Term: L3-AII & L3-ElnI

Semester: 5

AY: 2023-2024

Abdelbacet Mhamdi

Dr.-Ing. in Electrical Engineering

Senior Lecturer at ISET Bizerte

abdelbacet.mhamdi@bizerte.r-iset.tn

Machine Learning

LAB MANUAL

Institut Supérieur des Études Technologiques de Bizerte

Available @ https://github.com/a-mhamdi/mlpy/

Honor code

THE UNIVERSITY OF NORTH CAROLINA AT CHAPEL HILL
Department of Physics and Astronomy
http://physics.unc.edu/undergraduate-program/labs/general-info/

“During this course, you will be working with one or more partners with whom you may discuss any points concerning laboratory work. However, you must write your lab report, in your own words.

Lab reports that contain identical language are not acceptable, so do not copy your lab partner’s writing.

If there is a problem with your data, include an explanation in your report. Recognition of a mistake and a well-reasoned explanation is more important than having high-quality data, and will be rewarded accordingly by your instructor. A lab report containing data that is inconsistent with the original data sheet will be considered a violation of the Honor Code.

Falsification of data or plagiarism of a report will result in prosecution of the offender(s) under the University Honor Code.

On your first lab report you must write out the entire honor pledge:

The work presented in this report is my own, and the data was obtained by my lab partner and me during the lab period.

On future reports, you may simply write *“Laboratory Honor Pledge”* and sign your name.”

Contents

In order to activate the virtual environment and launch **Jupyter Notebook**, we recommend you to proceed as follow

① Press simultaneously the keys ⊞ & R on the keyboard. This will open the dialog box **`Run`**;

② Then enter **`cmd`** in the command line and confirm with ↵ key on the keyboard;

③ Type the instruction **`mlpy.bat`** in the console prompt line;

```
Command Prompt

C:\Users\admin> mlpy.bat
```

④ Finally press the ↵ key.

LEAVE THE SYSTEM CONSOLE ACTIVE.

1 | *Python* Onramp

Student's name	. . . .	. . . .	. . . .
Score */20*	. . . .	. . . .	. . . .

Detailed Credits

Anticipation *(4 points)*	. . . .	. . . .	. . . .
Management *(2 points)*	. . . .	. . . .	. . . .
Testing *(7 points)*	. . . .	. . . .	. . . .
Data Logging *(3 points)*	. . . .	. . . .	. . . .
Interpretation *(4 points)*	. . . .	. . . .	. . . .

Motivations

- ★ *Python* is a popular programming language in the field of machine learning because it is relatively easy to learn and has a wide range of libraries and frameworks that support machine learning tasks.
- ★ *Python* has a large and active community of developers, which means that there are many resources available online, such as tutorials, documentation, and online forums, to help students learn and troubleshoot their code.
- ★ Many machine learning tools and frameworks, such as *TensorFlow* and *scikit-learn*, are written in *Python*, which makes it easy to integrate these tools into *Python* programs.
- ★ *Python* is a versatile language that can be used for a wide range of applications beyond machine learning, including web development, data analysis, and scientific computing. Learning *Python* can therefore open up many career opportunities for students.

The notebook is available at `https://github.com/a-mhamdi/mlpy/` → *Codes* → *Python* → *py-onramp.ipynb*

Numerical variables & types

[1]:
```python
a = 1 # An integer
print('The variable a = {} is of type {}'.format(a, type(a)))
```

```
The variable a = 1 is of type <class 'int'>
```

[2]:
```
b = -1.25 # A floating number
print('The variable b = {} is of type {}'.format(b, type(b)))
```

```
The variable b = -1.25 is of type <class 'float'>
```

[3]:
```
c = 1+0.5j # A complex number
print('The variable c = {} is of type {}'.format(c, type(c)))
```

```
The variable c = (1+0.5j) is of type <class 'complex'>
```

Strings

[4]:
```
msg = "My 1st lab!"
print(msg, type(msg), sep = '\n***\n') # \n: Carriage Return & Line Feed
print(msg + 3* '\nPython is awesome')
```

```
My 1st lab!
***
<class 'str'>
My 1st lab!
Python is awesome
Python is awesome
Python is awesome
```

[5]:
```
longMsg = """This is a long message,
spanned over multiple lines"""
print(longMsg)
```

```
This is a long message,
spanned over multiple lines
```

Indexing and slicing

[6]:
```
# Positive indexing
print(msg, msg[1:5], sep = ' -----> ')
# Negative indexing
print(msg, msg[-5:-1], sep = ' -----> ')
```

```
My 1st lab! -----> y 1s
My 1st lab! ----->  lab
```

String transformations

[7]:
```
msg = 'A message'
print(len(msg))
print(msg.lower())
print(msg.upper())
print(msg.split(' '))
print(msg.replace('mes', 'MES'))
print('a' in msg) #  Check if the variable `msg` contains the letter 'a'
```

```
9
a message
```

```
A MESSAGE
['A', 'message']
A MESsage
True
```

[8]:
```
price, number, perso = 300, 7, 'A customer'
print('{} asks for {} pieces. They cost {} TND!'.format(perso, number,␣
 ↪price))
print('{1} demande {2} pièces. They cost {0} TND!'.format(price, perso,␣
 ↪number))
```

```
A customer asks for 7 pieces. They cost 300 TND!
A customer demande 7 pièces. They cost 300 TND!
```

Binary, octal & hexadecimal

[9]:
```
x = 0b0101 # 0b : binary
print(x, type(x), sep = '\t----\t') # \t : tabular
y = 0xAF # 0x : hexadecimal
print(y, type(y), sep = '\t' + '---'*5 + '\t')
z = 0o010 # 0o : octal
print(z, type(z), sep = ', ')
```

```
5       ----    <class 'int'>
175     --------------- <class 'int'>
8, <class 'int'>
```

Boolean

[10]:
```
a = True
b = False
print(a)
print(b)
```

```
True
False
```

[11]:
```
print("50 > 20 ? : {} \n50 < 20 ? : {} \n50 = 20 ? : {}\n50 /= 20 ? : {}"
      .format(50 > 20, 50 < 20, 50 == 20, 50 != 20)
     )
```

```
50 > 20 ? : True
50 < 20 ? : False
50 = 20 ? : False
50 /= 20 ? : True
```

[12]:
```
print(bool(123), bool(0), bool('Lab'), bool())
```

```
True False True False
```

[13]:
```
var1 = 100
print(isinstance(var1, int))
```

```
var2 = -100.35
print(isinstance(var2, int))
print(isinstance(var2, float))
```

```
True
False
True
```

Lists, tuples & dictionaries

In Python, a list is an ordered collection of items that can be of any data type (including other lists). Lists are defined using square brackets, with items separated by commas. For example:

[14]:
```
shopping_list = ['milk', 'eggs', 'bread', 'apples']
```

A tuple is also an ordered collection of items, but it is immutable, meaning that the items it contains cannot be modified once the tuple is created. Tuples are defined using parentheses, with items separated by commas. For example:

