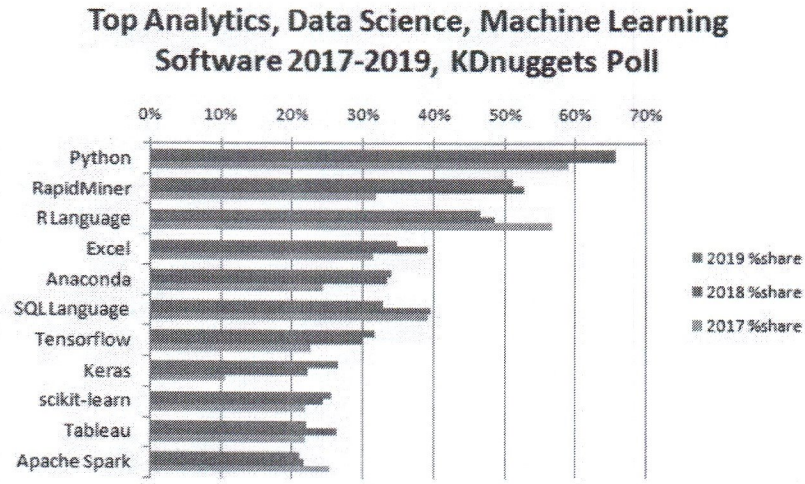
A (Quick!) Introduction to Python

> "*Python is an experiment in how much freedom programmers need.*" (Guido Van Rossum, Python creator)

Why Python?

- open-source
- simple and intuitive
- versatile: from scripting to webapp (e.g. Django framework)
- interpreted (we can use notebooks!)
- largely used for Data Science and ML



Other tutorials

You can find a lot of resources on the Web

- Python Courses by PoUL https://poul.org/2020/05/corsi_python/
- Google's Python Class https://developers.google.com/edu/python/
- Interactive Introduction to Python https://www.learnpython.org/

and even resources of resources

- Python for Programmers https://wiki.python.org/moin/BeginnersGuide/Programmers

How to install

- Option 1: **don't!** with colab, you have an online environment, ready to use.
- Option 2: install a python *distribution*; many are available, we suggest Anaconda for a complete environment, or Miniconda, for a minimal installation.

What is a notebook?

- An easy way to combine text and code.
- The basic component is called "cell": each cell can be either **code** or **text**.
- You can run code cells, update them, and run them again.
- Variables are shared between cells, code is not compiled, but interpreted.

Python Version

In [ ]:
```
!python3 --version
```

```
Python 3.7.10
```

Hello World!

- You can print stuff using the **print** function.
- The argument of the print function must be a string, enclosed between ' or ".
- No ";" required.

In [ ]:
```
# This is a comment in Python

print('Hello World')
```

```
Hello World
```

Variables and Built-in Types

- **Intepreted** language: no need to declare variables.
- **Dynamic typing**: a variable can change its type at runtime.

In [ ]:
```
animal = 'duck' # string
wings = 2 # integer
```

```
avg_weight = (710 + 1440 / 2) # float
can_fly = True # bool


print('A', animal, 'has', wings, 'wings, an average weight of ', avg_weight,' grams.')
# print('A {} has {} wings and an average weight of {} grams.'.format(animal, wings, avg_weight))
print('Can it fly?')
print(can_fly)
```

```
A duck has 2 wings, an average weight of  1430.0  grams.
Can it fly?
True
```

We can inspect at runtime the type of a variable:

In [ ]:
```
print(type(can_fly))
```

```
<class 'bool'>
```

In [ ]:
```
print(animal, type(animal))
animal = 1 # changing the type of a variable is fine
print(animal, type(animal))
```

```
duck <class 'str'>
1 <class 'int'>
```

In [ ]:
```
animal = None # None is the null object, can be used as an initialization sometimes
print(animal, type(animal))
```

```
None <class 'NoneType'>
```

Sequences

Sequences are *ordered* collections of objects, there are three basic ones:

- lists: *mutable* sequences
- tuples: *immutable* sequences
- range: *immutable* sequence of numbers, used in loops

In [ ]:
```
# a list
birds = ['ducks', 'pidgeons', 'chickens', 'blackbirds'] # elements separated by commas
print(birds)
```

```
['ducks', 'pidgeons', 'chickens', 'blackbirds']
```

In [ ]:
```
# a tuple
stuff = (1, 'duck', [1,2,3])
print(stuff)
```

```
(1, 'duck', [1, 2, 3])
```

In [ ]:
```
# a range
some_numbers = range(7)
print(some_numbers)
```

```
range(0, 7)
```

In [ ]:
```
# a range (inside)
some_numbers = range(7)
print(some_numbers, '->', *some_numbers) # we unpack the range with * operator
more_numbers = range(2, 11, 2) # range(start, stop[, step])
print(more_numbers, '->', *more_numbers)
```

```
range(0, 7) -> 0 1 2 3 4 5 6
range(2, 11, 2) -> 2 4 6 8 10
```

We can do some common operations on sequences, thanks to their ordering

In [ ]:
```
# Indexing
print(stuff[0]) # yes, we start counting from 0!
print(birds[-1]) # -1 refers to the last element
```

```
1
blackbirds
```

In [ ]:
```
# Slicing
print(birds)
print(birds[:3]) # extremes are excluded
print(birds[2:])
print(birds[::2]) # a step can also be set
```

```
['ducks', 'pidgeons', 'chickens', 'blackbirds']
['ducks', 'pidgeons', 'chickens']
['chickens', 'blackbirds']
['ducks', 'chickens']
```

In [ ]:
```
# Length
print(len(some_numbers))
print(len(stuff))
print(len(birds))
```

```
7
3
4
```

In [ ]:
```
# Max and min
print(max([2, 3, 5, 1]))
print(min(birds), max(birds)) #lexicographic order for strings
```

```
5
blackbirds pidgeons
```

In [ ]:
```
# Concatenation
animals = ('parrots', 'ducks') + ('pigs',)
print(animals)

birds = birds + ['parrots', 'ducks']
print(birds)
```

```
('parrots', 'ducks', 'pigs')
['ducks', 'pidgeons', 'chickens', 'blackbirds', 'parrots', 'ducks']
```

Immutable types cannot be modified after creation, but lists are mutable, hence some more operations are allowed:

- add an element to a list:

In [ ]:
```
birds.append('eagles')
print(birds)
```

```
['ducks', 'pidgeons', 'chickens', 'blackbirds', 'parrots', 'ducks', 'eagles']
```

- modify elements

In [ ]:
```
birds[1] = 'hawks'
print(birds)
```

```
['ducks', 'hawks', 'chickens', 'blackbirds', 'parrots', 'ducks', 'eagles']
```

- remove an element:

In [ ]:
```
birds.remove('ducks')
print(birds)
```

```
['hawks', 'chickens', 'blackbirds', 'parrots', 'ducks', 'eagles']
```

Strings are characters lists, hence you can do the same operations on them.

In [ ]:
```
string = "parrot"
print(string[:3])
print(string + ' is a bird.')
```

```
par
parrot is a bird.
```

Dict

A mapping object maps *hashable* (immutable) values to arbitrary objects.

In [ ]:
```
observations = {'ducks': 3,
                'pidgdeons': 10,
                'chickens': 0}

# if keys are string we can write equivalently
#observations = dict(ducks=3, pidgeons=10, chickens=0)

print(observations)
print('The value of pidgdeons is', observations['pidgdeons'])
```

```
{'ducks': 3, 'pidgdeons': 10, 'chickens': 0}
The value of pidgdeons is 10
```

Add an element to a dict:

In [ ]:
```
observations['blackbird'] = 1
print(observations)
```

```
{'ducks': 3, 'pidgdeons': 10, 'chickens': 0, 'blackbird': 1}
```

Modify an element:

In [ ]:
```
observations['chickens'] = 1
print(observations)
```

```
{'ducks': 3, 'pidgdeons': 10, 'chickens': 1, 'blackbird': 1}
```

Remove an element:

In [ ]:
```
del observations['chickens']
print(observations)
```

```
{'ducks': 3, 'pidgdeons': 10, 'blackbird': 1}
```

Set

A set object is an unordered collection of distinct *hashable* objects.

