Université d'Alger 1 – Benyoucef Benkhadda

Faculté des Sciences

Département d'Informatique

Master 1 – ISII 2021/2022

Artificial Intelligence

Machine Learning Project

Team N° 10

- Taleb Mehdi
- Sennoun Merouane
- Zouai Serine Maria
- Aouanouk Imane
- Zemmouri Arslane

Table of Contents

Division of work

Taleb Mehdi:

Preprocessing (Classification)

SVM (Classification)

Decision Trees (Classification)

Sennoun Merouane:

Preprocessing (Classification)

Logistic Regression (Classification)

SVM (Classification)

Zouai Serine Maria:

Preprocessing (Regression)

Normal Equation (Regression)

Aouanouk lmane:

Preprocessing (Regression)

Linear Regression (Regression)

Zemmouri Arslane:

Introduction + Project Descriptions

Neural Network (Classification)

Neural Network (Regression)

1. Introduction

The project that we had to undertake is based on a subcategory of machine learning and artificial intelligence, which is Supervised Learning. Essentially, supervised learning is when the computer is taught by example. It learns from past data and applies the learning to present data to predict future events. In this case, both input and desired output data provide help to the prediction of future events.

Supervised learning can also be divided into two subcategories, classification and regression. Which are the two sorts of problems that we had to operate.

We had to apply the specific analysis methods to each category, to solve the two problems and train different models capable of predicting the best possible results.

On the other hand, we also had to comply with specific working methods to solve a machine learning problem. The use of a notebook is therefore essential for this, because it is used to present the analysis process step by step by arranging the code, images, text, output etc. in a step-by-step manner. Jupyter Notebook provides a full set of features that makes it one the best components of Python Machine Learning. Another tool that we used, this time for the arrangement and organization of work in general, is Git and GitHub, such as each member of the group made changes on the part of the notebook that corresponded with a specific part of the project (preprocessing, graphic visualization etc.)

In the following parts, we are going to discuss the achieved tasks in each of the two problems, explaining the dataset content, the different steps of preprocessing realized to improve its quality, and testing and visualizing the obtained results produced from the built models.

2. Project Descriptions

2.1. Regression Problem

One of the two problems that we had to process is a regression problem. A regression analysis must be considered when we deal with a problem that requires a prediction of a continuous value. Our regression problem is about valuating real estate properties given certain characteristics about them and their surroundings. Basically, the main task is to predict a house price, which is continuous value, considering a group of numerical/categorical features describing this house.

2.2. Classification Problem

The second problem is a classification problem. This is the kind of problems that require the result of the prediction to be a categorical value. In our case it is a binary classification, for the reason that the model can only predict a 0 or 1 value for a given set of feature values concerning an instance. Our classification problem is about predicting if Immunotherapy, when used for wart treatment, produces a positive or negative result on a patient, given different information about this one. A prediction of "1" being a result of a successful treatment and "0" of an unsuccessful one.

3. Real Estate Valuation

3.1. Dataset Description

The real estate valuation is a regression problem. The market historical data set of real estate valuation are collected from Sindian Dist., New Taipei City, Taiwan.

The data set was randomly split into the training data set (2/3 samples) and the testing data set (1/3 samples).

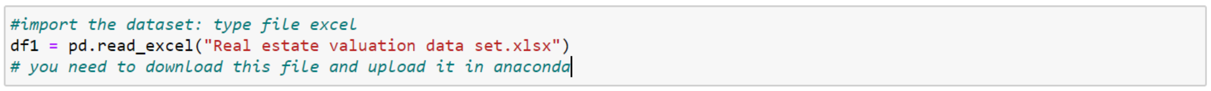
Attribute Information:

The inputs are as follows:
X1=the transaction date (for example, 2013.250=2013 March, 2013.500=2013 June, etc.)
X2=the house age (unit: year)
X3=the distance to the nearest MRT station (unit: meter)
X4=the number of convenience stores in the living circle on foot (integer)
X5=the geographic coordinate, latitude. (Unit: degree)
X6=the geographic coordinate, longitude. (Unit: degree)

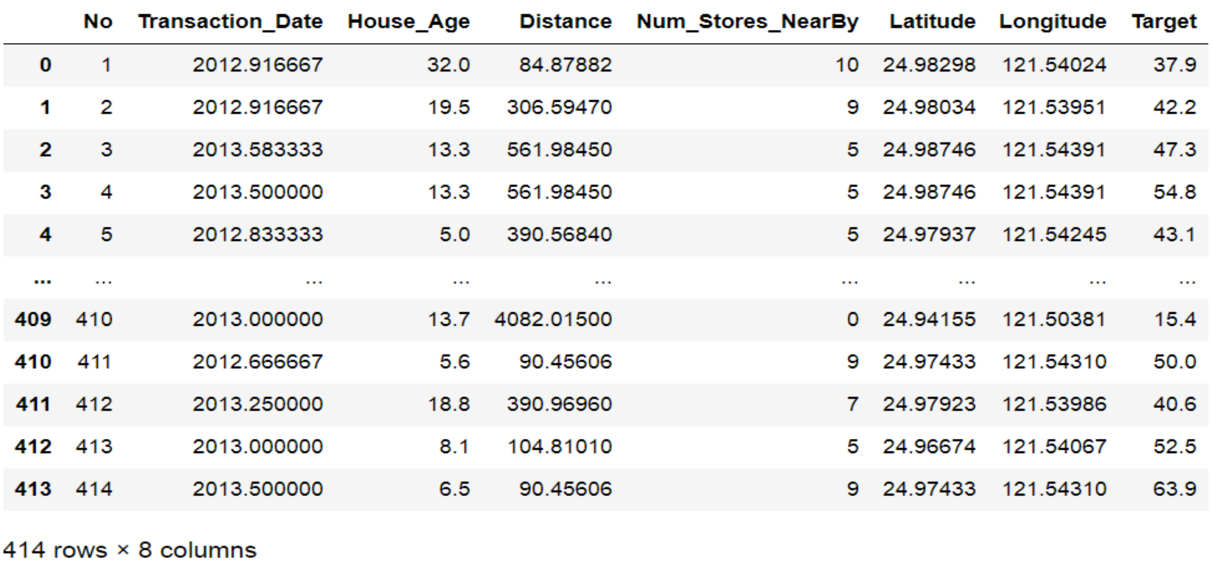
The output is as follows:
Y= house price of unit area (10000 New Taiwan Dollar/Ping, where Ping is a local unit, 1 Ping = 3.3 meter squared)

3.2. Preprocessing

We start our preprocessing by displaying our dataset df1 and using the Pandas method as follows:



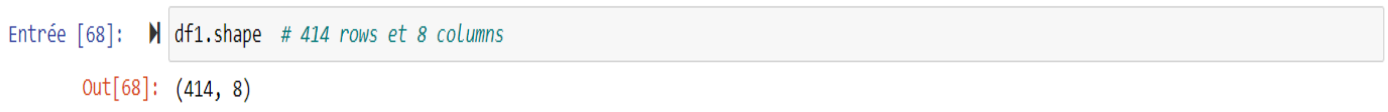
```
#import the dataset: type file excel
df1 = pd.read_excel("Real estate valuation data set.xlsx")
# you need to download this file and upload it in anaconda
```



	No	Transaction_Date	House_Age	Distance	Num_Stores_NearBy	Latitude	Longitude	Target
0	1	2012.916667	32.0	84.87882	10	24.98298	121.54024	37.9
1	2	2012.916667	19.5	306.59470	9	24.98034	121.53951	42.2
2	3	2013.583333	13.3	561.98450	5	24.98746	121.54391	47.3
3	4	2013.500000	13.3	561.98450	5	24.98746	121.54391	54.8
4	5	2012.833333	5.0	390.56840	5	24.97937	121.54245	43.1
...	...	...	...	...	...	...	...	...
409	410	2013.000000	13.7	4082.01500	0	24.94155	121.50381	15.4
410	411	2012.666667	5.6	90.45606	9	24.97433	121.54310	50.0
411	412	2013.250000	18.8	390.96960	7	24.97923	121.53986	40.6
412	413	2013.000000	8.1	104.81010	5	24.96674	121.54067	52.5
413	414	2013.500000	6.5	90.45606	9	24.97433	121.54310	63.9

414 rows × 8 columns

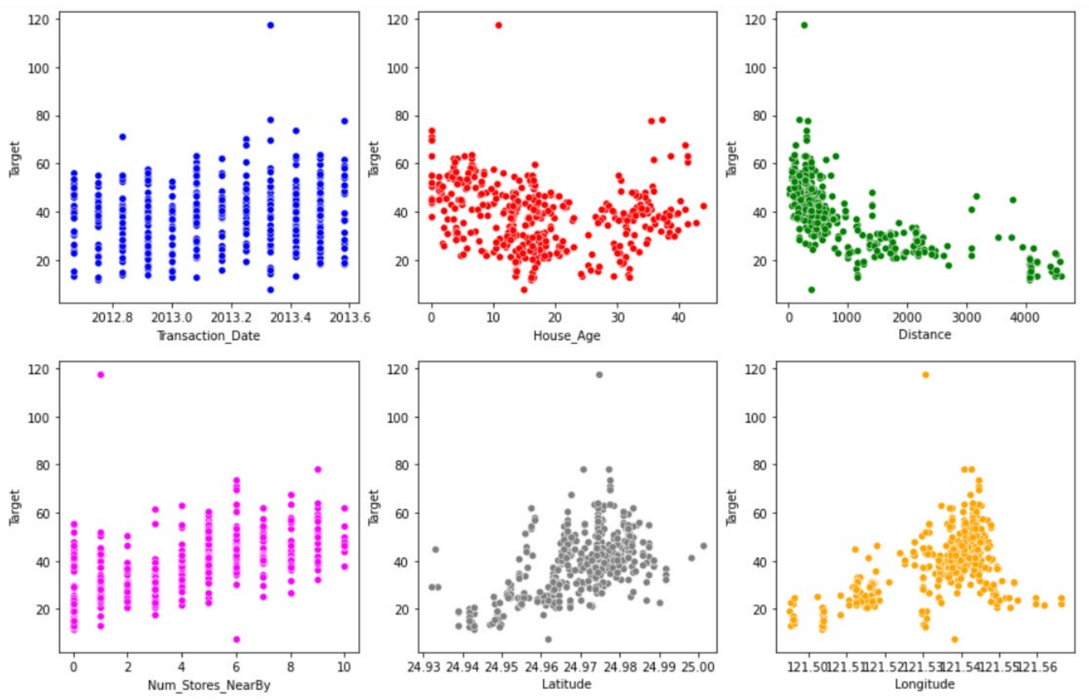
The shape of our Dataset df1 is:



```
Entrée [68]: df1.shape  # 414 rows et 8 columns

    Out[68]: (414, 8)
```

3.2.1 Graphic Representation

We use Scatter subplot for each feature as follows:



3.2.2 Missing values

In the description of the regression dataset, it is mentioned that there is no missing values and there is a command that allows us to determine the number of missing values : df1.isna().sum()

```
df1.isna().sum()    # same result with this request : df1.isnull().sum()
# As you can notice all the values obtained are equal to zero therefore there are no missing values
```

```
No                  0
Transaction_Date    0
House_Age           0
Distance            0
Num_Stores_NearBy   0
Latitude            0
Longitude           0
Target              0
dtype: int64
```

3.2.3 Remove an unnecessary attribute such as the feature "No”

	Transaction_Date	House_Age	Distance	Num_Stores_NearBy	Latitude	Longitude	Target
0	2012.916667	32.0	84.87882	10	24.98298	121.54024	37.9
1	2012.916667	19.5	306.59470	9	24.98034	121.53951	42.2
2	2013.583333	13.3	561.98450	5	24.98746	121.54391	47.3
3	2013.500000	13.3	561.98450	5	24.98746	121.54391	54.8
4	2012.833333	5.0	390.56840	5	24.97937	121.54245	43.1

3.2.4 Encoding

We show the type of our data by the command df1.dtypes and we found that the dtype of all our features is not categorical.

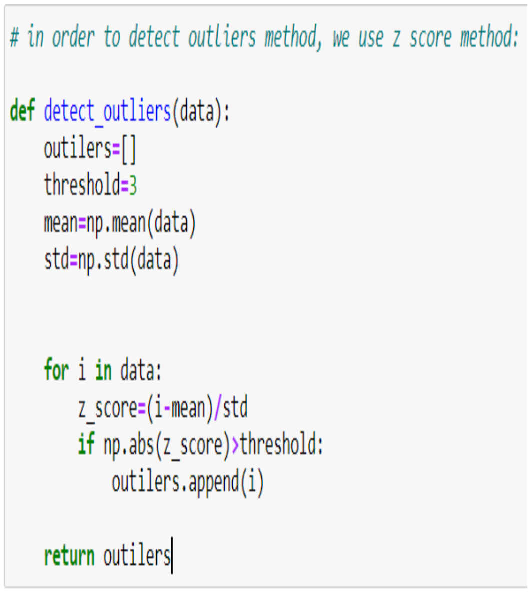
3.2.5 Duplicate Values

```
print(df1.duplicated().values.sum())#this command allows us to know if we have duplicate values and as we can notice the val
```

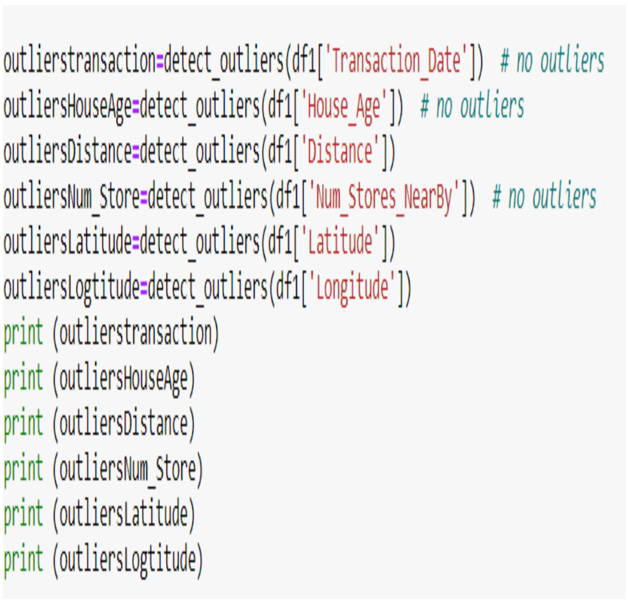
```
0
```

3.2.6 Remove outliers

We use the z_score method to detect outliers:



```
# in order to detect outliers method, we use z score method:

def detect_outliers(data):
    outilers=[]
    threshold=3
    mean=np.mean(data)
    std=np.std(data)


    for i in data:
        z_score=(i-mean)/std
        if np.abs(z_score)>threshold:
            outilers.append(i)

    return outilers
```



```
outlierstransaction=detect_outliers(df1['Transaction_Date'])  # no outliers
outliersHouseAge=detect_outliers(df1['House_Age'])  # no outliers
outliersDistance=detect_outliers(df1['Distance'])
outliersNum_Store=detect_outliers(df1['Num_Stores_NearBy'])  # no outliers
outliersLatitude=detect_outliers(df1['Latitude'])
outliersLogtitude=detect_outliers(df1['Longitude'])
print (outlierstransaction)
print (outliersHouseAge)
print (outliersDistance)
print (outliersNum_Store)
print (outliersLatitude)
print (outliersLogtitude)
```

```
[]
[]
[5512.038, 6396.283, 6306.153, 5512.038, 6488.021]
[]
[25.01459]
[121.48458, 121.47883, 121.47516, 121.48458, 121.47353]
```

Df1 after removing outliers:

	Transaction_Date	House_Age	Distance	Num_Stores_NearBy	Latitude	Longitude	Target
0	2012.916667	32.0	84.87882	10	24.98298	121.54024	37.9
1	2012.916667	19.5	306.59470	9	24.98034	121.53951	42.2
2	2013.583333	13.3	561.98450	5	24.98746	121.54391	47.3
3	2013.500000	13.3	561.98450	5	24.98746	121.54391	54.8
4	2012.833333	5.0	390.56840	5	24.97937	121.54245	43.1
...	...	...	...	...	...	...	...
403	2013.000000	13.7	4082.01500	0	24.94155	121.50381	15.4
404	2012.666667	5.6	90.45606	9	24.97433	121.54310	50.0
405	2013.250000	18.8	390.96960	7	24.97923	121.53986	40.6
406	2013.000000	8.1	104.81010	5	24.96674	121.54067	52.5
407	2013.500000	6.5	90.45606	9	24.97433	121.54310	63.9

408 rows × 7 columns

3.2.7 Feature scaling

In order to make our values in one interval [0 1] , we use the feature scaling method MinMaxScaler.

