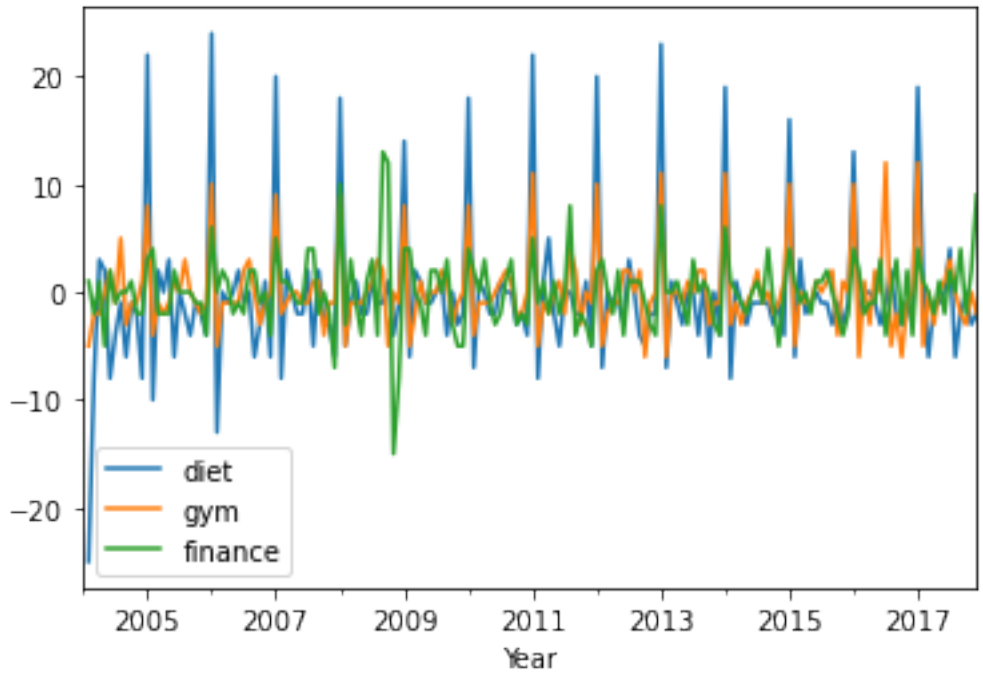
- trends components are negatively correlated.
- seasonal components would positively correlated and their

The actual correlation coefficient is actually capturing both of those.

Seasonal correlation: correlation of the first-order differences of these time series

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

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

```
             diet       gym   finance
diet     1.000000  0.758707  0.373828
gym      0.758707  1.000000  0.301111
finance  0.373828  0.301111  1.000000
```



Plot correlation matrix

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

```
             diet       gym   finance
diet     1.000000  0.758707  0.373828
gym      0.758707  1.000000  0.301111
finance  0.373828  0.301111  1.000000
```