In [ ]:
```
bird_set = {'eagle', 'duck', 'chicken'}
print(bird_set)
mammal_set = {'pig', 'squirrel', 'cow'}
print(mammal_set)
```

```
{'chicken', 'eagle', 'duck'}
{'squirrel', 'pig', 'cow'}
```

In [ ]:
```
# add an element to the set
bird_set.add('eagle')
print(bird_set)
bird_set.add('hawk')
print(bird_set)
```

```
{'chicken', 'eagle', 'duck'}
{'chicken', 'eagle', 'hawk', 'duck'}
```

In [ ]:
```
# union
animal_set = mammal_set.union(bird_set)
print(animal_set)
```

```
{'chicken', 'eagle', 'hawk', 'duck', 'cow', 'squirrel', 'pig'}
```

In [ ]:
```
# difference
cute_set = {'squirrel', 'rabbit'}
cannot_stand_up_set = mammal_set.difference(cute_set)
print(cannot_stand_up_set)
```

```
{'pig', 'cow'}
```

Sequences, dicts and sets are *collections* of object, we can always verify if an object belong to a collection:

In [ ]:
```
'cow' in cute_set
```

Out[ ]:
```
False
```

In [ ]:
```
4 in range(1,10,2) # odds number
```

Out[ ]:
```
False
```

In [ ]:
```
'blackbirds' in birds
```

Out[ ]:
```
True
```

If you want to know more about built-in types, visit: https://docs.python.org/3.8/library/stdtypes.html

Branches and Cycles

if

The *if* construct allow to branch your code:

In [ ]:
```
bird = {'legs': 2,
        'wings': 2,
        'arms': 0}

squirrel = {'legs': 2,
            'wings': 0,
            'arms': 2}
```

In [ ]:
```
animal = squirrel

if animal['wings'] > 0: # colon begin the instructions block
  print('This animal has {} wings'.format(animal['wings'])) # the block must be indented
else:
  print('This animal has no wings...')
```

```
This animal has no wings...
```

No paranthesis nor brackets are needed, Python recognizes the code blocks by *indentation*!

In [ ]:
```
animal = bird

if animal['arms'] > 0:
  print('This animal has {} arms'.format(animal['arms']))
elif animal['wings'] > 0:
  print('This animal has no arms... but it has {} wings!'.format(animal['wings']))
```

```
This animal has no arms... but it has 2 wings!
```

for

The *for* construct allows one to iterate over sequences:

In [ ]:
```
for even_number in range(0,10,2):
  print(even_number)
```

```
0
2
4
6
8
```

In [ ]:
```
for i, bird in enumerate(birds):
  print(i, bird)
```

```
0 hawks
1 chickens
2 blackbirds
3 parrots
4 ducks
5 eagles
```

In [ ]:
```
for i, bird in enumerate(birds): # enumerate associate an index to sequence elements
  if i == 0:
    suffix = 'st'
  elif i == 1:
    suffix = 'nd'
  elif i == 2:
    suffix = 'rd'
  else:
    suffix = 'th'
  print('The {}{} bird species is {}'.format(i+1, suffix, bird))
```

```
The 1st bird species is hawks
The 2nd bird species is chickens
The 3rd bird species is blackbirds
The 4th bird species is parrots
The 5th bird species is ducks
The 6th bird species is eagles
```

In [ ]:
```
ck_birds = []
for bird in birds:
  if 'ck' in bird:
    ck_birds.append(bird)
print(ck_birds)
```

```
['chickens', 'blackbirds', 'ducks']
```

a useful shortcut to iterate operations over lists are *list comprehension*

In [ ]:
```
ck_birds = [bird for bird in birds if 'ck' in bird] # we create a list with the collection birds
print(ck_birds)
type(ck_birds)
```

```
['chickens', 'blackbirds', 'ducks']
```

Out[ ]:
```
list
```

In [ ]:
```
squares = []
for number in range(10): # with a single argument, 0 is the default start
  square = number ** 2
```

```
  squares.append(square)
print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

In [ ]:
```
squares = [x**2 for x in range(10)]
print(squares)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

while

repeat the instructions block until the condition is true

In [ ]:
```
birds_l = birds
seen_birds = []
while len(seen_birds) < 3 and len(birds_l) > 0:
  bird = birds_l[-1]
  birds = birds_l[:-1]
  seen_birds.append(bird)
  print(seen_birds)
```

```
['eagles']
['eagles', 'eagles']
['eagles', 'eagles', 'eagles']
```

Functions

In [ ]:
```
def fly(animal):
  if animal in birds:
    print('fly {}!'.format(animal))
  else:
    print('They cannot fly...')

fly('pig')
```

```
They cannot fly...
```

In [ ]:
```
def parabola(x, a, b, c):
  return a * x**2 + b * x + c

print(parabola(1, 2, 1, 0))
```

```
3
```

In [ ]:
```
# a more complex example
def build_parabola(a, b, c):
  def _parabola(x):
    return a * x**2 + b * x + c
  return _parabola # what is happening here? we are returning a function (a closure to be precise, since its context is included)
```

In [ ]:
```
a = int(input()) # read from standard input and cost to int2
b = int(input())
c = int(input())
parabola = build_parabola(a,b,c)
```

```
10
4
6
```

In [ ]:
```
[parabola(x) for x in range(10)]
```

Out[ ]:
```
[6, 20, 54, 108, 182, 276, 390, 524, 678, 852]
```

Classes

In [ ]:
```
class Animal:

  def __init__(self, name, legs, wings, arms): # constructor
    self.name = name
    self.legs = legs
    self.arms = arms
    self.wings = wings

  def walk(self):
    if self.legs > 0:
      print('A {} is walking'.format(self.name))
    else:
      print('A {} cannot walk...'.format(self.name))

  def __repr__(self):
    return self.name
```

In [ ]:
```
dog = Animal('husky', 4, 0, 0)
print(dog)

# use methods
dog.walk()

# access to attributes
print(dog.wings)
```

```
husky
A husky is walking
0
```

In [ ]:
```
# Inheritance

class Dog(Animal):

  def __init__(self, name): # modify and reuse the constructor
    super().__init__(name, 4, 0, 0)

  # implement new methods
  def bark(self):
    print('woof!')
```

```
  # inherits father methods

  # and ovverrides some

  def __repr__(self):
    return u"\U0001F415"
```

In [ ]:
```
doge = Dog('Shiba Inu')
print(doge)
doge.bark()
doge.walk()
```

```
🐕
woof!
A Shiba Inu is walking
```

Some principles

- *Easier Ask for Forgiveness than Permission* (EAFP https://docs.python.org/3/glossary.html#term-eafp): assume the existence of valid keys or attributes and catches exceptions if the assumption proves false.
- *Duck Typing* ("If it walks like a duck and it quacks like a duck, then it must be a duck"): do not check types, check attributes!

In [ ]:
```
class Duck(Animal):
  def __init__(self, name):
    super().__init__(name, 2, 0, 2)


  def fly(self):
      print("Duck flying")

duffy = Animal('Duffy', 2, 0, 2)

def fly():
  print('Duck flying')

duffy.fly = fly


# We do not check wheter the object belong to a class
# but only if it can perform a certain action
for animal in (Duck('Donald'), doge, duffy):
  try:
    animal.fly()
  except:
    print('{} is clearly not a duck!'.format(animal))
```

```
Duck flying
🐕 is clearly not a duck!
Duck flying
```

To summarize, Python allow you some freedom in typing, use it!

Latest versions of Python allows for some *static typing*, if you are curious about it, check **mypy**: http://mypy-lang.org/

Useful Libraries

NumPy

"The fundamental package for scientific computing with Python"

It provides efficient implementations to deal with vectors, matrices, linear algebra, basic statistics...

- full documentation at https://numpy.org
- a visual guide at https://betterprogramming.pub/numpy-illustrated-the-visual-guide-to-numpy-3b1d4976de1d
- a more complete tutorial at https://realpython.com/numpy-tutorial/

How to use it?

In [ ]:
```
import numpy as np
```

Arrays and matrices

In [ ]:
```
# array
v = np.array([0,1,2,3,4])
print('We created a', v.shape, 'dimensional matrix\n', v, '\nof type', type(v))
```

```
We created a (5,) dimensional matrix
 [0 1 2 3 4]
of type <class 'numpy.ndarray'>
```

In [ ]:
```
# matrix
M = np.array([[0,1,0], [1,0,0], [1,1,0], [1,1,1]])
print('We created a', M.shape, 'dimensional matrix\n', M, '\nof type', type(M))
```

```
We created a (4, 3) dimensional matrix
 [[0 1 0]
 [1 0 0]
 [1 1 0]
 [1 1 1]]
of type <class 'numpy.ndarray'>
```

In [ ]:
```
# indexing
a = v[3]        # the 4th element of v
b = v[-1]       # the last element of v
c = M[2,1]      # an element of M (3rd row, 2nd column)
d = M[:,1]      # the 2nd column of M
e = M[1:3,0:2]  # extract a sub-matrix
```

In [ ]:
```
# transpose
```

```
M_transpose = M.T   # or M.transpose()
print(M_transpose)
```

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

In [ ]:
```
# element-wise operations
v2 = v * 2  # element-wise multiplication
v3 = v / 4  # element-wise division
M2 = M * 10 # element-wise multiplication
print('v2:', v2)
print('v3:', v3)
print('M2:', M2)
```

```
v2: [0 2 4 6 8]
v3: [0.   0.25 0.5  0.75 1.  ]
M2: [[ 0 10  0]
 [10  0  0]
 [10 10  0]
 [10 10 10]]
```

In [ ]:
```
# scalar product
a = np.array([2, 1])
b = np.array([3, 4])
scalar_product = a.dot(b)

# matrix product
A = np.array([[2, 1, 4], [2, 2, 3]])
B = np.array([[3, 4], [3, 6], [7, 2]])
matrix_product = A.dot(B)
print('A', A.shape, ':\n', A, '\n\nB', B.shape, ':\n', B, '\n\nA*B', matrix_product.shape, ':\n', matrix_product)
```

```
A (2, 3) :
 [[2 1 4]
 [2 2 3]]

B (3, 2) :
 [[3 4]
 [3 6]
 [7 2]]

A*B (2, 2) :
 [[37 22]
 [33 26]]
```

