Data Mining Project

By the supervision of Dr Abeer Amer

Names:

Donia Ahmed Hassan El Said 20201444709

Mayar Kamel Samir Othman 20201378683

Mennatullah Magdy Mahmoud Mostafa 20201378920

Nada Hossameldin Abdelhalim Abdelaziz 20201312674

Sally Muhammed Ahmed Ghozal 20201378673

Sarah Mohammed Mahmoud Mohammed 20201312604

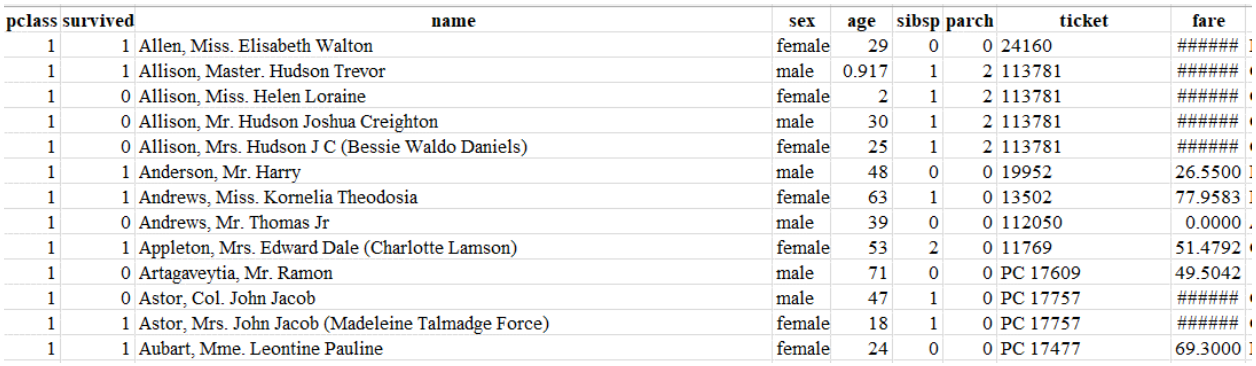
Yara Hassan Mohsen Abd ElHakam 20201447189

Table of Content:

Introduction: Dataset and Objectives

For the following python project, we have used the dataset for the Titanic Ship passengers, more specifically their information and tickets.

Here is a prompt of our data shown below:

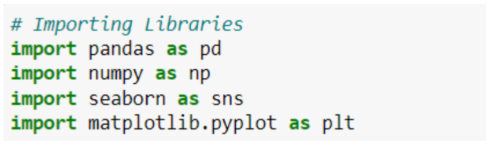


pclass	survived	name	sex	age	sibsp	parch	ticket	fare
1	1	Allen, Miss. Elisabeth Walton	female	29	0	0	24160	######
1	1	Allison, Master. Hudson Trevor	male	0.917	1	2	113781	######
1	0	Allison, Miss. Helen Loraine	female	2	1	2	113781	######
1	0	Allison, Mr. Hudson Joshua Creighton	male	30	1	2	113781	######
1	0	Allison, Mrs. Hudson J C (Bessie Waldo Daniels)	female	25	1	2	113781	######
1	1	Anderson, Mr. Harry	male	48	0	0	19952	26.5500
1	1	Andrews, Miss. Kornelia Theodosia	female	63	1	0	13502	77.9583
1	0	Andrews, Mr. Thomas Jr	male	39	0	0	112050	0.0000
1	1	Appleton, Mrs. Edward Dale (Charlotte Lamson)	female	53	2	0	11769	51.4792
1	0	Artagaveytia, Mr. Ramon	male	71	0	0	PC 17609	49.5042
1	0	Astor, Col. John Jacob	male	47	1	0	PC 17757	######
1	1	Astor, Mrs. John Jacob (Madeleine Talmadge Force)	female	18	1	0	PC 17757	######
1	1	Aubart, Mme. Leontine Pauline	female	24	0	0	PC 17477	69.3000

However, our data seemed to require a bit of cleaning, more specifically removal of duplicates, correlated attributes or irrelevant ones as well as handling missing data.

<u>Part 1 Explanation</u>

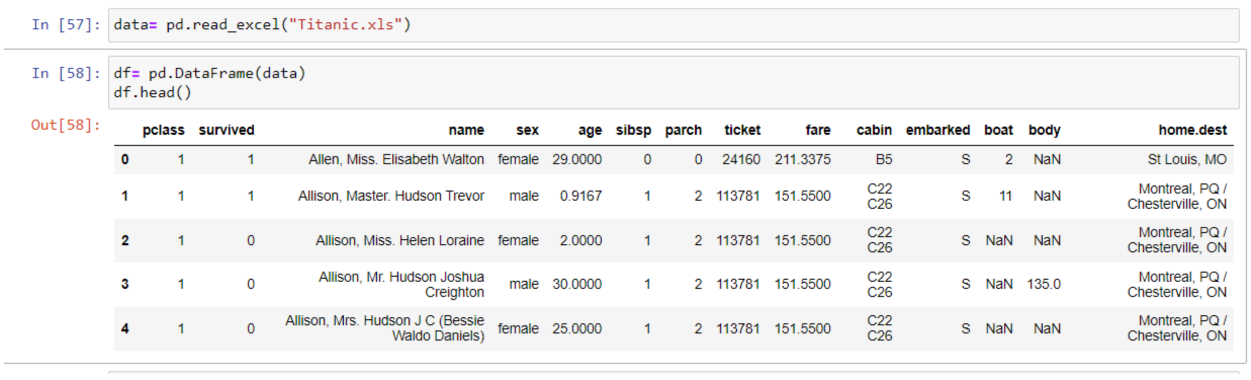
Thankfully python comes with many built in libraries to ease that process for us. To use the libraries, we started by importing them.



```python
# Importing Libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

- Pandas, which is used to read, process, and write our CSV data files.
- NumPy, which helps us with our mathematical operations and arrays.
- Seaborn and Matplotlib, which visualize our data.

We start off our code by loading in our dataset, or the csv file using pandas then printing it into the code using **df.head()** as shown below



```
In [57]: data= pd.read_excel("Titanic.xls")
```

```
In [58]: df= pd.DataFrame(data)
         df.head()
```

Out[58]:

	pclass	survived	name	sex	age	sibsp	parch	ticket	fare	cabin	embarked	boat	body	home.dest
0	1	1	Allen, Miss. Elisabeth Walton	female	29.0000	0	0	24160	211.3375	B5	S	2	NaN	St Louis, MO
1	1	1	Allison, Master. Hudson Trevor	male	0.9167	1	2	113781	151.5500	C22 C26	S	11	NaN	Montreal, PQ / Chesterville, ON
2	1	0	Allison, Miss. Helen Loraine	female	2.0000	1	2	113781	151.5500	C22 C26	S	NaN	NaN	Montreal, PQ / Chesterville, ON
3	1	0	Allison, Mr. Hudson Joshua Creighton	male	30.0000	1	2	113781	151.5500	C22 C26	S	NaN	135.0	Montreal, PQ / Chesterville, ON
4	1	0	Allison, Mrs. Hudson J C (Bessie Waldo Daniels)	female	25.0000	1	2	113781	151.5500	C22 C26	S	NaN	NaN	Montreal, PQ / Chesterville, ON

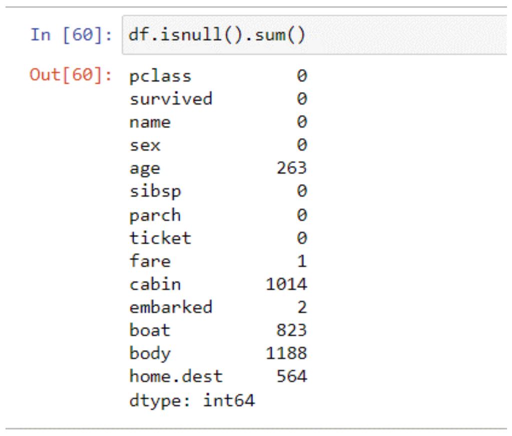
Each column represents different attributes, with the pclass being a proxy for socio-economic status (SES), 1 equalling the upper class, 2 for middle, and 3 for the lower class. The sibsp represents family relations in this way:

- Sibling = brother, sister, stepbrother, stepsister
- Spouse = husband, wife (mistresses and fiancés were ignored)

There is also the parch which defines the family relations as following:

- Parent = mother, father
- Child = daughter, son, stepdaughter, stepson
- Some children travelled only with a nanny, therefore parch=0 for them.

We then check the null values in each of our columns using by using **sum()** to returns the number of missing values in the data set.

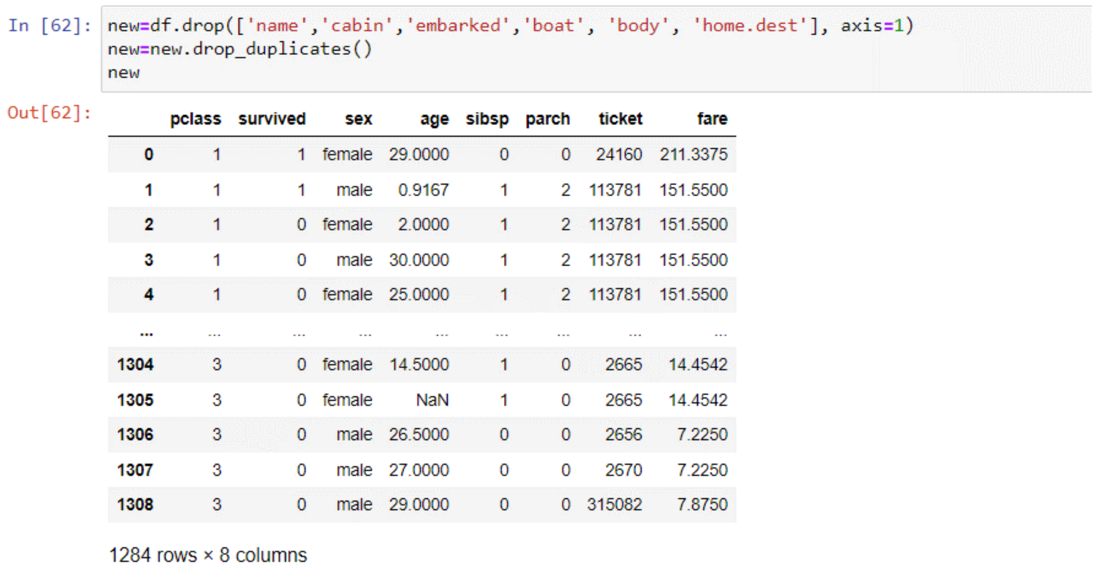


```
In [60]: df.isnull().sum()

Out[60]: pclass          0
         survived        0
         name            0
         sex             0
         age           263
         sibsp           0
         parch           0
         ticket          0
         fare            1
         cabin        1014
         embarked        2
         boat          823
         body         1188
         home.dest     564
         dtype: int64
```

.

Columns that have null values are 'age', 'fare', 'cabin', 'embarked', 'boat', 'body', and 'home.dest' . Now we go on to removing the irrelevant attributes in our dataset, those we do not seem important information wise. For this part we used the **drop()** function. We also use the same function to remove any duplicates we have to make our data cleaner and avoid redundancy.

As shown below these are our columns now:



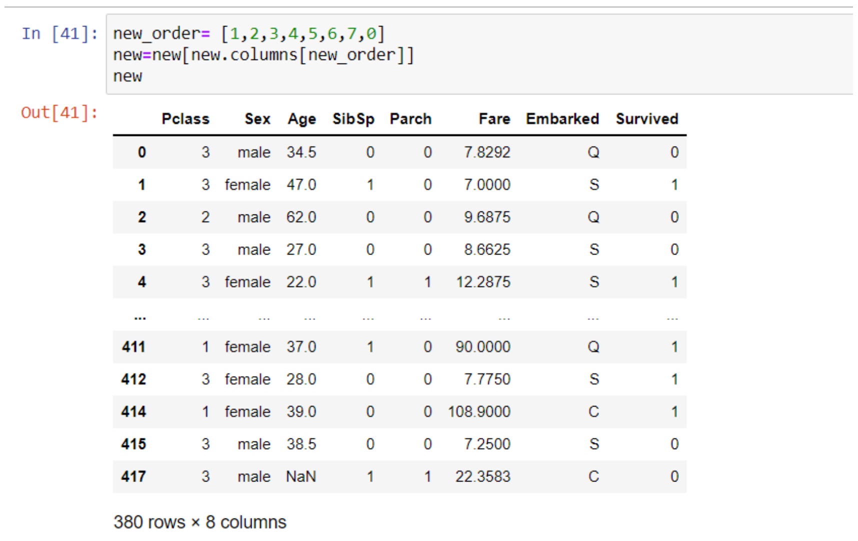
```
In [62]: new=df.drop(['name','cabin','embarked','boat', 'body', 'home.dest'], axis=1)
         new=new.drop_duplicates()
         new
```

Out[62]:

	pclass	survived	sex	age	sibsp	parch	ticket	fare
0	1	1	female	29.0000	0	0	24160	211.3375
1	1	1	male	0.9167	1	2	113781	151.5500
2	1	0	female	2.0000	1	2	113781	151.5500
3	1	0	male	30.0000	1	2	113781	151.5500
4	1	0	female	25.0000	1	2	113781	151.5500
...	...	...	...	...	...	...	...	...
1304	3	0	female	14.5000	1	0	2665	14.4542
1305	3	0	female	NaN	1	0	2665	14.4542
1306	3	0	male	26.5000	0	0	2656	7.2250
1307	3	0	male	27.0000	0	0	2670	7.2250
1308	3	0	male	29.0000	0	0	315082	7.8750

1284 rows × 8 columns

In the next part we wanted to re-order our column in order of relevancy, so we put the column of survived at the end to avoid the dropping of this column. By convention this class will be the last to show the attributes of the rest. For this part we simply reorder it to have our column at the end as shown below.