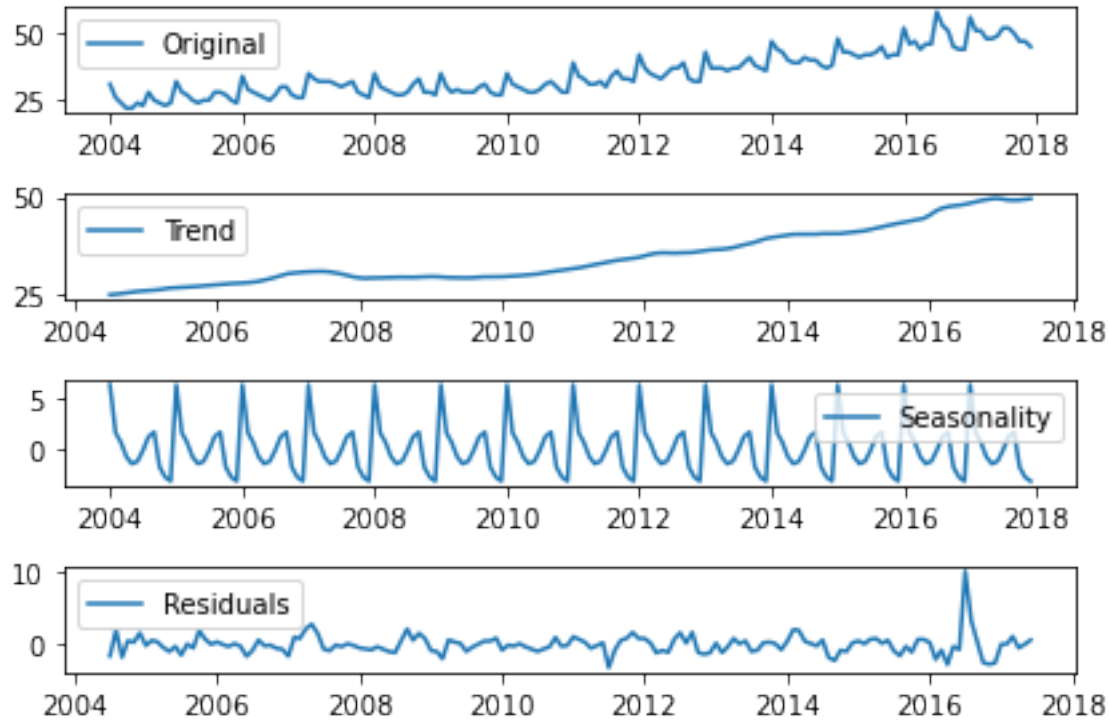
Decomposing time serie in trend, seasonality and residuals

```
from statsmodels.tsa.seasonal import seasonal_decompose

x = gym

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

plt.subplot(411)
plt.plot(x, label='Original')
plt.legend(loc='best')
plt.subplot(412)
plt.plot(trend, label='Trend')
plt.legend(loc='best')
plt.subplot(413)
plt.plot(seasonal,label='Seasonality')
plt.legend(loc='best')
plt.subplot(414)
```

(continues on next page)

(continued from previous page)

```
plt.plot(residual, label='Residuals')
plt.legend(loc='best')
plt.tight_layout()
```



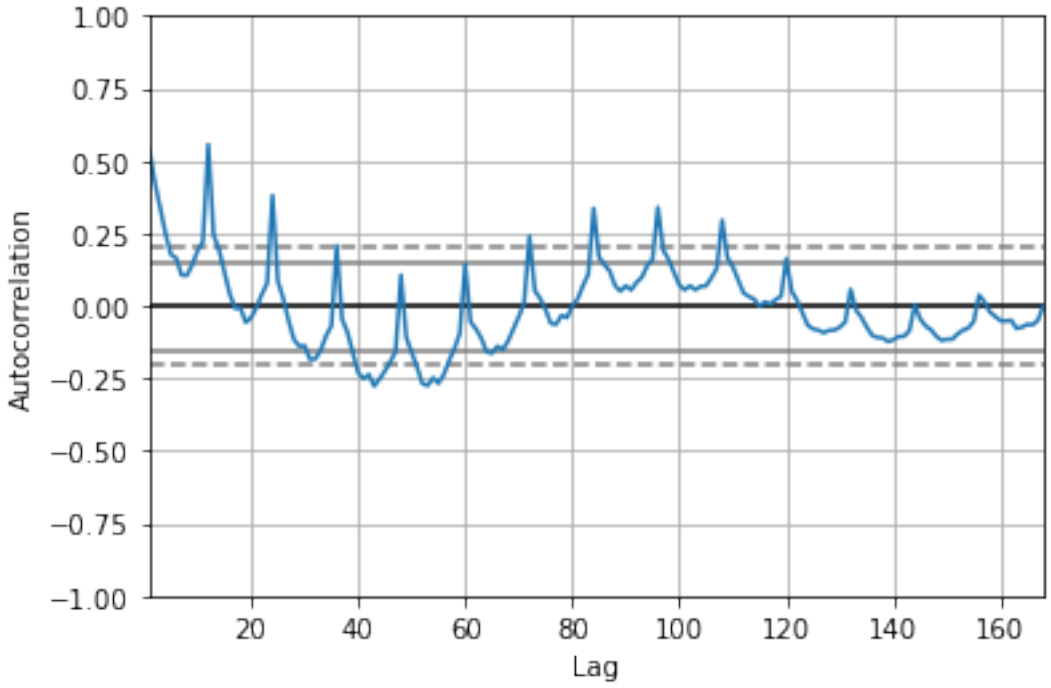
4.5.10 Autocorrelation

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 t1...t2 with series at instant t1-5...t2-5 (t1-5 and t2 being end points).