Df1 after scaling:

	Transaction_Date	House_Age	Distance	Num_Stores_NearBy	Latitude	Longitude	Target
0	0.272727	0.730594	0.013420	1.0	0.736972	0.634410	0.275705
1	0.272727	0.445205	0.061805	0.9	0.698755	0.624157	0.314832
2	1.000000	0.303653	0.117538	0.5	0.801824	0.685955	0.361237
3	0.909091	0.303653	0.117538	0.5	0.801824	0.685955	0.429481
4	0.181818	0.114155	0.080130	0.5	0.684713	0.665449	0.323021
...	...	...	...	...	...	...	...
403	0.363636	0.312785	0.885707	0.0	0.137232	0.122753	0.070974
404	0.000000	0.127854	0.014637	0.9	0.611754	0.674579	0.385805
405	0.636364	0.429224	0.080218	0.7	0.682687	0.629073	0.300273
406	0.363636	0.184932	0.017770	0.5	0.501882	0.640449	0.408553
407	0.909091	0.148402	0.014637	0.9	0.611754	0.674579	0.512284

408 rows × 7 columns

3.3. Linear Regression Gradient Descent:

Linear Regression is a linear model and one of supervised machine learning algorithm which is used to model a relationship between one dependent variable and two or more independent variables.

In our Dataset there are multiple input variables so we refer to the method as multiple linear regression.

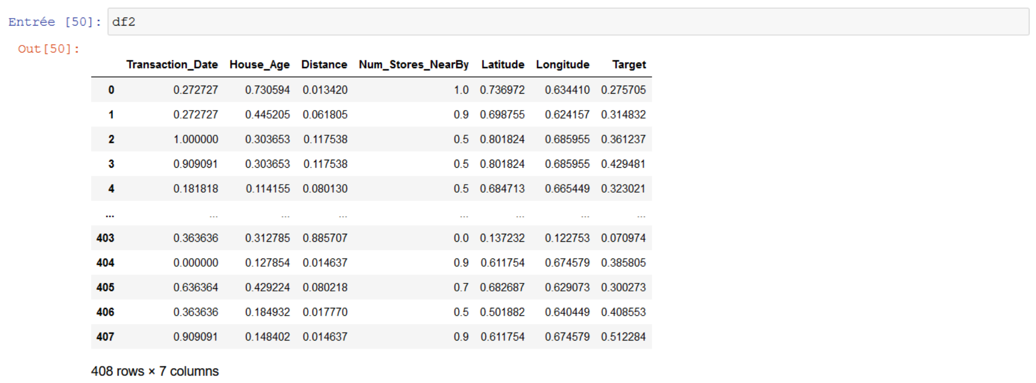
In this part, we will apply the multivariate linear regression using Gradient descent on our Dataset.

Gradient descent is an optimization algorithm used to minimize an objective function **J(θ)** parameterized by a model's parameters **θ ∈ R** by updating the parameters.

To implement the gradient descent algorithm to our regression problem we followed the following steps:

NB: we have used **the feature scaling** in order to make our values in one interval **[0,1]**

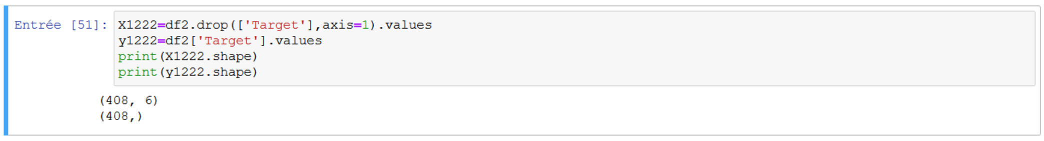
3.3.1 Preparing the X and Y



```
Entrée [50]: df2
Out[50]:
```

	Transaction_Date	House_Age	Distance	Num_Stores_NearBy	Latitude	Longitude	Target
0	0.272727	0.730594	0.013420	1.0	0.736972	0.634410	0.275705
1	0.272727	0.445205	0.061805	0.9	0.698755	0.624157	0.314832
2	1.000000	0.303653	0.117538	0.5	0.801824	0.685955	0.361237
3	0.909091	0.303653	0.117538	0.5	0.801824	0.685955	0.429481
4	0.181818	0.114155	0.080130	0.5	0.684713	0.665449	0.323021
...	...	...	...	...	...	...	...
403	0.363636	0.312785	0.885707	0.0	0.137232	0.122753	0.070974
404	0.000000	0.127854	0.014637	0.9	0.611754	0.674579	0.385805
405	0.636364	0.429224	0.080218	0.7	0.682687	0.629073	0.300273
406	0.363636	0.184932	0.017770	0.5	0.501882	0.640449	0.408553
407	0.909091	0.148402	0.014637	0.9	0.611754	0.674579	0.512284

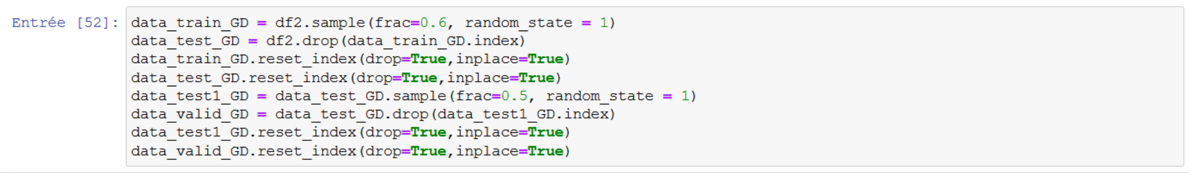
408 rows × 7 columns



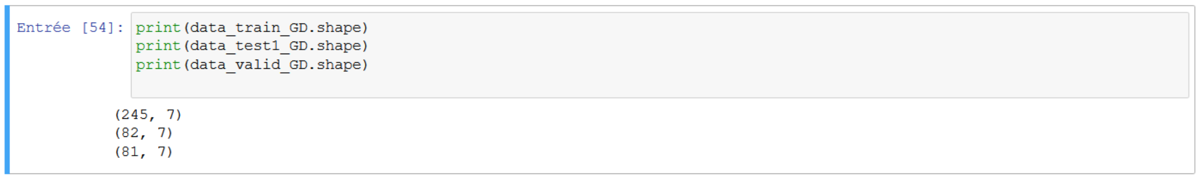
```
Entrée [51]: X1222=df2.drop(['Target'],axis=1).values
             y1222=df2['Target'].values
             print(X1222.shape)
             print(y1222.shape)

             (408, 6)
             (408,)
```

3.3.2 Splitting the dataset

We split our Dataset into train data and test data and validation data which 60% of the data will be used for training the model while 20% will be used for testing the model and 20% for the validation of our model. After splitting the dataset, we get the following shapes:



```
Entrée [52]: data_train_GD = df2.sample(frac=0.6, random_state = 1)
             data_test_GD = df2.drop(data_train_GD.index)
             data_train_GD.reset_index(drop=True,inplace=True)
             data_test_GD.reset_index(drop=True,inplace=True)
             data_test1_GD = data_test_GD.sample(frac=0.5, random_state = 1)
             data_valid_GD = data_test_GD.drop(data_test1_GD.index)
             data_test1_GD.reset_index(drop=True,inplace=True)
             data_valid_GD.reset_index(drop=True,inplace=True)
```



```
Entrée [54]: print(data_train_GD.shape)
             print(data_test1_GD.shape)
             print(data_valid_GD.shape)

             (245, 7)
             (82, 7)
             (81, 7)
```

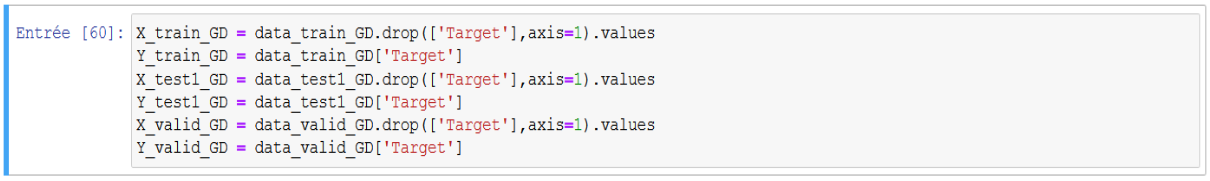
here we will notice that our data test and data valid are not equal because our data contain 408 rows and 7 attributes. So, when we split the data, we will have:

60% data training: 245 rows and 7 attributes

20% data testing: 82 rows and 7 attributes

And the rest 20% data validation: 81 rows and 7 attributes

Now, after splitting our dataset, we define X train, Y train, X test, Y test, X valid, Y valid



```
Entrée [60]: X_train_GD = data_train_GD.drop(['Target'],axis=1).values
             Y_train_GD = data_train_GD['Target']
             X_test1_GD = data_test1_GD.drop(['Target'],axis=1).values
             Y_test1_GD = data_test1_GD['Target']
             X_valid_GD = data_valid_GD.drop(['Target'],axis=1).values
             Y_valid_GD = data_valid_GD['Target']
```

3.3.3 Gradient descent, the cost function and the hypothesis

we must use the cost function to minimize errors and optimize the thetas, to build the cost function here's the following formula:

$$J(\theta) = \frac{1}{2m} \sum (h_\theta(x^{(i)}) - y^{(i)})^2$$

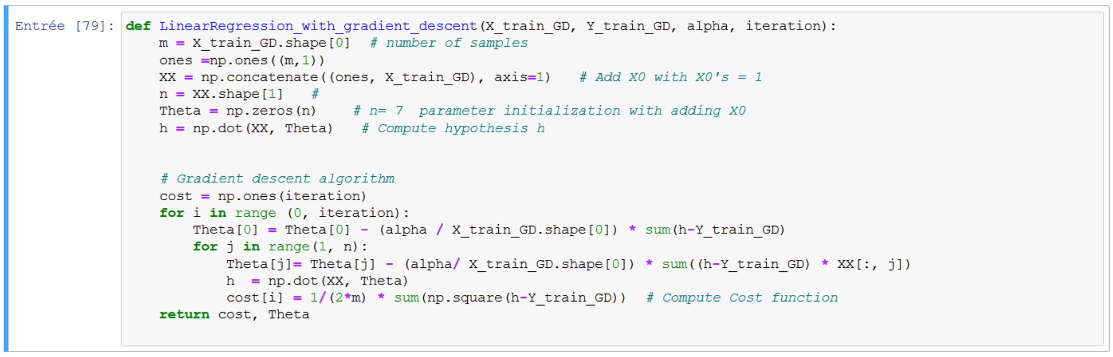
and h(x) is the hypothesis:

$$\text{Hypothesis: } h_\theta = \theta^T x$$

The gradient descent updates the thetas after every iteration until it converges to zero. here's the following formula:

And Alpha here mean the learning rate.

$$\text{Repeat}\{ \quad \theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)} \quad j = 0, 1, \ldots, n \quad \}$$

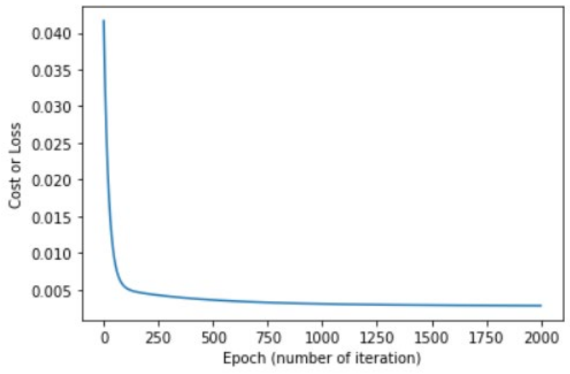


```
Entrée [79]: def LinearRegression_with_gradient_descent(X_train_GD, Y_train_GD, alpha, iteration):
                 m = X_train_GD.shape[0]   # number of samples
                 ones =np.ones((m,1))
                 XX = np.concatenate((ones, X_train_GD), axis=1)    # Add X0 with X0's = 1
                 n = XX.shape[1]    #
                 Theta = np.zeros(n)     # n= 7  parameter initialization with adding X0
                 h = np.dot(XX, Theta)    # Compute hypothesis h


                 # Gradient descent algorithm
                 cost = np.ones(iteration)
                 for i in range (0, iteration):
                     Theta[0] = Theta[0] - (alpha / X_train_GD.shape[0]) * sum(h-Y_train_GD)
                     for j in range(1, n):
                         Theta[j]= Theta[j] - (alpha/ X_train_GD.shape[0]) * sum((h-Y_train_GD) * XX[:, j])
                         h  = np.dot(XX, Theta)
                         cost[i] = 1/(2*m) * sum(np.square(h-Y_train_GD))   # Compute Cost function
                 return cost, Theta
```

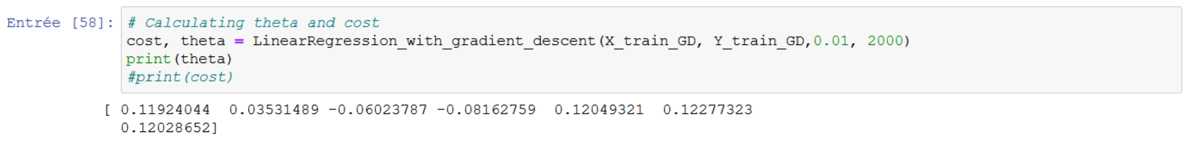
3.3.4 Testing different values of alpha and plotting the cost function

In this step we tested different values of alpha in order to find the best value to minimize the cost, we must set the learning rate to an appropriate value (not high and not low value) and we found that alpha = 0.01 is the best value for our dataset (we choose 2000 iterations).

The image below shows us that the value of J(θ) is decreasing for every iteration (gradient descent converge) which means that our model applies a good learning.



And the appropriate thetas are:

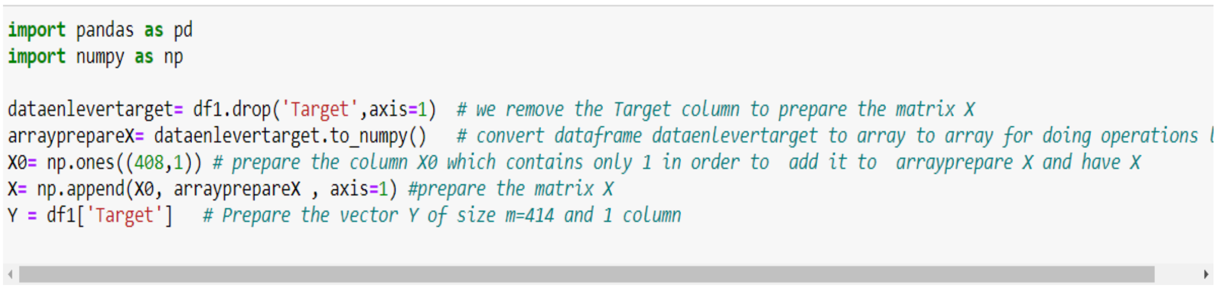


```
Entrée [58]: # Calculating theta and cost
             cost, theta = LinearRegression_with_gradient_descent(X_train_GD, Y_train_GD,0.01, 2000)
             print(theta)
             #print(cost)

[ 0.11924044  0.03531489 -0.06023787 -0.08162759  0.12049321  0.12277323
  0.12028652]
```

3.4. Normal Equation

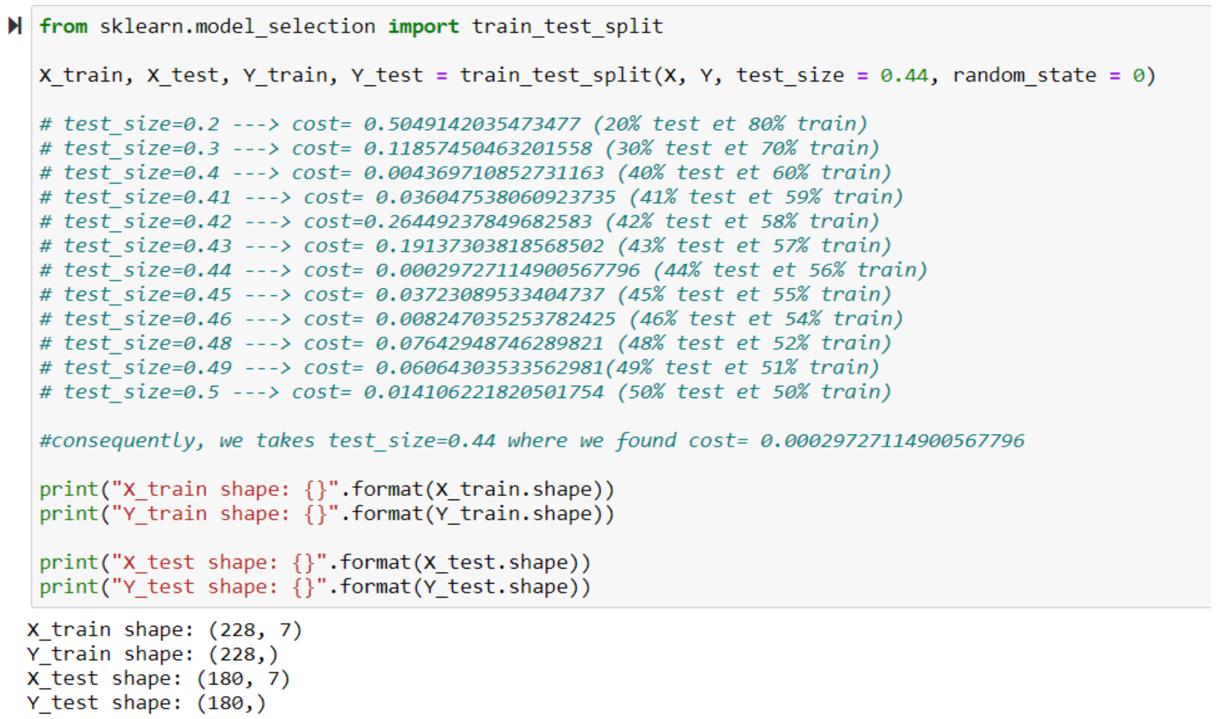
The Normal equation is a method that allows us to find the Teta θ analytically and will give us a much better way to solve for the optimal value of θ in our regression problem. We don't need to choose **α** and we don't need for feature scaling. We just need to calculate: inv(X.T * X) * X.T*Y.