```
In [41]: new_order= [1,2,3,4,5,6,7,0]
         new=new[new.columns[new_order]]
         new
```

Out[41]:

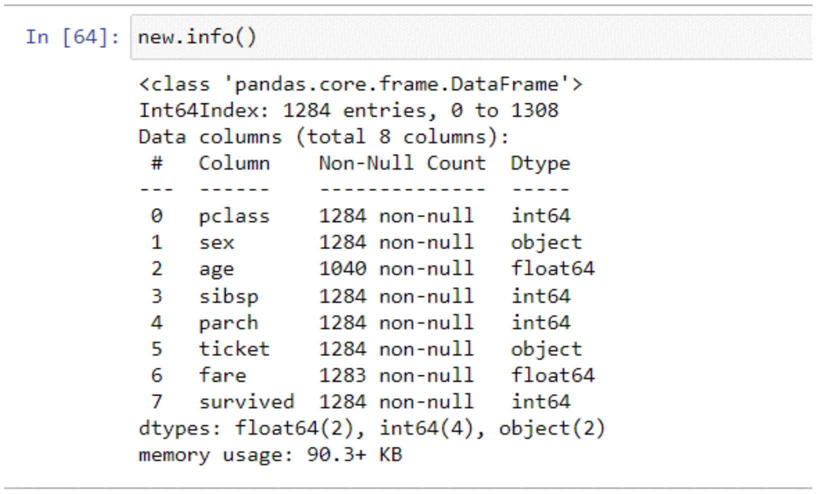
	Pclass	Sex	Age	SibSp	Parch	Fare	Embarked	Survived
0	3	male	34.5	0	0	7.8292	Q	0
1	3	female	47.0	1	0	7.0000	S	1
2	2	male	62.0	0	0	9.6875	Q	0
3	3	male	27.0	0	0	8.6625	S	0
4	3	female	22.0	1	1	12.2875	S	1
...	...	...	...	...	...	...	...	...
411	1	female	37.0	1	0	90.0000	Q	1
412	3	female	28.0	0	0	7.7750	S	1
414	1	female	39.0	0	0	108.9000	C	1
415	3	male	38.5	0	0	7.2500	S	0
417	3	male	NaN	1	1	22.3583	C	0

380 rows × 8 columns

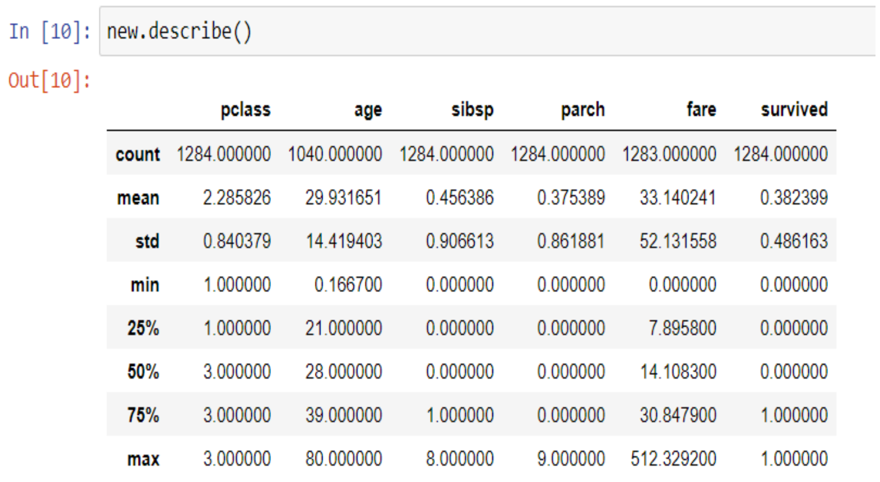
To get a quick analysis of our data frame so far, we use an **info()** function to check our data.

Then a **describe()** function to get a statistical summary of them.

As shown below these are the results it gives us:



```
In [64]: new.info()

<class 'pandas.core.frame.DataFrame'>
Int64Index: 1284 entries, 0 to 1308
Data columns (total 8 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   pclass    1284 non-null   int64
 1   sex       1284 non-null   object
 2   age       1040 non-null   float64
 3   sibsp     1284 non-null   int64
 4   parch     1284 non-null   int64
 5   ticket    1284 non-null   object
 6   fare      1283 non-null   float64
 7   survived  1284 non-null   int64
dtypes: float64(2), int64(4), object(2)
memory usage: 90.3+ KB
```

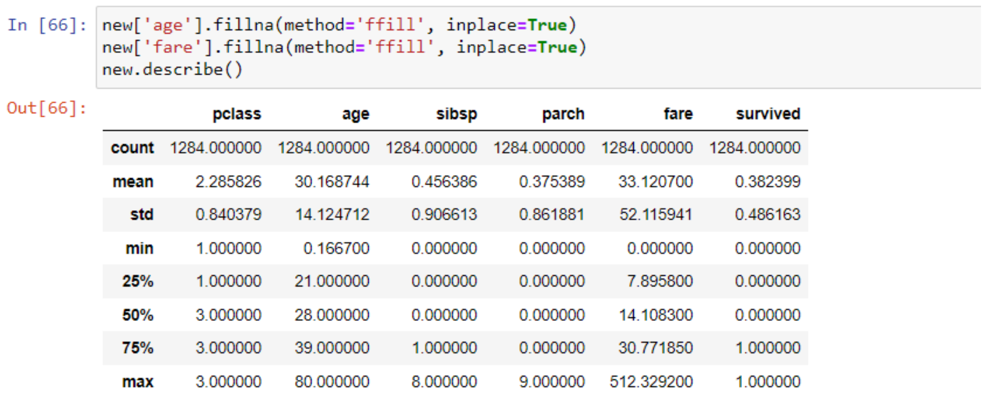


```
In [10]: new.describe()
```

Out[10]:

	pclass	age	sibsp	parch	fare	survived
count	1284.000000	1040.000000	1284.000000	1284.000000	1283.000000	1284.000000
mean	2.285826	29.931651	0.456386	0.375389	33.140241	0.382399
std	0.840379	14.419403	0.906613	0.861881	52.131558	0.486163
min	1.000000	0.166700	0.000000	0.000000	0.000000	0.000000
25%	1.000000	21.000000	0.000000	0.000000	7.895800	0.000000
50%	3.000000	28.000000	0.000000	0.000000	14.108300	0.000000
75%	3.000000	39.000000	1.000000	0.000000	30.847900	1.000000
max	3.000000	80.000000	8.000000	9.000000	512.329200	1.000000

For finding null values, we only did a check. However, we were yet to fill them. To do that we used forward fill method which takes the preceding non null value and replaces the null value with it.

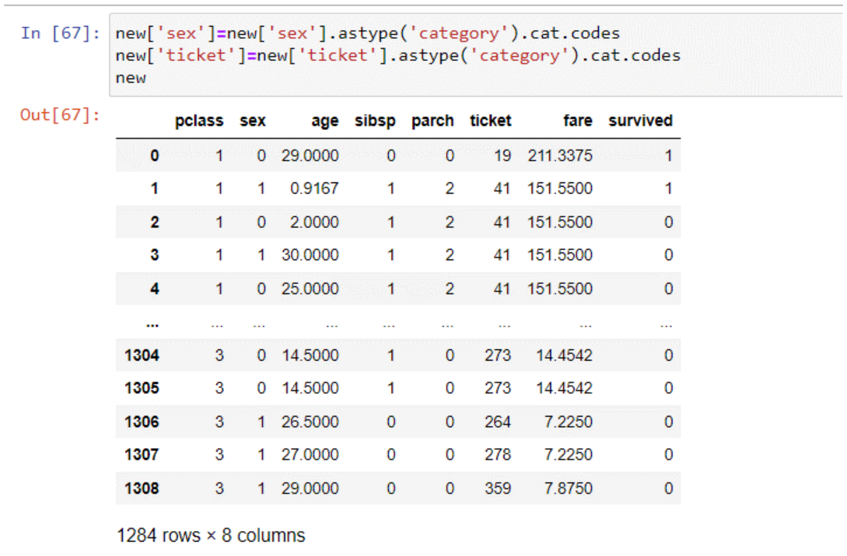


```
In [66]: new['age'].fillna(method='ffill', inplace=True)
         new['fare'].fillna(method='ffill', inplace=True)
         new.describe()
```

Out[66]:

	pclass	age	sibsp	parch	fare	survived
count	1284.000000	1284.000000	1284.000000	1284.000000	1284.000000	1284.000000
mean	2.285826	30.168744	0.456386	0.375389	33.120700	0.382399
std	0.840379	14.124712	0.906613	0.861881	52.115941	0.486163
min	1.000000	0.166700	0.000000	0.000000	0.000000	0.000000
25%	1.000000	21.000000	0.000000	0.000000	7.895800	0.000000
50%	3.000000	28.000000	0.000000	0.000000	14.108300	0.000000
75%	3.000000	39.000000	1.000000	0.000000	30.771850	1.000000
max	3.000000	80.000000	8.000000	9.000000	512.329200	1.000000

Now our next step is dealing with the correlated data types in our dataset, we start the process by converting our categorical attributes to numerical ones which enables us to calculate the correlation coefficient. To do so we used the function **.cat .codes** which converts the attributes from a string representation into an integer representation (1-1 Correspondence).



In [67]:

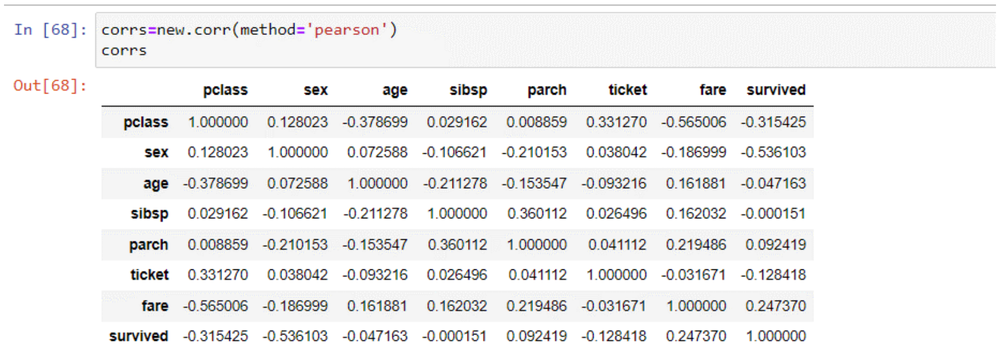
```
new['sex']=new['sex'].astype('category').cat.codes
new['ticket']=new['ticket'].astype('category').cat.codes
new
```

Out[67]:

	pclass	sex	age	sibsp	parch	ticket	fare	survived
0	1	0	29.0000	0	0	19	211.3375	1
1	1	1	0.9167	1	2	41	151.5500	1
2	1	0	2.0000	1	2	41	151.5500	0
3	1	1	30.0000	1	2	41	151.5500	0
4	1	0	25.0000	1	2	41	151.5500	0
...	...	...	...	...	...	...	...	...
1304	3	0	14.5000	1	0	273	14.4542	0
1305	3	0	14.5000	1	0	273	14.4542	0
1306	3	1	26.5000	0	0	264	7.2250	0
1307	3	1	27.0000	0	0	278	7.2250	0
1308	3	1	29.0000	0	0	359	7.8750	0

1284 rows × 8 columns

After being done with the conversions we now correlate by calculating the correlation coefficient between two of our values and storing it in a new data frame. To do so we use the Pearson method imported from the libraries:



In [68]:

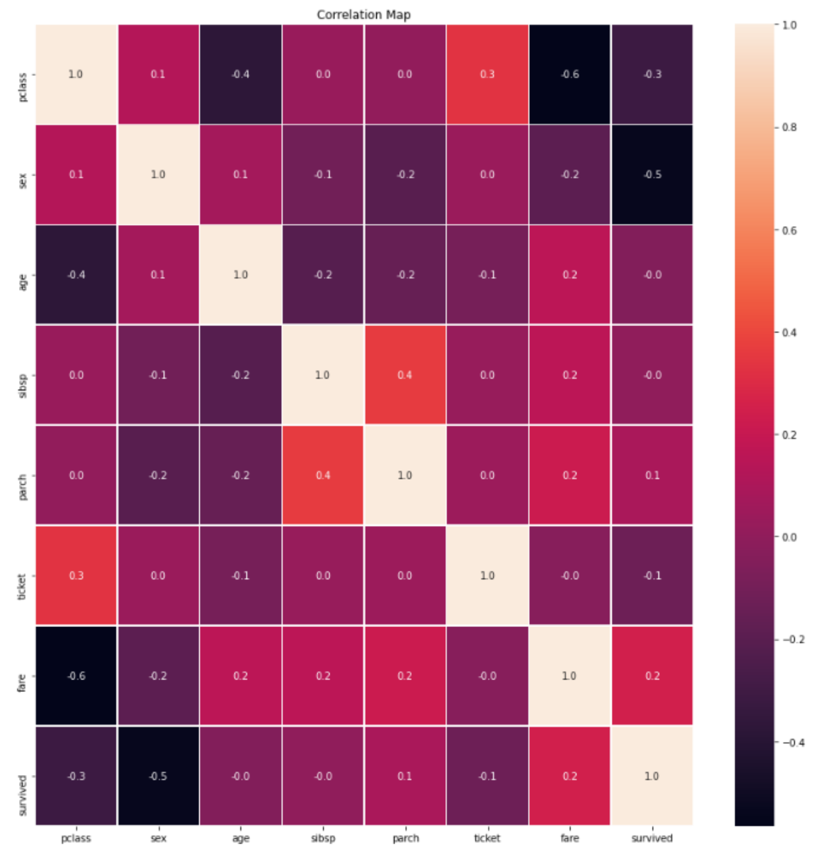
```
corrs=new.corr(method='pearson')
corrs
```

Out[68]:

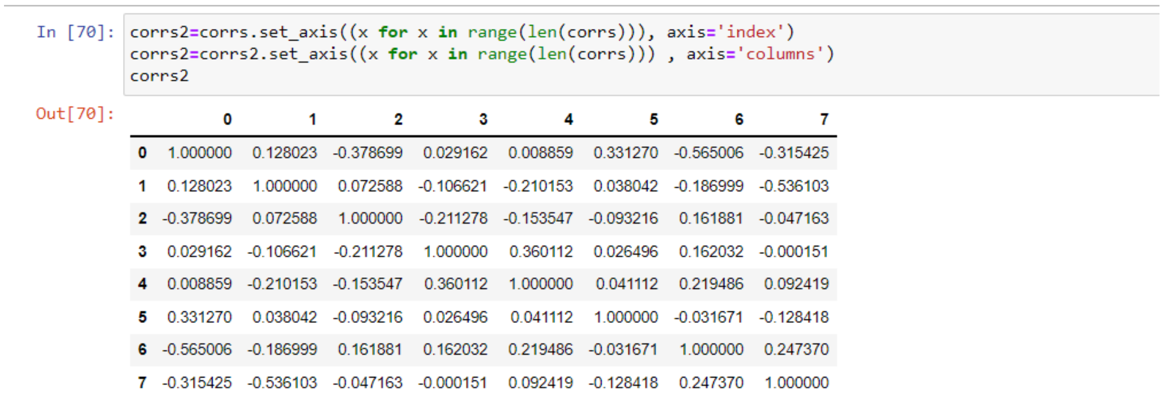
	pclass	sex	age	sibsp	parch	ticket	fare	survived
pclass	1.000000	0.128023	-0.378699	0.029162	0.008859	0.331270	-0.565006	-0.315425
sex	0.128023	1.000000	0.072588	-0.106621	-0.210153	0.038042	-0.186999	-0.536103
age	-0.378699	0.072588	1.000000	-0.211278	-0.153547	-0.093216	0.161881	-0.047163
sibsp	0.029162	-0.106621	-0.211278	1.000000	0.360112	0.026496	0.162032	-0.000151
parch	0.008859	-0.210153	-0.153547	0.360112	1.000000	0.041112	0.219486	0.092419
ticket	0.331270	0.038042	-0.093216	0.026496	0.041112	1.000000	-0.031671	-0.128418
fare	-0.565006	-0.186999	0.161881	0.162032	0.219486	-0.031671	1.000000	0.247370
survived	-0.315425	-0.536103	-0.047163	-0.000151	0.092419	-0.128418	0.247370	1.000000

After being done, we want to see our correlated and uncorrelated values and their rates, to determine the positive and negative correlations in our data. To do so, we used a heatmap as shown below, imported from the seaborn library:

```
In [69]: f,ax = plt.subplots(figsize=(15, 15))
         sns.heatmap(corrs, annot=True, linewidths=.5, fmt= '.1f',ax=ax)
         plt.title("Correlation Map")
         plt.show()
```



For easier access to our new data frame, defined as **corrs** above, we indexed the corrs data frame to a numeric representation and stored it in a new data frame called corrs2 using **set_axis**.

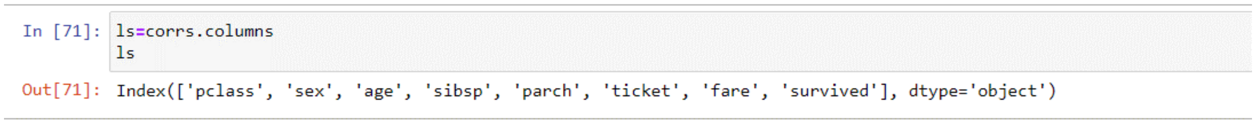


```
In [70]: corrs2=corrs.set_axis((x for x in range(len(corrs))), axis='index')
         corrs2=corrs2.set_axis((x for x in range(len(corrs))) , axis='columns')
         corrs2
```

Out[70]:

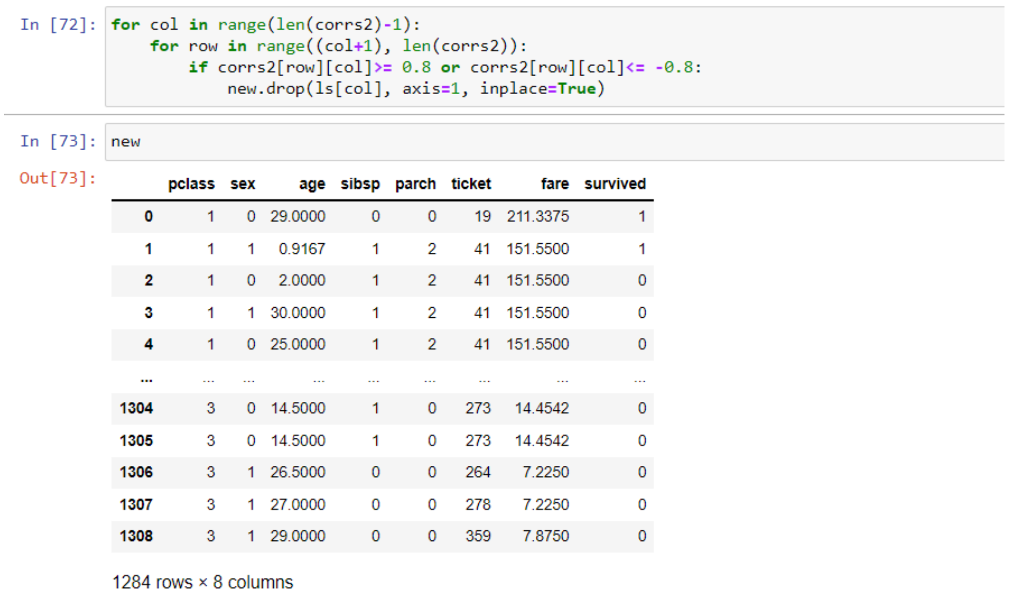
	0	1	2	3	4	5	6	7
0	1.000000	0.128023	-0.378699	0.029162	0.008859	0.331270	-0.565006	-0.315425
1	0.128023	1.000000	0.072588	-0.106621	-0.210153	0.038042	-0.186999	-0.536103
2	-0.378699	0.072588	1.000000	-0.211278	-0.153547	-0.093216	0.161881	-0.047163
3	0.029162	-0.106621	-0.211278	1.000000	0.360112	0.026496	0.162032	-0.000151
4	0.008859	-0.210153	-0.153547	0.360112	1.000000	0.041112	0.219486	0.092419
5	0.331270	0.038042	-0.093216	0.026496	0.041112	1.000000	-0.031671	-0.128418
6	-0.565006	-0.186999	0.161881	0.162032	0.219486	-0.031671	1.000000	0.247370
7	-0.315425	-0.536103	-0.047163	-0.000151	0.092419	-0.128418	0.247370	1.000000

We then store the columns we already had in **corrs** in a new list called 'ls'



```
In [71]: ls=corrs.columns
         ls
Out[71]: Index(['pclass', 'sex', 'age', 'sibsp', 'parch', 'ticket', 'fare', 'survived'], dtype='object')
```

Then we define a for loop function which traverses the elements in our correlation matrix. However, we do not have to traverse all of them since the correlation matrix is a symmetric matrix. So, our for loop traverses the lower triangle of the matrix only (without the main diagonal since its values = 1). Then it checks each of our correlation attributes using the if condition to see if it is positively or negatively attributed. It removes either of the said attributes if the condition = True. However, none of our attributes were removed.



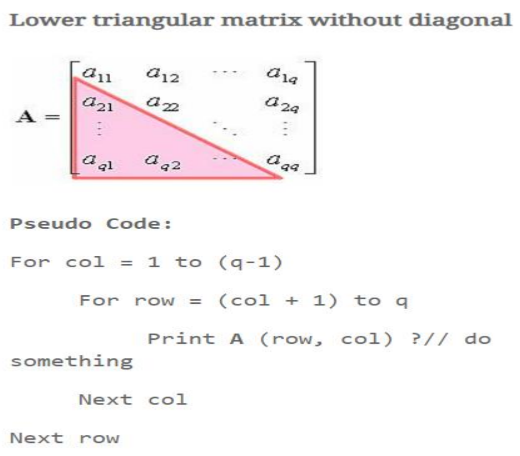
```
In [72]: for col in range(len(corrs2)-1):
             for row in range((col+1), len(corrs2)):
                 if corrs2[row][col]>= 0.8 or corrs2[row][col]<= -0.8:
                     new.drop(ls[col], axis=1, inplace=True)

In [73]: new
```

Out[73]:

	pclass	sex	age	sibsp	parch	ticket	fare	survived
0	1	0	29.0000	0	0	19	211.3375	1
1	1	1	0.9167	1	2	41	151.5500	1
2	1	0	2.0000	1	2	41	151.5500	0
3	1	1	30.0000	1	2	41	151.5500	0
4	1	0	25.0000	1	2	41	151.5500	0
...	...	...	...	...	...	...	...	...
1304	3	0	14.5000	1	0	273	14.4542	0
1305	3	0	14.5000	1	0	273	14.4542	0
1306	3	1	26.5000	0	0	264	7.2250	0
1307	3	1	27.0000	0	0	278	7.2250	0
1308	3	1	29.0000	0	0	359	7.8750	0

1284 rows × 8 columns

We created the previous function according to this pseudo code:

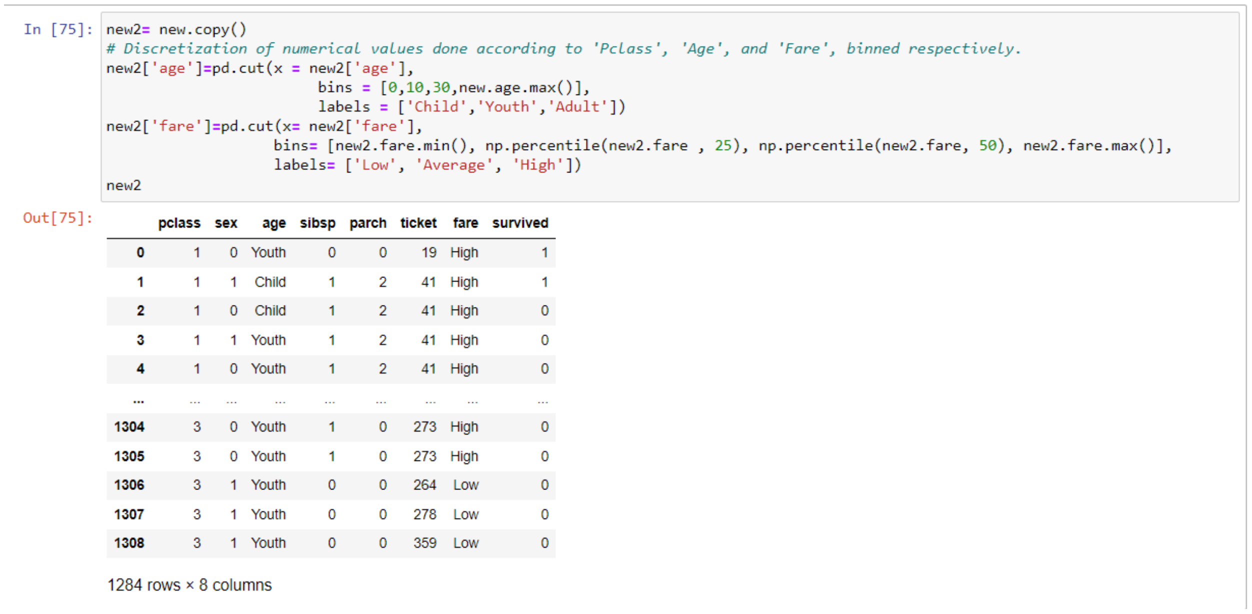


Lower triangular matrix without diagonal

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1q} \\ a_{21} & a_{22} & & a_{2q} \\ \vdots & & \ddots & \vdots \\ a_{q1} & a_{q2} & \cdots & a_{qq} \end{bmatrix}$$

```
Pseudo Code:
For col = 1 to (q-1)
      For row = (col + 1) to q
            Print A (row, col) ?// do
something
      Next col
Next row
```

Next part is the discretization, which is the process through which we can transform continuous variables, models or functions into a discrete form. We do this by creating a set of contiguous intervals (or bins) that go across the range of our desired variable/model/function. Continuous data is measured, while Discrete data is counted.



```
In [75]: new2= new.copy()
         # Discretization of numerical values done according to 'Pclass', 'Age', and 'Fare', binned respectively.
         new2['age']=pd.cut(x = new2['age'],
                                bins = [0,10,30,new.age.max()],
                                labels = ['Child','Youth','Adult'])
         new2['fare']=pd.cut(x= new2['fare'],
                            bins= [new2.fare.min(), np.percentile(new2.fare , 25), np.percentile(new2.fare, 50), new2.fare.max()],
                            labels= ['Low', 'Average', 'High'])
         new2
```

Out[75]:

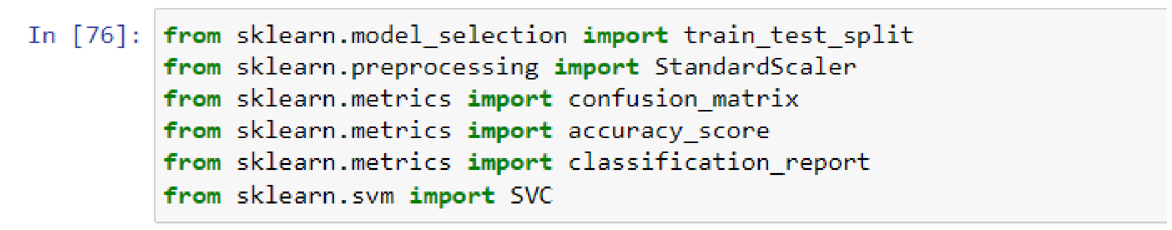
	pclass	sex	age	sibsp	parch	ticket	fare	survived
0	1	0	Youth	0	0	19	High	1
1	1	1	Child	1	2	41	High	1
2	1	0	Child	1	2	41	High	0
3	1	1	Youth	1	2	41	High	0
4	1	0	Youth	1	2	41	High	0
...	...	...	...	...	...	...	...	...
1304	3	0	Youth	1	0	273	High	0
1305	3	0	Youth	1	0	273	High	0
1306	3	1	Youth	0	0	264	Low	0
1307	3	1	Youth	0	0	278	Low	0
1308	3	1	Youth	0	0	359	Low	0

1284 rows × 8 columns

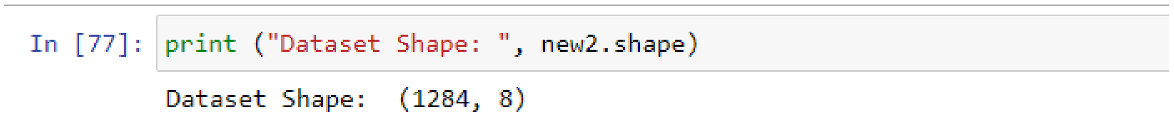
In the discretization code, we saved a copy of 'new' dataset in another variable 'new2' to discretize some attribute values in it. Then we discretized numerical values done according to 'pclass', 'age', and 'fare', binned respectively. We then called our main function to show the new table, which has values such as age and fare turned into strings.

Part 2 Explanation

For the second part of the project, we will be using classification techniques as a form of supervised learning. This model is built by inputting a set of training data for which the classes are pre-labeled in order for the algorithm to learn from. To achieve these results, we will start by importing the needed libraries and determining the dataset shape as follows:

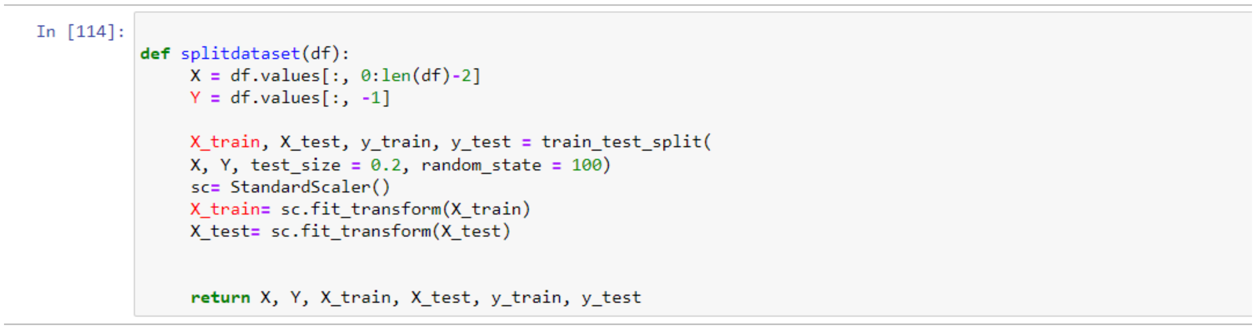