Some notable functions

In [ ]:
```
# special matrices
M = np.zeros((2,3))
print(M.shape, 'matrix with zero entries:\n', M)
M = np.ones((3,2))
print(M.shape, 'matrix with ones entries:\n', M)
M = np.eye(3)
print(M.shape, 'identity matrix:\n', M)
M = np.diag([1, 2, 3])
print(M.shape, 'diagonal matrix with specified diagonal:\n', M)
```

```
(2, 3) matrix with zero entries:
 [[0. 0. 0.]
 [0. 0. 0.]]
(3, 2) matrix with ones entries:
 [[1. 1.]
 [1. 1.]
 [1. 1.]]
(3, 3) identity matrix:
 [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
(3, 3) diagonal matrix with specified diagonal:
 [[1 0 0]
 [0 2 0]
 [0 0 3]]
```

In [ ]:
```
# maximum, minimum, mean of a vector
v = np.array([1, 3, -3, 10, 4, -1])
max = np.max(v)
argmax = np.argmax(v)
min = np.min(v)
argmin = np.argmin(v)
mean = np.mean(v)
std = np.std(v)
print('The vector', v, 'has maximum', max, 'at position', argmax,'and minimum',
      min, 'at position', argmin, '\n its mean is', mean, 'and standard deviation is', std)
```

```
The vector [ 1  3 -3 10  4 -1] has maximum 10 at position 3 and minimum -3 at position 2
 its mean is 2.3333333333333335 and standard deviation is 4.14996653266291
```

In [ ]:
```
# maximum, minimum, mean of a matrix (same as before)
v = np.array([[1, 3, -3], [10, 4, -1]])
max = np.max(v)
max_first_row = np.max(v[0,:])
mean = np.mean(v)
std = np.std(v)
print('The matrix\n', v, '\nhas maximum', max, ', its first row has maximum', max_first_row)
```

```
The matrix
 [[ 1  3 -3]
 [10  4 -1]]
has maximum 10 , its first row has maximum 3
```

In [ ]:
```
# some logical functions
v = np.array([3., -5., 7., 4.5, -1.5])
any_greater_than_2 = np.any(v > 2)
all_greater_than_2 = np.all(v > 2)
print('At least one element in', v, 'is greater than 2 ->', any_greater_than_2)
print('All the elements in', v, 'are greater than 2 ->', all_greater_than_2)
```

```
At least one element in [ 3.  -5.   7.   4.5 -1.5] is greater than 2 -> True
All the elements in [ 3.  -5.   7.   4.5 -1.5] are greater than 2 -> False
```

In [ ]:
```
# concatenate
v1 = np.array([1, 2])
```

```
v2 = np.array([-1, -2, -3])
v3 = np.concatenate((v1, v2), axis=0)
print(v3)
```

```
[ 1  2 -1 -2 -3]
```

In [ ]:
```
# arange (python range on steroids)
v = np.arange(-2.5, 2.5, 0.5)
print(v, type(v))
```

```
[-2.5 -2.  -1.5 -1.  -0.5  0.   0.5  1.   1.5  2. ] <class 'numpy.ndarray'>
```

Linear algebra and statistic

Linear algebra with *linalg*

In [ ]:
```
M = np.array([[1, 2],[3, 4]])
rank = np.linalg.matrix_rank(M)   # matrix rank
inverse = np.linalg.inv(M)        # matrix inversion
print('M has rank', rank, 'and inverse\n', inverse)
```

```
M has rank 2 and inverse
 [[-2.   1. ]
 [ 1.5 -0.5]]
```

In [ ]:
```
# extract eigenvalues and eigenvectors of M
eigenvalues, eigenvectors = np.linalg.eig(M)
print('Eigenvalues\n', eigenvalues)
print('Eigenvectors\n', eigenvectors)
```

```
Egienvalues
 [-0.37228132  5.37228132]
Eigenvectors
 [[-0.82456484 -0.41597356]
 [ 0.56576746 -0.90937671]]
```

In [ ]:
```
# solve a linear system Ax = b
A = np.array([[1, 2],[3, 4]])
b = np.array([3, -3])
x = np.linalg.solve(A, b)
print(x)
```

```
[-9.  6.]
```

Random sampling with *random*

In [ ]:
```
# we can set a seed for the number generator
np.random.seed(0)
```

In [ ]:
```
# sample a random dataset
data1 = np.random.rand(2, 2)    # (2,2) matrix with random entries in [0,1]
data2 = np.random.randn(2, 2)   # (2,2) matrix with random entries from a standard normal distribution
```

In [ ]:
```
# sample from a parametric distribution
np.random.normal(5, 2.5)        # normal distribution with 5 mean and 2.5 std
```

Out[ ]:
```
4.741952870516105
```

A whole lot of distributions https://docs.scipy.org/doc//numpy-1.10.4/reference/routines.random.html

Pandas

Well-known data manipulation tool. We essentially use it to manage *dataframes*, which is a common data sctructure with symbolic indexing (read more at https://pandas.pydata.org)

In [ ]:
```
import pandas as pd
```

We can easily import a datset from a local file or a web source (the file is typically in csv format)

In [ ]:
```
# e.g., the iris dataset
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data"
names = ['sepal-length', 'sepal-width', 'petal-length', 'petal-width', 'class']
df = pd.read_csv(url, names=names)
```

In [ ]:
```
# it is now stored in a dataframe
print(df.info, type(df))
```

```
<bound method DataFrame.info of      sepal-length  sepal-width  petal-length  petal-width           class
0             5.1          3.5           1.4          0.2     Iris-setosa
1             4.9          3.0           1.4          0.2     Iris-setosa
2             4.7          3.2           1.3          0.2     Iris-setosa
3             4.6          3.1           1.5          0.2     Iris-setosa
4             5.0          3.6           1.4          0.2     Iris-setosa
..            ...          ...           ...          ...             ...
145           6.7          3.0           5.2          2.3  Iris-virginica
146           6.3          2.5           5.0          1.9  Iris-virginica
147           6.5          3.0           5.2          2.0  Iris-virginica
148           6.2          3.4           5.4          2.3  Iris-virginica
149           5.9          3.0           5.1          1.8  Iris-virginica

[150 rows x 5 columns]> <class 'pandas.core.frame.DataFrame'>
```

In [ ]:
```
# look at the head (first 10 rows) of the dataframe
df.head(10)
```

Out[ ]:

	sepal-length	sepal-width	petal-length	petal-width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa

	sepal-length	sepal-width	petal-length	petal-width	class
7	5.0	3.4	1.5	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa

In [ ]:
```
# access the dataframe with numeric indexing (similar to NumPy matrices)
df.iloc[10:15, 1:3]
```

Out[ ]:

	sepal-width	petal-length
10	3.7	1.5
11	3.4	1.6
12	3.0	1.4
13	3.0	1.1
14	4.0	1.2

In [ ]:
```
# access the dataframe with symbolic indexing (similar to dictionaries)
df["sepal-width"]
```

Out[ ]:
```
0      3.5
1      3.0
2      3.2
3      3.1
4      3.6
      ...
145    3.0
146    2.5
147    3.0
148    3.4
149    3.0
Name: sepal-width, Length: 150, dtype: float64
```

In [ ]:
```
# access the dataframe with logical conditions (as it is common in Matlab)
df[(df["sepal-width"] > 3.0) & (df["petal-length"] < 1.5)]
```

Out[ ]:

	sepal-length	sepal-width	petal-length	petal-width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa
14	5.8	4.0	1.2	0.2	Iris-setosa
16	5.4	3.9	1.3	0.4	Iris-setosa
17	5.1	3.5	1.4	0.3	Iris-setosa
22	4.6	3.6	1.0	0.2	Iris-setosa
28	5.2	3.4	1.4	0.2	Iris-setosa
33	5.5	4.2	1.4	0.2	Iris-setosa
35	5.0	3.2	1.2	0.2	Iris-setosa
36	5.5	3.5	1.3	0.2	Iris-setosa
40	5.0	3.5	1.3	0.3	Iris-setosa
42	4.4	3.2	1.3	0.2	Iris-setosa
47	4.6	3.2	1.4	0.2	Iris-setosa
49	5.0	3.3	1.4	0.2	Iris-setosa

These are the very basics, but you will not need much more!

Scikit-Learn

"Machine Learning in Python"

A huge collection of Machine Learning methods already implemented in Python, including the algorithms we will see during the lectures and exercise sessions (and many more https://scikit-learn.org/stable/).