3.4.1 Preparing the X and Y in order to apply the previous formula:



```
import pandas as pd
import numpy as np

dataenlevertarget= df1.drop('Target',axis=1)  # we remove the Target column to prepare the matrix X
arrayprepareX= dataenlevertarget.to_numpy()   # convert dataframe dataenlevertarget to array to array for doing operations l
X0= np.ones((408,1)) # prepare the column X0 which contains only 1 in order to  add it to  arrayprepare X and have X
X= np.append(X0, arrayprepareX , axis=1) #prepare the matrix X
Y = df1['Target']   # Prepare the vector Y of size m=414 and 1 column
```

3.4.2 Splitting the data in train and test datasets

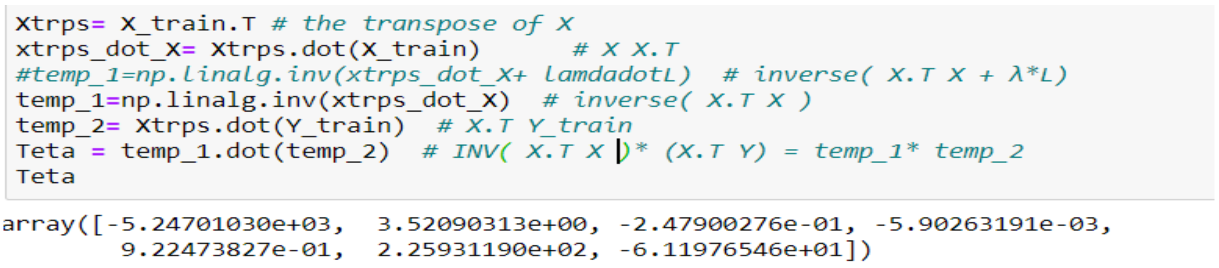


```
from sklearn.model_selection import train_test_split

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.44, random_state = 0)

# test_size=0.2 ---> cost= 0.5049142035473477 (20% test et 80% train)
# test_size=0.3 ---> cost= 0.11857450463201558 (30% test et 70% train)
# test_size=0.4 ---> cost= 0.004369710852731163 (40% test et 60% train)
# test_size=0.41 ---> cost= 0.036047538060923735 (41% test et 59% train)
# test_size=0.42 ---> cost=0.26449237849682583 (42% test et 58% train)
# test_size=0.43 ---> cost= 0.19137303818568502 (43% test et 57% train)
# test_size=0.44 ---> cost= 0.00029727114900567796 (44% test et 56% train)
# test_size=0.45 ---> cost= 0.03723089533404737 (45% test et 55% train)
# test_size=0.46 ---> cost= 0.008247035253782425 (46% test et 54% train)
# test_size=0.48 ---> cost= 0.07642948746289821 (48% test et 52% train)
# test_size=0.49 ---> cost= 0.060643035333562981(49% test et 51% train)
# test_size=0.5 ---> cost= 0.014106221820501754 (50% test et 50% train)

#consequently, we takes test_size=0.44 where we found cost= 0.00029727114900567796

print("X_train shape: {}".format(X_train.shape))
print("Y_train shape: {}".format(Y_train.shape))

print("X_test shape: {}".format(X_test.shape))
print("Y_test shape: {}".format(Y_test.shape))
```

```
X_train shape: (228, 7)
Y_train shape: (228,)
X_test shape: (180, 7)
Y_test shape: (180,)
```

After many tests, we found that the test_size=0.44 give us the best value of cost.

3.4.3 Calculate the θ

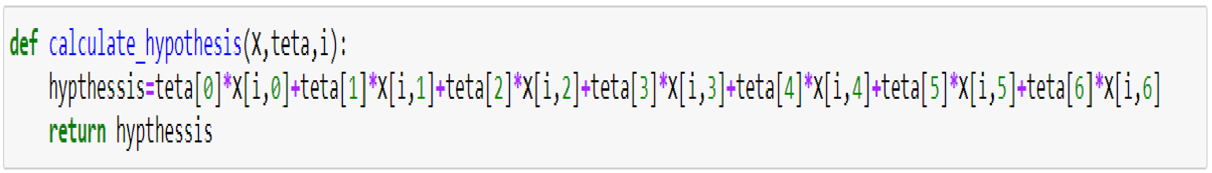


```
Xtrps= X_train.T # the transpose of X
xtrps_dot_X= Xtrps.dot(X_train)      # X X.T
#temp_1=np.linalg.inv(xtrps_dot_X+ lamdadotL)  # inverse( X.T X + λ*L)
temp_1=np.linalg.inv(xtrps_dot_X)  # inverse( X.T X )
temp_2= Xtrps.dot(Y_train)  # X.T Y_train
Teta = temp_1.dot(temp_2)  # INV( X.T X )* (X.T Y) = temp_1* temp_2
Teta
```

```
array([-5.24701030e+03,  3.52090313e+00, -2.47900276e-01, -5.90263191e-03,
        9.22473827e-01,  2.25931190e+02, -6.11976546e+01])
```

3.4.4 Function calculate-hypothesis

In order to calculate the hypothesis, which are used later in order to calculate the mean-square error (cost), we implement this method:



```
def calculate_hypothesis(X,teta,i):
    hypthessis=teta[0]*X[i,0]+teta[1]*X[i,1]+teta[2]*X[i,2]+teta[3]*X[i,3]+teta[4]*X[i,4]+teta[5]*X[i,5]+teta[6]*X[i,6]
    return hypthessis
```

3.4.5 Function Cost

Now, we will calculate the cost between the Y_test and the prediction of our X_test using the previous function

```
def compute_cost(X,Y,teta): #calculate the meen squar error wich includes the function calculate_hypothesis
    J=0.0
    m=np.size(Y,0)
    for i in range(m):
        hypothessis= calculate_hypothesis(X,Teta,i)
        squared_error=np.power((hypothessis-Y.iloc[i]),2) # (h(θ)i-yi)^2
        J=J+squared_error
        J=J*(1.0/(2*m)) # (1/2*m )* sum de i allant de 1 à m de notre (h(θ)i-yi)^2
    return J
```

We will now calculate the cost_test and the cost_train in order to compare between the two values and see if we have the possibility to have overfitting in our dataset .

```
cost=compute_cost(X_test,Y_test,Teta) #calculate the meen squar error wich includes the function calculate_hypothesis
cost
```

```
0.00029727114900567796
```

```
cost2=compute_cost(X_train,Y_train,Teta) #calculate the meen squar error wich includes the function calculate_hypothesis in
cost2
#we can notify that cost2 which represent the meen squar error train < cost which represent the meen squar error test
```

```
0.0008030134702190326
```

As we can see, the cost (X_test, Y_test, Teta) < cost (X_train, Y_train, Teta), so we have a good result in our cost_test and it is not just in cost_train.

NB :
<you find the regularization in Appendix B.1 Regularization in Normal Equation >

3.5. Neural Network

3.5.1. Splitting the Dataset

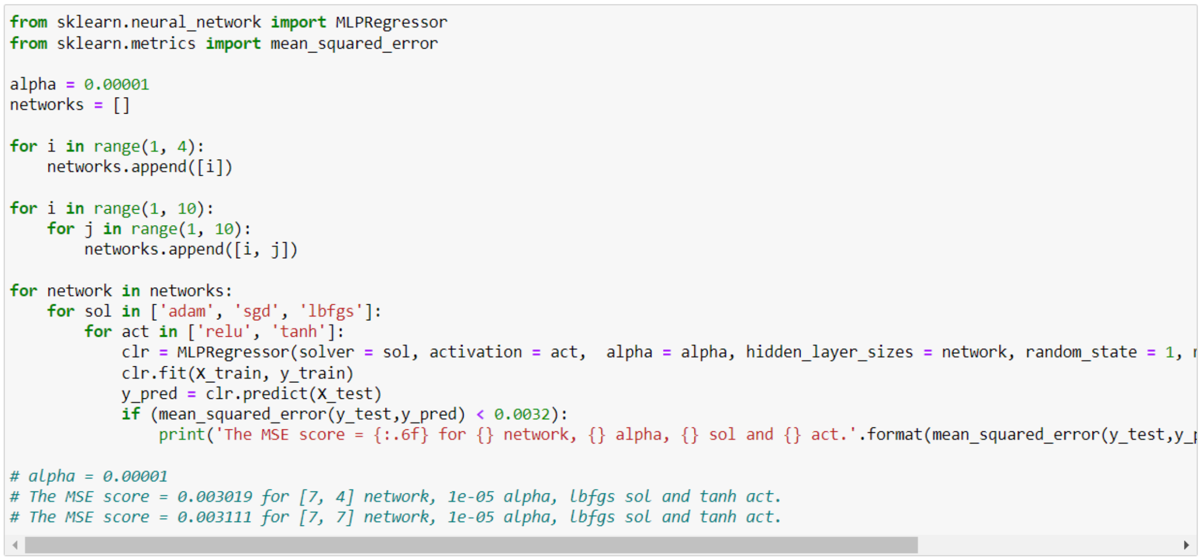
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 0)

print ("Train set X shape = ", X_train.shape, "\nTrain set y shape = ", y_train.shape)

print ("Test set X shape  = ", X_test.shape, "\nTest set y shape  = ", y_test.shape)
```

```
Train set X shape =  (326, 6)
Train set y shape =  (326,)
Test set X shape  =  (82, 6)
Test set y shape  =  (82,)
```

3.5.2. Tuning parameters for the model

The goal of this part of the process is to find the best parameters for the multi-layer perceptron regressor so it can predict the best possible continuous output data based on input data (number of layers and neurons, alpha value etc.)

To do that, we used for a couple first tests a set of loops that go through some values of different parameters that the model can use.



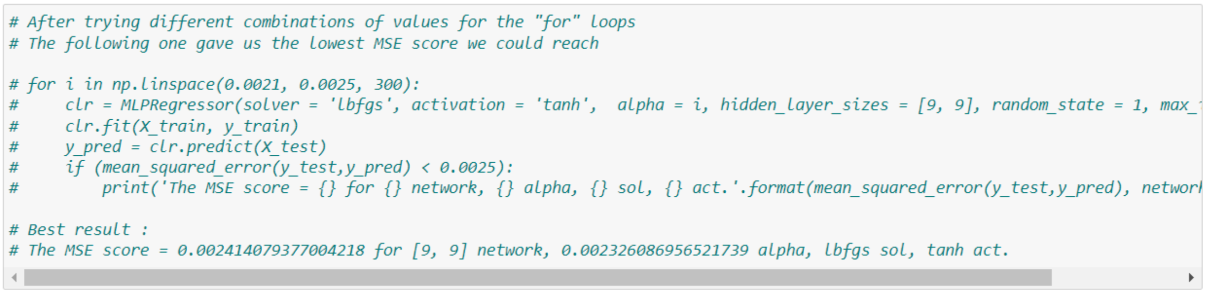
```
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_squared_error

alpha = 0.00001
networks = []

for i in range(1, 4):
    networks.append([i])

for i in range(1, 10):
    for j in range(1, 10):
        networks.append([i, j])

for network in networks:
    for sol in ['adam', 'sgd', 'lbfgs']:
        for act in ['relu', 'tanh']:
            clr = MLPRegressor(solver = sol, activation = act,  alpha = alpha, hidden_layer_sizes = network, random_state = 1, r
            clr.fit(X_train, y_train)
            y_pred = clr.predict(X_test)
            if (mean_squared_error(y_test,y_pred) < 0.0032):
                print('The MSE score = {:.6f} for {} network, {} alpha, {} sol and {} act.'.format(mean_squared_error(y_test,y_

# alpha = 0.00001
# The MSE score = 0.003019 for [7, 4] network, 1e-05 alpha, lbfgs sol and tanh act.
# The MSE score = 0.003111 for [7, 7] network, 1e-05 alpha, lbfgs sol and tanh act.
```

After several tests with this method, we noticed after that:

- Logistic activation function does not produce any interesting results at all no matter the chosen alpha values (0.00001 etc.), we had to remove it for the remaining tests.
- All three solver can produce relatively similar results, so we must try testing all three solvers with more larger sets of values for the rest of the parameters.

We had to keep testing all solvers with more precise parameters (alpha, network, with “relu” and “tanh” activation function).

After several tests, we managed to find the lowest mean square error value (0.002414) we could reach, which is an acceptable value, and a relatively good r-squared (0.08359), for a [9, 9] neural network structure, with the "lbfgs" solver and the "tanh" activation function. We also notice that the model



```
# After trying different combinations of values for the "for" loops
# The following one gave us the lowest MSE score we could reach

# for i in np.linspace(0.0021, 0.0025, 300):
#     clr = MLPRegressor(solver = 'lbfgs', activation = 'tanh',  alpha = i, hidden_layer_sizes = [9, 9], random_state = 1, max_
#     clr.fit(X_train, y_train)
#     y_pred = clr.predict(X_test)
#     if (mean_squared_error(y_test,y_pred) < 0.0025):
#         print('The MSE score = {} for {} network, {} alpha, {} sol, {} act.'.format(mean_squared_error(y_test,y_pred), networ

# Best result :
# The MSE score = 0.002414079377004218 for [9, 9] network, 0.002326086956521739 alpha, lbfgs sol, tanh act.
```

3.5.3. Final Results

```
print('Mean Squarred Error scores :')
clr = MLPRegressor(solver='lbfgs', activation = 'tanh', alpha = 0.002326086956521739, hidden_layer_sizes=[9, 9], max_iter= 2500,
clr.fit(X_train, y_train)
y_pred = clr.predict(X_test)
y_train_pred = clr.predict(X_train)
print('Train set MSE score: '+'{}'.format(mean_squared_error(y_train,y_train_pred)))
print('Test set MSE score : '+'{}'.format(mean_squared_error(y_test,y_pred)))
```

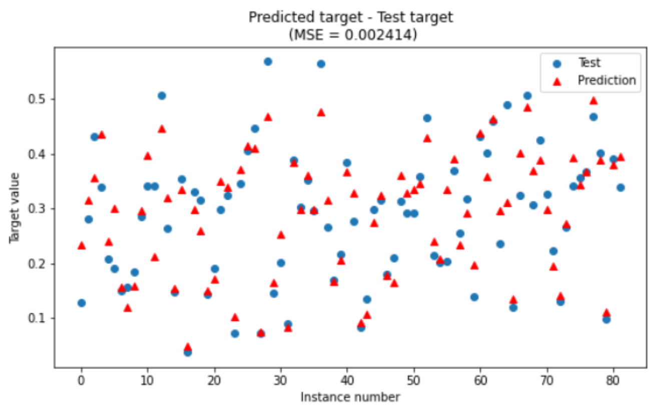
```
Mean Squarred Error scores :
Train set MSE score: 0.004016697204253358
Test set MSE score : 0.002414079377004218
```