```
In [76]: from sklearn.model_selection import train_test_split
         from sklearn.preprocessing import StandardScaler
         from sklearn.metrics import confusion_matrix
         from sklearn.metrics import accuracy_score
         from sklearn.metrics import classification_report
         from sklearn.svm import SVC
```



```
In [77]: print ("Dataset Shape: ", new2.shape)

         Dataset Shape:  (1284, 8)
```

Then we define a function to the split dataset by the following process:



```
In [114]:
def splitdataset(df):
    X = df.values[:, 0:len(df)-2]
    Y = df.values[:, -1]

    X_train, X_test, y_train, y_test = train_test_split(
    X, Y, test_size = 0.2, random_state = 100)
    sc= StandardScaler()
    X_train= sc.fit_transform(X_train)
    X_test= sc.fit_transform(X_test)


    return X, Y, X_train, X_test, y_train, y_test
```

Firstly: Separating the values of the target variable (label / class 'survived') from the whole other values of other variables (columns), and stored in variable Y, while the other values are stored in variable X.

Secondly: Initialization of four new variables where the returned values of 'train_test_split' function are stored as follows:

- 'X_train' where 80% of the values of the whole dataset except for the target variable are stored [for training phase].
- 'X_test' where 20% of the values of the whole dataset except for the target variable are stored [for testing phase].
- 'Y_train' where 80% of the values of the target variable are stored [for training phase].
- 'Y_test' where 20% of the values of the target variable are stored [for testing phase].

Note: 'train_test_split' is a built-in function in python which takes some parameters as:

1. Arrays: sequence of indexable with same length / shape[0]. Allowed inputs are lists, NumPy arrays, SciPy-sparse matrices or pandas data frames.

(Set to X, and Y [the variables where the values of the whole dataset are splitted as mentioned above]),

2. Test_size: float or int, default= None. If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split (set to 0.2 as required).

If int, represents the absolute number of test samples. If None, the value is set to the complement of the train size. If train_size is also None, it will be set to 0.25.

3. Random_state which accepts int values, RandomState instance or None, default=None. It controls the shuffling applied to the data before applying the split (set to 100).

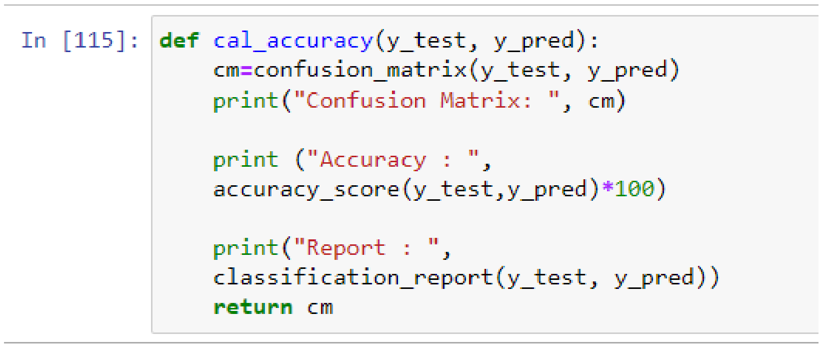
4. Some other parameters: train_size float or int, default=None. If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split.

If int, represents the absolute number of train samples. If None, the value is automatically set to the complement of the test size.

Thirdly: Initiation of an object 'sc' from the class 'StandardScaler', then applying 'fit_transform function' (which is called through 'sc' object) on the values stored in 'X_train' and 'X_test' which standarize those values to have a mean equals to zero, and variance/ standard deviation equal to one [values~N(0,1)].

Finally: Returning of the 6 variables 'X', 'Y', 'X_train', 'X_test', 'Y_train', and 'Y_test' to be used in further steps (eg: Classification Techniques).

We then define a function for accuracy calculation using the following steps:



```
In [115]: def cal_accuracy(y_test, y_pred):
              cm=confusion_matrix(y_test, y_pred)
              print("Confusion Matrix: ", cm)

              print ("Accuracy : ",
              accuracy_score(y_test,y_pred)*100)

              print("Report : ",
              classification_report(y_test, y_pred))
              return cm
```

Firstly: Storing the confusion matrix in 'cm' variable, and then printing it using 'confusion_matrix' is a built-in function which accepts some parameters as:

1. 'y_test' variable.

2. 'y_pred' variable where the predicted values of the unknown class / label using any classification technique are stored and returns a matrix containing the values of True Negative (TN), False Positive (FP), False Negative (FN), and True Positive (TP) respectively. Those values are calculated according to the comparison between the predicted values of the unknown class 'y_pred' using any classifier and the true known values of the class 'survived' stored in 'y_test'.

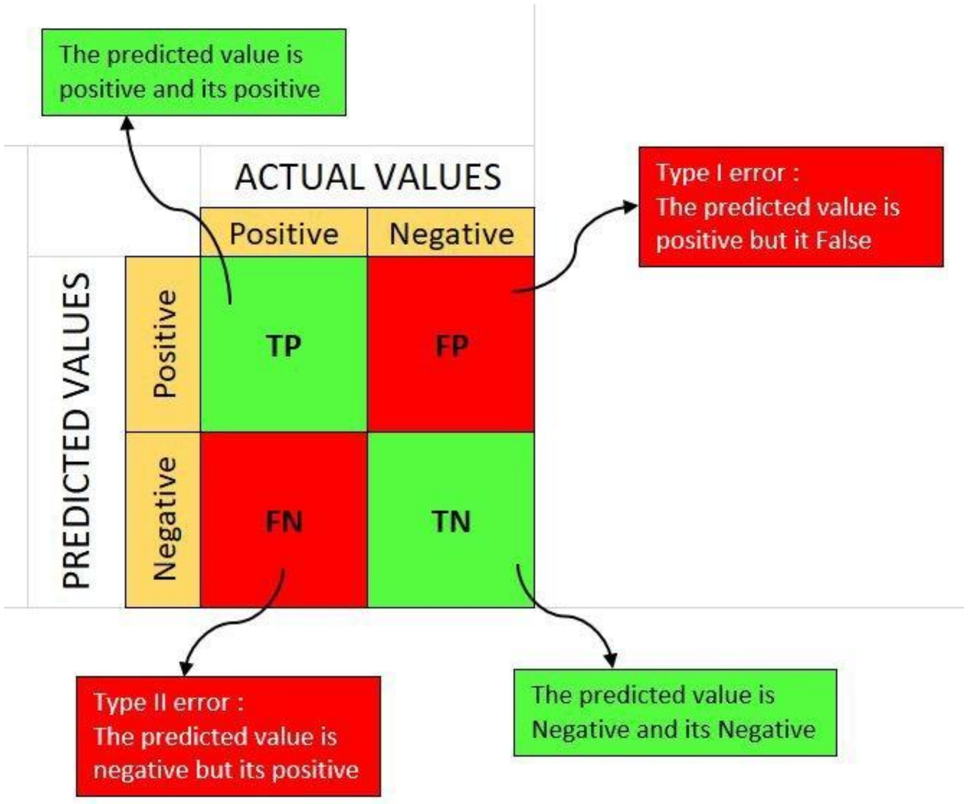
Note: Positive and negative in this case are generic names for the predicted class. There are four ways to check if the predictions are right or wrong:

True Negative (TN): when a case was negative and predicted negative.

True Positive (TP): when a case was positive and predicted positive.

False Negative (FP): when a case was positive but predicted negative.

False Positive (FP): when a case was negative but predicted positive.


The predicted value is positive and its positive
ACTUAL VALUES
Positive
Negative
PREDICTED VALUES
Positive
Negative
TP
FP
FN
TN
Type I error : The predicted value is positive but it False
Type II error : The predicted value is negative but its positive
The predicted value is Negative and its Negative

Secondly: Printing the accuracy percentage of the model (classifier) using 'accuracy_score' function which accepts 'y_test' and 'y_pred' as parameters, and returns the ratio of the sum of TN and TP (predicted correctly) to the total number of predicted values.

Accuracy = (TP+TN)/(TP+FP+FN+TN)

Thirdly: Printing a classification report using 'classification_report' function which takes 'Y_test' and 'Y_pred' as parameters.

A Classification report is used to measure the quality of predictions from a classification algorithm. How many predictions are True and how many are False. More specifically, True Positives, False Positives, True negatives and False Negatives are used to predict the metrics of a classification report.

The report shows the main classification metrics precision, recall, f1-score, and support of the trained classifier on a per-class basis. The metrics are calculated by using true and false positives, true and false negatives.

1. Precision – What percent of your predictions were correct? [Accuracy of positive predictions.]

Precision is the ability of a classifier not to label an instance positive that is actually negative. For each class it is defined as the ratio of true positives to the sum of true and false positives.

Precision = TP/(TP + FP)

2. Recall – What percent of the positive cases did you catch? [Fraction of positives that were correctly identified.]

Recall is the ability of a classifier to find all positive instances. For each class it is defined as the ratio of true positives to the sum of true positives and false negatives.

Recall = TP/(TP+FN)

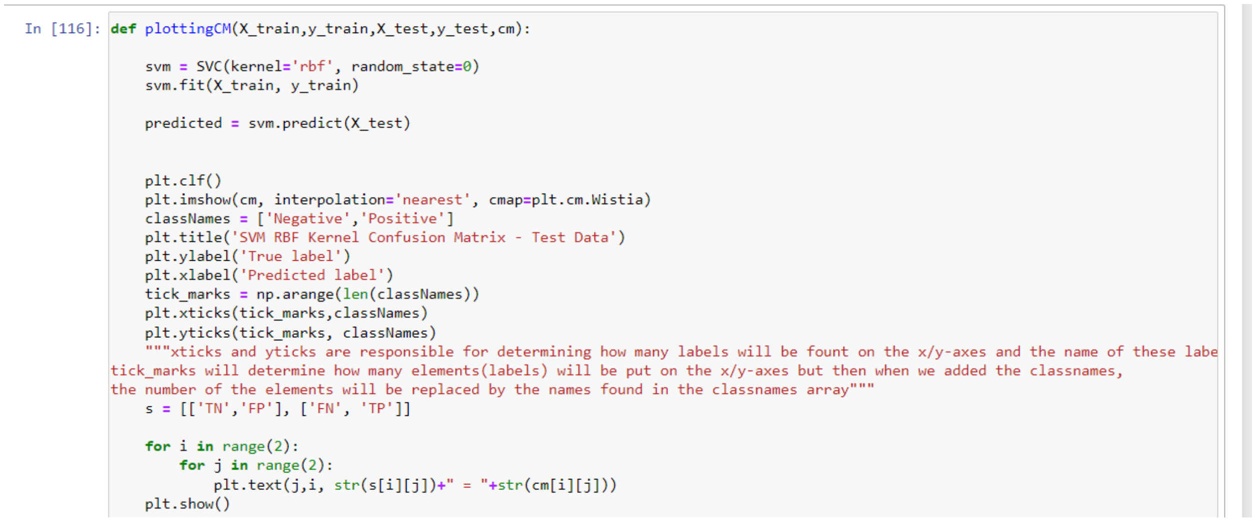
3. F1 score – What percent of positive predictions were correct?

The F1 score is a weighted harmonic mean of precision and recall such that the best score is 1.0 and the worst is 0.0. Generally speaking, F1 scores are lower than accuracy measures as they embed precision and recall into their computation. As a rule of thumb, the weighted average of F1 should be used to compare classifier models, not global accuracy.

F1 Score = 2*(Recall * Precision) / (Recall + Precision)

4. Support – Support is the number of actual occurrences of the class in the dataset. It doesn't vary between models; it just diagnoses the performance evaluation process.

Finally: Returning the values stored in 'cm' variable (confusion matrix) to be visualized in further steps.



```
In [116]: def plottingCM(X_train,y_train,X_test,y_test,cm):

              svm = SVC(kernel='rbf', random_state=0)
              svm.fit(X_train, y_train)

              predicted = svm.predict(X_test)


              plt.clf()
              plt.imshow(cm, interpolation='nearest', cmap=plt.cm.Wistia)
              classNames = ['Negative','Positive']
              plt.title('SVM RBF Kernel Confusion Matrix - Test Data')
              plt.ylabel('True label')
              plt.xlabel('Predicted label')
              tick_marks = np.arange(len(classNames))
              plt.xticks(tick_marks,classNames)
              plt.yticks(tick_marks, classNames)
              """xticks and yticks are responsible for determining how many labels will be fount on the x/y-axes and the name of these labe
          tick_marks will determine how many elements(labels) will be put on the x/y-axes but then when we added the classnames,
          the number of the elements will be replaced by the names found in the classnames array"""
              s = [['TN','FP'], ['FN', 'TP']]