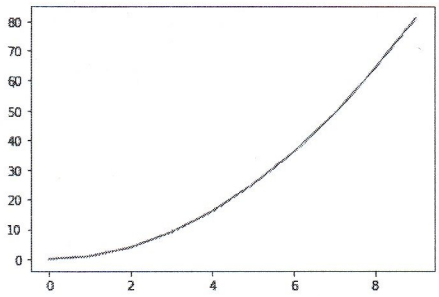
In [ ]:
```
# just a simple example, wait for the exercise sessions to know more
from sklearn import datasets, linear_model

# create the dataset
dataset_X = np.array([[1, 1, 1], [3, 2, 3], [-1, 0, 3]])
dataset_y = np.array([0.7, 1, 0.3])

# create a linear regression object
regr = linear_model.LinearRegression()

# train the model with the function fit
regr.fit(dataset_X, dataset_y)

# et voilà, we have our linear model coefficients
print('Coefficients: \n', regr.coef_)
```

```
Coefficients:
 [ 0.14   0.07  -0.025]
```

Matplotlib

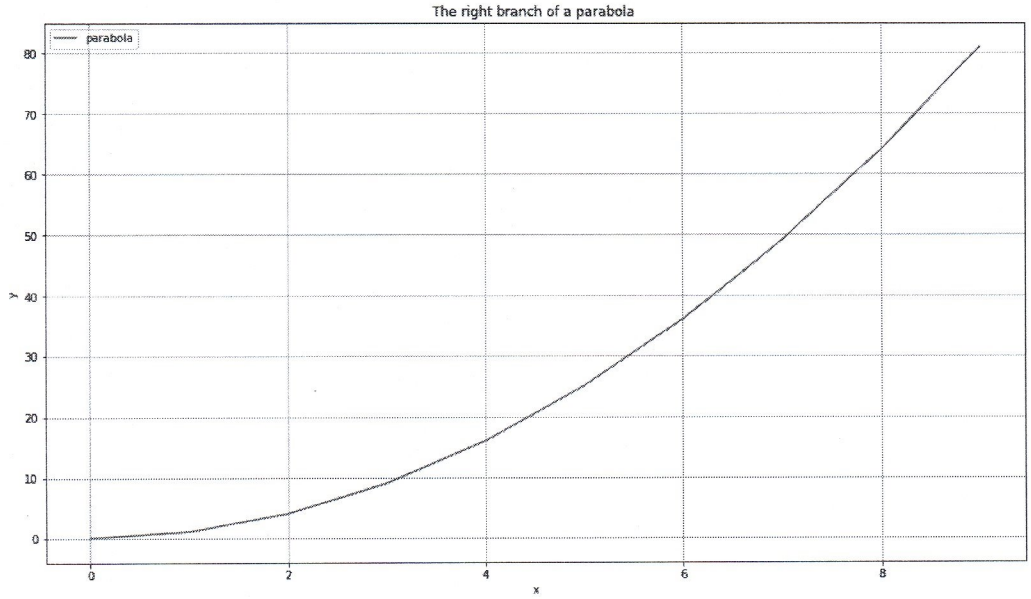
Matplotlib is a library for creating visualizations in python

In [ ]:
```
import matplotlib.pyplot as plt
plt.plot([x**2 for x in range(10)])
plt.show()
```



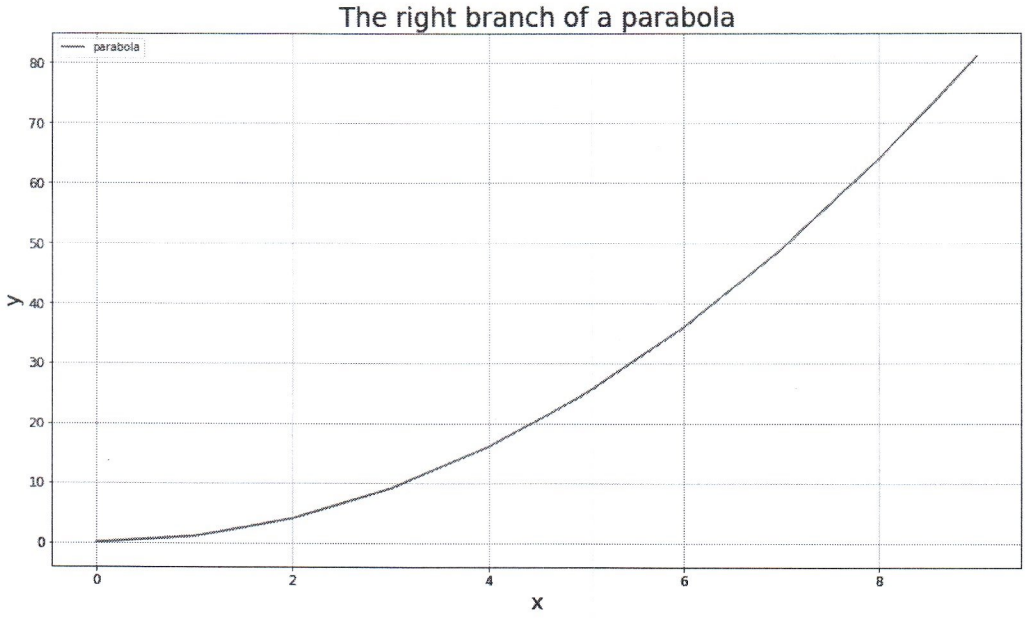
That's quick, but we can do something better:

In [ ]:
```
fig = plt.figure(figsize=(16, 9))
plt.plot([x**2 for x in range(10)], label='parabola')
plt.xlabel('x')
plt.ylabel('y')
plt.title('The right branch of a parabola')
plt.legend()
plt.grid()
plt.show()
```



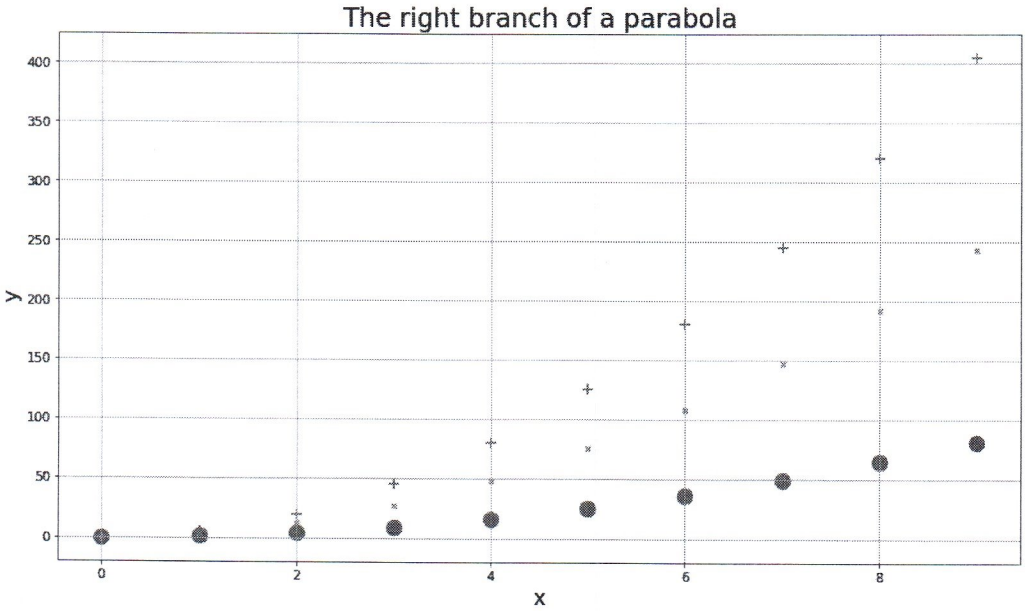
In [ ]:
```
# you can write on a file your preferences
with open('mystyle.mplstyle','w+') as f:
  f.write('axes.titlesize : 24\
           \naxes.labelsize : 20\
           \nlines.linewidth : 2\
           \nlines.markersize : 5\
           \nxtick.labelsize : 12\
           \nytick.labelsize : 12')
```

In [ ]:
```
# and then use them for all your plots
with plt.style.context('mystyle.mplstyle'):
  fig = plt.figure(figsize=(16, 9))
  plt.plot([x**2 for x in range(10)], label='parabola')
  plt.xlabel('x')
  plt.ylabel('y')
  plt.title('The right branch of a parabola')
  plt.legend()
  plt.grid()
  plt.show()
```



Scatter plots:

In [ ]:
```
with plt.style.context('mystyle.mplstyle'):
 fig = plt.figure(figsize=(16, 9))
 x = list(range(10))
 y = [x**2 for x in range(10)]
 plt.scatter(x, y, s=200)

 y = [3*x**2 for x in range(10)]
 plt.scatter(x, y, marker='x', s=20)

 y = [5*x**2 for x in range(10)]
 plt.scatter(x, y, marker='+', s=100)

 plt.xlabel('x')
 plt.ylabel('y')
 plt.title('The right branch of a parabola')
 plt.grid()
 plt.show()
```



3D Plots:

In [ ]:
```
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
fig = plt.figure(figsize=(16, 9))
ax = fig.add_subplot(111, projection='3d')

def randrange(n, vmin, vmax):
    '''
    Helper function to make an array of random numbers having shape (n, )
    with each number distributed Uniform(vmin, vmax).
    '''
    return (vmax - vmin)*np.random.rand(n) + vmin

n = 100

# For each set of style and range settings, plot n random points in the box
# defined by x in [23, 32], y in [0, 100], z in [zlow, zhigh].
for c, m, zlow, zhigh in [('r', 'o', -50, -25), ('b', '^', -30, -5)]:
    xs = randrange(n, 23, 32)
    ys = randrange(n, 0, 100)
    zs = randrange(n, zlow, zhigh)
```

```
    ax.scatter(xs, ys, zs, c=c, marker=m)


ax.set_xlabel('x_0')
ax.set_ylabel('x_1')
ax.set_zlabel('x_2')
ax.view_init(elev=20., azim=32)
plt.grid()
plt.show()
```

In [ ]:

```
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
fig = plt.figure(figsize=(16, 9))
ax = fig.add_subplot(111, projection='3d')
s = 70



theta = np.linspace(-4 * np.pi, 4 * np.pi, 100)
z = np.linspace(-2, 2, 100)
r = z**2 + 1
x = r * np.sin(theta)
y = r * np.cos(theta)
ax.plot(x, y, z, label='parametric curve')

ax.legend()

ax.set_xlabel('x_0')
ax.set_ylabel('x_1')
ax.set_zlabel('x_2')
ax.view_init(elev=20., azim=32)
plt.grid()
plt.show()
```

1 Hitchhiker Guide to the Machine Learning Course

In this document we review some preliminary concepts of linear algebra, and statistics used in the course *Machine Leaning*, and how these tools can be used in Pyhton. This is not supposed to be an exhaustive document on these topics, for more information you should refer to:

- Petersen, K.B., Pedersen, M.S., "The matrix cookbook", 2008;
- Bishop, C.M., "Pattern recognition and machine learning", 2006, Springer;
- Montgomery, D.C., Runger, G.C., "Applied statistics and probability for engineers", 2010, John Wiley & Sons.

1.1 Linear Algebra

Let us recall some basics on the use of linear algebra about derivation and the definition of an eigenvector problem.

1.1.1 Matrix Derivatives

Consider a dataset composed of a set of N inputs $\mathbf{x}_i := (x_{i1}, \ldots, x_{iD})$, with $x_{ij} \in \mathbb{R}$, each of which has dimension D and a target $t_i \in \mathbb{R}$, corresponding to input $\mathbf{x}_i$. The goal of the least square method is to find a column vector $\mathbf{w} := (w_1, \ldots, w_D)^\top$ that minimize the following loss function:

$$L(\mathbf{w}) := \frac{1}{2} \sum_{i=1}^{N} \left(t_i - \sum_{j=1}^{D} x_{ij} w_j \right)^2 . \tag{1.1}$$

Equivalently, we can use a matrix notation to express the same goal. More specifically,

we define the input matrix as:

$$X = \begin{pmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_N \end{pmatrix} = \begin{pmatrix} x_{11} & \cdots & x_{1D} \\ \vdots & & \vdots \\ x_{N1} & \cdots & x_{ND} \end{pmatrix},$$

and the target vector as $\mathbf{t} = (t_1, \dots t_N)^\top$. The loss one want to minimize in the least square problem becomes:

$$L(\mathbf{w}) := \frac{1}{2}||\mathbf{t} - X\mathbf{w}||_2^2 = \frac{1}{2}(\mathbf{t} - X\mathbf{w})^\top(\mathbf{t} - X\mathbf{w}). \tag{1.2}$$

Let us verify that Equation (1.2) is equivalent to Equation (1.1). By the definition of product between two matrices we have:

$$r_i = (\mathbf{t} - X\mathbf{w})_i := t_i - (x_{i1}, \dots, x_{iD}) \begin{pmatrix} w_1 \\ \vdots \\ w_D \end{pmatrix} = t_i - \sum_{j=1}^{D} x_{ij} w_j,$$

where $(\cdot)_i$ represents ith element of a vector and defining the residual vector $\mathbf{r} := (r_1, \dots, r_N)^\top$ we have:

$$L(\mathbf{w}) = \frac{1}{2}\mathbf{r}^\top\mathbf{r} = \frac{1}{2}\sum_{i=1}^{N}(r_i)^2 = \frac{1}{2}\sum_{i=1}^{N}\left(t_i - \sum_{j=1}^{D} x_{ij} w_j\right)^2,$$

which is the initial expression for the loss.

To minimize the loss we might compute the its derivatives w.r.t. each element of $\mathbf{w}$ and equal it to zero. To do that, we define the gradient as follows:

$$\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} := \left(\frac{\partial L(\mathbf{w})}{\partial w_1}, \dots, \frac{\partial L(\mathbf{w})}{\partial w_D}\right). \tag{1.3}$$

Depending on the form of the loss we will be able to compute it in closed form or not. Usually, the matrix notation is less intuitive, but requires less computations to perform the same operations. Again, let us check the equivalence between the two forms. Consider an element of the gradient and the formulation of the loss in Equation (1.1). We have:

$$\begin{aligned}\left(\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}}\right)_h &= \frac{\partial L(\mathbf{w})}{\partial w_h} = \frac{\partial}{\partial w_h}\left[\frac{1}{2}\sum_{i=1}^{N}\left(t_i - \sum_{j=1}^{D} x_{ij} w_j\right)^2\right] \\ &= \frac{1}{2}\sum_{i=1}^{N}\frac{\partial}{\partial w_h}\left[\left(t_i - \sum_{j=1}^{D} x_{ij} w_j\right)^2\right] = \sum_{i=1}^{N}\left[-x_{ih}\left(t_i - \sum_{j=1}^{D} x_{ij} w_j\right)\right].\end{aligned}$$

Conversely, using the rule to derive matrix (which are similar to the one used to derive scalar functions), and using Equation (1.2) we have:

$$\begin{aligned}\frac{\partial L(\mathbf{w})}{\partial \mathbf{w}} &= \frac{\partial}{\partial \mathbf{w}}\left[\frac{1}{2}(\mathbf{t}-X\mathbf{w})^\top(\mathbf{t}-X\mathbf{w})\right] \\ &= -X^\top(\mathbf{t}-X\mathbf{w}).\end{aligned}$$

Let us compare this last derivatives with the previously derived one:

$$\begin{aligned}\left(-X^\top(\mathbf{t}-X\mathbf{w})\right)_h &= (-x_{1h},\ldots,-x_{Nh})(\mathbf{t}-X\mathbf{w}) \\ &= (-x_{1h},\ldots,-x_{Nh})\begin{pmatrix} r_1 \\ \vdots \\ r_N \end{pmatrix} = \sum_{i=1}^{N} -x_{ih}\, r_i = \sum_{i=1}^{N}\left[-x_{ih}\left(t_i - \sum_{j=1}^{D} x_{ij}w_j\right)\right],\end{aligned}$$

which concludes our check.

Consequently, the minimum point of the previous loss function is computed looking at its stationary points:

$$X^\top(\mathbf{t}-X\mathbf{w}) = 0 X^\top X\mathbf{w} = X^\top\mathbf{t} \tag{1.4}$$

1.1.2 Eigenvalues and Eigenvectors

Given a square matrix $A \in \mathbb{R}^{N\times N}$, its *eigenvalues* $\lambda_1,\ldots,\lambda_N$ and corresponding *eigenvectors* $\mathbf{v}_1,\ldots,\mathbf{v}_N$ are defined from the *eigenvector equations*:

$$A\mathbf{v}_i = \lambda_i\mathbf{v}_i,$$

for each $i \in \{1,\ldots,N\}$. Intuitively, an eigenvector $\mathbf{v}_i$ is a direction in the space that is only stretched when the transformation A is applied. The corresponding eigenvalue λ_i is the factor this vector is stretched. See Figure 1.1 for a simple example.

Using the matricial formulation, we have that to find the generic pair $(\lambda,\mathbf{v})$ it is possible to solve the following:

$$\begin{aligned}A\mathbf{v} &= \lambda\mathbf{v} \\ (A-\lambda I_N)\mathbf{v} &= 0,\end{aligned}$$

where I_N is the identity matrix of order N. The previous problem has a non-null solution only if the rank of the matrix $A-\lambda I_N$ is full, or, equivalently if its determinant is non-null. The equation corresponding to this condition is:

$$|A-\lambda I_N| = 0,$$

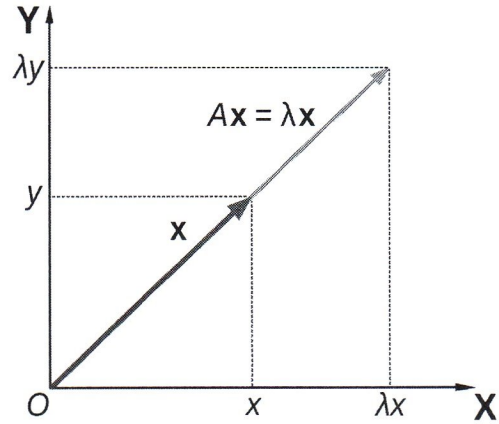


Figure 1.1: All the vectors in the direction of the eigenvector $\mathbf{x}$ are only stretched of a factor λ by the application of the linear transformation A. Vectors in other directions might also suffer from a change in terms of direction.

where we denoted the determinant of a matrix using $|\cdot|$. This is also known as the characteristic equation. Since it is a polynomial of order N in λ, it must have N solutions, which results in being the N eigenvalues of the matrix A.