[15]:
```
point = (3, 5)
```

A dictionary is a collection of key-value pairs, where the keys are unique and used to look up the corresponding values. Dictionaries are defined using curly braces, with the key-value pairs separated by commas. The keys and values are separated by a colon. For example:

[16]:
```
phonebook = {'Alice': '555-1234', 'Bob': '555-5678', 'Eve': '555-9101'}
```

You can access the items in a list or tuple using an index, and you can access the values in a dictionary using the corresponding keys. For example:

[17]:
```
# Accessing the second item in a list
print(shopping_list[1])  # prints 'eggs'

# Accessing the first item in a tuple
print(point[0])  # prints 3

# Accessing the phone number for 'Bob' in the phonebook dictionary
print(phonebook['Bob'])  # prints '555-5678'
```

```
eggs
3
555-5678
```

List

[18]:
```
lst = ['a', 'b', 'c', 1, True] # An aggregate of various types
print(lst)
```

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

[19]:
```
print(len(lst)) # Length of `lst` variable
print(lst[1:3]) # Accessing elements of `lst`
```

```
lst[0] = ['1', 0] # Combined list
print(lst)
print(lst[3:])
print(lst[:3])
```

```
5
['b', 'c']
[['1', 0], 'b', 'c', 1, True]
[1, True]
[['1', 0], 'b', 'c']
```

[20]:
```
lst.append('etc') # Insert 'etc' at the end
print(lst)
```

```
[['1', 0], 'b', 'c', 1, True, 'etc']
```

[21]:
```
lst.insert(1, 'xyz') # Inserting 'xyz'
print(lst)
```

```
[['1', 0], 'xyz', 'b', 'c', 1, True, 'etc']
```

[22]:
```
lst.pop(1)
print(lst)
```

```
[['1', 0], 'b', 'c', 1, True, 'etc']
```

[23]:
```
lst.pop()
print(lst)
```

```
[['1', 0], 'b', 'c', 1, True]
```

[24]:
```
del lst[0]
print(lst)
```

```
['b', 'c', 1, True]
```

[25]:
```
lst.append('b')
print(lst)
lst.remove('b')
print(lst)
```

```
['b', 'c', 1, True, 'b']
['c', 1, True, 'b']
```

[26]:
```
# Loop
for k in lst:
    print(k)
```

```
c
1
True
b
```

[27]:
```python
lst.clear()
print(lst)
```

```
[]
```

Method	*Description*
copy()	Returns a copy of the list
list()	Transforms into a list
extend ()	Extends a list by adding elements at its end
count()	Returns the occurrences of the specified value
index()	Returns the index of the first occurrence of a specified value
reverse()	Reverse a list
sort()	Sort a list

Tuples

[28]:
```python
tpl = (1, 2, 3)
print(tpl)
```

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

[29]:
```python
tpl = (1, '1', 2, 'text')
print(tpl)
```

```
(1, '1', 2, 'text')
```

[30]:
```python
print(len(tpl))
```

```
4
```

[31]:
```python
print(tpl[1:])
```

```
('1', 2, 'text')
```

[32]:
```python
try:
    tpl.append('xyz') # Throws an error
except Exception as err:
    print(err)
```

```
'tuple' object has no attribute 'append'
```

[33]:
```python
try:
    tpl.insert(1, 'xyz') # Throws an error
except Exception as err:
    print(err)
```

```
'tuple' object has no attribute 'insert'
```

[34]:
```
my_lst = list(tpl)
my_lst.append('xyz')
print(my_lst, type(my_lst), sep = ', ')
```

```
[1, '1', 2, 'text', 'xyz'], <class 'list'>
```

[35]:
```
nv_tpl = tuple(my_lst) # Convert 'my_lst' into a tuple 'nv_tpl'
print(nv_tpl, type(nv_tpl), sep = ', ')
```

```
(1, '1', 2, 'text', 'xyz'), <class 'tuple'>
```

[36]:
```
# Loop
for k in nv_tpl:
    print(k)
```

```
1
1
2
text
xyz
```

[37]:
```
rs_tpl = tpl + nv_tpl
print(rs_tpl)
```

```
(1, '1', 2, 'text', 1, '1', 2, 'text', 'xyz')
```

Dictionaries

[38]:
```
# dct = {"key": "value"}
dct = {
    "Term" : "GM",
    "Speciality" : "ElnI",
    "Sem" : "4"
}
print(dct, type(dct), sep = ', ')
```

```
{'Term': 'GM', 'Speciality': 'ElnI', 'Sem': '4'}, <class 'dict'>
```

[39]:
```
print(dct["Sem"])
sem = dct.get("Sem")
print(sem)
```

```
4
4
```

[40]:
```
dct["Term"] = "GE"
print(dct)
```

```
{'Term': 'GE', 'Speciality': 'ElnI', 'Sem': '4'}
```

[41]:
```
# Loop
for d in dct:
    print(d, dct[d], sep = '\t|\t')
```

```
Term	|	GE
Speciality	|	ElnI
Sem	|	4
```

[42]:
```
for k in dct.keys():
    print(k)
```

```
Term
Speciality
Sem
```

[43]:
```
for v in dct.values():
    print(v)
```

```
GE
ElnI
4
```

NumPy

NumPy is a *Python* library that is used for scientific computing and data analysis. It provides support for large, multi-dimensional arrays and matrices of numerical data, and a large library of mathematical functions to operate on these arrays.

One of the main features of *NumPy* is its *N*-dimensional array object, which is used to store and manipulate large arrays of homogeneous data (*i.e.*, data of the same type, such as integers or floating point values). The array object provides efficient operations for performing element-wise calculations, indexing, slicing, and reshaping.

NumPy also includes a number of functions for performing statistical and mathematical operations on arrays, such as mean, standard deviation, and dot product. It also includes functions for linear algebra, random number generation, and Fourier transforms.

NumPy is a fundamental package for scientific computing with *Python*, and is widely used in a variety of applications including machine learning, data analysis, and scientific simulations. It is an essential library for working with large, multi-dimensional arrays and matrices of numerical data in *Python*.