              for i in range(2):
                  for j in range(2):
                      plt.text(j,i, str(s[i][j])+" = "+str(cm[i][j]))
              plt.show()
```

For the next part of our classification, we will be using the **Decision Tree**. Which is a tree-like model (flowchart-like model), used in classification where the class labels are known so it is a supervised-learning tool.

It consists of:

1) Internal nodes: that represent test on the attributes.

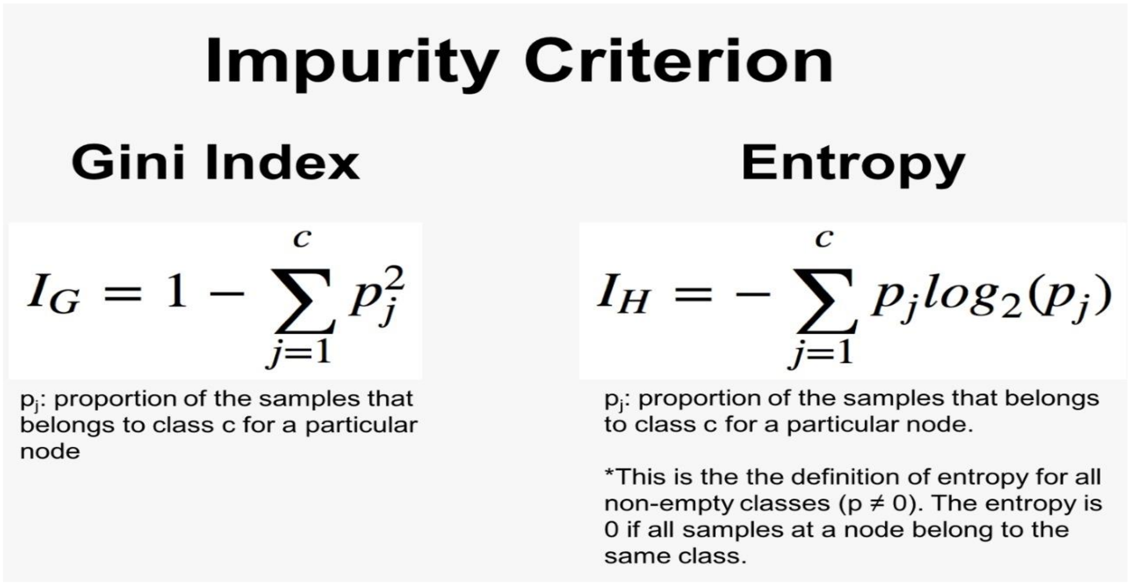
2) Branches: which represent the outcomes of the test.

3) Leaf nodes (terminal node): that represent the class labels.

The coding part:

We have used several functions to perform the decision tree classification:

1) Training using Gini/Entropy:



Impurity Criterion

Gini Index

$$I_G = 1 - \sum_{j=1}^{c} p_j^2$$

p_j: proportion of the samples that belongs to class c for a particular node

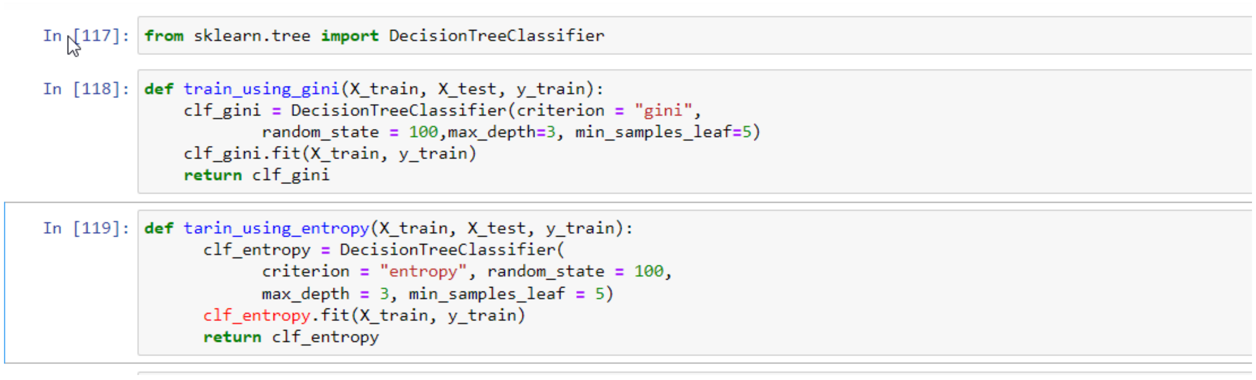
Entropy

$$I_H = - \sum_{j=1}^{c} p_j log_2(p_j)$$

p_j: proportion of the samples that belongs to class c for a particular node.

*This is the the definition of entropy for all non-empty classes ($p \neq 0$). The entropy is 0 if all samples at a node belong to the same class.

This function takes three parameters which are the x_train (the features with known labels) , x_test (the known labels) and y-train (the features that we want to predict their classes). Then we fit the model to the x_train and y-train to be used the prediction of the labels of the x_test as shown in the following figure:



```
In [117]: from sklearn.tree import DecisionTreeClassifier

In [118]: def train_using_gini(X_train, X_test, y_train):
              clf_gini = DecisionTreeClassifier(criterion = "gini",
                      random_state = 100,max_depth=3, min_samples_leaf=5)
              clf_gini.fit(X_train, y_train)
              return clf_gini

In [119]: def tarin_using_entropy(X_train, X_test, y_train):
                clf_entropy = DecisionTreeClassifier(
                      criterion = "entropy", random_state = 100,
                      max_depth = 3, min_samples_leaf = 5)
                clf_entropy.fit(X_train, y_train)
                return clf_entropy
```

2) The Prediction method, where we use the built-in function predict.

Then we calculate the accuracy of the model as shown in the figure.

3) Cal_accuracy: where we compare between the predicted labels and the labels which were already known but not used and we represent this by using

- Accuracy_score
- Confusion_matrix that shows the labels that were correctly or wrongly predicted.

- Classification_report

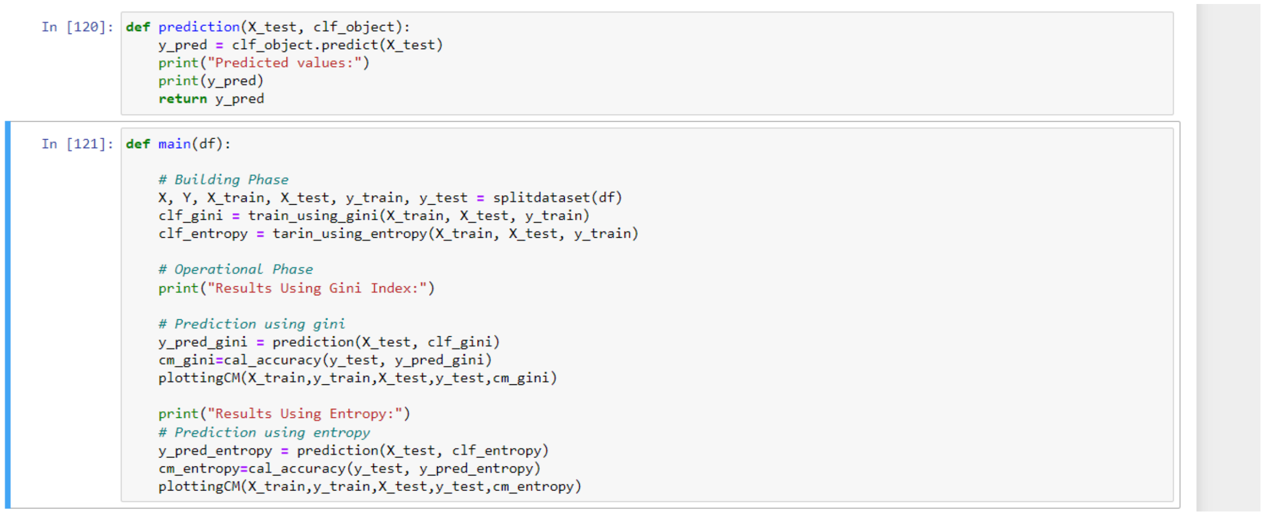
We also visualized the confusion matrix to be more readable and easy to be understood as shown in plotting CM function that takes x_train, y_train, x_test, y_test and the confusion matrix as parameters.

Where we draw it by using **SVC()** function and then fit it on the x_train and y_train and use it to predict the x_test, then we make sure that the figure is clean by using **plt.clf()**. Also, we display the data as image using **plt.imshow()** where we name the plot using **plt.title** , x-axis using **plt.xlabel()** label and the y-axis using **plt.ylabel()**.

Classnames: contain the names that will be found inside the visualization itself.

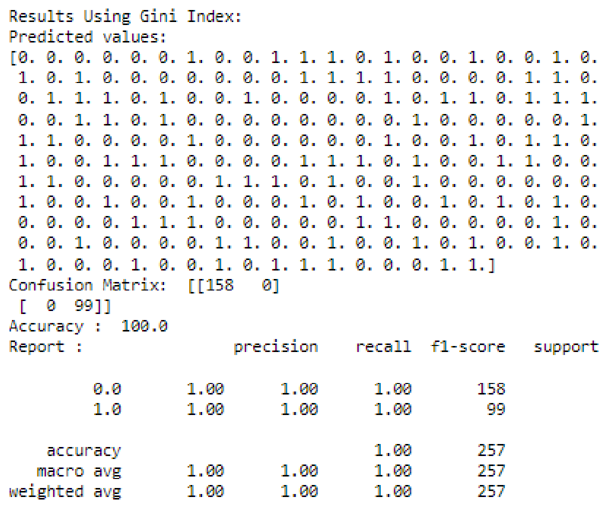
xticks and yticks are responsible for determining how many labels will be found on the x/y-axes and the name of these labels tick_marks will determine how many elements(labels) will be put on the x/y-axes but then when we added the classnames. The number of the elements will be replaced by the names found in the classnames array.

This function works on the given confusion matrix using nested for loops that passes through the values of the matrix one by one and concatenating them with the values found in “s array". Then we plot this graph using **plt.show()**.

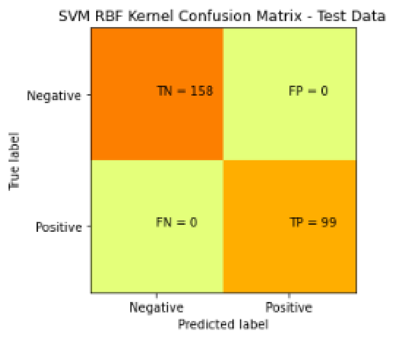


```
In [120]: def prediction(X_test, clf_object):
              y_pred = clf_object.predict(X_test)
              print("Predicted values:")
              print(y_pred)
              return y_pred

In [121]: def main(df):

              # Building Phase
              X, Y, X_train, X_test, y_train, y_test = splitdataset(df)
              clf_gini = train_using_gini(X_train, X_test, y_train)
              clf_entropy = tarin_using_entropy(X_train, X_test, y_train)

              # Operational Phase
              print("Results Using Gini Index:")

              # Prediction using gini
              y_pred_gini = prediction(X_test, clf_gini)
              cm_gini=cal_accuracy(y_test, y_pred_gini)
              plottingCM(X_train,y_train,X_test,y_test,cm_gini)

              print("Results Using Entropy:")
              # Prediction using entropy
              y_pred_entropy = prediction(X_test, clf_entropy)
              cm_entropy=cal_accuracy(y_test, y_pred_entropy)
              plottingCM(X_train,y_train,X_test,y_test,cm_entropy)
```

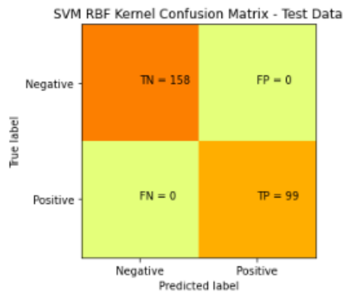
We call all these methods in the main method.



```
Results Using Gini Index:
Predicted values:
[0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 1. 1. 0. 1. 0. 0. 1. 0. 0. 1. 0. 0. 1. 0.
 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 0. 1. 1. 0. 0. 0. 1.
 0. 1. 1. 1. 0. 1. 0. 0. 1. 0. 0. 0. 0. 1. 0. 1. 1. 0. 1. 1. 1. 1. 1. 1.
 0. 0. 1. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 1. 1. 0. 0.
 1. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 0. 1. 1. 0. 0. 0. 1.
 1. 0. 0. 1. 1. 1. 0. 0. 0. 0. 1. 1. 1. 0. 1. 0. 0. 1. 1. 0. 0. 0. 1. 1.
 1. 1. 0. 0. 0. 0. 0. 1. 1. 1. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1.
 1. 0. 0. 1. 0. 0. 1. 0. 0. 0. 1. 0. 1. 0. 0. 1. 0. 1. 0. 1. 0. 1. 1. 0.
 0. 0. 0. 0. 1. 1. 1. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 1. 0. 0. 1. 0.
 0. 0. 1. 0. 0. 0. 0. 1. 1. 0. 0. 1. 0. 0. 1. 0. 1. 0. 0. 1. 0. 1. 1. 0.
 1. 0. 0. 0. 1. 0. 0. 1. 0. 1. 1. 1. 0. 0. 0. 1. 1.]
Confusion Matrix:  [[158   0]
 [  0  99]]
Accuracy :  100.0
Report :                precision    recall  f1-score   support

         0.0       1.00      1.00      1.00       158
         1.0       1.00      1.00      1.00        99

    accuracy                           1.00       257
   macro avg       1.00      1.00      1.00       257
weighted avg       1.00      1.00      1.00       257
```



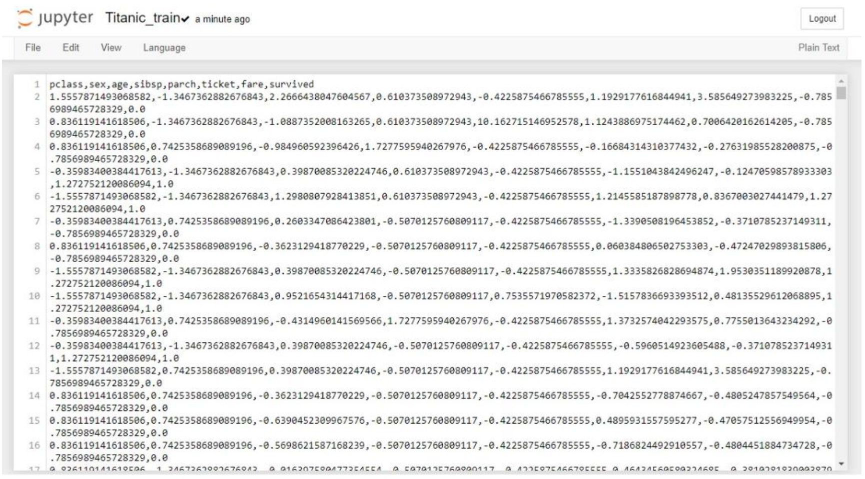
```
Results Using Entropy:
Predicted values:
[0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 1. 1. 0. 1. 0. 0. 1. 0. 0. 1. 0. 0. 1. 0.
 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 0. 1. 1. 0. 0. 0. 1.
 0. 1. 1. 1. 0. 1. 0. 0. 1. 0. 0. 0. 0. 1. 0. 1. 1. 0. 1. 1. 1. 1. 1. 1.
 0. 0. 1. 1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 1. 1. 0. 0.
 1. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0. 1. 0. 1. 1. 0. 0. 0. 1.
 1. 0. 0. 1. 1. 1. 0. 0. 0. 0. 1. 1. 1. 0. 1. 0. 0. 1. 1. 0. 0. 0. 1. 1.
 1. 1. 0. 0. 0. 0. 0. 1. 1. 1. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 1.
 1. 0. 0. 1. 0. 0. 1. 0. 0. 0. 1. 0. 1. 0. 0. 1. 0. 1. 0. 1. 0. 1. 1. 0.
 0. 0. 0. 0. 1. 1. 1. 0. 0. 0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 1. 0. 0. 1. 0.
 0. 0. 1. 0. 0. 0. 0. 1. 1. 0. 0. 1. 0. 0. 1. 0. 1. 0. 0. 1. 0. 1. 1. 0.
 1. 0. 0. 0. 1. 0. 0. 1. 0. 1. 1. 1. 0. 0. 0. 1. 1.]
Confusion Matrix:  [[158   0]
 [  0  99]]
Accuracy :  100.0
Report :                precision    recall  f1-score   support

         0.0       1.00      1.00      1.00       158
         1.0       1.00      1.00      1.00        99

    accuracy                           1.00       257
   macro avg       1.00      1.00      1.00       257
weighted avg       1.00      1.00      1.00       257
```



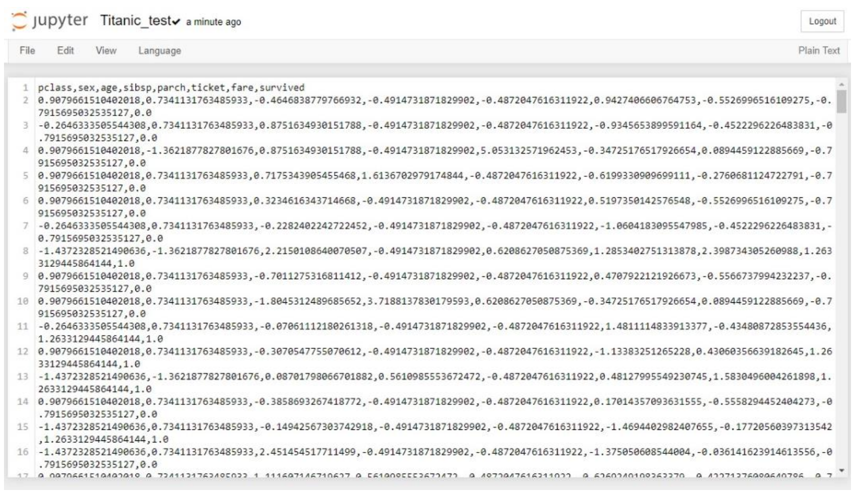
We splitted the dataset into train dataset and test dataset.