Plot

```
# from pandas.plotting import autocorrelation_plot
from pandas.plotting import autocorrelation_plot

x = df["diet"].astype(float)
autocorrelation_plot(x)
```

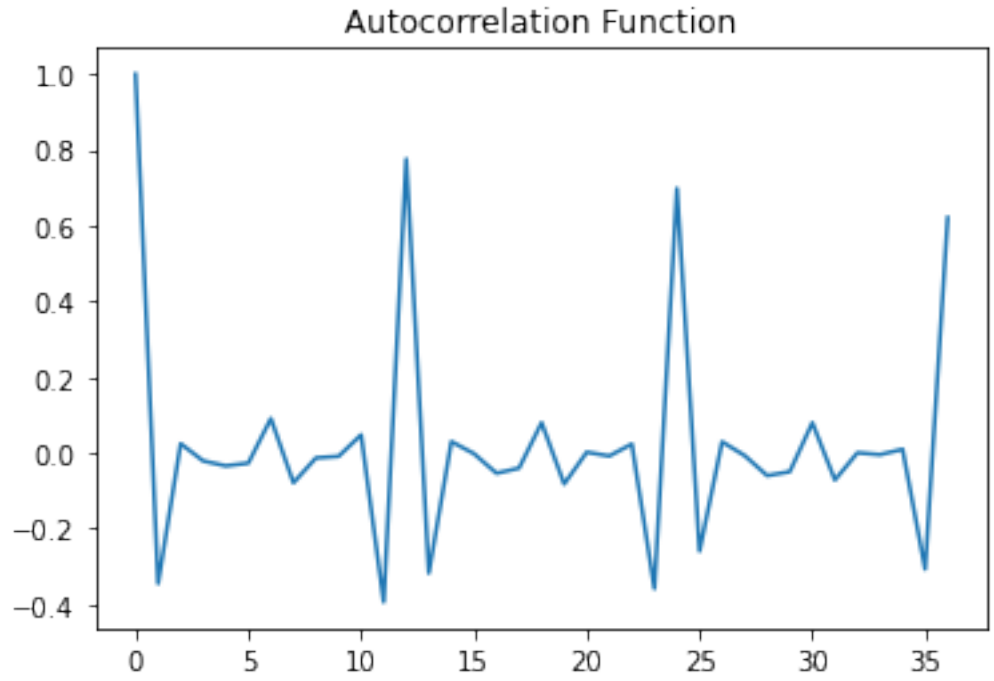
```
<AxesSubplot:xlabel='Lag', ylabel='Autocorrelation'>
```



Compute Autocorrelation Function (ACF)

```
from statsmodels.tsa.stattools import acf

x_diff = x.diff().dropna() # first item is NA
lag_acf = acf(x_diff, nlags=36, fft=True)
plt.plot(lag_acf)
plt.title('Autocorrelation Function')
```

```
Text(0.5, 1.0, 'Autocorrelation Function')
```



ACF peaks every 12 months: Time series is correlated with itself shifted by 12 months.

4.5.11 Time series forecasting with Python using Autoregressive Moving Average (ARMA) models

Source:

- https://www.packtpub.com/mapt/book/big_data_and_business_intelligence/9781783553358/7/ch07lvl1sec77/arma-models
- http://en.wikipedia.org/wiki/Autoregressive%E2%80%93moving-average_model
- ARIMA: https://www.analyticsvidhya.com/blog/2016/02/time-series-forecasting-codes-python/

ARMA models are often used to forecast a time series. These models combine autoregressive and moving average models. In moving average models, we assume that a variable is the sum of the mean of the time series and a linear combination of noise components.