Official documentation can be found at https://numpy.org/

[44]:
```
import numpy as np
```

NumPy vs List

[45]:
```
a_np = np.arange(6) # NumPy
print("a_np = ", a_np)
print(type(a_np))
a_lst = list(range(0,6)) # List
print("a_lst = ", a_lst)
print(type(a_lst))
# Comparison
print("2 * a_np = ", a_np * 2)
print("2 * a_lst = ", a_lst * 2)
```

```
a_np =  [0 1 2 3 4 5]
<class 'numpy.ndarray'>
a_lst =  [0, 1, 2, 3, 4, 5]
<class 'list'>
2 * a_np =  [ 0  2  4  6  8 10]
2 * a_lst =  [0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5]
```

[46]:
```
v_np = np.array([1, 2, 3, 4, 5, 6]) # NB : parentheses then brackets, i.e,␣
 ↪([])
print(v_np)
```

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

[47]:
```
v_np = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
print(v_np)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

[48]:
```
print(type(v_np))
```

```
<class 'numpy.ndarray'>
```

[49]:
```
print(v_np[0])
```

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

[50]:
```
v_np.ndim # Dimensions of v_np
```

[50]:
```
2
```

[51]:
```
v_np.shape # Number of lignes and columns, may be more
```

[51]:
```
(3, 4)
```

[52]:
```
v_np.size # How many elements are in `v_np`
```

[52]:
```
12
```

If we need to create a matrix $(3, 3)$, we can do as follows:

[53]:
```
u = np.arange(9).reshape(3,3)
print(u)
```

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

Let us see some known operations to do on matrices

[54]:
```
M = np.array([[1, 2], [1, 2]])
print(M)
```

```
[[1 2]
 [1 2]]
```

[55]:
```python
N = np.array([[0, 3], [4, 5]])
print(N)
```

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

Addition

[56]:
```python
print(M + N)
print(np.add(M, N))
```

```
[[1 5]
 [5 7]]
[[1 5]
 [5 7]]
```

Subtraction

[57]:
```python
print(M-N)
print(np.subtract(M, N))
```

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

Element-wise Product

Element-wise multiplication, also known as **Hadamard product**, is an operation that multiplies each element of one matrix with the corresponding element of another matrix. It is denoted by the symbol $\odot$ or `.*` in some programming languages.

For example, consider the following matrices:

$$A = \begin{bmatrix} a_1, & a_2, & a_3 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} b_1, & b_2, & b_3 \end{bmatrix}$$

The element-wise product of these matrices is:

$$A \odot B = \begin{bmatrix} a_1 b_1, & a_2 b_2, & a_3 b_3 \end{bmatrix}$$

$$\begin{bmatrix} 1 & 2 \\ 1 & 2 \end{bmatrix} . \times \begin{bmatrix} 0 & 3 \\ 4 & 5 \end{bmatrix} = \begin{bmatrix} 0 & 6 \\ 4 & 10 \end{bmatrix}$$

We need element-wise multiplication in many applications. For example, in image processing, element-wise multiplication is used to modify the intensity values of an image by multiplying each pixel value with a scalar value. In machine learning, element-wise multiplication is used in the implementation of various neural network layers, such as convolutional layers and fully connected layers. Element-wise multiplication is also used in many other mathematical and scientific applications.

[58]:
```
print(M * N)
print(np.multiply(M, N))
```

```
[[ 0  6]
 [ 4 10]]
[[ 0  6]
 [ 4 10]]
```

Dot Product

$$\begin{bmatrix} 1 & 2 \\ 1 & 2 \end{bmatrix} \times \begin{bmatrix} 0 & 3 \\ 4 & 5 \end{bmatrix} = \begin{bmatrix} 8 & 13 \\ 8 & 13 \end{bmatrix}$$

[59]:
```
print(M.dot(N))
print(np.dot(M, N))
```

```
[[ 8 13]
 [ 8 13]]
[[ 8 13]
 [ 8 13]]
```

Element-wise Division

$$\begin{bmatrix} 0 & 3 \\ 4 & 5 \end{bmatrix} ./ \begin{bmatrix} 1 & 2 \\ 1 & 2 \end{bmatrix} = \begin{bmatrix} 0:1 & 3:2 \\ 4:1 & 5:2 \end{bmatrix}$$

[60]:
```
print(N / M)
print(np.divide(N, M))
```

```
[[0.  1.5]
 [4.  2.5]]
[[0.  1.5]
 [4.  2.5]]
```

Determinant of a matrix

[61]:
```
print("Determinant of M:")
print(np.linalg.det(M))
print("Determinant of N:")
print(np.linalg.det(N))
```

```
Determinant of M:
0.0
Determinant of N:
-12.0
```

Matplotlib

Matplotlib is a 2D data visualization library in *Python* that allows users to create a wide range of static, animated, and interactive visualizations in *Python*. It is one of the most widely used data visualization libraries in the *Python* data science ecosystem and is particularly useful for creating line plots, scatter plots, bar plots, error bars, histograms, bar charts, pie charts, box plots, and many other types of visualizations.

Matplotlib is designed to be easy to use and highly customizable, with a wide range of options for customizing the look and feel of the plots it produces. It can be used to create visualizations for a wide range of applications, including scientific, technical, and business applications. It is also widely used in data journalism and data communication, and is a powerful tool for communicating data-driven insights to a wide audience.

Matplotlib is built on top of *NumPy* and is often used in conjunction with other libraries in the PyData ecosystem, such as *Pandas* and *Seaborn*, to create complex visualizations of data. It is also compatible with a number of different backends, such as the *Jupyter notebook*, *Qt*, and *Tkinter*, which allows it to be used in a wide range of environments and contexts.

The full documentation and an exhaustive list of samples can be found at `https://matplotlib.org/`

[62]:
```
import numpy as np
import matplotlib.pyplot as plt

plt.style.use("ggplot")
plt.rcParams['figure.figsize'] = [15, 10]
```

We begin by creating a sinusoidal waveform denoted by *x*, period is 1 sec. The offset is 1.

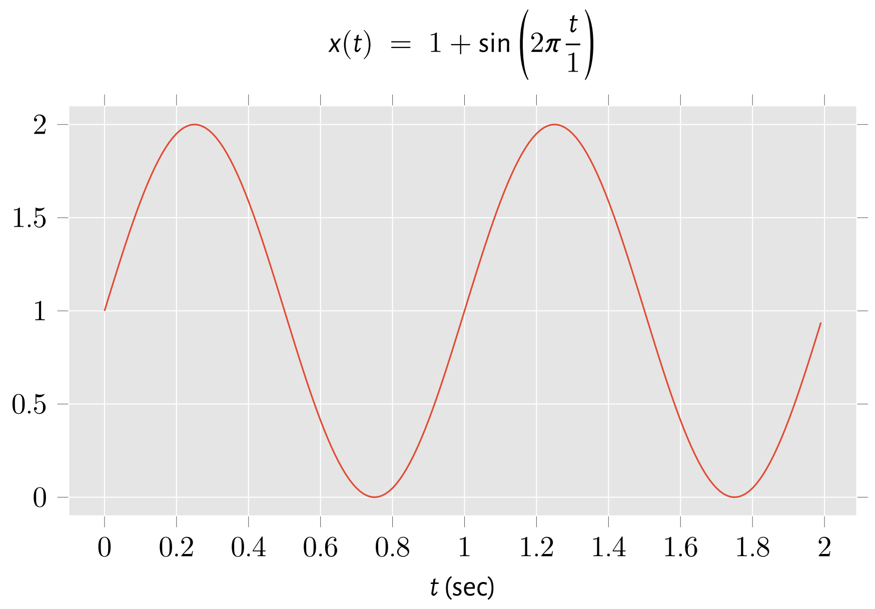
[63]:
```
# Continuous function
t = np.arange(0.0, 2.0, 0.01)
x = 1 + np.sin(2 * np.pi * t) # Frequency = 1Hz
```

The set of instructions that allow to plot (x) are:

[64]:
```
plt.plot(t, x)