```
print('R Squared score :')
print('Train set R2 score: '+'{}'.format(r2_score(y_train,y_train_pred)))
print('Test set R2 score : '+'{}'.format(r2_score(y_test,y_pred)))
```

```
R Squared score :
Train set R2 score: 0.7317590110415303
Test set R2 score : 0.8359174721672065
```

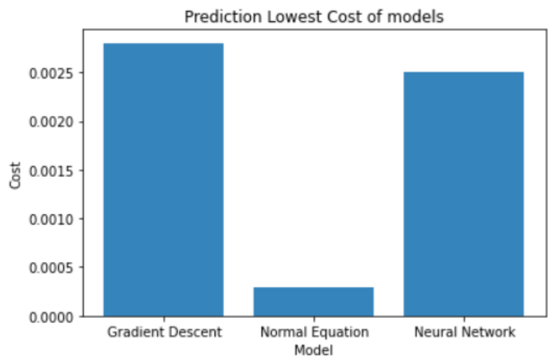
```
# Plotting the predicted targets and the test targets

plt.figure(figsize=(9, 5))
plt.scatter(range(0, 82), y_test)
plt.scatter(range(0, 82), y_pred, marker = "^", color = "red")
plt.legend(["Test", "Prediction"])
plt.xlabel("Instance number")
plt.ylabel("Target value")
plt.title("Predicted target - Test target\n (MSE = 0.002414)")
```

```
Text(0.5, 1.0, 'Predicted target - Test target\n (MSE = 0.002414)')
```



In conclusion, our Neural Network model gives us a Mean Squared Error value of 0.002413, and a R-Squared value of 0.835917.

3.6 Comparing models



At the end, we can conclude that the best suitable training algorithm for our dataset is Normal Equation because the cost function of Normal Equation is small than others.

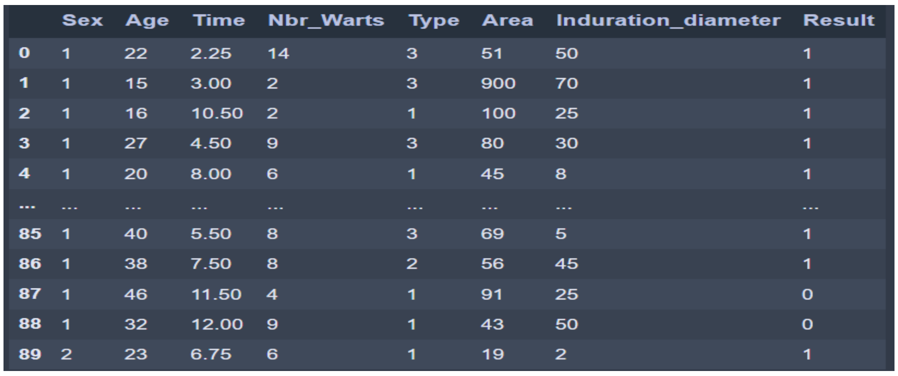
4. Immunotherapy Treatment Result

4.1. Dataset Description

This dataset contains information about wart treatment results of 90 patients using immunotherapy.

4.2. Preprocessing

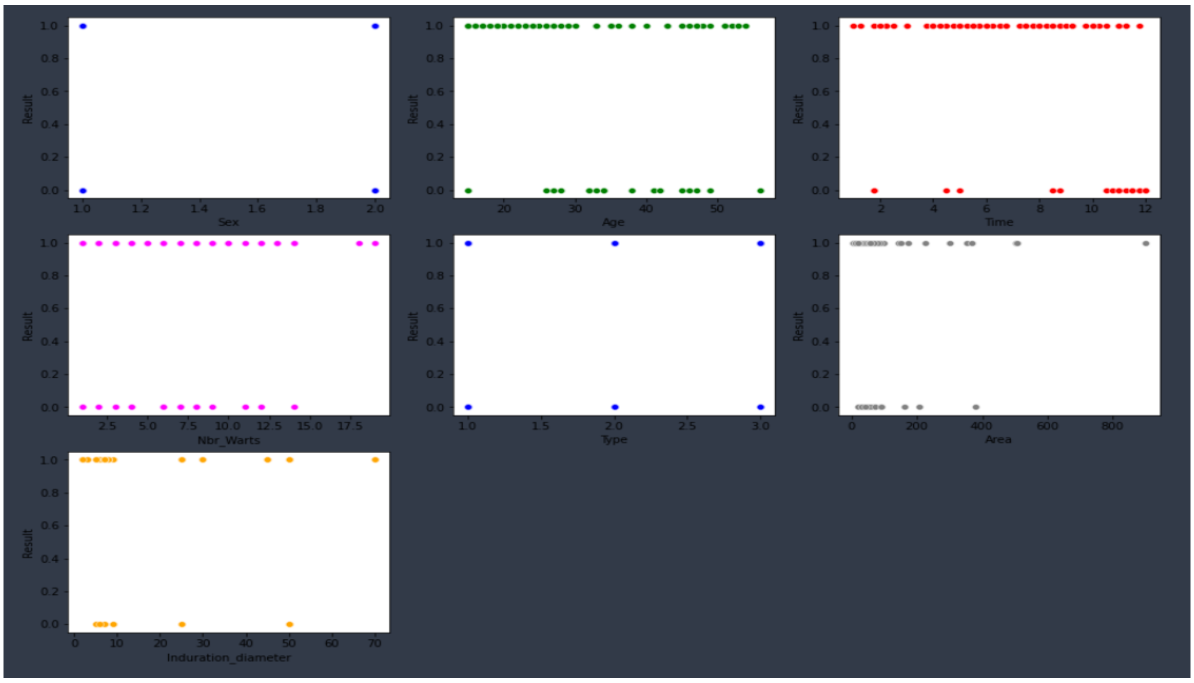
The imported Immunotherapy dataset “df”:



	Sex	Age	Time	Nbr_Warts	Type	Area	Induration_diameter	Result
0	1	22	2.25	14	3	51	50	1
1	1	15	3.00	2	3	900	70	1
2	1	16	10.50	2	1	100	25	1
3	1	27	4.50	9	3	80	30	1
4	1	20	8.00	6	1	45	8	1
...	...	...	...	...	...	...	...	...
85	1	40	5.50	8	3	69	5	1
86	1	38	7.50	8	2	56	45	1
87	1	46	11.50	4	1	91	25	0
88	1	32	12.00	9	1	43	50	0
89	2	23	6.75	6	1	19	2	1

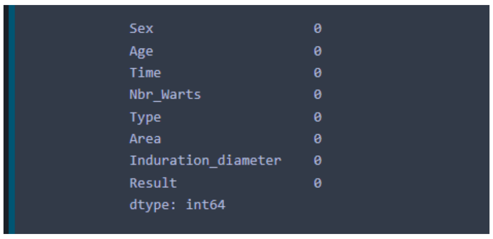
4.2.1 Graphic Representation

We represent all the 7 features in comparison to ‘Result’ in 7 subplots:



4.2.2 Missing values

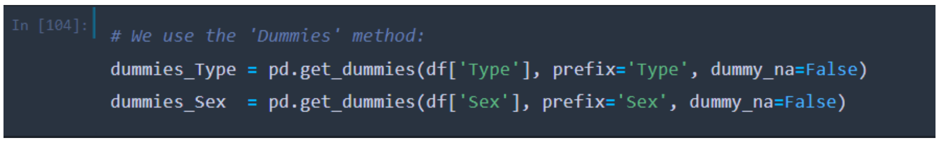
We use the command **isna().sum()** to check if there are any missing values



```
Sex                    0
Age                    0
Time                   0
Nbr_Warts              0
Type                   0
Area                   0
Induration_diameter    0
Result                 0
dtype: int64
```

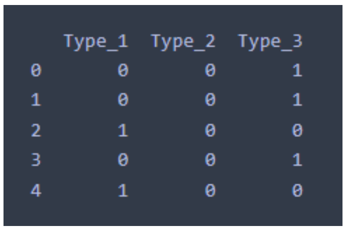
4.2.3 Removing Categorical values

The features 'Sex' and 'Type' are of Categorical type, we have to transform them to a numerical type (int) using the 'Dummies' method:



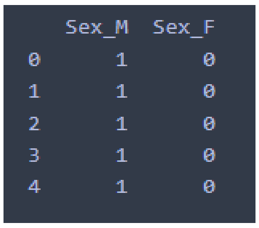
```
In [104]:  # We use the 'Dummies' method:
            dummies_Type = pd.get_dummies(df['Type'], prefix='Type', dummy_na=False)
            dummies_Sex  = pd.get_dummies(df['Sex'], prefix='Sex', dummy_na=False)
```

'Type' dummies:



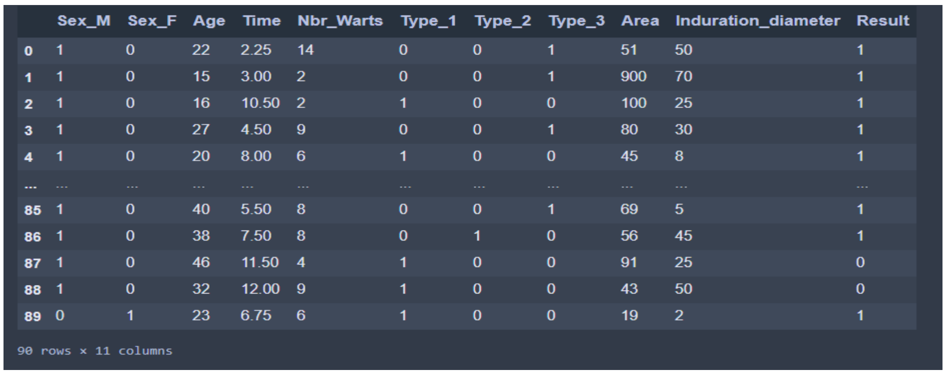
```
   Type_1  Type_2  Type_3
0       0       0       1
1       0       0       1
2       1       0       0
3       0       0       1
4       1       0       0
```

'Sex' dummies:



```
   Sex_M  Sex_F
0      1      0
1      1      0
2      1      0
3      1      0
4      1      0
```

-We remove the features 'Type' and 'Sex' from the initial dataset df and we merge all the features:



	Sex_M	Sex_F	Age	Time	Nbr_Warts	Type_1	Type_2	Type_3	Area	Induration_diameter	Result
0	1	0	22	2.25	14	0	0	1	51	50	1
1	1	0	15	3.00	2	0	0	1	900	70	1
2	1	0	16	10.50	2	1	0	0	100	25	1
3	1	0	27	4.50	9	0	0	1	80	30	1
4	1	0	20	8.00	6	1	0	0	45	8	1
...	...	...	...	...	...	...	...	...	...	...	...
85	1	0	40	5.50	8	0	0	1	69	5	1
86	1	0	38	7.50	8	0	1	0	56	45	1
87	1	0	46	11.50	4	1	0	0	91	25	0
88	1	0	32	12.00	9	1	0	0	43	50	0
89	0	1	23	6.75	6	1	0	0	19	2	1

```
90 rows × 11 columns
```

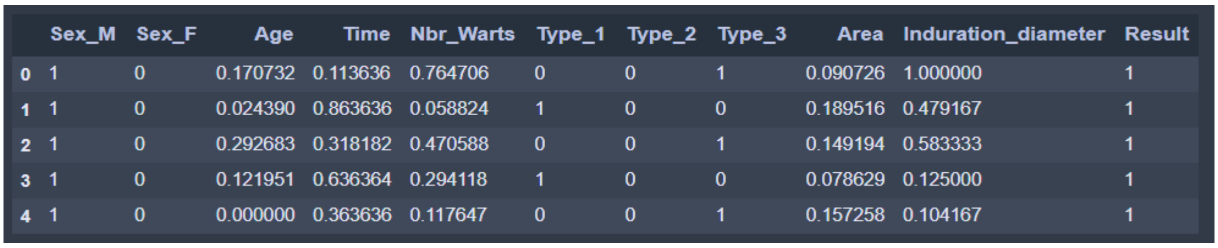
4.2.4 Detecting Outliers

We use the **'z_score'** method to detect the outliers (you can find it in the Appendix D1)

4.2.5 Feature scaling

To make all values of features between 0 and 1 we use MinMaxScaler from sklearn.preprocessing.

Df after the feature scaling:



	Sex_M	Sex_F	Age	Time	Nbr_Warts	Type_1	Type_2	Type_3	Area	Induration_diameter	Result
0	1	0	0.170732	0.113636	0.764706	0	0	1	0.090726	1.000000	1
1	1	0	0.024390	0.863636	0.058824	1	0	0	0.189516	0.479167	1
2	1	0	0.292683	0.318182	0.470588	0	0	1	0.149194	0.583333	1
3	1	0	0.121951	0.636364	0.294118	1	0	0	0.078629	0.125000	1
4	1	0	0.000000	0.363636	0.117647	0	0	1	0.157258	0.104167	1

(And this is also the final result after all of the pre-processing methods).

4.3. Logistic Regression

4.3.1 Splitting the dataset

After splitting the dataset, we get the following shapes:

```
X_valid, X_test, Y_valid, Y_test = train_test_split(X_rest,Y_rest,test_size=0.5,random_state=0)
```

```
print(X_train.shape)
print(X_valid.shape)
print(X_test.shape)
```

```
(67, 10)
(8, 10)
(9, 10)
```

4.3.2 definition of the functions

To be able to perform the Gradient Descent, we need to implement the following functions first.

```
#def of the sigmoid function
def sigmoid(x):
    return 1/(1 + np.exp(-x))
```

```
#def of the hypothesis
def lr_hypothesis(x, theta):
    return np.dot(theta.T, x)
```

```
#def of the cost function
def compute_cost(Y, A):
    m = X_train.shape[1]
    cost = -(1/m)*np.sum(Y*np.log(A) + (1-Y)*np.log(1-A))
    return cost
```

4.3.3 Implementation of the Gradient Descent

Now we use the functions we defined previously to perform the Gradient Descent and train our model using the train part of the dataset. The Gradient will perform a set amount of times "iterations" the calculations shown below in order to reduce the cost and give the best values of theta.

```
#perform Gradient Descent
def gradientDescent(X, Y, alpha, iterations):
    CostVect = []
    m = X_train.shape[1]
    n = X_train.shape[0]
    theta = np.zeros((n,1))
    B = 0
    for i in range(iterations):
        Z = lr_hypothesis(X, theta) + B
        A = sigmoid(Z)
        cost = compute_cost(Y, A)
        dW = (1/m)*np.dot(A-Y, X.T)
        dB =  (1/m)*np.sum(A-Y)
        theta = theta - alpha*dW.T
        B = B - alpha*dB
        CostVect.append(cost)
        if(i%(iterations/10) == 0):
            print("cost after ", i, "iterations is : ", cost)
    print("Value of cost at the end of the Gradient Descent : ",cost)
    return theta, B, CostVect
```

4.3.4 Testing different values of alpha

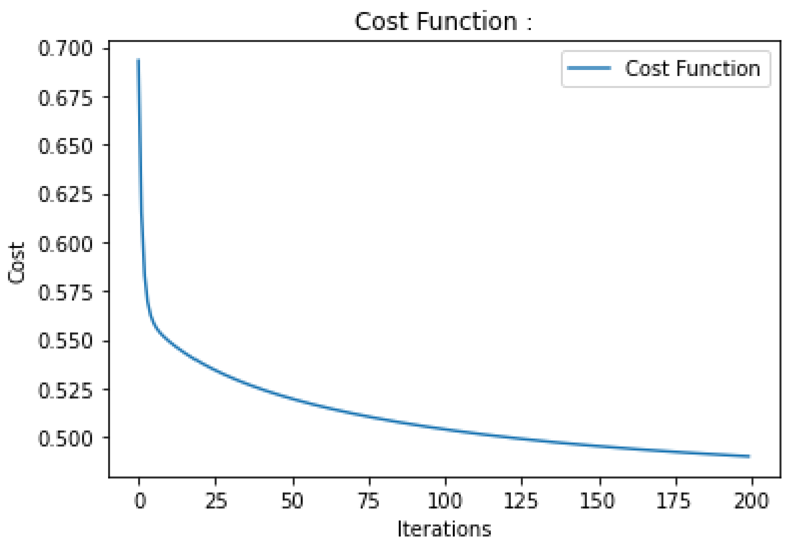
The goal here is to minimize the cost, so we tried different values of "alpha" and "iterations" in order to get the best result possible and found that alpha = 0.6 is the best value for our dataset. If we go further than 0.6, we will no longer see a big decrease in the final value of the cost.