```
In [112]: append_to_csv(X_train, y_train, 'Titanic_train')
          append_to_csv(X_test, y_test, 'Titanic_test')
```

And these are how they look:

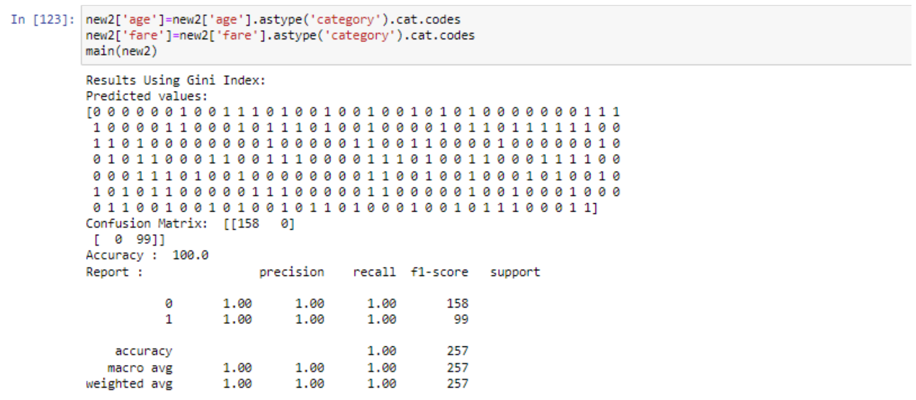


```
pclass,sex,age,sibsp,parch,ticket,fare,survived
1.5557871493068582,-1.3467362882676843,2.2666438047604567,0.610373508972943,-0.4225875466785555,1.1929177616844941,3.585649273983225,-0.7856989465728329,0.0
0.836119141618506,-1.3467362882676843,-1.0887352008163265,0.610373508972943,10.162715146952578,1.1243886975174462,0.7006420162614205,-0.7856989465728329,0.0
0.836119141618506,0.7425358689089196,-0.984960592396426,1.7277595940267976,-0.4225875466785555,-0.16684314310377432,-0.27631985528200875,-0.7856989465728329,0.0
-0.35983400384417613,-1.3467362882676843,0.39870085320224746,0.610373508972943,-0.4225875466785555,-1.1551043842496247,-0.12470598578933303,1.272752120086094,1.0
-1.5557871493068582,-1.3467362882676843,1.2980807928413851,0.610373508972943,-0.4225875466785555,1.2145585187898778,0.8367003027441479,1.272752120086094,1.0
-0.35983400384417613,0.7425358689089196,0.2603347086423801,-0.5070125760809117,-0.4225875466785555,-1.3390508196453852,-0.3710785237149311,-0.7856989465728329,0.0
0.836119141618506,0.7425358689089196,-0.3623129418770229,-0.5070125760809117,-0.4225875466785555,0.060384806502753303,-0.47247029893815806,-0.7856989465728329,0.0
-1.5557871493068582,-1.3467362882676843,0.39870085320224746,-0.5070125760809117,-0.4225875466785555,1.3335826828694874,1.9530351189920878,1.272752120086094,1.0
-1.5557871493068582,-1.3467362882676843,0.9521654314417168,-0.5070125760809117,0.7535571970582372,-1.5157836693393512,0.4813552961206889 5,1.272752120086094,1.0
-0.35983400384417613,0.7425358689089196,-0.4314960141569566,1.7277595940267976,-0.4225875466785555,1.3732574042293575,0.775501364323429 2,-0.7856989465728329,0.0
-0.35983400384417613,-1.3467362882676843,0.39870085320224746,-0.5070125760809117,-0.4225875466785555,-0.5960514923605488,-0.3710785237149311,1.272752120086094,1.0
-1.5557871493068582,0.7425358689089196,0.39870085320224746,-0.5070125760809117,-0.4225875466785555,1.1929177616844941,3.585649273983225,-0.7856989465728329,0.0
0.836119141618506,0.7425358689089196,-0.3623129418770229,-0.5070125760809117,-0.4225875466785555,-0.7042552778874667,-0.4805247857549564,-0.7856989465728329,0.0
0.836119141618506,0.7425358689089196,-0.6390452309967576,-0.5070125760809117,-0.4225875466785555,0.4895931557595277,-0.4705751255694995 4,-0.7856989465728329,0.0
0.836119141618506,0.7425358689089196,-0.5698621587168239,-0.5070125760809117,-0.4225875466785555,-0.7186824492910557,-0.4804451884734728,-0.7856989465728329,0.0
0.836119141618506,-1.3467362882676843,-0.016307580477354554,-0.5070125760809117,-0.4225875466785555,0.4643456958032468 5,-0.3810281830903879
```



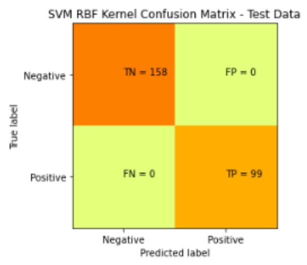
```
pclass,sex,age,sibsp,parch,ticket,fare,survived
0.9079661510402018,0.7341131763485933,-0.4646838779766932,-0.4914731871829902,-0.4872047616311922,0.9427406606764753,-0.5526996516109275,-0.7915695032535127,0.0
-0.2646333505544308,0.7341131763485933,0.8751634930151788,-0.4914731871829902,-0.4872047616311922,-0.9345653899591164,-0.4522296226483831,-0.7915695032535127,0.0
0.9079661510402018,-1.3621877827801676,0.8751634930151788,-0.4914731871829902,5.053132571962453,-0.34725176517926654,0.0894459122885669,-0.7915695032535127,0.0
0.9079661510402018,0.7341131763485933,0.7175343905455468,1.6136702979174844,-0.4872047616311922,-0.6199330909699111,-0.2760681124722791,-0.7915695032535127,0.0
0.9079661510402018,0.7341131763485933,0.3234616343714668,-0.4914731871829902,-0.4872047616311922,0.5197350142576548,-0.5526996516109275,-0.7915695032535127,0.0
-0.2646333505544308,0.7341131763485933,-0.2282402242722452,-0.4914731871829902,-0.4872047616311922,-1.0604183095547985,-0.4522296226483831,-0.7915695032535127,0.0
-1.4372328521490636,-1.3621877827801676,2.2150108640070507,-0.4914731871829902,0.6208627050875369,1.2853402751313878,2.398734305260988,1.2633129445864144,1.0
0.9079661510402018,0.7341131763485933,-0.7011275316811412,-0.4914731871829902,-0.4872047616311922,0.4707922121926673,-0.5566737994232237,-0.7915695032535127,0.0
0.9079661510402018,0.7341131763485933,-1.8045312489685652,3.7188137830179593,0.6208627050875369,-0.34725176517926654,0.0894459122885669,-0.7915695032535127,0.0
-0.2646333505544308,0.7341131763485933,-0.07061112180261318,-0.4914731871829902,-0.4872047616311922,1.4811114833913377,-0.43480872853554436,1.2633129445864144,1.0
0.9079661510402018,0.7341131763485933,-0.3070547755070612,-0.4914731871829902,-0.4872047616311922,-1.13383251265228,0.43060356639182645,1.2633129445864144,1.0
-1.4372328521490636,-1.3621877827801676,0.08701798066701882,0.5610985553672472,-0.4872047616311922,0.48127995549230745,1.5830496004261898,1.2633129445864144,1.0
0.9079661510402018,0.7341131763485933,-0.3858693267418772,-0.4914731871829902,-0.4872047616311922,0.17014357093631555,-0.5558294452404273,-0.7915695032535127,0.0
-1.4372328521490636,0.7341131763485933,-0.14942567303742918,-0.4914731871829902,-0.4872047616311922,-1.4694402982407655,-0.17720560397313542,1.2633129445864144,1.0
-1.4372328521490636,0.7341131763485933,2.451454517711499,-0.4914731871829902,-0.4872047616311922,-1.375050608544004,-0.036141623914613556,-0.7915695032535127,0.0
```

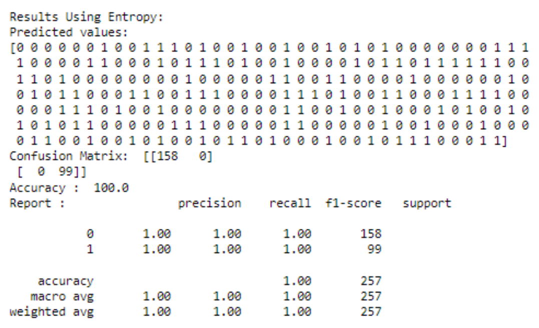
We also perform this classification tool on the dataset with the discretized attributes.



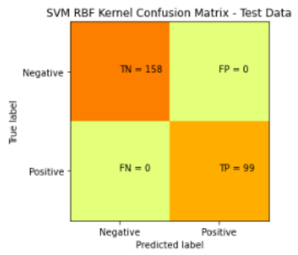
```
In [123]: new2['age']=new2['age'].astype('category').cat.codes
          new2['fare']=new2['fare'].astype('category').cat.codes
          main(new2)
```

```
Results Using Gini Index:
Predicted values:
[0 0 0 0 0 0 1 0 0 1 1 1 0 1 0 0 1 0 0 1 0 0 1 0 1 0 1 0 0 0 0 0 0 0 1 1 1
 1 0 0 0 0 1 1 0 0 0 1 0 1 1 1 0 1 0 0 1 0 0 0 0 1 0 1 1 0 1 1 1 1 1 1 0 0
 1 1 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 1 0
 0 1 0 1 1 0 0 0 1 1 0 0 1 1 1 0 0 0 0 1 1 1 0 1 0 0 1 1 0 0 0 1 1 1 1 0 0
 0 0 0 1 1 1 0 1 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 1 0 0 0 1 0 1 0 0 1 0
 1 0 1 0 1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0
 0 1 1 0 0 1 0 0 1 0 1 0 0 1 0 1 1 0 1 0 0 0 1 0 0 1 0 1 1 1 0 0 0 1 1]
Confusion Matrix:  [[158   0]
 [  0  99]]
Accuracy :  100.0
Report :               precision    recall  f1-score   support

           0       1.00      1.00      1.00       158
           1       1.00      1.00      1.00        99

    accuracy                           1.00       257
   macro avg       1.00      1.00      1.00       257
weighted avg       1.00      1.00      1.00       257
```





```
Results Using Entropy:
Predicted values:
[0 0 0 0 0 0 1 0 0 1 1 1 0 1 0 0 1 0 0 1 0 0 1 0 1 0 1 0 0 0 0 0 0 0 1 1 1
 1 0 0 0 0 1 1 0 0 0 1 0 1 1 1 0 1 0 0 1 0 0 0 0 1 0 1 1 0 1 1 1 1 1 1 0 0
 1 1 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 1 0 0 1 1 0 0 0 0 1 0 0 0 0 0 0 1 0
 0 1 0 1 1 0 0 0 1 1 0 0 1 1 1 0 0 0 0 1 1 1 0 1 0 0 1 1 0 0 0 1 1 1 1 0 0
 0 0 0 1 1 1 0 1 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0 1 0 0 0 1 0 1 0 0 1 0
 1 0 1 0 1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 1 1 0 0 0 0 0 1 0 0 1 0 0 0 1 0 0 0
 0 1 1 0 0 1 0 0 1 0 1 0 0 1 0 1 1 0 1 0 0 0 1 0 0 1 0 1 1 1 0 0 0 1 1]
Confusion Matrix:  [[158   0]
 [  0  99]]
Accuracy :  100.0
Report :               precision    recall  f1-score   support

           0       1.00      1.00      1.00       158
           1       1.00      1.00      1.00        99

    accuracy                           1.00       257
   macro avg       1.00      1.00      1.00       257
weighted avg       1.00      1.00      1.00       257
```



For the next part in our classification, we have the **KNN** algorithm. KNN algorithm is one of the simplest Machine Learning algorithms based on Supervised Learning technique. It assumes the similarity between the new case/data and available cases and put the new case into the category that is most like the available categories. It stores all the available data and classifies a new data point based on the similarity. This means when new data appears then it can be easily classified into a well suite category by using KNN algorithm.

Advantages:

• The algorithm is simple and easy to implement.

• There's no need to build a model, tune several parameters, or make additional assumptions.

Disadvantages:

• The algorithm gets significantly slower as the number of examples and/or predictors/independent variables increase.

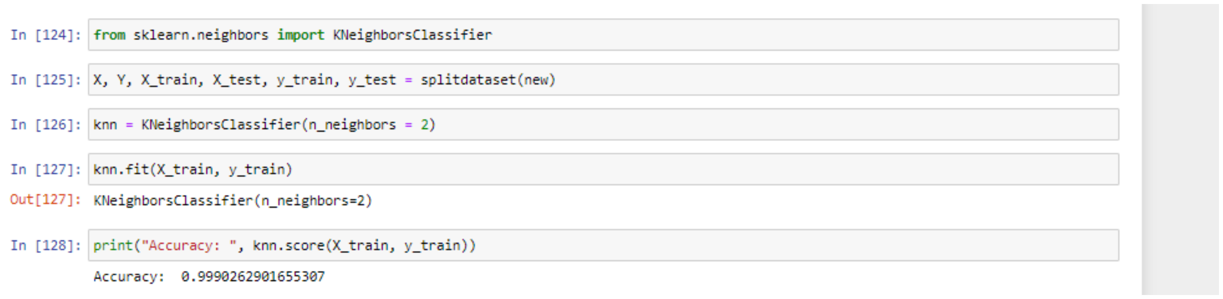
KNN implementation:

1. Load the data (previously loaded).

2. Splitting Data: To understand model performance, dividing the dataset into a training set and a test set by using function splitdataset(new).

3. Initialize K to your chosen number of neighbors [k=2].

4. Use K KNeighborsClassifier Module.