# Give the graph a title
plt.title(r"$x(t) = 1+\sin\left(2\pi\frac{t}{1}\right)$")
plt.xlabel("$t$ (sec)") # Label the axis
```

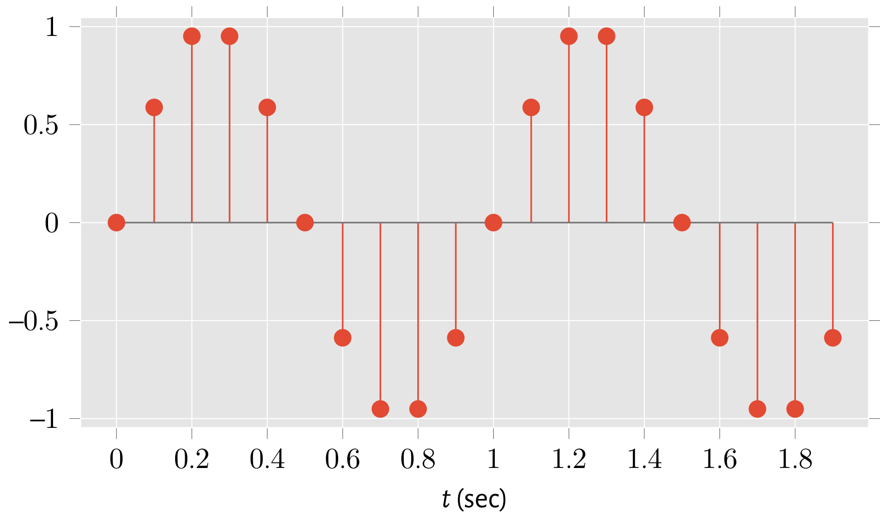
[64]:
```
Text(0.5, 0, '$t$ (sec)')
```



[65]:
```
# Discret Function
t = np.arange(0.0, 2.0, 0.1)
y = np.sin(2*np.pi*t) # Same thing! Sinusoidal signal
```

[66]:
```
plt.stem(t, y)
plt.xlabel("$t$ (sec)")
```

[66]:
```
Text(0.5, 0, '$t$ (sec)')
```



2 | Linear Regression

Student's name	. . . .	. . . .	. . . .
Score **/20**	. . . .	. . . .	. . . .

Detailed Credits

Anticipation ***(4 points)***	. . . .	. . . .	. . . .
Management ***(2 points)***	. . . .	. . . .	. . . .
Testing ***(7 points)***	. . . .	. . . .	. . . .
Data Logging ***(3 points)***	. . . .	. . . .	. . . .
Interpretation ***(4 points)***	. . . .	. . . .	. . . .

Motivations

- ★ Linear regression is a fundamental statistical technique that is widely used in many fields, including economics, finance, biology, and computer science. It is a simple and effective way to model the relationship between a dependent variable and one or more independent variables.
- ★ Linear regression is relatively easy to understand and implement, making it a good starting point for students who are new to statistical modeling. It is also a good foundation for learning more advanced statistical techniques, such as multiple regression or logistic regression.
- ★ Linear regression can be a useful tool for making predictions and understanding the underlying trends in data. It can help students to better understand and analyze data, and to make informed decisions based on their findings.

The notebook is available at `https://github.com/a-mhamdi/mlpy/` ⟶ *Codes* ⟶ *Python* ⟶ *linear-regression.ipynb*

[1]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import os
```

Load the datasets.

[2]:
```python
dataset = pd.read_csv("./datasets/Weight_Height.csv")
```

Check the dataset.

[3]:
```
dataset.head()
```

[3]:
```
  Gender     Height      Weight
0   Male  73.847017  241.893563
1   Male  68.781904  162.310473
2   Male  74.110105  212.740856
3   Male  71.730978  220.042470
4   Male  69.881796  206.349801
```

Check the dimensions of the loaded dataset.

[4]:
```
dataset.shape
```

[4]:
```
(10000, 3)
```

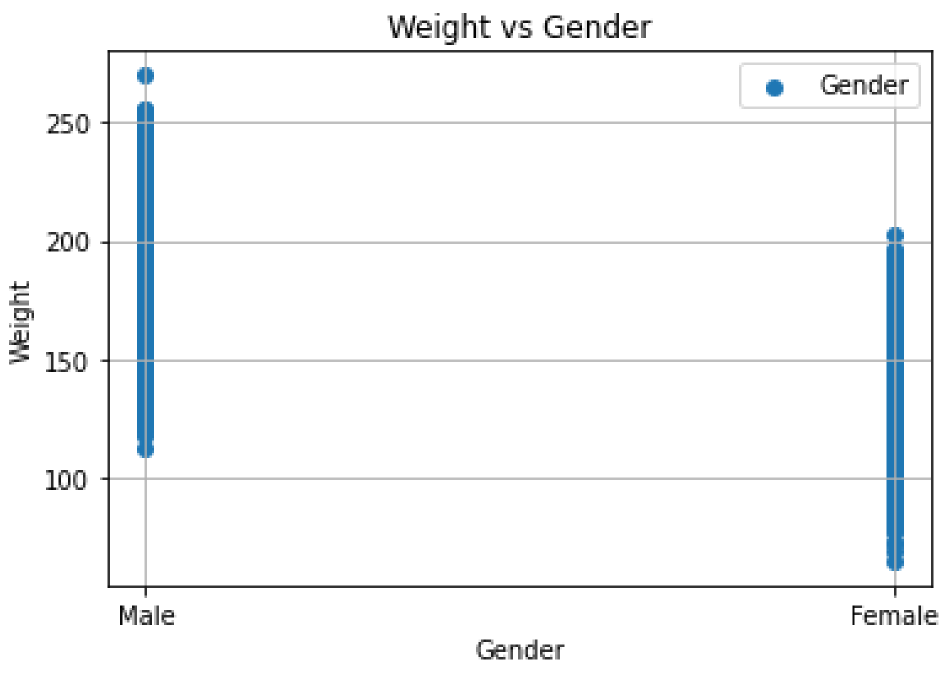
Check if there are null values in the dataset.