The autoregressive and moving average models can have different orders. In general, we can define an ARMA model with p autoregressive terms and q moving average terms as follows:

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

Choosing p and q

Plot the partial autocorrelation functions for an estimate of p, and likewise using the autocorrelation functions for an estimate of q.

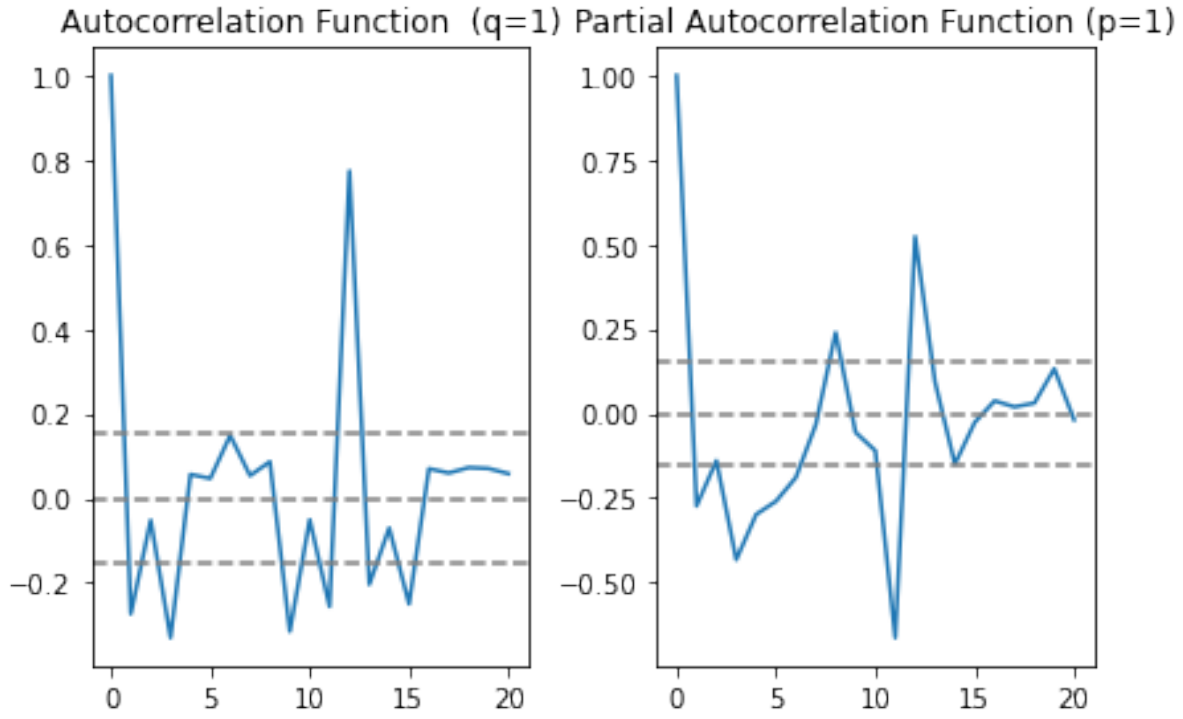
Partial Autocorrelation Function (PACF): This measures the correlation between the TS with a lagged version of itself but after eliminating the variations already explained by the intervening comparisons. Eg at lag 5, it will check the correlation but remove the effects already explained by lags 1 to 4.

```
from statsmodels.tsa.stattools import acf, pacf

x = df["gym"].astype(float)

x_diff = x.diff().dropna() # first item is NA
# ACF and PACF plots:

lag_acf = acf(x_diff, nlags=20, fft=True)
lag_pacf = pacf(x_diff, nlags=20, method='ols')

#Plot ACF:
plt.subplot(121)
plt.plot(lag_acf)
plt.axhline(y=0,linestyle='--',color='gray')
plt.axhline(y=-1.96/np.sqrt(len(x_diff)),linestyle='--',color='gray')
plt.axhline(y=1.96/np.sqrt(len(x_diff)),linestyle='--',color='gray')
plt.title('Autocorrelation Function  (q=1)')
```