```
In [124]: from sklearn.neighbors import KNeighborsClassifier

In [125]: X, Y, X_train, X_test, y_train, y_test = splitdataset(new)

In [126]: knn = KNeighborsClassifier(n_neighbors = 2)

In [127]: knn.fit(X_train, y_train)
Out[127]: KNeighborsClassifier(n_neighbors=2)

In [128]: print("Accuracy: ", knn.score(X_train, y_train))
          Accuracy:  0.9990262901655307
```

KNeighborsClassifier Module Functions:

• Calculate the distance between the query example and the current example from the data.

• Add the distance and the index of the example to an ordered collection.

• Sort the ordered collection of distances and indices from smallest to largest (in ascending order) by the distances. (Euclidean distance).

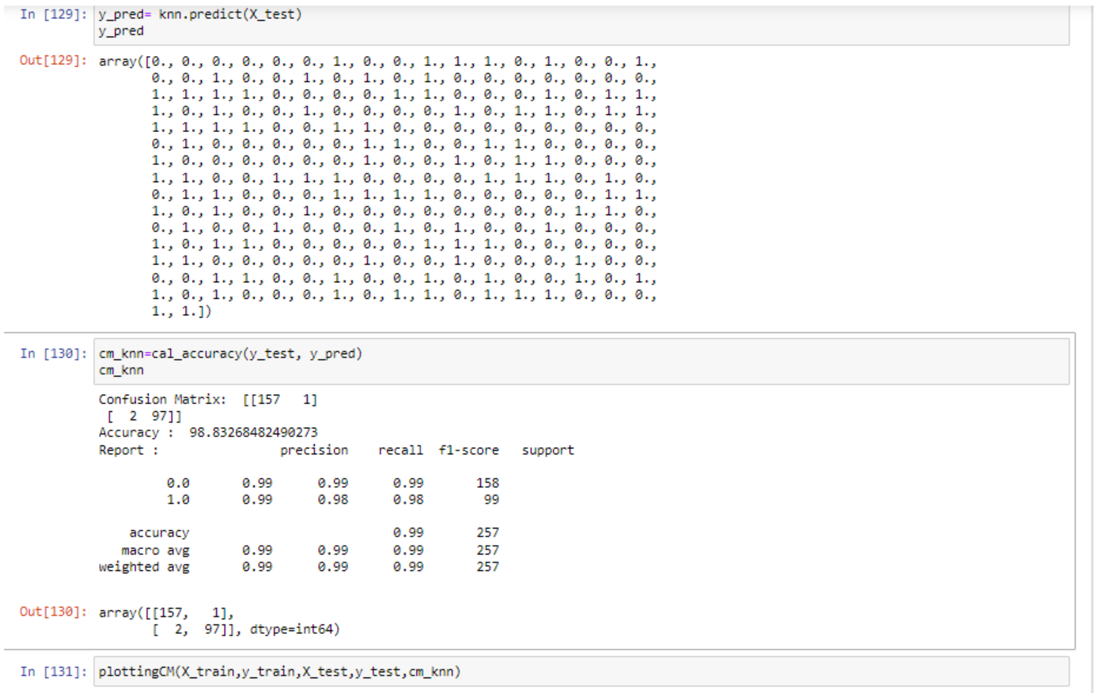
• Pick the first K entries from the sorted collection.

• Get the labels of the selected K entries.

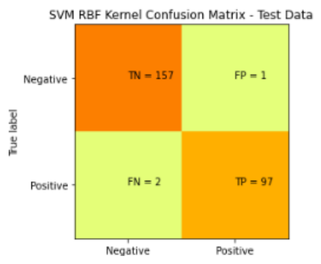
• If classification, return the mode of the K labels. [Survived or not]

Training and Testing:

We fit our model on the train set using **fit()** and perform prediction on the test set using **predict** KNeighborsClassifier Module.



```
In [129]: y_pred= knn.predict(X_test)
          y_pred

Out[129]: array([0., 0., 0., 0., 0., 0., 1., 0., 0., 1., 1., 1., 0., 1., 0., 0., 1.,
                 0., 0., 1., 0., 0., 1., 0., 1., 0., 1., 0., 0., 0., 0., 0., 0., 0.,
                 1., 1., 1., 1., 0., 0., 0., 0., 1., 1., 0., 0., 0., 1., 0., 1., 1.,
                 1., 0., 1., 0., 0., 1., 0., 0., 0., 0., 1., 0., 1., 1., 0., 1., 1.,
                 1., 1., 1., 1., 0., 0., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
                 0., 1., 0., 0., 0., 0., 0., 1., 1., 0., 0., 1., 1., 0., 0., 0., 0.,
                 1., 0., 0., 0., 0., 0., 0., 1., 0., 0., 1., 0., 1., 1., 0., 0., 0.,
                 1., 1., 0., 0., 1., 1., 1., 0., 0., 0., 0., 1., 1., 1., 0., 1., 0.,
                 0., 1., 1., 0., 0., 0., 1., 1., 1., 1., 0., 0., 0., 0., 0., 1., 1.,
                 1., 0., 1., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 0.,
                 0., 1., 0., 0., 1., 0., 0., 0., 1., 0., 1., 0., 0., 1., 0., 0., 0.,
                 1., 0., 1., 1., 0., 0., 0., 0., 0., 1., 1., 1., 0., 0., 0., 0., 0.,
                 1., 1., 0., 0., 0., 0., 0., 1., 0., 0., 1., 0., 0., 0., 1., 0., 0.,
                 0., 0., 1., 1., 0., 0., 1., 0., 0., 1., 0., 1., 0., 0., 1., 0., 1.,
                 1., 0., 1., 0., 0., 0., 1., 0., 1., 1., 0., 1., 1., 1., 0., 0., 0.,
                 1., 1.])

In [130]: cm_knn=cal_accuracy(y_test, y_pred)
          cm_knn

          Confusion Matrix:  [[157   1]
           [  2  97]]
          Accuracy :  98.83268482490273
          Report :                precision    recall  f1-score   support

                   0.0       0.99      0.99      0.99       158
                   1.0       0.99      0.98      0.98        99

              accuracy                           0.99       257
             macro avg       0.99      0.99      0.99       257
          weighted avg       0.99      0.99      0.99       257


Out[130]: array([[157,   1],
                 [  2,  97]], dtype=int64)

In [131]: plottingCM(X_train,y_train,X_test,y_test,cm_knn)
```



Moving on, we will be using the **Naïve Bayes** classifier. Naïve bayes is a classification technique based on Bayes' Theorem with an assumption of independence among predictors. In simple terms, a Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. It performs probabilistic prediction, prediction of class membership probabilities.

Where the formula for predicting whether the class label will survive or not is:

- (P(x|ci)*p(ci))/p(x)

Where ci is m class label for whether they survived or not:

- And p(xi|ci) is the probability of each attribute given the class label in which they survived or not.

And since p(x) is constant for all classes, then only;

- P(xi|ci)*p(ci)

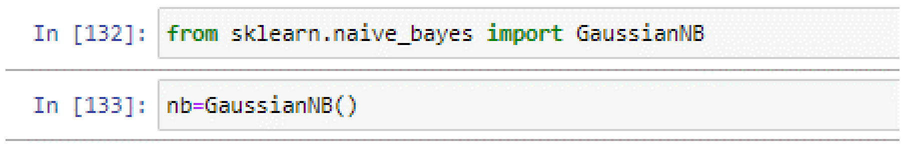
Then we will take p(xi|survived)*p(survived)

And p(xi|didn’t survive)*p(didn’t survive)

And we will take the maximum and predict whether they survived or not.

Python implementation and explanation of this algorithm is:

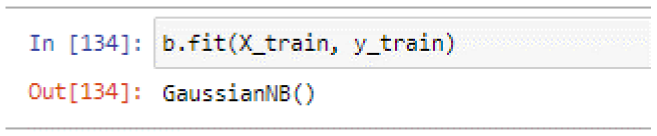
We will import **GaussianNB** from **sklearn.naive.bayes**, which helps us to know the performance of the classification model on a set of test data and it is very important for classification models.



```
In [132]: from sklearn.naive_bayes import GaussianNB

In [133]: nb=GaussianNB()
```

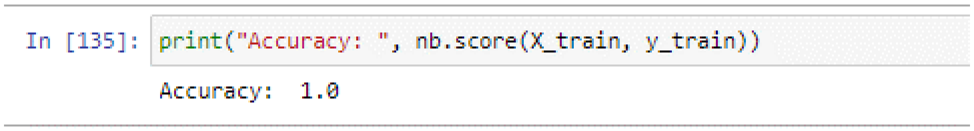
We initiate an object 'nb' from '=GaussianNB' class. Then we perform a train test split on our dataset by calling the functions above which splits the code into 80% training and 20% testing and then training them by calling the functions x_train, y_train.

Step1: Model training



```
In [134]: b.fit(X_train, y_train)
Out[134]: GaussianNB()
```

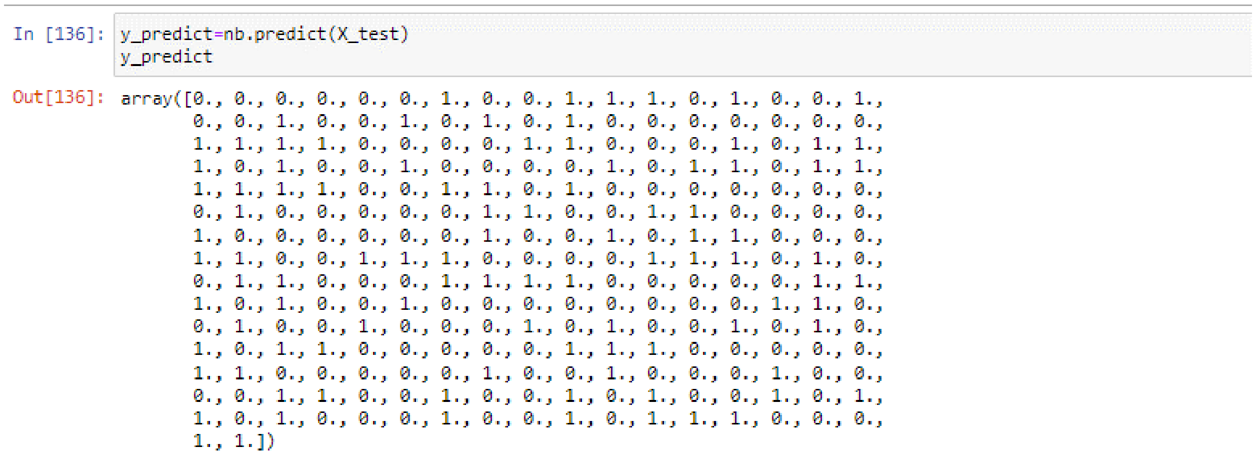
We are building the data model by training and fitting x_train, y_train.



```
In [135]: print("Accuracy: ", nb.score(X_train, y_train))

          Accuracy:  1.0
```

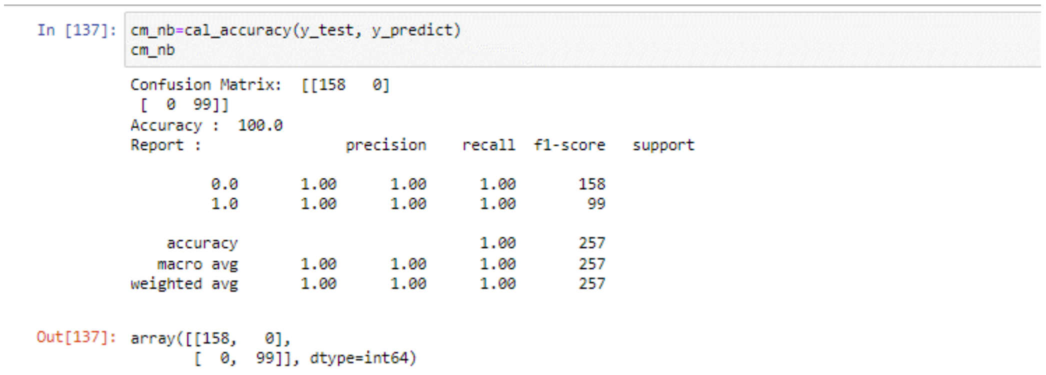
We get the accuracy of the trained data set which is equal to 1.

Step 2: Classification (prediction)



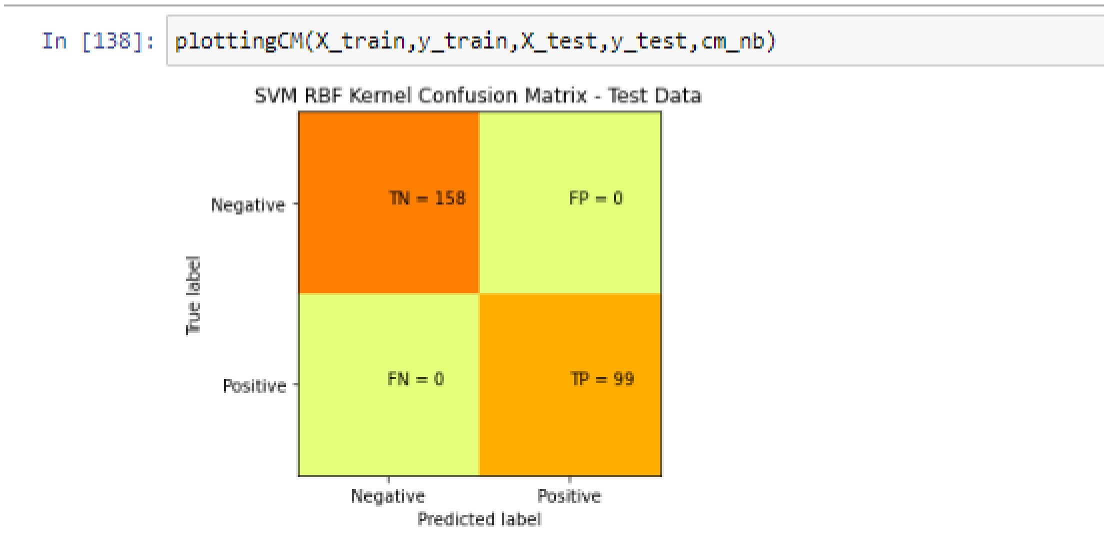
```
In [136]: y_predict=nb.predict(X_test)
          y_predict

Out[136]: array([0., 0., 0., 0., 0., 0., 1., 0., 0., 1., 1., 1., 0., 1., 0., 0., 1.,
                 0., 0., 1., 0., 0., 1., 0., 1., 0., 1., 0., 0., 0., 0., 0., 0., 0.,
                 1., 1., 1., 1., 0., 0., 0., 0., 1., 1., 0., 0., 0., 1., 0., 1., 1.,
                 1., 0., 1., 0., 0., 1., 0., 0., 0., 0., 1., 0., 1., 1., 0., 1., 1.,
                 1., 1., 1., 1., 0., 0., 1., 1., 0., 1., 0., 0., 0., 0., 0., 0., 0.,
                 0., 1., 0., 0., 0., 0., 0., 1., 1., 0., 0., 1., 1., 0., 0., 0., 0.,
                 1., 0., 0., 0., 0., 0., 0., 1., 0., 0., 1., 0., 1., 1., 0., 0., 0.,
                 1., 1., 0., 0., 1., 1., 1., 0., 0., 0., 0., 1., 1., 1., 0., 1., 0.,
                 0., 1., 1., 0., 0., 0., 1., 1., 1., 1., 0., 0., 0., 0., 0., 1., 1.,
                 1., 0., 1., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 1., 1., 0.,
                 0., 1., 0., 0., 1., 0., 0., 0., 1., 0., 1., 0., 0., 1., 0., 1., 0.,
                 1., 0., 1., 1., 0., 0., 0., 0., 0., 1., 1., 1., 0., 0., 0., 0., 0.,
                 1., 1., 0., 0., 0., 0., 0., 1., 0., 0., 1., 0., 0., 0., 1., 0., 0.,
                 0., 0., 1., 1., 0., 0., 1., 0., 0., 1., 0., 1., 0., 0., 1., 0., 1.,
                 1., 0., 1., 0., 0., 0., 1., 0., 0., 1., 0., 1., 1., 1., 0., 0., 0.,
                 1., 1.])
```

We will predict the x_test results and get y_test.