```
#cost = 0.5417328938326874        when alpha = 0.05
#cost = 0.5285871335280208        when alpha = 0.1
#cost = 0.5039761442833408        when alpha = 0.3
#cost = 0.4899756627599029        when alpha = 0.6

#cost = 0.520016930268673         with iterations = 50
#cost = 0.5040530028527112        with iterations = 100
#cost = 0.4953224633557965        with iterations = 150
#cost = 0.4899756627599029        with iterations = 200
```

We can see below the decrease of the cost value while training the model.

```
cost after  0 iterations is :  0.6931471805599454
cost after  20 iterations is :  0.5383401636142672
cost after  40 iterations is :  0.5244854747404475
cost after  60 iterations is :  0.5154489246310349
cost after  80 iterations is :  0.5088786814234666
cost after  100 iterations is :  0.503828189813101
cost after  120 iterations is :  0.49982200560531
cost after  140 iterations is :  0.49658152821554824
cost after  160 iterations is :  0.4939252372797384
cost after  180 iterations is :  0.4917261988529806
Value of cost at the end :  0.4899756627599029
```

And here is the plot of the cost function.



At the end, we get an accuracy of 77.77% when predicting the test set, which is a good value. (Picture of the accuracy function in the appendix)

We can now use the predefined function in "sklearn" to compare our results, and we can see that we got an identical accuracy of 77.77%. (Picture of the code function in the appendix)

4.4. Neural Network

4.4.1. Splitting the dataset

Train sets and tests sets are split as follows:

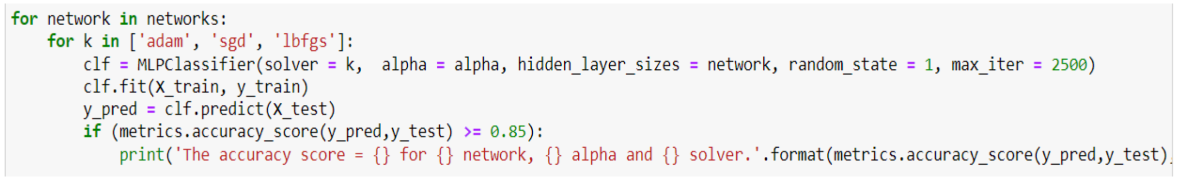
```
Train set X shape =  (67, 10)
Train set y shape =  (67,)
Test set X shape  =  (17, 10)
Test set y shape  =  (17,)
```

4.4.2. Tuning parameters for the model

The goal of this part of the process is to find the best parameters for the multi-layer perceptron classifier so it can predict the best possible output data based on input data (number of layers and neurons, alpha value etc.).

To do that, we used for a few first tests a set of loops that go through some values of different parameters that the model can use.

- A more detailed version of this section of code is to be found in the appendix A.1.



```
for network in networks:
    for k in ['adam', 'sgd', 'lbfgs']:
        clf = MLPClassifier(solver = k,  alpha = alpha, hidden_layer_sizes = network, random_state = 1, max_iter = 2500)
        clf.fit(X_train, y_train)
        y_pred = clf.predict(X_test)
        if (metrics.accuracy_score(y_pred,y_test) >= 0.85):
            print('The accuracy score = {} for {} network, {} alpha and {} solver.'.format(metrics.accuracy_score(y_pred,y_test)
```

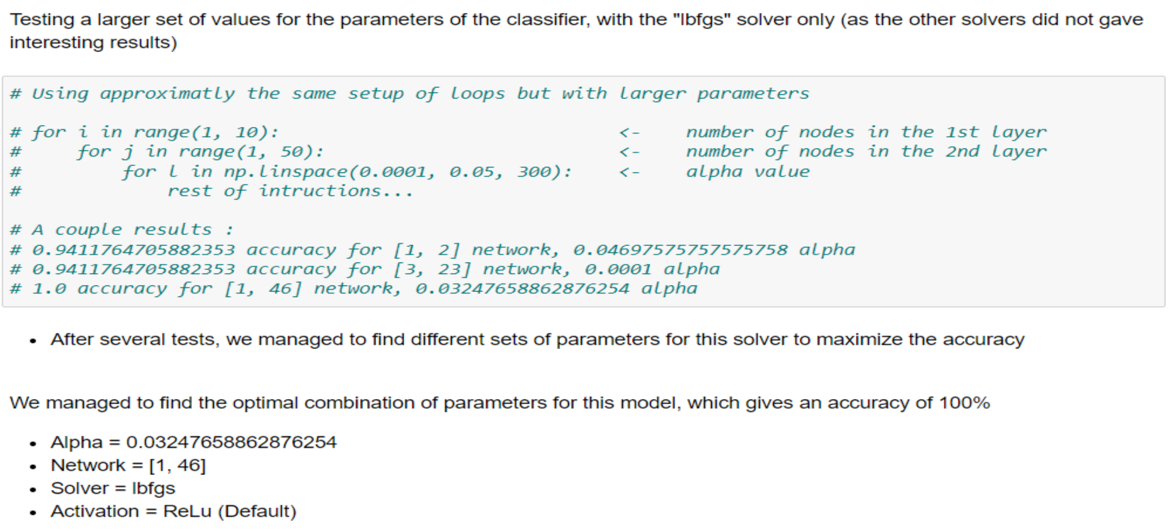
After a few tests with this method, we noticed that:

- We can use “relu” activation function as a parameter for the classifier for the remaining tests.
- No matter the alpha value chosen manually, we can reach a maximum accuracy value of 94.11%, we must add a loop to test a larger set of alphas to try to increase the accuracy.

We must keep testing all solvers with more precise parameters (alpha, network, but the same “relu” activation function).

After several tests, we noticed that the “adam” and “sgd” solvers could not give good results no matter the range of parameters we fitted into the “for” loops.

Conclusion: In the final executions, we managed to find an optimal set of parameters for the classifier so it can give an accuracy of 100%, as follows:



Testing a larger set of values for the parameters of the classifier, with the "lbfgs" solver only (as the other solvers did not gave interesting results)

```
# Using approximatly the same setup of loops but with larger parameters

# for i in range(1, 10):                               <-    number of nodes in the 1st layer
#     for j in range(1, 50):                           <-    number of nodes in the 2nd layer
#         for l in np.linspace(0.0001, 0.05, 300):     <-    alpha value
#             rest of intructions...

# A couple results :
# 0.9411764705882353 accuracy for [1, 2] network, 0.04697575757575758 alpha
# 0.9411764705882353 accuracy for [3, 23] network, 0.0001 alpha
# 1.0 accuracy for [1, 46] network, 0.03247658862876254 alpha
```

- After several tests, we managed to find different sets of parameters for this solver to maximize the accuracy

We managed to find the optimal combination of parameters for this model, which gives an accuracy of 100%

- Alpha = 0.03247658862876254
- Network = [1, 46]
- Solver = lbfgs
- Activation = ReLu (Default)

GridSearchCV:

We tried to optimize the model parameters using the GridSerachCV algorithm.

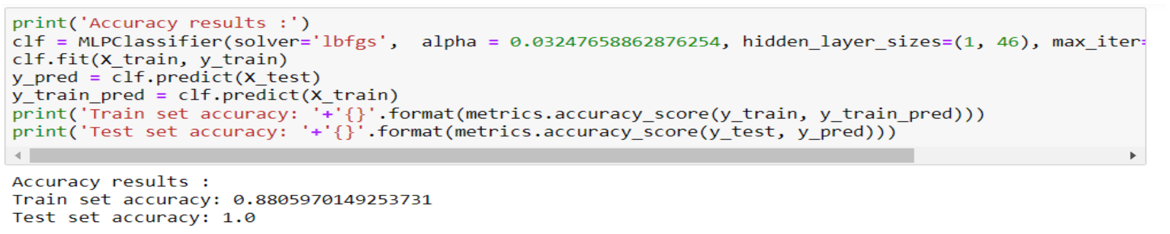
In brief, it does the process of performing parameter tuning in order to determine the optimal values for a given model, as we pass predefined values for parameters to the GridSearchCV function by defining a dictionary in which we mention a particular parameter along with the values it can take.

We initialized the searching algorithm by giving it the model we're trying to optimize, a parameter space that contains different values for the different parameters that the model can have, and other parameters. After the search, the algorithm returned the best parameters it could find.

We concluded that using this algorithm was not an interesting way to find the optimal parameters for our model, as the accuracy produced with the best parameters the algorithm could find fitted in the mlp classifier is lower as the accuracy we found with the method above.

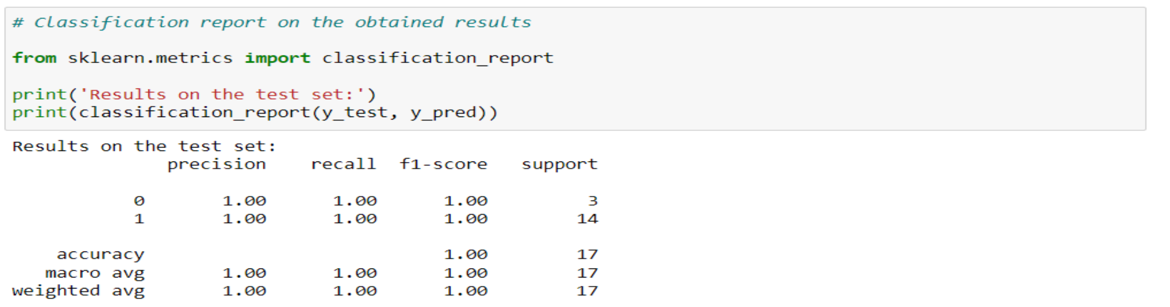
- All the details of this part are to be found in the appendix A.2.

4.4.3. Final Results



```
print('Accuracy results :')
clf = MLPClassifier(solver='lbfgs',  alpha = 0.03247658862876254, hidden_layer_sizes=(1, 46), max_iter=
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
y_train_pred = clf.predict(X_train)
print('Train set accuracy: '+'{}'.format(metrics.accuracy_score(y_train, y_train_pred)))
print('Test set accuracy: '+'{}'.format(metrics.accuracy_score(y_test, y_pred)))
```

```
Accuracy results :
Train set accuracy: 0.8805970149253731
Test set accuracy: 1.0
```

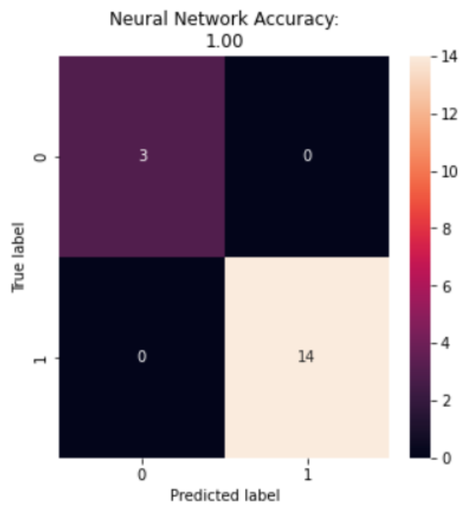


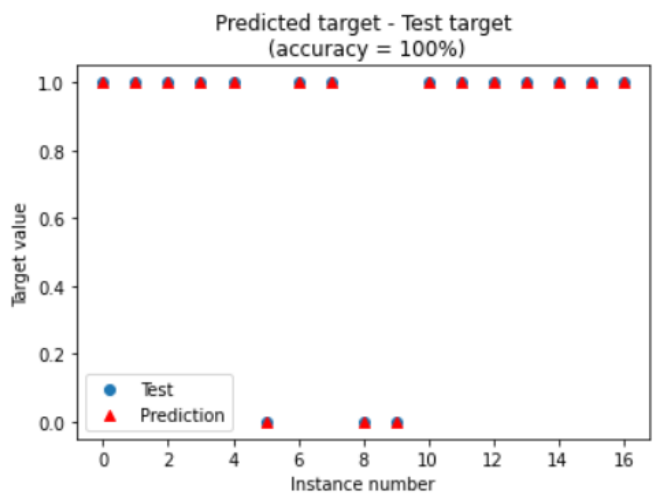
```
# Classification report on the obtained results

from sklearn.metrics import classification_report

print('Results on the test set:')
print(classification_report(y_test, y_pred))
```

```
Results on the test set:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00         3
           1       1.00      1.00      1.00        14

    accuracy                           1.00        17
   macro avg       1.00      1.00      1.00        17
weighted avg       1.00      1.00      1.00        17
```

4.4.4. Confusion Matrix - Plotting the results

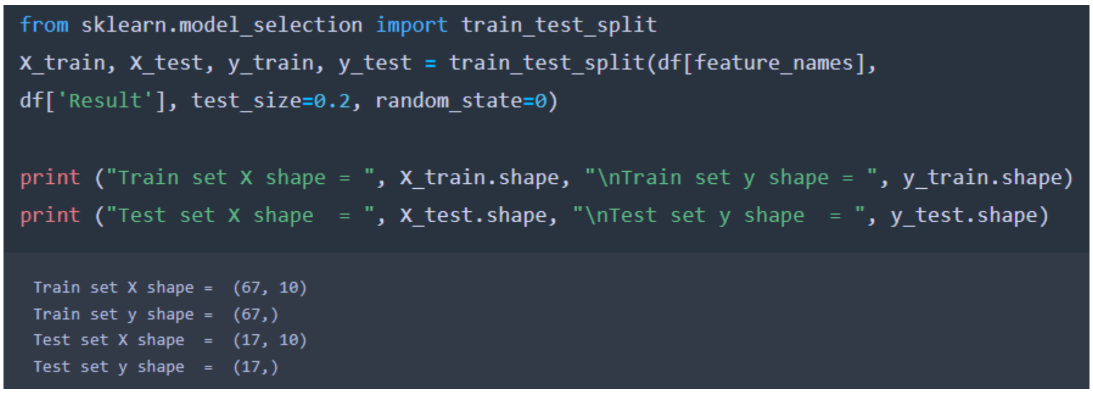
- Code details in the appendix A.3.





4.5 SVM

4.5.1 Train and Test Datasets

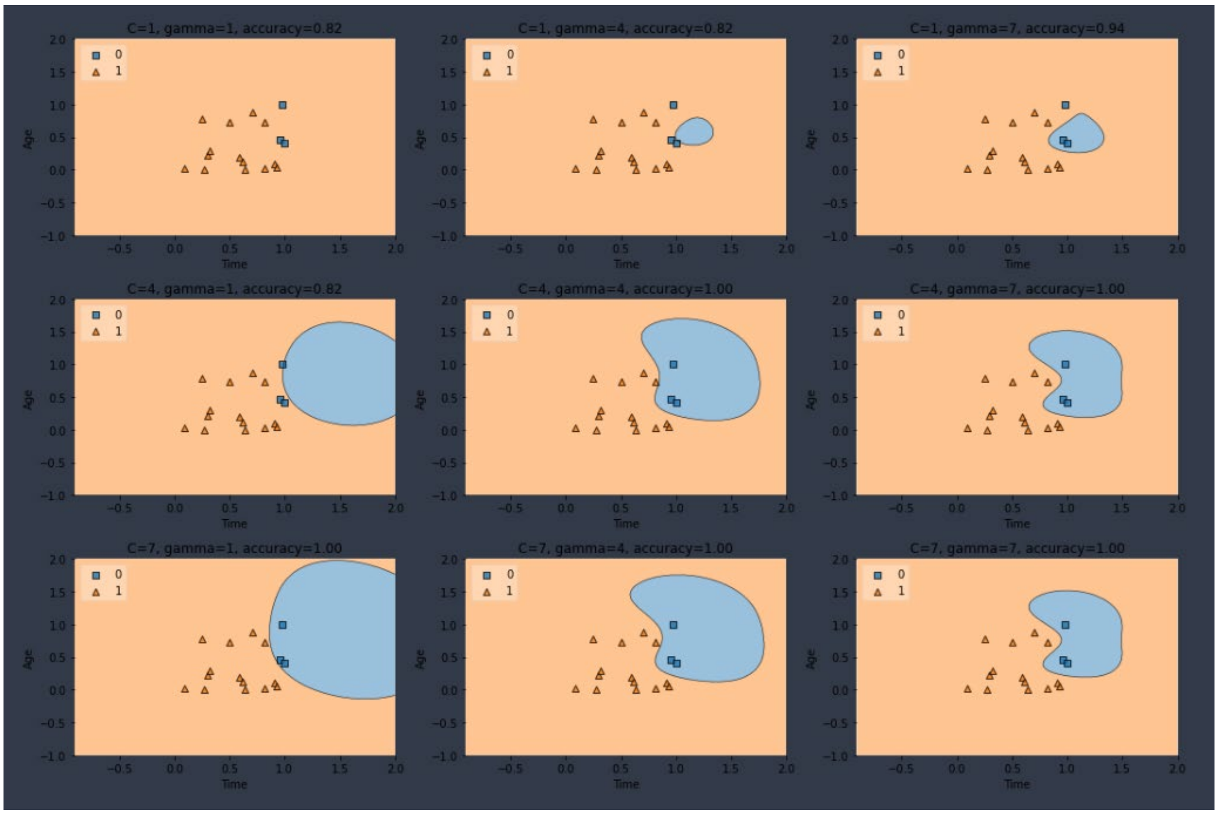