(continues on next page)

(continued from previous page)

```
#Plot PACF:
plt.subplot(122)
plt.plot(lag_pacf)
plt.axhline(y=0,linestyle='--',color='gray')
plt.axhline(y=-1.96/np.sqrt(len(x_diff)),linestyle='--',color='gray')
plt.axhline(y=1.96/np.sqrt(len(x_diff)),linestyle='--',color='gray')
plt.title('Partial Autocorrelation Function (p=1)')
plt.tight_layout()
```



In this plot, the two dotted lines on either sides of 0 are the confidence interevals. These can be used to determine the p and q values as:

- p: The lag value where the PACF chart crosses the upper confidence interval for the first time, in this case p=1.
- q: The lag value where the ACF chart crosses the upper confidence interval for the first time, in this case q=1.

Fit ARMA model with statsmodels

1. Define the model by calling `ARMA()` and passing in the p and q parameters.
2. The model is prepared on the training data by calling the `fit()` function.
3. Predictions can be made by calling the `predict()` function and specifying the index of the time or times to be predicted.

```
from statsmodels.tsa.arima_model import ARMA
# from statsmodels.tsa.arima.model import ARIMA
```

(continues on next page)

(continued from previous page)

```
model = ARMA(x, order=(1, 1)).fit() # fit model

print(model.summary())
plt.plot(x)
plt.plot(model.predict(), color='red')
plt.title('RSS: %.4f'% sum((model.fittedvalues-x)**2))
```

```
/home/ed203246/anaconda3/lib/python3.7/site-packages/statsmodels/tsa/arima_model.
↪py:472: FutureWarning:
statsmodels.tsa.arima_model.ARMA and statsmodels.tsa.arima_model.ARIMA have
been deprecated in favor of statsmodels.tsa.arima.model.ARIMA (note the .
between arima and model) and
statsmodels.tsa.SARIMAX. These will be removed after the 0.12 release.

statsmodels.tsa.arima.model.ARIMA makes use of the statespace framework and
is both well tested and maintained.

To silence this warning and continue using ARMA and ARIMA until they are
removed, use:

import warnings
warnings.filterwarnings('ignore', 'statsmodels.tsa.arima_model.ARMA',
                        FutureWarning)
warnings.filterwarnings('ignore', 'statsmodels.tsa.arima_model.ARIMA',
                        FutureWarning)

  warnings.warn(ARIMA_DEPRECATION_WARN, FutureWarning)
/home/ed203246/anaconda3/lib/python3.7/site-packages/statsmodels/tsa/base/tsa_
↪model.py:527: ValueWarning: No frequency information was provided, so inferred␣
↪frequency MS will be used.
  % freq, ValueWarning)
```

```
                              ARMA Model Results
==============================================================================
Dep. Variable:                    gym   No. Observations:                  168
Model:                     ARMA(1, 1)   Log Likelihood                -436.852
Method:                       css-mle   S.D. of innovations              3.229
Date:                Fri, 04 Dec 2020   AIC                            881.704
Time:                        13:05:20   BIC                            894.200
Sample:                    01-01-2004   HQIC                           886.776
                         - 12-01-2017
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         36.4315      8.827      4.127      0.000      19.131      53.732
ar.L1.gym      0.9967      0.005    220.566      0.000       0.988       1.006
ma.L1.gym     -0.7494      0.054    -13.931      0.000      -0.855      -0.644
                                    Roots
```

(continues on next page)

(continued from previous page)

```
================================================================================
                 Real          Imaginary           Modulus         Frequency
--------------------------------------------------------------------------------
AR.1           1.0033           +0.0000j            1.0033            0.0000
MA.1           1.3344           +0.0000j            1.3344            0.0000
--------------------------------------------------------------------------------
```

```
Text(0.5, 1.0, 'RSS: 1794.4653')
```