[5]:
```
dataset.isnull().sum()
```

[5]:
```
Gender    0
Height    0
Weight    0
dtype: int64
```

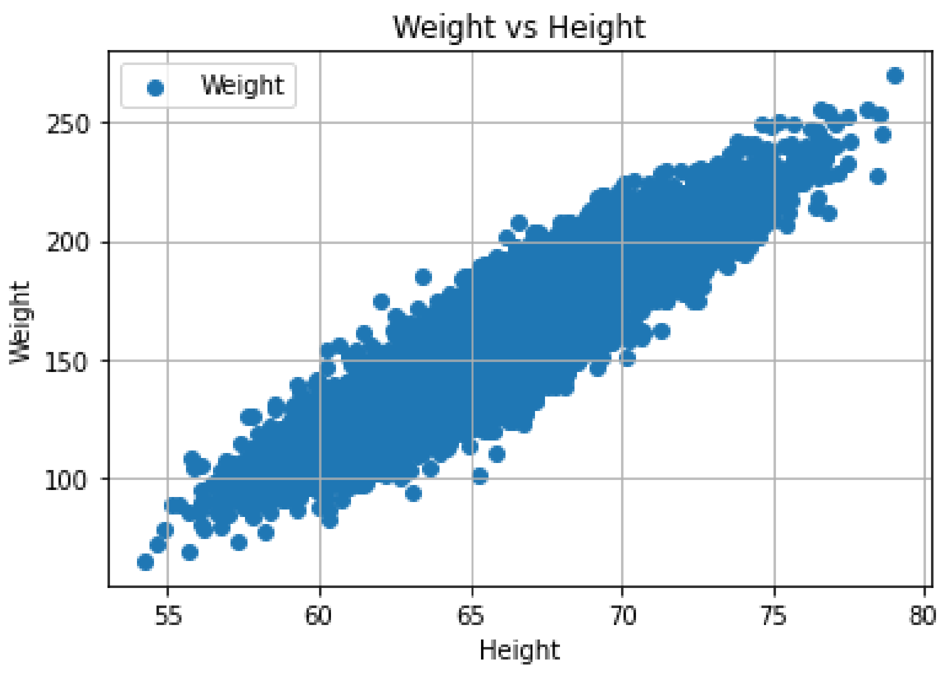
Plot *Gender* vs *Weight*.

[6]:
```
x1 = dataset.iloc[:, 0].values
y1 = dataset.iloc[:, 2].values
plt.scatter(x1, y1, label='Gender')
plt.xlabel('Gender')
plt.ylabel('Weight')
plt.title('Weight vs Gender')
plt.grid()
plt.legend()
```



Plot *Height* vs *Weight*.

[7]:
```
x2 = dataset.iloc[:, 1].values
y2 = dataset.iloc[:, 2].values
plt.scatter(x2,y2,label='Weight')
plt.xlabel('Height')
plt.ylabel('Weight')
plt.title('Weight vs Height')
plt.grid()
plt.legend()
```



[8]:
```
X = dataset.iloc[:, 1].values
```

Target values y

[9]:
```
y = dataset.iloc[:, 2].values
```

[10]:
```
X_train, X_test, y_train, y_test = train_test_split(X.reshape(-1,1), y,␣
↪test_size=0.2, random_state=123)
```

Create linear regression model.

[11]:
```
w_h_regressor = LinearRegression()
```

[12]:
```
w_h_regressor.fit(X_train, y_train)
```

[12]:
```
LinearRegression()
```

A full description of the available methods can be found at the official website of scikit-learn.

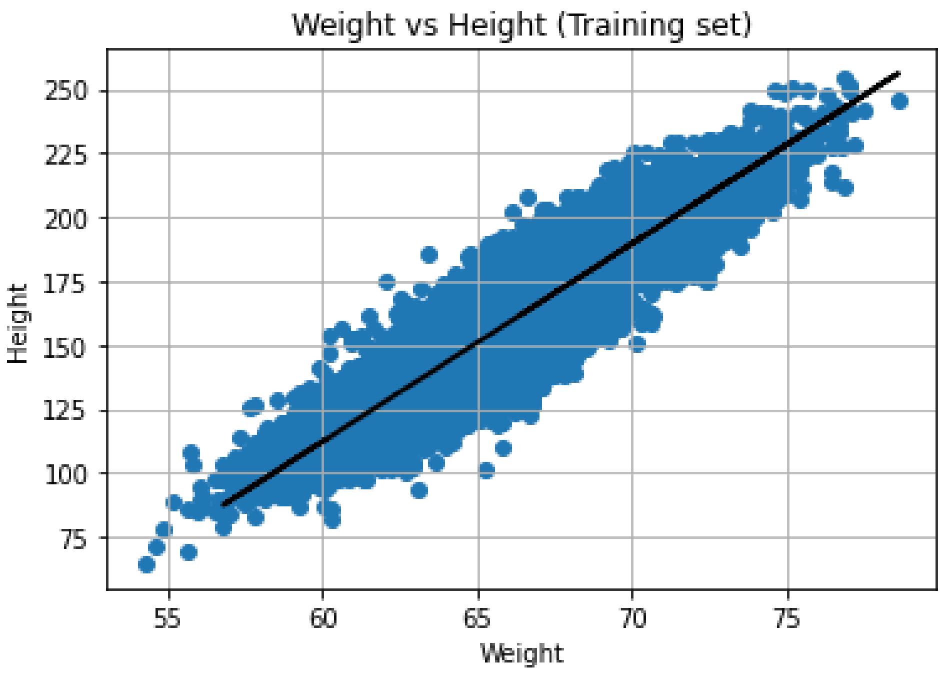
Syntax	Description
`fit(X, y[, sample_weight])`	Fit linear model.
`get_params([deep])`	Get parameters for this estimator.
`predict(X)`	Predict using the linear model.
`score(X, y[, sample_weight])`	Return the coefficient of determination of the prediction.
`set_params(**params)`	Set the parameters of this estimator.

Predict the training set.

[13]:
```
y_pred = w_h_regressor.predict(X_train)
```

Display the training set results

[14]:
```
plt.scatter(X_train, y_train)
plt.plot(X_train, y_pred, color='black', linewidth=2)
plt.title('Weight vs Height (Training set)')
plt.xlabel('Weight')
plt.ylabel('Height')
plt.grid()
```

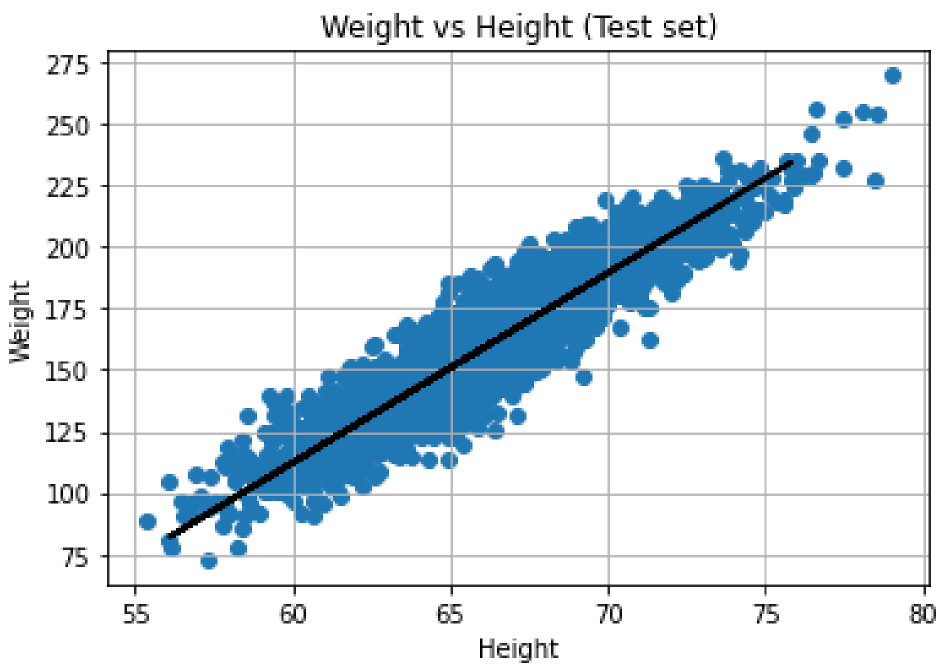


Predict the test set.