```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(df[feature_names],
df['Result'], test_size=0.2, random_state=0)

print ("Train set X shape = ", X_train.shape, "\nTrain set y shape = ", y_train.shape)
print ("Test set X shape  = ", X_test.shape, "\nTest set y shape  = ", y_test.shape)
```

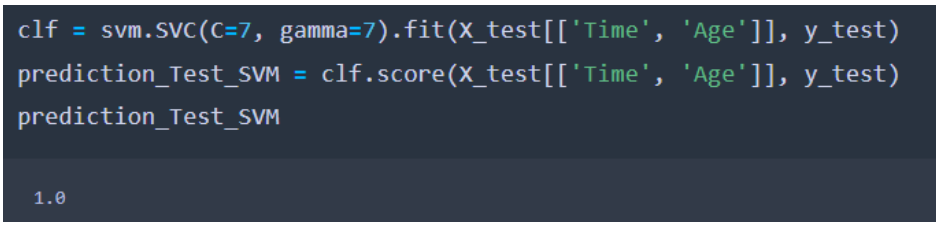
```
Train set X shape =  (67, 10)
Train set y shape =  (67,)
Test set X shape  =  (17, 10)
Test set y shape  =  (17,)
```

4.5.2 Graphical representation

We plot the following graph using the kernel 'rbf' and 'plot_decision_regions':



We found multiple combinations of C and gamma that resulted in 100% train prediction accuracy, we chose C=7 and gamma=7:

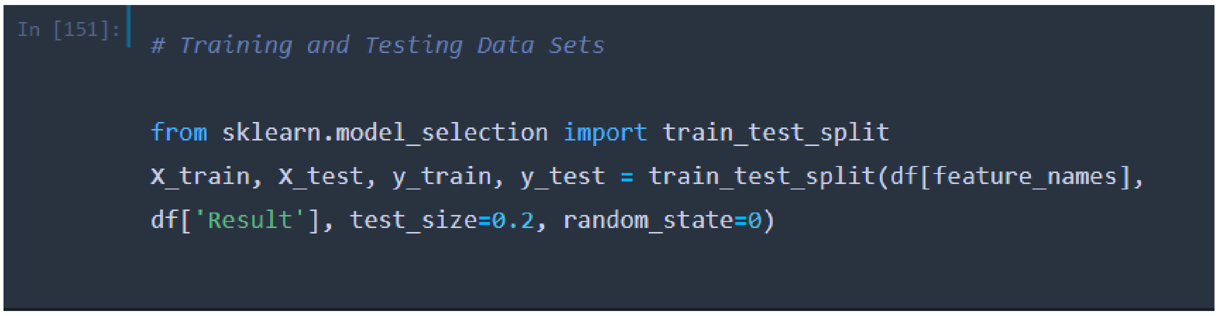


```
clf = svm.SVC(C=7, gamma=7).fit(X_test[['Time', 'Age']], y_test)
prediction_Test_SVM = clf.score(X_test[['Time', 'Age']], y_test)
prediction_Test_SVM
```

```
1.0
```

4.6 Decision Trees

4.6.1 Train and Test Datasets



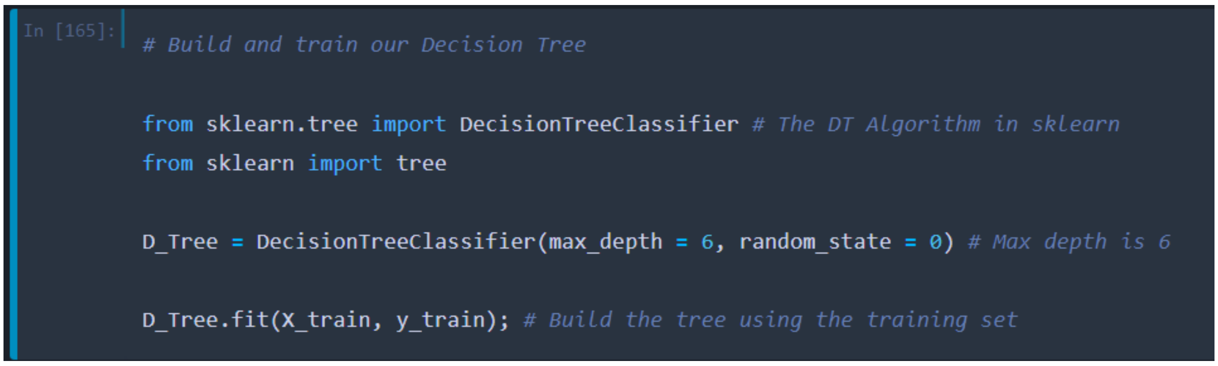
```
In [151]:  # Training and Testing Data Sets

           from sklearn.model_selection import train_test_split
           X_train, X_test, y_train, y_test = train_test_split(df[feature_names],
           df['Result'], test_size=0.2, random_state=0)
```

After trying many test-sizes, 0.2 gave the best test accuracy:

Test Size	Train Accuracy	Test Accuracy
0.1	98.67 %	88.89 %
0.15	98.59 %	92.31 %
0.2	100 %	94.12 %
0.25	100 %	80.95 %
0.3	98.28 %	92.31 %
0.35	98.15 %	93.33 %
0.4	100 %	91.18 %

4.6.2 Building the Decision Tree

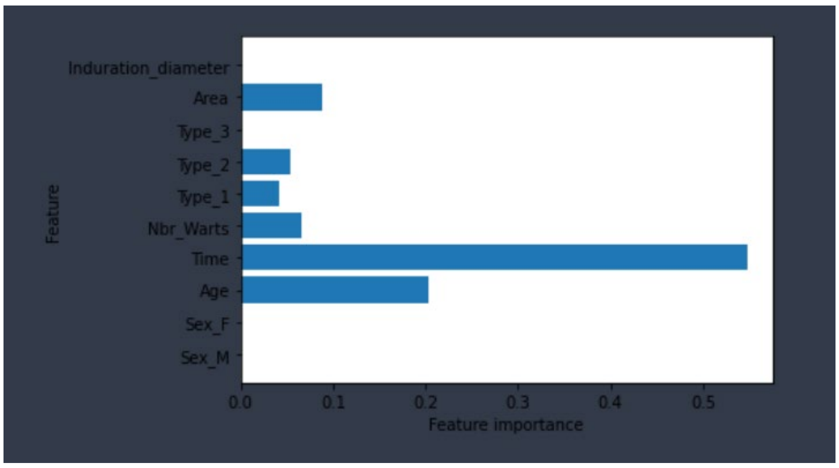
We found that the best max_depth is 6.



```
In [165]:  # Build and train our Decision Tree

           from sklearn.tree import DecisionTreeClassifier # The DT Algorithm in sklearn
           from sklearn import tree

           D_Tree = DecisionTreeClassifier(max_depth = 6, random_state = 0) # Max depth is 6

           D_Tree.fit(X_train, y_train); # Build the tree using the training set
```

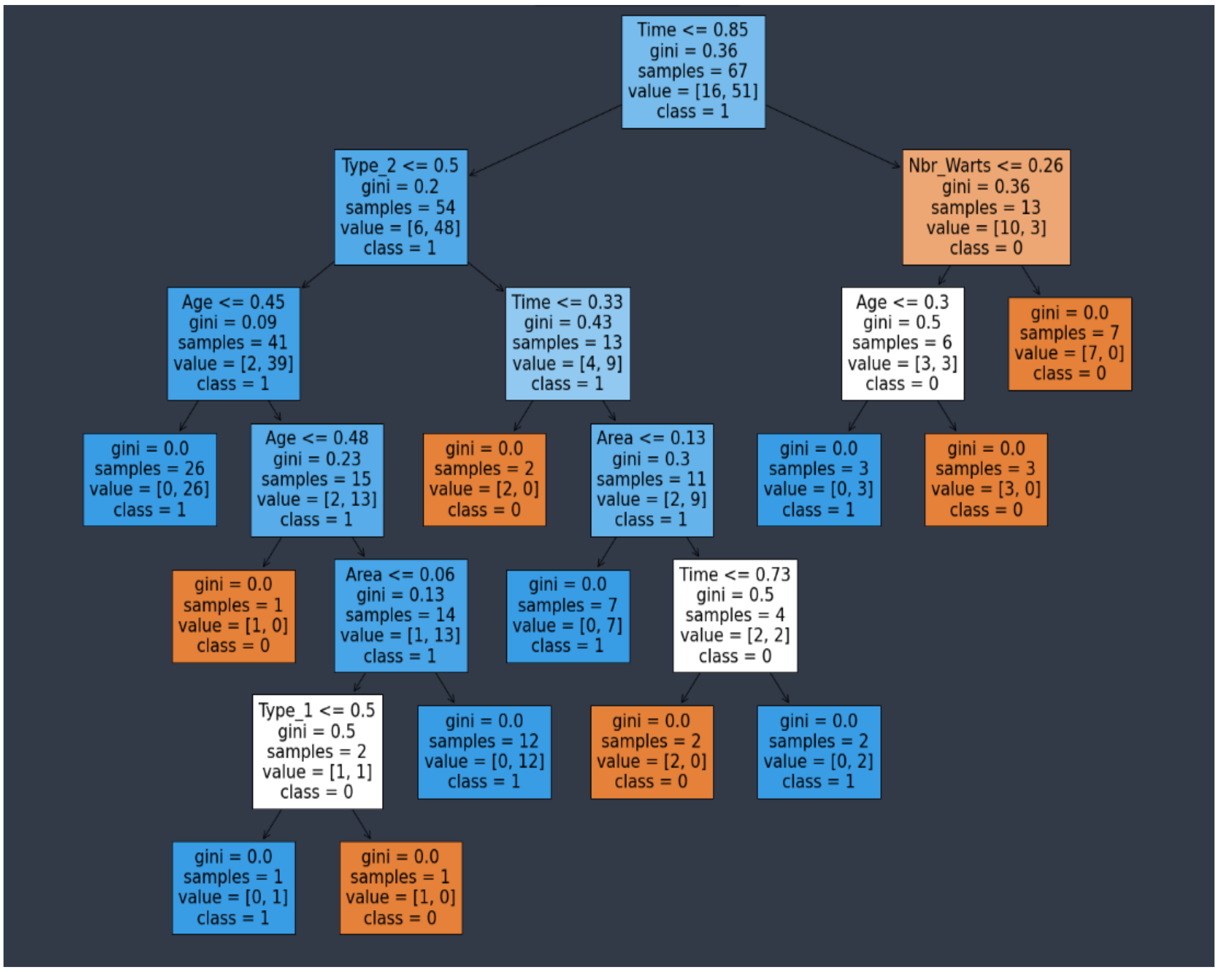
4.6.3 Feature importance in the Decision Tree

We used D_Tree.feature_importances and plt.barh() to create this graph :

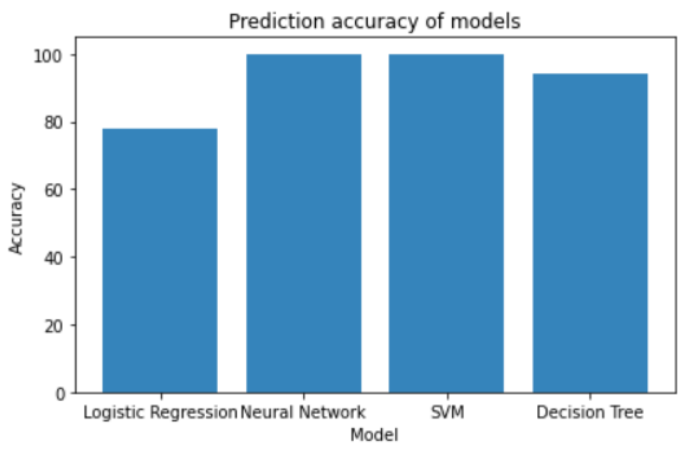


We notice that ‘Time’, ’Age’ and ‘Area’ are the most important features, while ‘Sex_F’, ‘Sex_M’ and ‘Type_3’ are not used.

4.6.4 Graphical representation


Time <= 0.85 gini = 0.36 samples = 67 value = [16, 51] class = 1
Type_2 <= 0.5 gini = 0.2 samples = 54 value = [6, 48] class = 1
Nbr_Warts <= 0.26 gini = 0.36 samples = 13 value = [10, 3] class = 0
Age <= 0.45 gini = 0.09 samples = 41 value = [2, 39] class = 1
Time <= 0.33 gini = 0.43 samples = 13 value = [4, 9] class = 1
Age <= 0.3 gini = 0.5 samples = 6 value = [3, 3] class = 0
gini = 0.0 samples = 7 value = [7, 0] class = 0
gini = 0.0 samples = 26 value = [0, 26] class = 1
Age <= 0.48 gini = 0.23 samples = 15 value = [2, 13] class = 1
gini = 0.0 samples = 2 value = [2, 0] class = 0
Area <= 0.13 gini = 0.3 samples = 11 value = [2, 9] class = 1
gini = 0.0 samples = 3 value = [0, 3] class = 1
gini = 0.0 samples = 3 value = [3, 0] class = 0
gini = 0.0 samples = 1 value = [1, 0] class = 0
Area <= 0.06 gini = 0.13 samples = 14 value = [1, 13] class = 1
gini = 0.0 samples = 7 value = [0, 7] class = 1
Time <= 0.73 gini = 0.5 samples = 4 value = [2, 2] class = 0
Type_1 <= 0.5 gini = 0.5 samples = 2 value = [1, 1] class = 0
gini = 0.0 samples = 12 value = [0, 12] class = 1
gini = 0.0 samples = 2 value = [2, 0] class = 0
gini = 0.0 samples = 2 value = [0, 2] class = 1
gini = 0.0 samples = 1 value = [0, 1] class = 1
gini = 0.0 samples = 1 value = [1, 0] class = 0

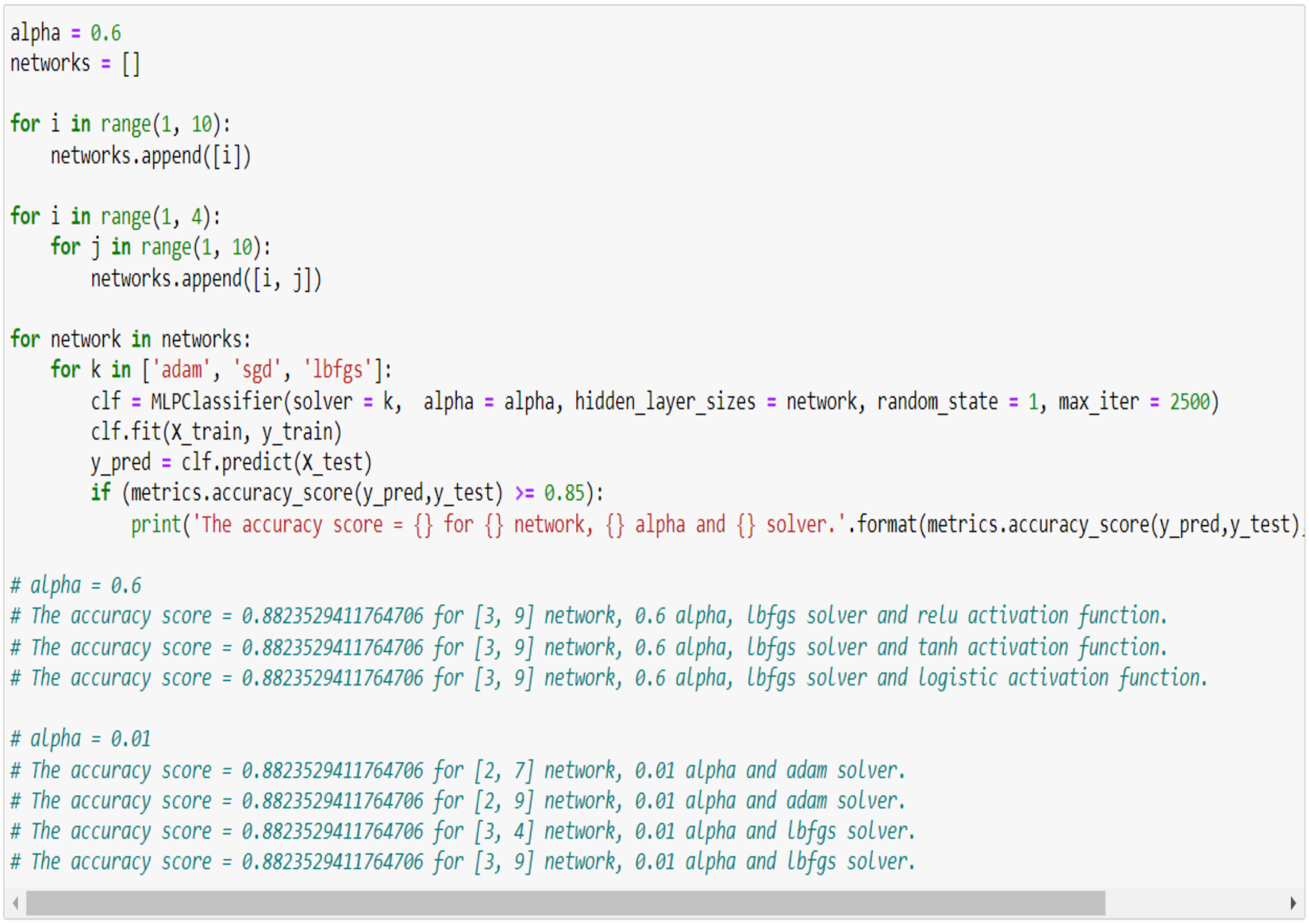
4.7 Comparing models



We conclude that SVM and Neural Network models gave the best test prediction accuracy (100%).

Appendix

A.1 Detailed set of loops example (Neural Network for Classification)