Once we solved the previous equation, it is possible to show that the eigenvalues $\lambda_1, \ldots, \lambda_N$ satisfy the following properties:

- the rank of the matrix A is equal to the number of nonzero eigenvalues;
- the determinant of A is equal to the product of its eigenvalues:

$$|A| = \prod_{i=1}^{N} \lambda_i;$$

- the trace of A is equal to the sum of its eigenvalues:

$$Tr(A) = \sum_{i=1}^{N} \lambda_i.$$

Moreover, the positivity of the eigenvectors also determine some properties of the linear transformation. In the specific:

- a matrix A is said to be *positive definite*, if $\mathbf{x}^\top A\mathbf{x} > 0$ for all possible vectors $\mathbf{x} \in \mathbb{R}^N \setminus \{\mathbf{0}\}$. Equivalently, a positive definite matrix has all positive eigenvalues, i.e., $\lambda_i > 0 \; \forall i$.

- a matrix A is said to be *semi-positive definite*, if $\mathbf{x}^\top A\mathbf{x} \geq 0$ for all possible vectors $\mathbf{x} \in \mathbb{R}^N \setminus \{\mathbf{0}\}$. Equivalently, a semi-positive definite matrix has all non-negative eigenvalues, i.e., $\lambda_i \geq 0 \; \forall i$.

*** Exercise 1.1**

Show that the second derivatives of the previously considered quadratic form is:

$$\frac{\partial^2 L(\mathbf{w})}{\partial \mathbf{w} \partial \mathbf{w}^\top} = X^\top X.$$

1.2 Discrete Random Variables

A discrete random variable X is a variable with values in a discrete set E whose value is determined by a stochastic phenomenon, i.e., we are not able to predict its value even if we are given precise information about the phenomenon. For instance, consider a 20-faced dice: the event of throwing it can be modeled as a random variable X taking values in the finite set of events $E = \{1, \ldots, 20\}$, since we are not able to predict precisely which value might occur, even if we are given all the characteristics of the dice (e.g., dimensions, initial position, speed).

To properly model this phenomenon, we define a *probability function* $\mathbb{P} : E \to [0, 1]$ which tells you how often the event i belonging to a discrete set of events E occurs (e.g., probability that by trowing the dice you get 3) as:

$$\mathbb{P}(X = i) := \frac{|i|}{|E|},$$

where $|i|$ is the measure of the favorable events set and $|E|$ is the measure of set of the possible events. For instance, for a 20-faced dice we have:

$$\mathbb{P}(X = i) = \frac{1}{20},$$

since all the faces are equally likely to occur. In this case we have some properties that the probability function should have:

- $0 \leq \mathbb{P}(X = i) \leq 1$: an event can occur at least with zero probability and at most with probability one (the favorable cases are at most equal to the possible ones);
- $\sum_{i \in E} \mathbb{P}(X = i) = 1$: if we consider probability of the set of all the possible events E we should get one.

While this is the probability of a single events, we might be interested in the probability of multiple events. For instance, one might be interested in the probability that the dice

roll is small, say less than a specific value. This probability is captured by the *cumulative function* $F : E \to [0, 1]$, which specifies the probability that the random variable is lower than i:

$$F(i) := \mathbb{P}(X \leq i) = \sum_{h=1}^{i} \mathbb{P}(X = h) = \sum_{h \in E, h \leq i} \frac{|h|}{|E|}.$$

This newly defined function satisfies:

- $0 \leq F(i) \leq 1$: the sum of the probabilities should be between zero and one:
- $F(i) = 0, \ \forall i < \min_{h \in E} h$: if we consider a value small enough (smaller than the smaller element in the event space) the cumulative function should have value zero;
- $F(i) = 1, \ \forall i \geq \max_{h \in E} h$: if we consider a value large enough (larger or equal than the element in the event space) the cumulative function should have value one.

In the 20-faced dice case we have:

$$F(i) = \sum_{h=1}^{i} \frac{1}{20} = \frac{i}{20}.$$

There are two quantities which might be of major interest in the study of a random variable: the *expected value* and the *variance*. In the case of a dice, the former tells you what is value on average one could get from trowing the dice repeatedly, the latter gives us information about the spread in the single results we would have. Formally, for a generic random variable we have:

$$\mathbb{E}[X] := \sum_{i \in E} i \, P(X = i);$$
$$Var(X) := \sum_{i \in E} (\mathbb{E}[X] - i)^2 P(X = i).$$

In the case of the 20-faced dice we have:

$$\mathbb{E}[X] := \sum_{i=1}^{20} \frac{i}{20} = \frac{1}{20} \frac{20(20+1)}{2} = \frac{21}{2};$$
$$Var(X) := \sum_{i=1}^{20} \frac{\left(\frac{21}{2} - i\right)^2}{20} = \frac{57}{4}.$$

Sometimes the "spread" of a random variable is evaluated with the *standard deviation*, which is the square root of the variance $std(X) = \sqrt{Var(X)}$. This is due to the fact that if we are considering random variables expressed in some unit of measure, the variance is not compatible with the measurement itself, but its squared root does.

Remark 1. *Notice that these are the values obtained by knowing the random variable. If we only have some samples coming from the random variable, we are not able to compute the expected value and the variance, but we would be able to estimate their real values. We will see how in what follows.*

1.3 Continuous Random Variables

All the concept presented for a random variable X taking discrete values in a set E can be extended to the ones taking values in a continuous 1D set $\Omega \subseteq \mathbb{R}$. For instance, when we perform a length measurement, its values belongs to the interval $[0, +\infty)$. Similarly to what we did for discrete random variables with the probability function, we define the a *probability density function* (pdf) as follows:

$$f(x) := \lim_{\delta x \to 0} \frac{\mathbb{P}(x \leq X \leq x + \delta x)}{\delta x},$$

since, in this case, the probability of the event $X = x$ (having zero measure in a 1D space) is zero. In this case, the properties of the pdf are:

$$f(x) \geq 0 \ \ \forall x \in \Omega,$$
$$\int_{x \in \Omega} f(x)dx = 1.$$

A definition similar to what has been provided with the cumulative function for discrete variables can be provided also in the continuous case. When we want to evaluate the probabilities of intervals, we might resort to the *Cumulative Distribution Function* (CDF), defined as:

$$F(x) := \int_{s \in \Omega, s \leq x} f(s)ds,$$

having the following properties:

$$0 \leq F(x) \leq 1 \ \forall x \in \Omega,$$
$$F\left(\min_{x \in \Omega} x - \varepsilon\right) = 0,$$
$$F\left(\max_{x \in \Omega} x\right) = 1,$$

where $\varepsilon > 0$.

Similarly to the discrete case, the expected value and the variance are defined as:

$$\mu = \mathbb{E}[X] := \int_{x \in \Omega} xf(x)dx;$$
$$\sigma^2 = Var(X) := \int_{x \in \Omega} (\mathbb{E}[X] - x)^2 f(x)dx.$$

Among the most used continuous distributions we have the Gaussian one $X \sim \mathcal{N}(\mu, \sigma^2)$ defined over $\Omega = \mathbb{R}$ and having:

$$f(x; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$
$$F(x; \mu, \sigma) = \int_{-\infty}^{x} f(t; \mu, \sigma) dt,$$

the uniform random variable $X \sim \mathcal{U}([0, 1])$ defined over $\Omega = [0; 1]$ and having:

$$f(x) = 1,$$
$$F(x) = x,$$

and the Beta random variable $X \sim Beta(\alpha, \beta)$ defined over $\Omega = [0; 1]$ and having:

$$f(x) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha, \beta)},$$

where $B(\alpha, \beta) := \int_0^1 x^{\alpha-1}(1-x)^{\beta-1}\, dx$ is a normalization term, and the CFD cannot be provided analytically.

1.4 Univariate Distributions in Python

In Python it is possible to define objects to handle different distributions in the package `scipy.stats`, among the others:

- `binom` Binomial distribution (discrete, `n` and `p` parameters);
- `norm` Gaussian distribution (continuous, `loc` and `scale` parameters);
- `unif` Uniform distribution (continuous, `loc` and `scale` parameters);
- `beta` Beta distribution (continuous, `a` and `b` parameters);

each of which will have specific parameters.