[15]:
```
y_pred = w_h_regressor.predict(X_test)
```

Display the test set results.

[16]:
```
plt.scatter(X_test, y_test)
plt.plot(X_test, y_pred, color='black', linewidth=2)
plt.title('Weight vs Height (Test set)')
plt.xlabel('Height')
plt.ylabel('Weight')
plt.grid()
```



Overall evaluation of the model

```
[17]: print('Coefficients: ', w_h_regressor.coef_)
```

```
Coefficients:  [7.72896259]
```

The mean squared error

```
[18]: print('Mean squared error is {}.'.format(np.mean((y_pred - y_test)** 2)))
```

```
Mean squared error is 143.22556010111649.
```

The more variance score approaches to 1, the more perfect is the prediction.

```
[19]: print('Variance score is {}.'.format(w_h_regressor.score(X_test, y_test)))
```

```
Variance score is 0.8649031737206692.
```

3 | k-NN for Classification

Student's name	. . . .	. . . .	. . . .
Score ***/20***	. . . .	. . . .	. . . .

Detailed Credits

Anticipation ***(4 points)***	. . . .	. . . .	. . . .
Management ***(2 points)***	. . . .	. . . .	. . . .
Testing ***(7 points)***	. . . .	. . . .	. . . .
Data Logging ***(3 points)***	. . . .	. . . .	. . . .
Interpretation ***(4 points)***	. . . .	. . . .	. . . .

Motivations

- ⋆ k-nearest neighbors (k-NN) is a simple and effective classification algorithm that is easy to understand and implement. It is based on the idea of using the class labels of the "nearest neighbors" to predict the class label of a new data point.
- ⋆ k-NN is a "lazy learner" that does not make any assumptions about the underlying data distribution, which makes it a good choice for working with complex or non-linear data. It is also robust to noise and can handle missing data. As a result, k-NN is often used as a baseline method for comparison with more advanced classification algorithms.

The notebook is available at `https://github.com/a-mhamdi/mlpy/` → *Codes* → *Python* → *clf-knn.ipynb*

Load the necessary python modules

[1]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Load the datasets

[2]:
```python
df = pd.read_csv('./datasets/Diabetes.csv')
```

Print the first 5 rows of the dataframe.

[3]:
```
df.head()
```

[3]:
```
   Pregnancies  Glucose  Diastolic  Triceps  Insulin   BMI    DPF  Age  \
0            6      148         72       35        0  33.6  0.627   50
1            1       85         66       29        0  26.6  0.351   31
2            8      183         64        0        0  23.3  0.672   32
3            1       89         66       23       94  28.1  0.167   21
4            0      137         40       35      168  43.1  2.288   33

   Diabetes
0         1
1         0
2         1
3         0
4         1
```

Let's observe the shape of the dataframe.

[4]:
```
df.shape
```

[4]:
```
(768, 9)
```

Let's extract the features and target as numpy arrays.

[5]:
```
X = df.drop('Diabetes',axis=1).values
y = df['Diabetes'].values
```

Split the data into two sets: train and test. We begin by importing the `train_test_split` from `sklearn` module.

[6]:
```
from sklearn.model_selection import train_test_split
```

[7]:
```
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.
 ↪4,random_state=42, stratify=y)
```

It is time now to create a classifier using k-Nearest Neighbors algorithm. At first, the class `KNeighborsClassifier` has to be loaded.

[8]:
```
from sklearn.neighbors import KNeighborsClassifier
```

Let's setup a knn classifier with only $k = 7$ neighbors.

[9]:
```
knn = KNeighborsClassifier(n_neighbors=7)
```

Fit the model.

[10]:
```
knn.fit(X_train,y_train)
```

[10]:
```
KNeighborsClassifier(n_neighbors=7)
```

It is always a good manner to gather some score metrics.

[11]:
```
knn.score(X_test,y_test)
```

```
[11]: 0.7305194805194806
```

Import `confusion_matrix`

```
[12]: from sklearn.metrics import confusion_matrix
```

Let's make some predictions using the classifier we built earlier.

```
[13]: y_pred = knn.predict(X_test)
```

```
[14]: confusion_matrix(y_test,y_pred)
```

```
[14]: array([[165,  36],
             [ 47,  60]])
```

A fancy way to display the confusion matrix, is to use the `crosstab` method.

```
[15]: pd.crosstab(y_test, y_pred, rownames=['True'], colnames=['Predicted'],␣
      ↪margins=True)
```

```
[15]: Predicted    0   1  All
      True
      0          165  36  201
      1           47  60  107
      All        212  96  308
```

By importing `classification_report`, we can get some insights on how the model behaves.

```
[16]: from sklearn.metrics import classification_report
```

As a reminder, **F1-Score**, **Accuracy**, **Recall** and **Precision** are calculated as follow:

$$f1 - \text{score} = \frac{2}{\frac{1}{\text{Recall}} + \frac{1}{\text{Precision}}}$$

$f1 - \text{score}$ denotes the *Harmonic Mean of Recall & Precision*

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}}$$

It denotes the ratio of how much we got right over all cases. Recall, on the other hand, designates the ratio of how much positives do we got right over all actual positive cases.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

Precision, at last, is how much positives we got right over all positive predictions. It is given by:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

```
[17]: print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           0       0.78      0.82      0.80       201
           1       0.62      0.56      0.59       107

    accuracy                           0.73       308
   macro avg       0.70      0.69      0.70       308
weighted avg       0.73      0.73      0.73       308
```

4 K-Means for Clustering

Student's name	....	....	....
Score */20*	....	....	....

Detailed Credits

Anticipation *(4 points)*	....	....	....
Management *(2 points)*	....	....	....
Testing *(7 points)*	....	....	....
Data Logging *(3 points)*	....	....	....
Interpretation *(4 points)*	....	....	....