```
alpha = 0.6
networks = []

for i in range(1, 10):
    networks.append([i])

for i in range(1, 4):
    for j in range(1, 10):
        networks.append([i, j])

for network in networks:
    for k in ['adam', 'sgd', 'lbfgs']:
        clf = MLPClassifier(solver = k,  alpha = alpha, hidden_layer_sizes = network, random_state = 1, max_iter = 2500)
        clf.fit(X_train, y_train)
        y_pred = clf.predict(X_test)
        if (metrics.accuracy_score(y_pred,y_test) >= 0.85):
            print('The accuracy score = {} for {} network, {} alpha and {} solver.'.format(metrics.accuracy_score(y_pred,y_test)

# alpha = 0.6
# The accuracy score = 0.8823529411764706 for [3, 9] network, 0.6 alpha, lbfgs solver and relu activation function.
# The accuracy score = 0.8823529411764706 for [3, 9] network, 0.6 alpha, lbfgs solver and tanh activation function.
# The accuracy score = 0.8823529411764706 for [3, 9] network, 0.6 alpha, lbfgs solver and logistic activation function.

# alpha = 0.01
# The accuracy score = 0.8823529411764706 for [2, 7] network, 0.01 alpha and adam solver.
# The accuracy score = 0.8823529411764706 for [2, 9] network, 0.01 alpha and adam solver.
# The accuracy score = 0.8823529411764706 for [3, 4] network, 0.01 alpha and lbfgs solver.
# The accuracy score = 0.8823529411764706 for [3, 9] network, 0.01 alpha and lbfgs solver.
```

A.2 GridSearchCV implementation (Neural Network for Classification)

A.2.1 Initializing the algorithm

As mentioned in GridSearchCV part of the report (4.4.2. Tuning parameters for the model), we have to pass predefined values for parameters to the GridSearchCV function by defining a dictionary in which we mention a particular parameter along with the values it can take.

```
networks = []
for i in range(1, 2):
    for j in range(1, 5):
        networks.append([i, j])
parameter_space = {
    'hidden_layer_sizes': networks,
    'activation': ['relu'],
    'solver': ['lbfgs'],
    'alpha': np.linspace(0.001, 0.05, 50),
    'learning_rate': ['constant','adaptive'],
}
```

```
#Initializing the GridsearchCV

from sklearn.model_selection import GridSearchCV

mlp_grid = MLPClassifier(max_iter=(100), random_state = 1)

clf_grid = GridSearchCV(mlp_grid, parameter_space, n_jobs=-1, cv=5)
clf_grid.fit(X_train, y_train)

# parameter_space : Dictionary or list of parameters of models or function in which GridSearchCV have to select the best.
# cv : In this we have to pass a interger value, as it signifies the number of splits that is needed for cross validation.
# n_jobs : This signifies the number of jobs to be run in parallel, -1 signifies to use all processor.
```

A.2.2 The final results

```
# To get the best parameters it could find

print('Best parameters found:\n', clf_grid.best_params_)
```

```
Best parameters found:
 {'activation': 'relu', 'alpha': 0.021, 'hidden_layer_sizes': [1, 2], 'learning
_rate': 'constant', 'solver': 'lbfgs'}
```

```
# To test the accuracy of the best parameters

clf_grid = MLPClassifier(solver='lbfgs', activation = 'relu', alpha = 0.021, hid
clf_grid.fit(X_train, y_train)
y_pred_grid = clf_grid.predict(X_test)

metrics.accuracy_score(y_pred_grid,y_test)
```

```
0.8235294117647058
```

```
print('Accuracy results :')
clf_grid = MLPClassifier(solver='lbfgs',  alpha = 0.021, hidden_layer_sizes=(1,
clf_grid.fit(X_train, y_train)
y_pred_grid = clf_grid.predict(X_test)
y_train_pred_grid = clf_grid.predict(X_train)
print('Train set accuracy: '+'{}'.format(metrics.accuracy_score(y_train, y_train
print('Test set accuracy: '+'{}'.format(metrics.accuracy_score(y_test, y_pred_gr
```

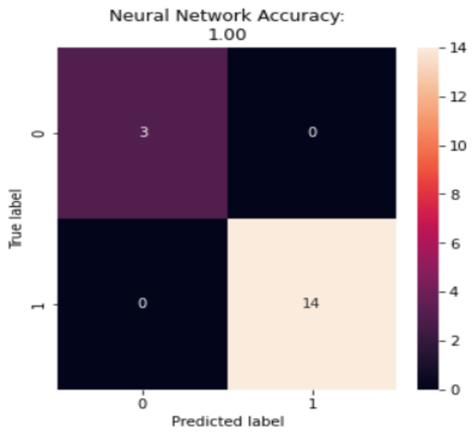
```
Accuracy results :
Train set accuracy: 0.7611940298507462
Test set accuracy: 0.8235294117647058
```

Finally, we notice that using GridSearchCV to find the best parameters for the classifier is not the optimal way to do it, doing it with for loops seems to be better, in our case at least.

This limitation is due to the fact that the search can only test the parameters that we fed into, and giving a large set of values in each parameter can produce a really high time-consuming search.

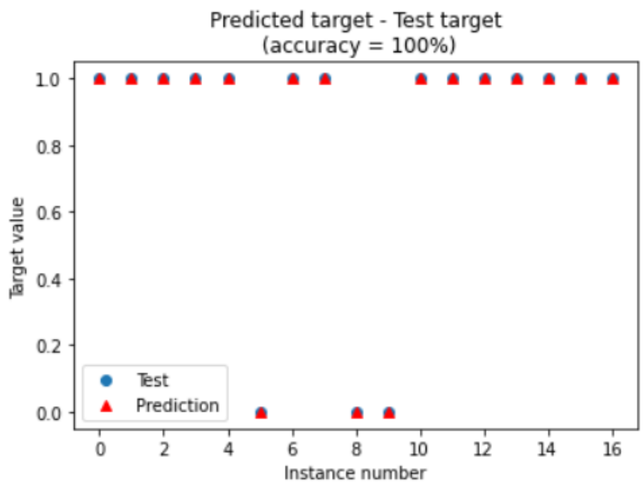
A.3 Confusion matrix and results plotting code (Neural Network for Classification)

```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

confusion_mc = confusion_matrix(y_test, y_pred)

df_cm = pd.DataFrame(confusion_mc, index = [i for i in range(0,2)], columns = [i for i in range(0,2)])

plt.figure(figsize=(5, 5))

sns.heatmap(df_cm, annot=True)

plt.title('Neural Network Accuracy:\n{0:.2f}'.format(accuracy_score(y_test, y_pred)))
plt.ylabel('True label')
plt.xlabel('Predicted label');
```



```
# Plotting the predicted targets and the test targets

plt.scatter(range(0, 17), y_test)
plt.scatter(range(0, 17), y_pred, marker = "^", color = "red")
plt.legend(["Test", "Prediction"])
plt.xlabel("Instance number")
plt.ylabel("Target value")
plt.title("Predicted target - Test target\n (accuracy = 100%)")
```

```
Text(0.5, 1.0, 'Predicted target - Test target\n (accuracy = 10
0%)')
```

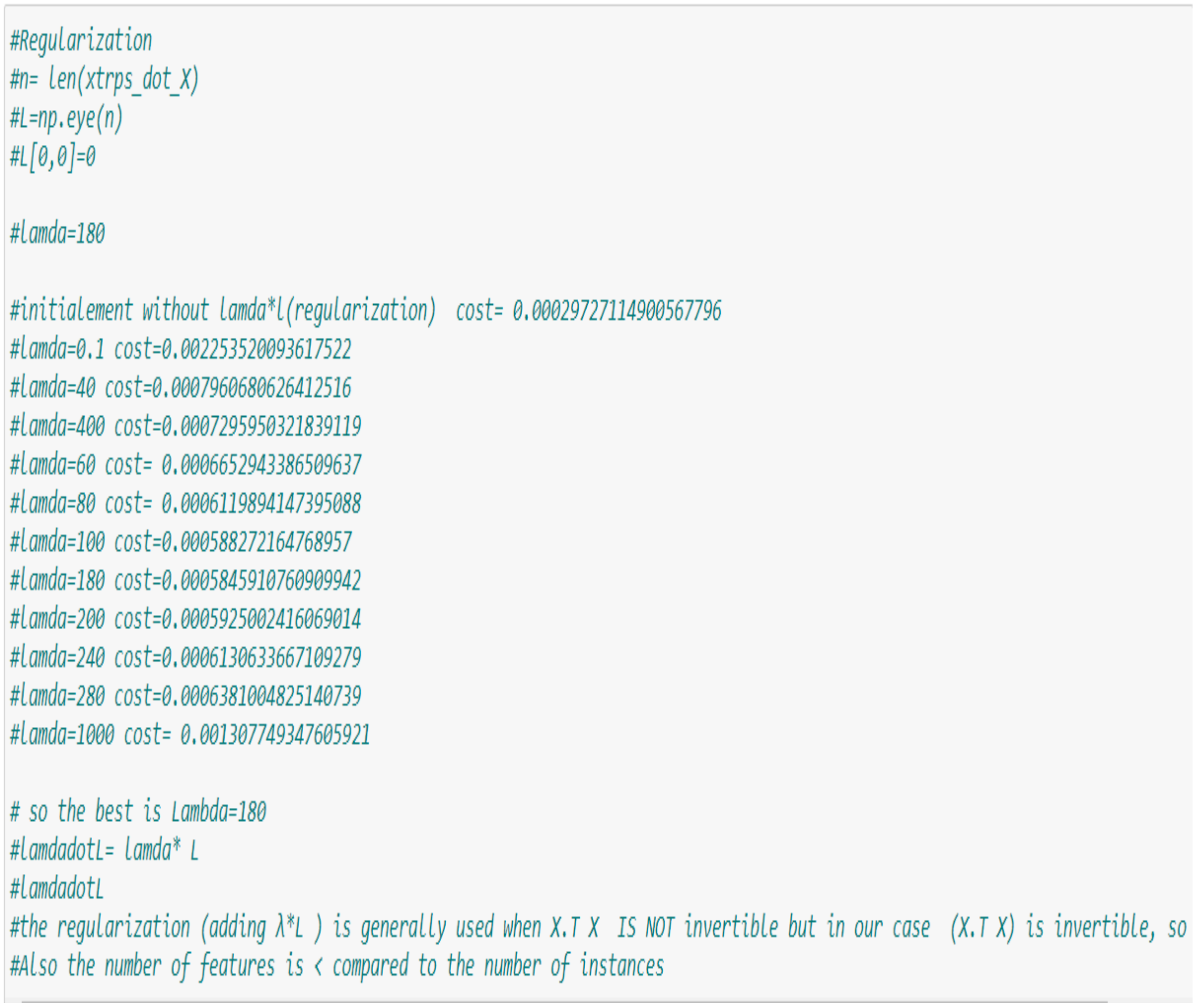


B.1 Regularization in Normal Equation

In order to apply regularization in Normal equation we just need to add λ*L in our formula like this:

reverse(X.T X + λ*L)

Now we test many values of λ and evaluate the cost



```
#Regularization
#n= len(xtrps_dot_X)
#L=np.eye(n)
#L[0,0]=0

#lamda=180

#initialement without lamda*l(regularization)  cost= 0.00029727114900567796
#lamda=0.1 cost=0.002253520093617522
#lamda=40 cost=0.0007960680626412516
#lamda=400 cost=0.0007295950321839119
#lamda=60 cost= 0.0006652943386509637
#lamda=80 cost= 0.0006119894147395088
#lamda=100 cost=0.000588272164768957
#lamda=180 cost=0.0005845910760909942
#lamda=200 cost=0.0005925002416069014
#lamda=240 cost=0.0006130633667109279
#lamda=280 cost=0.0006381004825140739
#lamda=1000 cost= 0.001307749347605921

# so the best is Lambda=180
#lamdadotL= lamda* L
#lamdadotL
#the regularization (adding λ*L ) is generally used when X.T X  IS NOT invertible but in our case  (X.T X) is invertible, so
#Also the number of features is < compared to the number of instances
```

After that, we replace the formula: **temp_1=np.linalg.inv(xtrps_dot_X) # inverse(X.T X)**

By: temp_1=np.linalg.inv(xtrps_dot_X+ lamdadotL) # inverse(X.T X + λ*L)

And leave the rest of the code as it is.

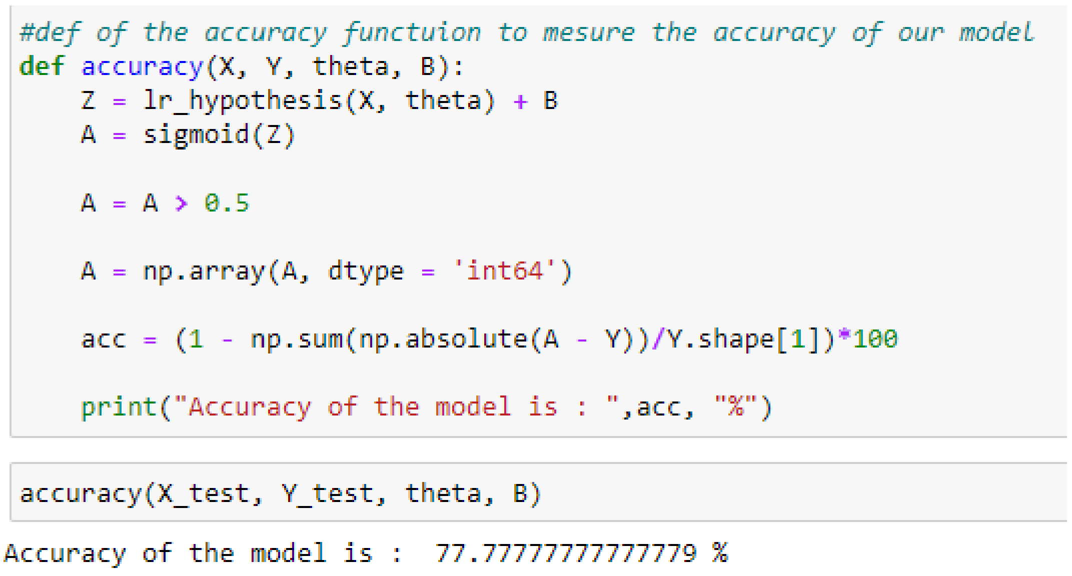
Remark: we prefer put regularization as a comment is our code for many reasons:

1- The regularization in normal equation is to add lambda*L in the formula X.T*X in order to apply the reverse, however, in our case the formula inv(X.T*X) without adding λ*L is possible .

2- The number of features n is < to the number of instances

C.1 Accuracy function

Here is the function we defined to get the accuracy of our model at predicting the test set after training it with the train set.



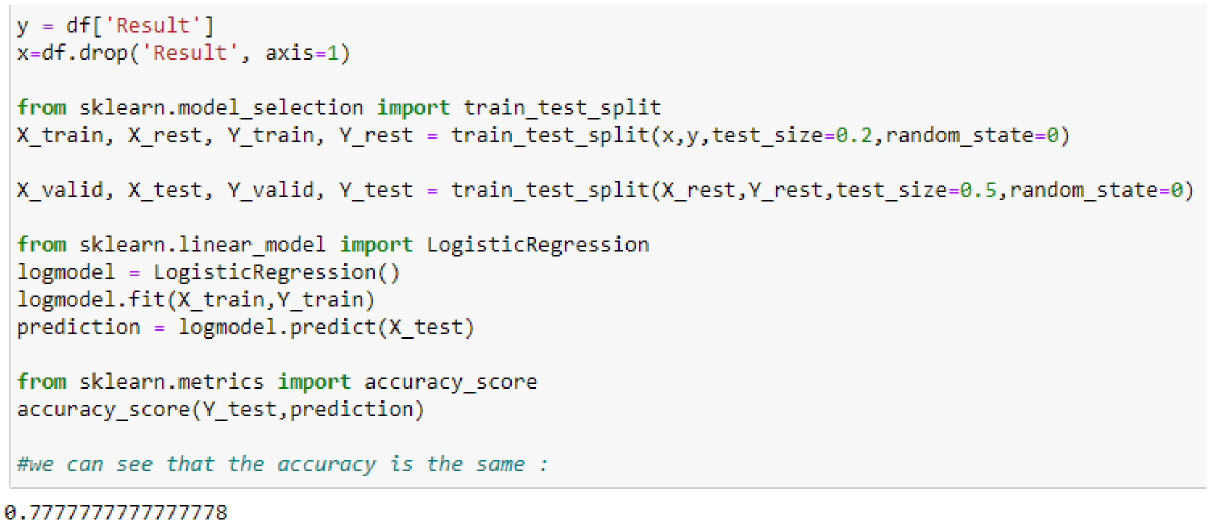
```
#def of the accuracy functuion to mesure the accuracy of our model
def accuracy(X, Y, theta, B):
    Z = lr_hypothesis(X, theta) + B
    A = sigmoid(Z)