Remark 2. *Here the parameter `scale` for the Gaussian distribution is the standard deviation. In some other functions the second parameter of the Gaussian distribution is the variance. Check the documentation to know which one of the two is needed.*

If we want to get the pdf of a given point from a random variable with Gaussian distribution with mean 3 and standard deviation 4 we can write:

```
1 from scipy.stats import norm
2 norm.pdf(5, loc = 3, scale = 4) # pdf at x = 5
```

From the same distribution we can also get the value of the pdf function at a specific value, the value of the CDF function at a specific value, or the CDF function takes a specific value, in the following way:

```
norm.pdf(5, loc = 3, scale = 4) # pdf at x = 5
norm.cdf(3, loc = 3, scale = 4) # CDF at x = 3
norm.ppf(0.05, loc = 3, scale = 4) # x s.t. the CDF is 0.05
```

Another useful functionality available in Python allows to draw samples from a specific distribution. For instance, if we want to sample 100 realizations of the Gaussian variable X we could do:

```
norm.rvs(loc = 3, scale = 4, size = 100)
```

1.5 Multivariate Distributions in Python

Up to now we described only distributions whose domain was a subset of $\mathbb{R}$. There exists also some distributions which takes values in $\Omega \subseteq \mathbb{R}^n$, with $n \in \mathbb{N}$, $n > 1$. They are usually called *multivariate distributions*. If we want to resort to such random variable in Python we should use different tools. For instance, for the multivariate Gaussian we have:

```
from scipy.stats import multivariate_normal
multivariate_normal.pdf([0, 0], mean = [0.5, -0.2], cov = [[2.0, 0.3], [0.3,
    0.5]])
multivariate_normal.rvs(mean = [0.5, -0.2], cov = [[2.0, 0.3], [0.3, 0.5]],
    size = 100)
```

where $x = (0, 0)$ is the point considered, `mean` is the mean vector, `cov` is the covariance matrix, and `size` is the number of instances to be sampled.

For instance, if we want to sample 100 points from a 5-variate normal distribution with mean $[1\ 1\ 1\ 1\ 1]^T$ with identity covariance matrix and plot these points, we can use:

```
import numpy as np
multivariate_normal.rvs(mean = [1 1 1 1 1], cov = np.eye(5), size = 100)
```

1.6 Central Limit Theorem

In the previous sections we assumed that the distribution was known, i.e., that the parameters characterizing the distribution were known. Otherwise we built some *estimators* to infer them from data coming from the specified distribution. For instance, given a set of N independent samples $\{x_1, \ldots, x_n\}$ coming from the same distribution

(or formally, i.i.d. samples), the consistent estimators for the expected value and for the (sample) variance are:

$$\bar{X} := \frac{\sum_{i=1}^{n} x_i}{n},$$
$$s^2 := \frac{\sum_{i=1}^{n} (\bar{X} - x_i)^2}{n-1},$$

respectively.[1]

Let us focus on the empirical expected value $\bar{X}$. We recall the *Central Limit Theorem* (CLT):

Theorem 1 (Central Limit Theorem). *Assume $\{X_1, \dots X_n\}$ is a sequence of independent and identically distributed (i.i.d.) random variables, with $\mathbb{E}[X_i] = \mu$ and $Var[X_i] = \sigma^2 < \infty$, then:*

$$\sqrt{n}\left(\frac{\sum_{i=1}^{n} X_i}{n} - \mu\right) \to \mathcal{N}(0; \sigma^2),$$

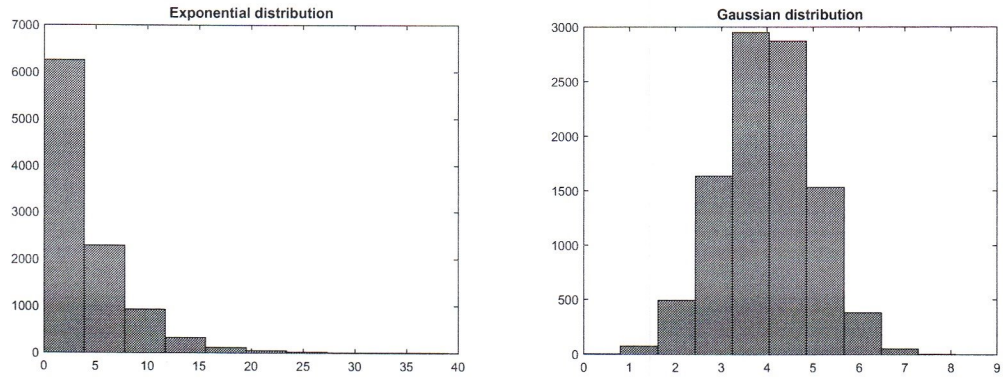
where the convergence holds in distribution.

To better understand this concept, let us sample from an *exponential* distribution with $\mu = 4$ and a *Gaussian* distribution with $\mu = 4$ and $\sigma = 1$. The histograms of the sampled distributions (we considered $n = 10000$ samples) are shown in Figure 1.2. Since the distributions have the same expected value, the empiric mean $\bar{X} = \frac{\sum_{i=1}^{n} X_i}{n}$ will concentrate around this value. If we repeatedly sample from these distributions and consider the empiric mean obtained at each repetition we will have the results shown in Figure 1.3, which confirms that the distribution of the empirical mean is approximated by a Gaussian, as stated in the CLT. Moreover, we can see how the first one is more spread since the variance of the exponential distribution is 4 times the one of the Gaussian we considered.

1.7 Confidence Intervals

Once we have some estimates of the distribution parameters we would like to understand how much we should rely on them. For instance, if we used few data to estimate the true expected value $\mathbb{E}[X]$ it is likely that the true value might be far from the estimated one, while if we used N large enough we are more certain about the true value. Another factor determining how much we can rely on the estimator is the variance of the phenomenon itself: with high variance we will have samples which are more

[1] The term *consistent* means that if we have an infinite number of samples we would converge (in probability) to the true value of the parameter.



(a) Exponential (b) Gaussian

Figure 1.2: Histograms of the samples coming from two different distributions.

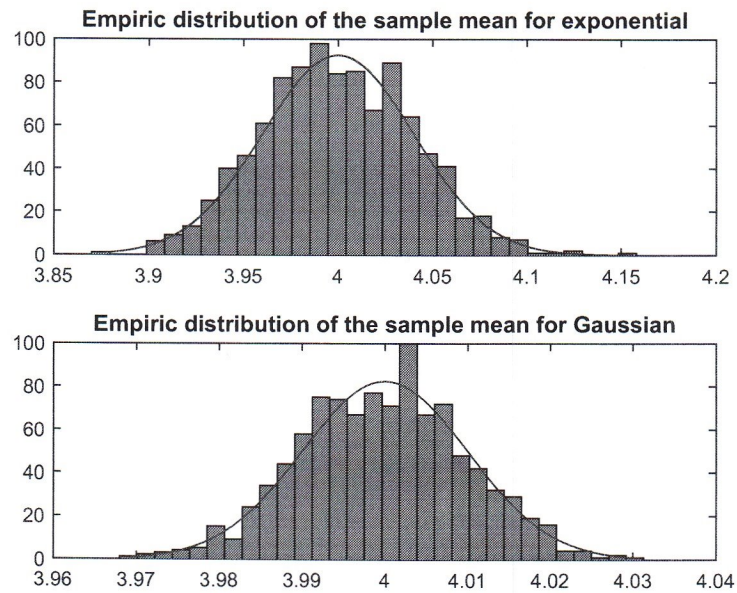


Figure 1.3: Histograms of the estimated means $\bar{X}$ coming from two different distributions.

likely to be far from each other and, thus, we are less certain about the value of the expected value (as it was shown in the previous section). Since we are in a stochastic environment, we need to set a level identifying that our estimator is "good enough". The probability that $\mathbb{E}[X]$ is exactly $\bar{X}$ is zero since the expected realization is a continuous random variable itself. Thus, we need to built some intervals, where we have high

confidence that the true mean $\mathbb{E}[X]$ is in.

Since we have the characterization of the distribution of the empirical mean, we can built intervals in which the expected value $\mathbb{E}[X]$ is with a specific confidence α. For instance, if we want to consider a confidence interval for the empirical mean, in the case we know the value of the standard deviation σ, with confidence at least $1-\alpha$, we have:

$$\bar{X} - \frac{z_{\alpha/2}\sigma}{\sqrt{n}} \leq \mu \leq \bar{X} + \frac{z_{\alpha/2}\sigma}{\sqrt{n}}, \tag{1.5}$$

where, $z_{\alpha/2}$ is the $1-\alpha/2$ quantile (i.e., the point where the CDF takes value $1-\alpha/2$ or $F^{-1}(1-\alpha/2)$) for a Normal distribution $\mathcal{N}(0,1)$, or equivalently:

$$\mathbb{P}\left(|\bar{X} - \mu| \geq \frac{z_{\alpha/2}\sigma}{\sqrt{n}}\right) \leq \alpha.$$

Remark 3. *The previous inequality is true only under Gaussian assumption. Since we are provided a finite number of samples, the confidence intervals are only an approximations of the real ones if we are considering random variables which are not Gaussian distributed.*

Remark 4. *In the derivation of the confidence interval we assumed that the standard deviation σ was known. In the case we resort to its estimates s^2, there exist different confidence intervals for the mean of the Gaussian distribution (see the Montgomery book for more details).*

If we are considering unbounded domains (e.g., $\Omega = \mathbb{R}$ or $\Omega = \mathbb{R}^+$), we also might resort to the following inequality to design confidence bounds:

Theorem 2 (Chebichev Inequality). *Suppose X is random variables with $\mathbb{E}[X] = \mu < \infty$ and $Var[X] = \sigma^2 < \infty$, then:*

$$\mathbb{P}\left(|\mu - X| \geq \frac{\sigma}{\sqrt{\alpha}}\right) \leq \alpha.$$

which, if we consider the empirical mean $\bar{X}$ and a symmetric bound, leads to:

$$\bar{X} - \frac{\sigma}{\sqrt{n}\sqrt{\alpha}} \leq \mu \leq \bar{X} + \frac{\sigma}{\sqrt{n}\sqrt{\alpha}} \tag{1.6}$$

Remark 5. *The Chebichev inequality is more general w.r.t. the one derived from CLT, since it can be applied to generic random variables. In fact, CLT holds asymptotically for every random variable and exactly for Gaussian ones.*

Finally, if we also assume that the random variables have have finite support (e.g., $\Omega = [a,b]$ or $\Omega = \{0,1\}$), we might rely on:

Theorem 3 (Chernoff-Hoeffding bound). *Assume to have a sequence* $\{X_1, X_2, \ldots, X_n\}$ *of* n *i.i.d. random variables with support in* $[a, b]$ *and* $\mathbb{E}[X_i] = \mu$, $\forall i$, *then for each* $\varepsilon > 0$ *we have:*

$$\mathbb{P}\left(|\bar{X} - \mu| \geq \varepsilon\right) \leq 2\exp\left\{-\frac{2n\varepsilon^2}{(b-a)^2}\right\} = \alpha.$$

This statistical bound leads to the following confidence intervals with confidence at least $1 - \alpha$ confidence:

$$\bar{X} - (b-a)\sqrt{\frac{-\log(\alpha/2)}{2n}} \leq \mu \leq \bar{X} + (b-a)\sqrt{\frac{-\log(\alpha/2)}{2n}}. \tag{1.7}$$

1.7.1 Hypothesis Testing

Sometimes we would like to make statement like:

The estimated parameter $\bar{X}$ *is equal to* μ.
The estimated parameter $\bar{X}'$ *is different from another estimated parameter* $\bar{X}''$.

In the case stochastic quantities are involved, the answer to such questions is the *test of hypotheses*. More specifically, we need to specify a null hypothesis H_0 and an alternative hypothesis H_1, e.g.,:

$$H_0 : \mu = \mu_0 \quad vs. \quad H_1 : \mu \neq \mu_0.$$

Given the data $\{x_1, \ldots, x_n\}$, we need that some of them support either the null H_0 or the alternative H_1 hypothesis. If we are able to compute an estimates for μ and we know its distribution, we are also able to say how likely is that the estimates has been drawn by the distribution. For instance, by considering the CLT we might say that:

$$\bar{X} \sim \mathcal{N}\left(\mu, \frac{\sigma^2}{n}\right) \rightarrow t = \frac{\bar{X} - \mu}{\frac{\sigma}{\sqrt{n}}} \sim \mathcal{N}(0, 1)$$

and, thus, that there is α probability that the test statistic t is lower than the quantile of order $\alpha/2$ of the standard Gaussian distribution or that it is larger than the quantile of order $1 - \alpha/2$. More formally:

$$\mathbb{P}(t < z_{\alpha/2} \vee t > z_{1-\alpha/2}) = \alpha.$$

If the test statistic has absolute value greater than $z_{1-\alpha/2}$ or smaller than $z_{\alpha/2}$ there is small (α) probability that the data are coming from a distribution where H_0 holds. Nonetheless, if we sample data from distribution in the case H_0 holds repeatedly we have that α of the times the test will say that the data are not coming from H_0. One

		Decision	
		Fail to reject H_0	reject H_0
True	H_0	Correct	Type I error
	H_1	Type II error	Correct

Table 1.1: Possible situations for a hypothesis test.

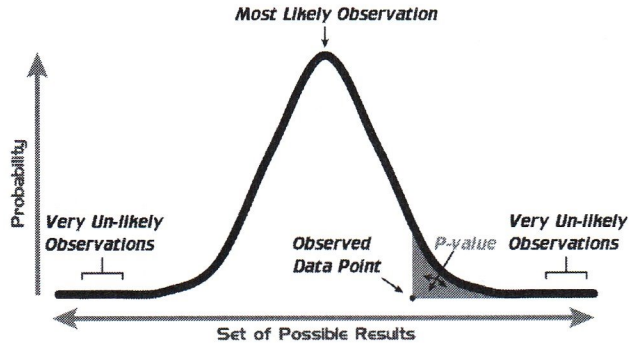


Figure 1.4: P-value in a right tailed test.

might think to use values of $\alpha \approx 0$ to reduce this probability and solve the problem. This would decrease the so called "type I error", but increase the "type II error", that even if the data are coming from a distribution for which H_1 holds we are not able to state that. Table 1.1 summarize all the possible situations.

The same reasoning used for the (two-tail) test could be used to develop test for hypothesis of the kind:

$$H_0 : \mu \leq \mu_0 \quad vs. \quad H_1 : \mu > \mu_0,$$
$$H_0 : \mu \geq \mu_0 \quad vs. \quad H_1 : \mu < \mu_0,$$

by considering only quantiles of order $1 - \alpha$ and α, respectively.

If we want to avoid to define a specific confidence α and let the data tell us how much we might be confident about their correspondence to a specific hypothesis, we could compute the p-value. The p-value is defined as the smallest confidence $\bar{\alpha}$ s.t. we are still able to reject the null hypothesis H_0. If we want to visualize the p-value in a right tailed hypothesis test (see Figure 1.4), it is the area under the distribution pdf and above the test statistics we computed

1.8 Frequentist vs. Bayesian Approach

The bounds we derived allow one to consider the cases in which we have only information about the bound of the variable which is considered. They do not allow to incorporate in a straightforward way information about the data distribution. In the case we have further information we might resort to a Bayesian approach for the parameter estimation. Indeed, by adopting the Bayesian framework, the value of the expected value of the random variable μ is a random variable itself. This is particularly interesting if we have information coming from the domain or from previously observed data.

For instance, let us say that we are considering a Bernoulli variable and we have some information coming from the past that tells us that a previously analysed phenomenon, **similar** to the one in analysis, had 3 successes over 10 trials. It would be wrong to consider these samples as drawn from the considered variable (i.e., using them to compute the empirical mean).

Consider the Bayes formula:

$$\begin{aligned}\mathbb{P}(\mu|x_1,\dots,x_t) &= \frac{\mathbb{P}(x_1,\dots,x_{t-1},x_t|\mu)\mathbb{P}(\mu)}{\mathbb{P}(x_1,\dots,x_t)}\\ &\propto \mathbb{P}(x_t|\mu)\mathbb{P}(x_1,\dots,x_{t-1}|\mu)\mathbb{P}(\mu)\\ &= \mathbb{P}(x_t|\mu)\mathbb{P}(x_{t-1}|\mu)\mathbb{P}(x_1,\dots,x_{t-2}|\mu)\mathbb{P}(\mu)\\ &= \mathbb{P}(\mu)\prod_{h=1}^{t}\mathbb{P}(x_h|\mu),\end{aligned}$$

where we assumed conditional independence of x_t from all the other data.[2] This way we are able to incorporate information starting from a prior distribution $\mathbb{P}(\mu)$ incrementally.

In the case we consider Bernoulli realizations, if we consider a Beta distribution as prior for the expected value μ ($\mu \sim \text{Beta}(3,7)$ in the example), we have that the posterior is still a Beta (i.e., Bernoulli and Beta are conjugate prior-posterior), thus allowing us to have an update rule for the next datum x_t.

Remark 6. *In the case we incorporate meaningful information the Bayesian learning process could be faster than the frequentist one. Though, if the information provided by the prior are misleading, the process could slow down and in some case prevent the estimation process to converge to the real value (e.g., if the prior assigns zero probability to the real value of the parameter).*

[2] We do not report the denominators since they do not change depending on μ, therefore they constitute only a normalization term for the posterior probability.

Exercise 1.2

Model a fair 6-faced dice. Then, find the best strategy for playing dice (you bet on the sum of a couple of dice and win only if the predicted number is equal to the result).

What happens if the dice has 20 faces.

Exercise 1.3

Consider a simplified version of the game of the *roulette* in which you can only bet on specific numbers and you win 35 times what you bet if you guessed right and 0 if you guessed wrong. Recall that in the roulette game you can bet on numbers from 0 to 36.

What is the expected value of playing 1 euro at the roulette? What about the variability of this phenomenon?

Write a script emulating a player, with and initial budget of 100 euros that always bets 1 euro per turn on the number 17, and record the amount of money she/he has over time. Plot the amount of money in her/his wallet with a red line. Are you sure that the experiment ends eventually?

Exercise 1.4

Given a set of $n = 1000$ realization from a Bernoulli distribution with $\mu = 0.5$ compute the following bounds for the expected value with confidence $\alpha = 0.01$:

- confidence interval from CLT;
- Hoeffding bounds;
- Chebichev bounds.

There is any theoretical difference among them?