Motivations

★ K-means clustering is a widely used method for partitioning a dataset into a set of clusters, where each cluster consists of data points that are similar to each other. This can be useful for a variety of applications, including data compression, anomaly detection, and customer segmentation.

★ K-means is a simple and efficient algorithm that is easy to implement and can be applied to large datasets. It is also relatively fast, making it a good choice for real-time applications.

★ K-means is a popular method for exploratory data analysis because it can reveal underlying patterns and structures in the data that may not be immediately apparent. It can also help to identify outliers and anomalies in the data, which can be useful for identifying errors or identifying new opportunities for analysis.

The notebook is available at `https://github.com/a-mhamdi/mlpy/` ⟶ *Codes* ⟶ *Python* ⟶ *clu-k-means.ipynb*

[1]:
```python
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

Let's begin by loading the *Mall_Customers* datasets.

[2]:
```python
df = pd.read_csv('./datasets/Mall_Customers.csv')
```

```
[3]: df.head()
```

```
[3]:    CustomerID   Genre  Age  Annual Income (k$)  Spending Score (1-100)
     0           1    Male   19                  15                      39
     1           2    Male   21                  15                      81
     2           3  Female   20                  16                       6
     3           4  Female   23                  16                      77
     4           5  Female   31                  17                      40
```

Renaming some columns is very handy for further data manipulation.

```
[4]: df.rename(columns={'Annual Income (k$)': 'Income', 'Spending Score (1-100)':␣
      ↪'Spending Score'}, inplace=True)
```

```
[5]: df.head()
```

```
[5]:    CustomerID   Genre  Age  Income  Spending Score
     0           1    Male   19      15              39
     1           2    Male   21      15              81
     2           3  Female   20      16               6
     3           4  Female   23      16              77
     4           5  Female   31      17              40
```

df.describe() allows to get useful insights from data.

```
[6]: df.describe()
```

```
[6]:        CustomerID         Age      Income  Spending Score
     count  200.000000  200.000000  200.000000      200.000000
     mean   100.500000   38.850000   60.560000       50.200000
     std     57.879185   13.969007   26.264721       25.823522
     min      1.000000   18.000000   15.000000        1.000000
     25%     50.750000   28.750000   41.500000       34.750000
     50%    100.500000   36.000000   61.500000       50.000000
     75%    150.250000   49.000000   78.000000       73.000000
     max    200.000000   70.000000  137.000000       99.000000
```

```
[7]: from sklearn import cluster
```

We will perform **K-Means** Clustering with 5 clusters using only 2 Variables.

```
[8]: clu_k = cluster.KMeans(n_clusters=5 ,init="k-means++")
```

```
[9]: clu_k = clu_k.fit(df[['Spending Score','Income']])
```

Coordinates of the centers.

```
[10]: clu_k.cluster_centers_
```

```
[10]: array([[49.51851852, 55.2962963 ],
             [82.12820513, 86.53846154],
             [17.11428571, 88.2       ],
             [79.36363636, 25.72727273],
             [20.91304348, 26.30434783]])
```

[11]:
```
df['Clusters'] = clu_k.labels_
```

[12]:
```
df.head()
```

[12]:
```
   CustomerID   Genre  Age  Income  Spending Score  Clusters
0           1    Male   19      15              39         4
1           2    Male   21      15              81         3
2           3  Female   20      16               6         4
3           4  Female   23      16              77         3
4           5  Female   31      17              40         4
```

[13]:
```
df['Clusters'].value_counts()
```

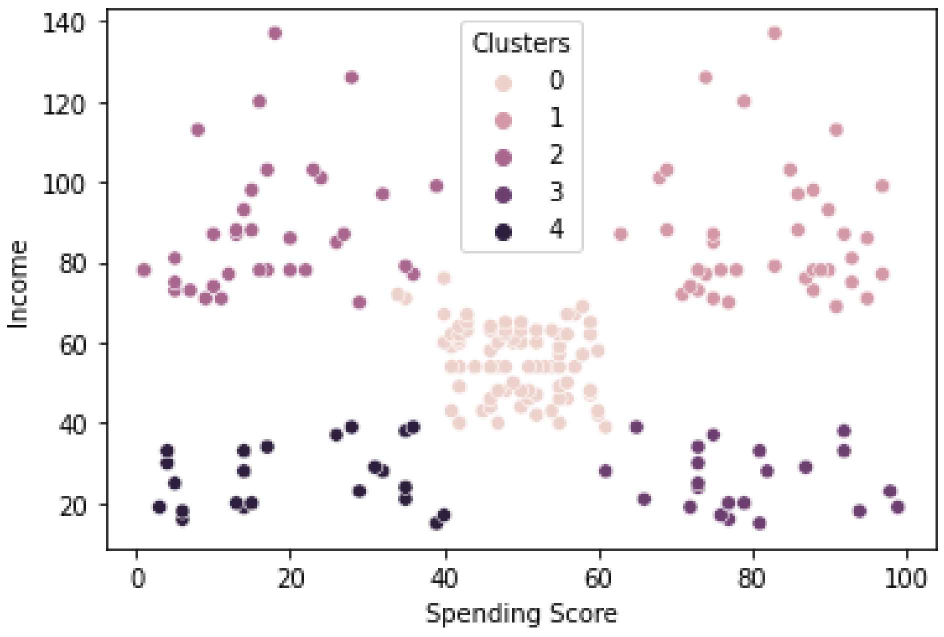
[13]:
```
0    81
1    39
2    35
4    23
3    22
Name: Clusters, dtype: int64
```

Let's save the new data frame to a new file.

[14]:
```
df.to_csv('./datasets/Mall_Clusters.csv', index = False)
```

Plot the 5 clusters on a chart.

[15]:
```
sns.scatterplot(x="Spending Score", y="Income", hue='Clusters',  data=df)
```



5 Binary Classifier using ANN

Student's name	. . . .	. . . .	. . . .
Score */20*	. . . .	. . . .	. . . .

Detailed Credits

Anticipation *(4 points)*	. . . .	. . . .	. . . .
Management *(2 points)*	. . . .	. . . .	. . . .
Testing *(7 points)*	. . . .	. . . .	. . . .
Data Logging *(3 points)*	. . . .	. . . .	. . . .
Interpretation *(4 points)*	. . . .	. . . .	. . . .

Motivations

★ Artificial neural networks (ANNs) are a powerful tool for binary classification tasks, which involve predicting a binary outcome (e.g., "yes" or "no") based on input data. ANNs are able to learn complex relationships between the input data and the output labels, which makes them well-suited for tasks with a large number of features or a complex underlying structure.

★ ANNs are highly flexible and can be trained on a wide range of data types, including continuous and categorical variables. They can also handle missing values and handle large amounts of data efficiently. This makes them a good choice for tasks where the data is noisy or high-dimensional.

The notebook is available at `https://github.com/a-mhamdi/mlpy/` → *Codes* → *Python* → *clf-ann.ipynb*

[1]:
```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Import `sklearn`.