    A = A > 0.5

    A = np.array(A, dtype = 'int64')

    acc = (1 - np.sum(np.absolute(A - Y))/Y.shape[1])*100

    print("Accuracy of the model is : ",acc, "%")
```

```
accuracy(X_test, Y_test, theta, B)
```

```
Accuracy of the model is :  77.77777777777779 %
```

C.2 Accuracy test with predefined function

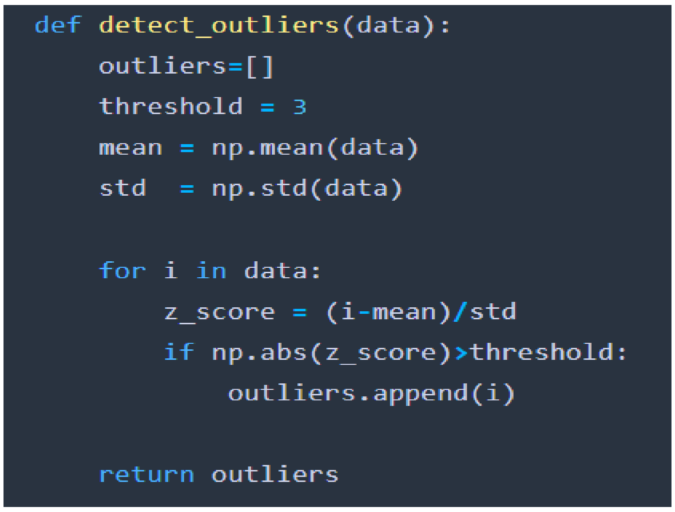
After finishing our train and test with the functions we defined, we use already existing functions to compare the results.



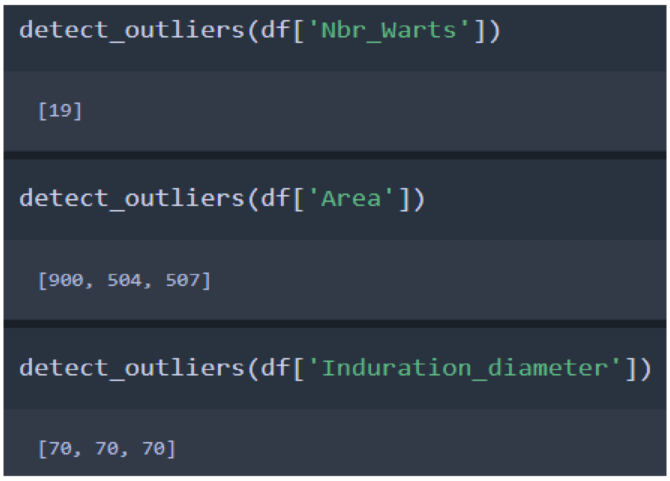
```
y = df['Result']
x=df.drop('Result', axis=1)

from sklearn.model_selection import train_test_split
X_train, X_rest, Y_train, Y_rest = train_test_split(x,y,test_size=0.2,random_state=0)

X_valid, X_test, Y_valid, Y_test = train_test_split(X_rest,Y_rest,test_size=0.5,random_state=0)

from sklearn.linear_model import LogisticRegression
logmodel = LogisticRegression()
logmodel.fit(X_train,Y_train)
prediction = logmodel.predict(X_test)

from sklearn.metrics import accuracy_score
accuracy_score(Y_test,prediction)

#we can see that the accuracy is the same :
```

```
0.7777777777777778
```

D.1 Detecting Outliers using ‘z_score’ method



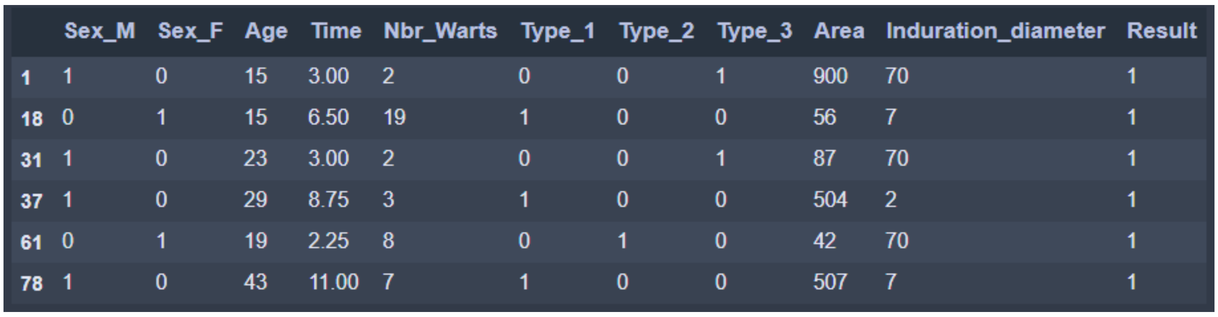
```
def detect_outliers(data):
    outliers=[]
    threshold = 3
    mean = np.mean(data)
    std  = np.std(data)

    for i in data:
        z_score = (i-mean)/std
        if np.abs(z_score)>threshold:
            outliers.append(i)

    return outliers
```



```
detect_outliers(df['Nbr_Warts'])

[19]

detect_outliers(df['Area'])

[900, 504, 507]

detect_outliers(df['Induration_diameter'])

[70, 70, 70]
```

Remark: we checked all the other features and they don’t contain any outliers.

Table of all the outliers of ‘Nbr_Warts’, ‘Area’ and ‘Induration_diameter’:



	Sex_M	Sex_F	Age	Time	Nbr_Warts	Type_1	Type_2	Type_3	Area	Induration_diameter	Result
1	1	0	15	3.00	2	0	0	1	900	70	1
18	0	1	15	6.50	19	1	0	0	56	7	1
31	1	0	23	3.00	2	0	0	1	87	70	1
37	1	0	29	8.75	3	1	0	0	504	2	1
61	0	1	19	2.25	8	0	1	0	42	70	1
78	1	0	43	11.00	7	1	0	0	507	7	1

E.1. Linear Regression Gradient Descent

1- Import libraries:

```
Entrée [119]: import pandas as pd
              import numpy as np
              import matplotlib.pyplot as plt
              from sklearn.linear_model import LinearRegression
              from sklearn.model_selection import train_test_split
              from sklearn.metrics import mean_squared_error, r2_score
```

2- Find the predict Y to our X train

Firstly, we add the column X0 to our dataset

```
Entrée [178]:
              Data1= pd.DataFrame(df2)

              #print(Data1)

              X0= np.ones((408,1)) # prepare the column X0 which contains only 1 in order to  add it to use it after
              #print(X0)

              X11= np.append(X0, Data1 , axis=1)
              #print(X11)

              data2=pd.DataFrame(X11, columns=['X0','Transaction_Date','House_Age','Distance','Num_Stores_NearBy','Latitude','Long
              #print(data2)
```

```
Entrée [180]: Y = data2['Target']
              print(Y)

         0      0.275705
         1      0.314832
         2      0.361237
         3      0.429481
         4      0.323021
                  ...
         403    0.070974
         404    0.385805
         405    0.300273
         406    0.408553
         407    0.512284
         Name: Target, Length: 408, dtype: float64
```

Next, we split our Dataset into train data and test data and validation data which 60% of the data will be used for training the model while 20% will be used for testing the model and 20% for the validation of our model.

```
Entrée [112]: #Spliting our Data2 into train data and test data and validation data
              #60% of the data will be used for training the model while 20% will be used for testing the model and 20% for the va


              data_train =data2.sample(frac=0.6, random_state = 1)
              data_test = data2.drop(data_train.index)
              data_train.reset_index(drop=True,inplace=True)
              data_test.reset_index(drop=True,inplace=True)
              data_test1 = data_test.sample(frac=0.5, random_state = 1)
              data_valid = data_test.drop(data_test1.index)
              data_test1.reset_index(drop=True,inplace=True)
              data_valid.reset_index(drop=True,inplace=True)
```

```
Entrée [140]: print(data_train.shape)
              print(data_test1.shape)
              print(data_valid.shape)

              #here we will notice that our data test and data valid are not equal because our data contain 408 rows and 8 attribu
              #So when we split the data we will have :
              # 60% data training : 245 rows and 8 attributes
              #20% data testing : 82 rows and 8 attributes
              #And the rest 20% data validation : 81 rows and 8 attributes

         (245, 8)
         (82, 8)
         (81, 8)
```

Now, after splitting our dataset, we define X train, Y train, X test, Y test, X valid, Y valid :

```
Entrée [114]: X_train = data_train.drop(['Target'],axis=1)#removing 'Target' from X_train
              Y_train = data_train[['Target']]
              #print(X_train)
              #print(Y_train)
```

```
Entrée [115]: X_test1 = data_test1.drop(['Target'],axis=1)#removing 'Target' from X_test1
              Y_test1 = data_test1[['Target']]
              #print(X_test1_X)
              #print(Y_test1_Y)
```

```
Entrée [141]: X_valid = data_valid.drop(['Target'],axis=1)#removing 'Target' from X_valid
              Y_valid = data_valid[['Target']]
              #print(X_valid)
              #print(Y_valid)
```

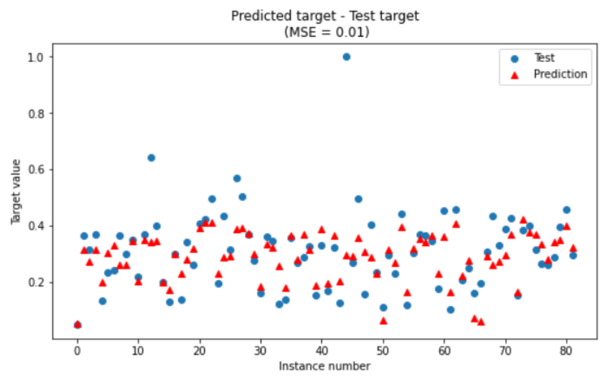
Next, we will predict Y to our X train:

```
Entrée [118]: reg = LinearRegression()
              reg.fit(X_train[['X0','Transaction_Date','House_Age','Distance','Num_Stores_NearBy','Latitude','Longitude']], Y_trai
              y_predicted = reg.predict(X_test1[['X0','Transaction_Date','House_Age','Distance','Num_Stores_NearBy','Latitude','Lo
              print("Mean squared error: %.2f" % mean_squared_error(Y_test1, y_predicted))#calculate MSE
              print('R²: %.2f' % r2_score(Y_test1, y_predicted))#calculate R²

          Mean squared error: 0.01
          R²: 0.42
```

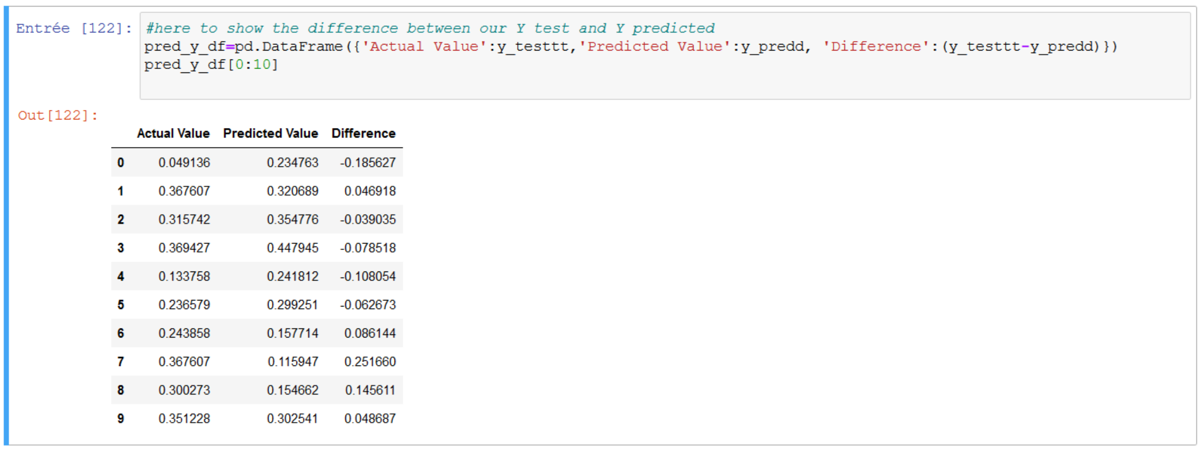
```
Entrée [182]: y_pred=ML.predict(X_test1)
              #print(y_pred)
              print(y_pred.shape)

          (82, 1)
```

Now we plot the predicted Y and our Y test:

```
Entrée [183]: #Plotting the predicted Y and Y test
              plt.figure(figsize=(9, 5))
              plt.scatter(range(0, 82), Y_test1)
              plt.scatter(range(0, 82), y_predicted, marker = "^", color = "red")
              plt.legend(["Test", "Prediction"])
              plt.xlabel("Instance number")
              plt.ylabel("Target value")
              plt.title("Predicted target - Test target\n (MSE = 0.01)")

Out[183]: Text(0.5, 1.0, 'Predicted target - Test target\n (MSE = 0.01)')
```



Now we compare the values of both:



```
Entrée [122]: #here to show the difference between our Y test and Y predicted
              pred_y_df=pd.DataFrame({'Actual Value':y_testtt,'Predicted Value':y_predd, 'Difference':(y_testtt-y_predd)})
              pred_y_df[0:10]

Out[122]:
```

```
   Actual Value  Predicted Value  Difference
0      0.049136         0.234763   -0.185627
1      0.367607         0.320689    0.046918
2      0.315742         0.354776   -0.039035
3      0.369427         0.447945   -0.078518
4      0.133758         0.241812   -0.108054
5      0.236579         0.299251   -0.062673
6      0.243858         0.157714    0.086144
7      0.367607         0.115947    0.251660
8      0.300273         0.154662    0.145611
9      0.351228         0.302541    0.048687
```

We notice that there is a small difference between Y predicted and our Y test (good).

References

Preprocessing:

https://datascientest.com/data-cleaning

Logistic Regression:

https://www.youtube.com/watch?v=nzNp05AyBM8&ab_channel=CodingLane

Neural Network for Classification - GridSerachCV:

https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

Pre-Modeling: Data Preprocessing and Feature Exploration in Python:

https://www.youtube.com/watch?v=V0u6bxQOUJ8

Machine Learning Tutorial Python - 10 Support Vector Machine (SVM):

https://www.youtube.com/watch?v=FB5EdxAGxQg

Linear Regression Gradient Descent:

https://learn.g2.com/linear-regression

https://ruder.io/optimizing-gradient-descent/

https://www.youtube.com/watch?v=xrPZbHrxrWo

Normal Equation:

https://www.youtube.com/watch?v=DQ6xfe75CDk