```
In [137]: cm_nb=cal_accuracy(y_test, y_predict)
          cm_nb

          Confusion Matrix:  [[158   0]
           [  0  99]]
          Accuracy :  100.0
          Report :               precision    recall  f1-score   support

                   0.0       1.00      1.00      1.00       158
                   1.0       1.00      1.00      1.00        99

              accuracy                           1.00       257
             macro avg       1.00      1.00      1.00       257
          weighted avg       1.00      1.00      1.00       257


Out[137]: array([[158,   0],
                 [  0,  99]], dtype=int64)
```

And then we will compare y_test, y_predict and get the tested (predicted) data accuracy cal_accuracy(y_test, y_predict) and the confusion matrix.



```
In [138]: plottingCM(X_train,y_train,X_test,y_test,cm_nb)
```

Therefore, the tested data accuracy is 100%.

Finally, we have our **K-Means** algorithm.

K-Means algorithm is an iterative algorithm that tries to partition the dataset into K pre-defined distinct non-overlapping subgroups (clusters) where each data point belongs to only one group. It tries to make the intra-cluster data points as similar as possible while also keeping the clusters as different (far) as possible. It assigns data points to a cluster such that the sum of the squared distance between the data points and the cluster's centroid (arithmetic mean of all the data points that belong to that cluster) is at the minimum. The less variation we have within clusters, the more homogeneous (similar) the data points are within the same cluster.

The way K-Means algorithm works is as follows:

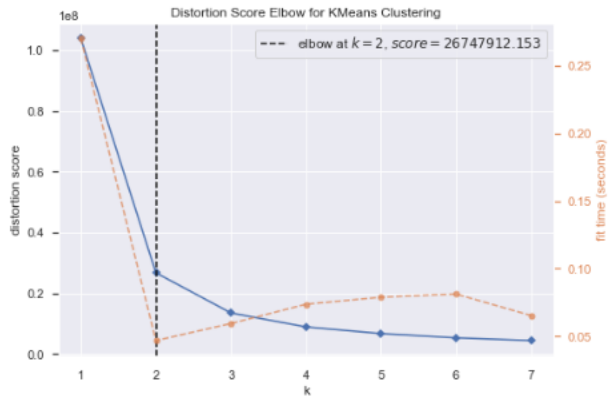
- Specify number of clusters K.
- Initialize centroids by first shuffling the dataset and then randomly selecting K data points for the centroids without replacement.

- Keep iterating until there is no change to the centroids. i.e assignment of data points to clusters isn't changing.
- Compute the sum of the squared distance between data points and all centroids.
- Assign each data point to the closest cluster (centroid).
- Compute the centroids for the clusters by taking the average of the all-data points that belong to each cluster.

The coding part:

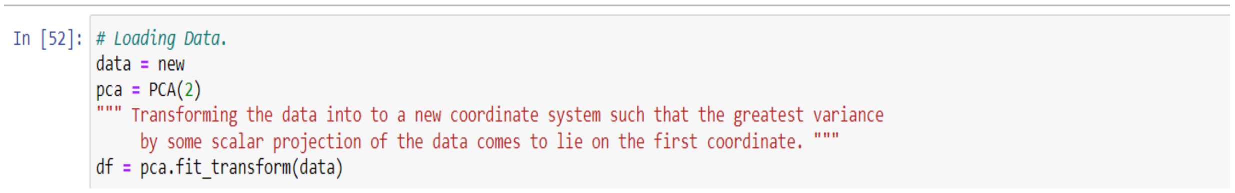
We imported **KElbowVisualizer** from **Yellowbrick** library to use the "Elbow Method" to select the optimal number of clusters by fitting the model with a range of values for K. If the line chart resembles an arm, then the "elbow" (the point of inflection on the curve) is a good indication that the underlying model fits best at that point. In the visualizer "elbow" will be annotated with a dashed line.

```
In [232]: model = KMeans()
          no_of_cols=new.shape[1]
          visualizer = KElbowVisualizer(model, k=(1,no_of_cols)).fit(new)
          visualizer.show()
          """The elbow method used to preidct the optimal value for the number of clusters such that when the curve starts to bend (elbow)
          this is the optimal k.
          Therefore the optimal k here is =2"""
```



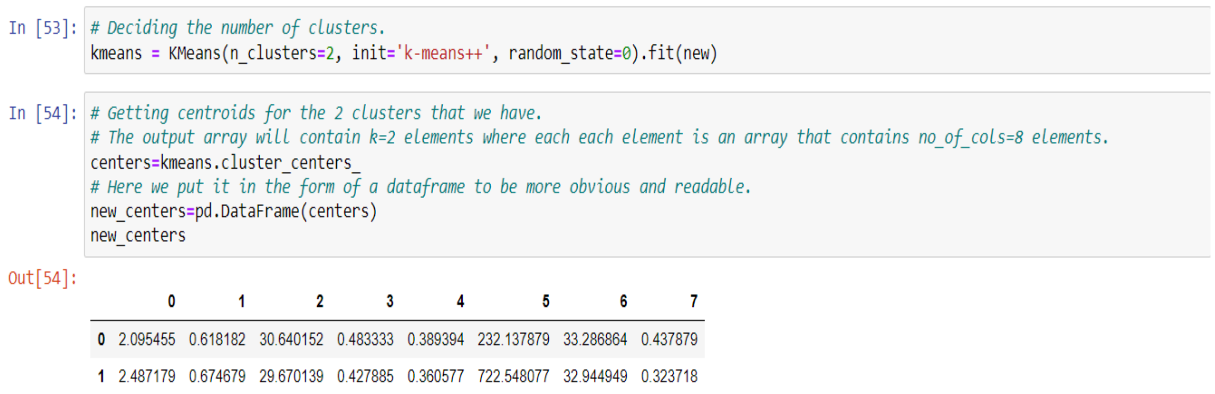
```
Out[232]: 'The elbow method used to preidct the optimal value for the number of clusters such that when the curve starts to bend (elbow)
          \nthis is the optimal k.\nTherefore the optimal k here is =2'
```

We imported **PCA** from **sklearn.decomposition** to reduce the linear dimensionality of the data by transforming the data into to a new coordinate system such that the greatest variance by some scalar projection of the data comes to lie on the first coordinate.



```
In [52]: # Loading Data.
         data = new
         pca = PCA(2)
         """ Transforming the data into to a new coordinate system such that the greatest variance
             by some scalar projection of the data comes to lie on the first coordinate. """
         df = pca.fit_transform(data)
```

We imported **KMeans** from **sklearn.cluster** to apply the K-Means algorithm. We decided the number of clusters to be 2 and we got the centroids, then we predicted the labels of the clusters and chose the unique labels where we counted the number of data points that belongs to each cluster.



```
In [53]: # Deciding the number of clusters.
         kmeans = KMeans(n_clusters=2, init='k-means++', random_state=0).fit(new)

In [54]: # Getting centroids for the 2 clusters that we have.
         # The output array will contain k=2 elements where each each element is an array that contains no_of_cols=8 elements.
         centers=kmeans.cluster_centers_
         # Here we put it in the form of a dataframe to be more obvious and readable.
         new_centers=pd.DataFrame(centers)
         new_centers

Out[54]:
          0         1          2         3         4           5          6         7
   0  2.095455  0.618182  30.640152  0.483333  0.389394  232.137879  33.286864  0.437879
   1  2.487179  0.674679  29.670139  0.427885  0.360577  722.548077  32.944949  0.323718
```

```
In [55]: """ The kmeans.labels_ is used to know which cluster that every data point belongs to; such that every data point will belong
         to whether cluster_0 or cluster_1. """
         kmeans_labels=kmeans.labels_
         # We add a new column in the dataframe called "cluster" to show which cluster does each object belong to.
         new['cluster']=kmeans_labels
         new
```

Out[55]:

	pclass	sex	age	sibsp	parch	ticket	fare	survived	cluster
0	1	0	29.0000	0	0	19	211.3375	1	0
1	1	1	0.9167	1	2	41	151.5500	1	0
2	1	0	2.0000	1	2	41	151.5500	0	0
3	1	1	30.0000	1	2	41	151.5500	0	0
4	1	0	25.0000	1	2	41	151.5500	0	0
...	...	...	...	...	...	...	...	...	...
1304	3	0	14.5000	1	0	273	14.4542	0	0
1305	3	0	14.5000	1	0	273	14.4542	0	0
1306	3	1	26.5000	0	0	264	7.2250	0	0
1307	3	1	27.0000	0	0	278	7.2250	0	0
1308	3	1	29.0000	0	0	359	7.8750	0	0

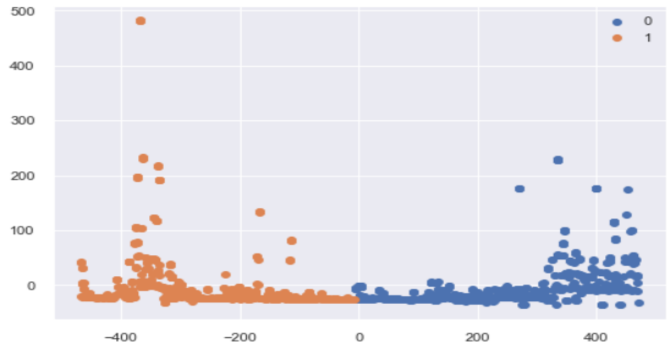
1284 rows × 9 columns

```
In [56]: # A summary for the previous dataframe:
         # The classes of the algorithm {0,1}.
         u_labels = np.unique(kmeans_labels)
         # This will count how many objects (data points) belong to the 2 clusters.
         Counter(kmeans.labels_)

Out[56]: Counter({0: 660, 1: 624})
```

Then, using the scatter plot we plotted the K-Means clusters.

Finally, we calculated the inertia to measure how well the dataset was clustered by K-Means and we printed the maximum number of iterations done by the K-Means.

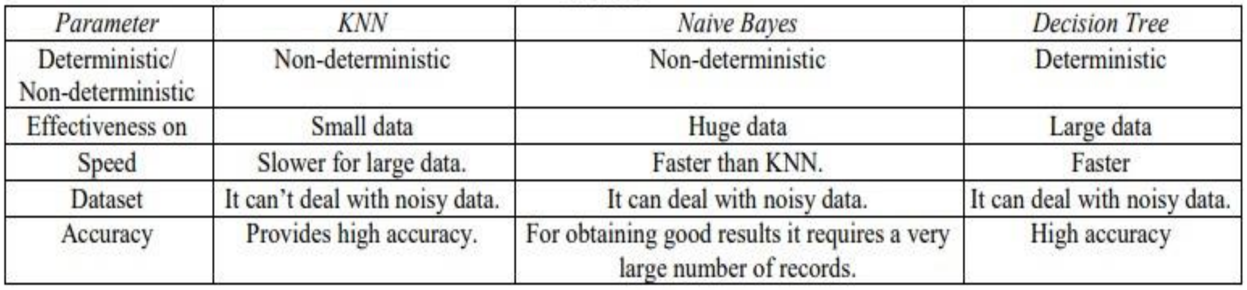
```
In [57]: # Predict the labels of clusters.
         label = kmeans.fit_predict(df)

         # Plotting the results:
         for i in u_labels:
             plt.scatter(df[label == i , 0] , df[label == i , 1] , label = i)
         plt.legend()
         plt.show()
```



```
In [58]: # Extra information:
         # Inertia measures how well a dataset was clustered by K-Means.
         print("Inertia:",kmeans.inertia_)
         # The maximum number of iterations done by the kmeans.
         print("maximum number of iterations:",kmeans.n_iter_)

         Inertia: 26500470.880814955
         maximum number of iterations: 7
```

Bonus Comparison:

Table I



Parameter	*KNN*	*Naive Bayes*	*Decision Tree*
Deterministic/ Non-deterministic	Non-deterministic	Non-deterministic	Deterministic
Effectiveness on	Small data	Huge data	Large data
Speed	Slower for large data.	Faster than KNN.	Faster
Dataset	It can't deal with noisy data.	It can deal with noisy data.	It can deal with noisy data.
Accuracy	Provides high accuracy.	For obtaining good results it requires a very large number of records.	High accuracy

References:

First: Pre-Processing

The Complete Titanic Dataset

Titanic Dataset

How to Loop over a symmetric Matrix

Dataset and its description (Meta Data)

Wide vs. Long data

Correlation in python

Discretization in python

NumPy, SciPy, and Pandas: Correlation with python

The Art of Effective Visualization of Multi-dimensional Data

Second: Classification

Classification Report

Decision Tree

Train_test_split

Introduction to decision trees

Decision tree implementation

KNN

k-nearest neighbor algorithm in Python

Analyzing dataset using KNN

KNN tutorial using Scikit learn

KNN from scratch

Naïve Bayes Classifier

Introduction to Naive Bayes algorithm

Naive Bayes Scratch Implementation using Python

K-Means Algorithm

Calculating accuracy score of k-means on the test data set

Plotting K-Means clusters

Visualizing clusters

K-means on titanic dataset

K-means clustering in python

K Means Clustering Simplified in Python

In Depth: k-Means Clustering

With this, we conclude our report for the data mining project. Sincere thanks to everyone who has worked on this project and to our Professor Abeer Amer, as well as her teaching assistants.