[2]:
```python
from sklearn.preprocessing import LabelEncoder, OneHotEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
```

Import keras.

```
[3]: from keras.models import Sequential
     from keras.layers import Dense
```

```
Using TensorFlow backend.
```

Load the data using pandas.

```
[4]: df = pd.read_csv('./datasets/Churn_Modelling.csv')
```

```
[5]: df.head(3)
```

```
[5]:    RowNumber  CustomerId   Surname  CreditScore Geography  Gender  Age  \
     0          1    15634602  Hargrave          619    France  Female   42
     1          2    15647311      Hill          608     Spain  Female   41
     2          3    15619304      Onio          502    France  Female   42

        Tenure    Balance  NumOfProducts  HasCrCard  IsActiveMember  \
     0       2       0.00              1          1               1
     1       1   83807.86              1          0               1
     2       8  159660.80              3          1               0

        EstimatedSalary  Exited
     0        101348.88       1
     1        112542.58       0
     2        113931.57       1
```

```
[6]: X = df.iloc[:, 3:13].values
     y = df.iloc[:, 13].values

     label_encoder_X_country = LabelEncoder()
     label_encoder_X_gender = LabelEncoder()

     X[:, 1] = label_encoder_X_country.fit_transform(X[:, 1])
     X[:, 2] = label_encoder_X_gender.fit_transform(X[:, 2])

     one_hot_encoder = ColumnTransformer([("Geography", OneHotEncoder(), [1])],␣
      ↪remainder = 'passthrough')

     X = one_hot_encoder.fit_transform(X)
     X = np.array(X, dtype=float)
     X = X[:, 1:]
```

Scale the features.

```
[7]: sc = StandardScaler()
     X = sc.fit_transform(X)
```

Split the datasets into training & testing sets.

```
[8]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
      ↪random_state=0)
```

Define the artificial neural network architecture.

[9]:
```
clf_ann = Sequential()
```

Input layer & first hidden layer

[10]:
```
num_features = X_train.shape[1]
clf_ann.add(Dense(6, input_shape = (num_features, ), activation = 'relu'))
```

Second hidden layer

[11]:
```
clf_ann.add(Dense(6, activation = 'relu'))
```

Output layer

[12]:
```
num_classes = 1
clf_ann.add(Dense(num_classes, activation = 'sigmoid'))
```

[13]:
```
clf_ann.compile('Adam', loss = 'binary_crossentropy', metrics=['accuracy'])
```

An overall description of the neural network architecture.

[14]:
```
clf_ann.summary()
```

```
Model: "sequential_1"

_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 6)                 72
_________________________________________________________________
dense_2 (Dense)              (None, 6)                 42
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 7
=================================================================
Total params: 121
Trainable params: 121
Non-trainable params: 0
_________________________________________________________________
```

Fit the classifier.

[15]:
```
clf_ann.fit(x=X_train, y=y_train, batch_size=200, epochs=20, verbose=1)
```

```
Epoch 1/20
8000/8000 [==============================] - 0s 21us/step - loss: 0.6352 -
accuracy: 0.6734
Epoch 2/20
8000/8000 [==============================] - 0s 7us/step - loss: 0.5697 -
accuracy: 0.7575
Epoch 3/20
8000/8000 [==============================] - 0s 7us/step - loss: 0.5312 -
accuracy: 0.7881
Epoch 4/20
8000/8000 [==============================] - 0s 6us/step - loss: 0.5058 -
```

```
accuracy: 0.7961
Epoch 5/20
8000/8000 [==============================] - 0s 8us/step - loss: 0.4867 -
accuracy: 0.8001
Epoch 6/20
8000/8000 [==============================] - 0s 7us/step - loss: 0.4722 -
accuracy: 0.8020
Epoch 7/20
8000/8000 [==============================] - 0s 6us/step - loss: 0.4607 -
accuracy: 0.8037
Epoch 8/20
8000/8000 [==============================] - 0s 8us/step - loss: 0.4520 -
accuracy: 0.8065
Epoch 9/20
8000/8000 [==============================] - 0s 9us/step - loss: 0.4454 -
accuracy: 0.8100
Epoch 10/20
8000/8000 [==============================] - 0s 9us/step - loss: 0.4405 -
accuracy: 0.8135
Epoch 11/20
8000/8000 [==============================] - 0s 7us/step - loss: 0.4366 -
accuracy: 0.8149
Epoch 12/20
8000/8000 [==============================] - 0s 7us/step - loss: 0.4336 -
accuracy: 0.8160
Epoch 13/20
8000/8000 [==============================] - 0s 7us/step - loss: 0.4310 -
accuracy: 0.8173
Epoch 14/20
8000/8000 [==============================] - 0s 5us/step - loss: 0.4287 -
accuracy: 0.8171
Epoch 15/20
8000/8000 [==============================] - 0s 5us/step - loss: 0.4270 -
accuracy: 0.8174
Epoch 16/20
8000/8000 [==============================] - 0s 5us/step - loss: 0.4255 -
accuracy: 0.8179
Epoch 17/20
8000/8000 [==============================] - 0s 5us/step - loss: 0.4240 -
accuracy: 0.8199
Epoch 18/20
8000/8000 [==============================] - 0s 5us/step - loss: 0.4229 -
accuracy: 0.8199
Epoch 19/20
8000/8000 [==============================] - 0s 5us/step - loss: 0.4215 -
accuracy: 0.8205
Epoch 20/20
8000/8000 [==============================] - 0s 6us/step - loss: 0.4202 -
accuracy: 0.8219
```

```
[15]: <keras.callbacks.callbacks.History at 0x7f46f71cefa0>
```

Evaluate the model.

```
[16]: scores = clf_ann.evaluate(x=X_test, y=y_test, batch_size=100, verbose=1)
```

```
2000/2000 [==============================] - 0s 10us/step
```

Let's predict an output.

```
[17]: y_pred = clf_ann.predict(X_test)
      y_pred = (y_pred > 0.5)
```

Define the confusion matrix.

```
[18]: cm = confusion_matrix(y_test, y_pred)
      tp, fp, fn, tn = cm.ravel()
```

```
[19]: print('Accuracy is about {}%.' .format(100*(tp+tn)/sum((sum(cm)))))
```

```
Accuracy is about 83.2%.
```

```
[20]: print('\
      The loss value is: {}.\n\n\
      The accuracy percentage is: {}%. '.format(scores[0], 100*scores[1]))
```

```
The loss value is: 0.4155806913971901.

The accuracy percentage is: 83.20000171661377%.
```

The overall scope of this manual is to introduce **Machine Learning**, through some numeric simulations, to the students at the department of **Electrical Engineering**.

The topics discussed in this manuscript are as follow:

① Getting started with *Python*

② Linear Regression

③ Classification

④ Clustering

⑤ ANN

Python; *Jupyter*; *NumPy*; *Matplotlib*; *scikit-learn*; machine learning; linear regression; classification; clustering; deep learning.