Linear regression

In machine learning linear regression is an algorithm based on supervised learning. Which performs a regression task. Regression models a target prediction value based on independent variables. Linear regression performs the task to predict a dependent variable value based on a given independent variable. Mainly it finds out a linear relationship between input and output.

Hypothesis function for Linear Regression:

$$Y = \alpha + \beta . X$$

Here,

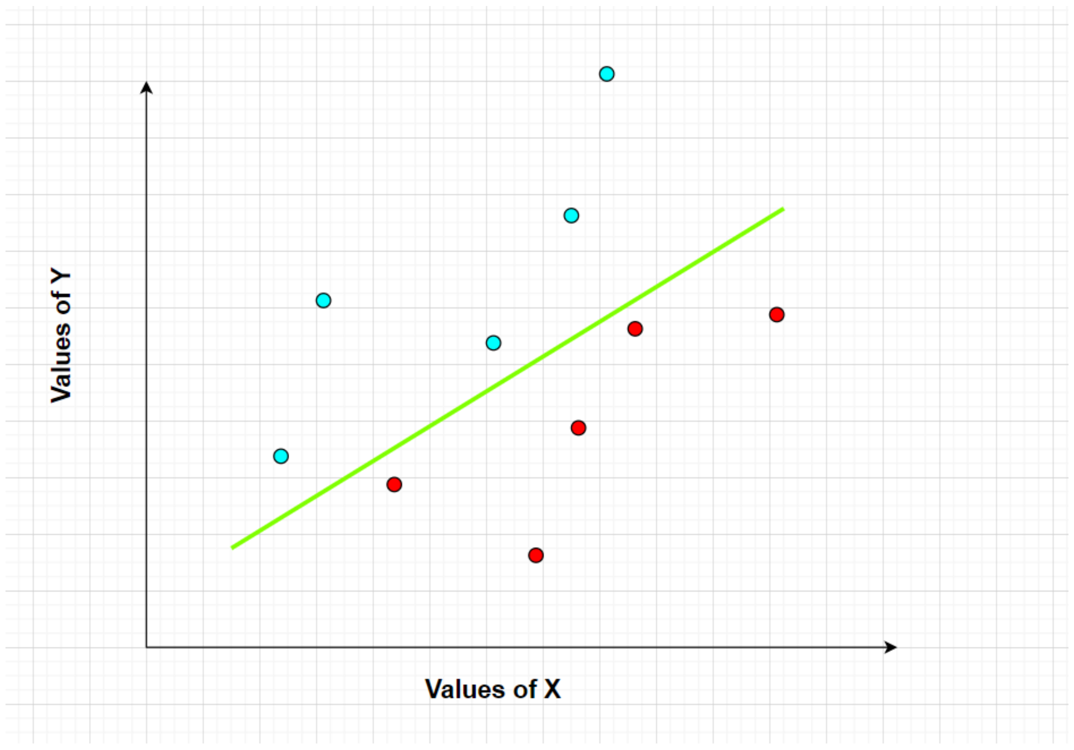
Y = Line of best fit

α = Intercept

β = Coefficient of X

X = Input training data (Given value)

When regression model of a data set is plotted on graph the Y becomes the regression line.



In the graph the green line (Y) represents the predicted values corresponding to X input values. The dots on the graph are the realistic values corresponding to X input values of a data set.

The distance between Y line and the dots are known as error terms. In order to make a regression model more accurate the sum of errors must be minimized. But, just the sum of errors will not provide proper outcome.

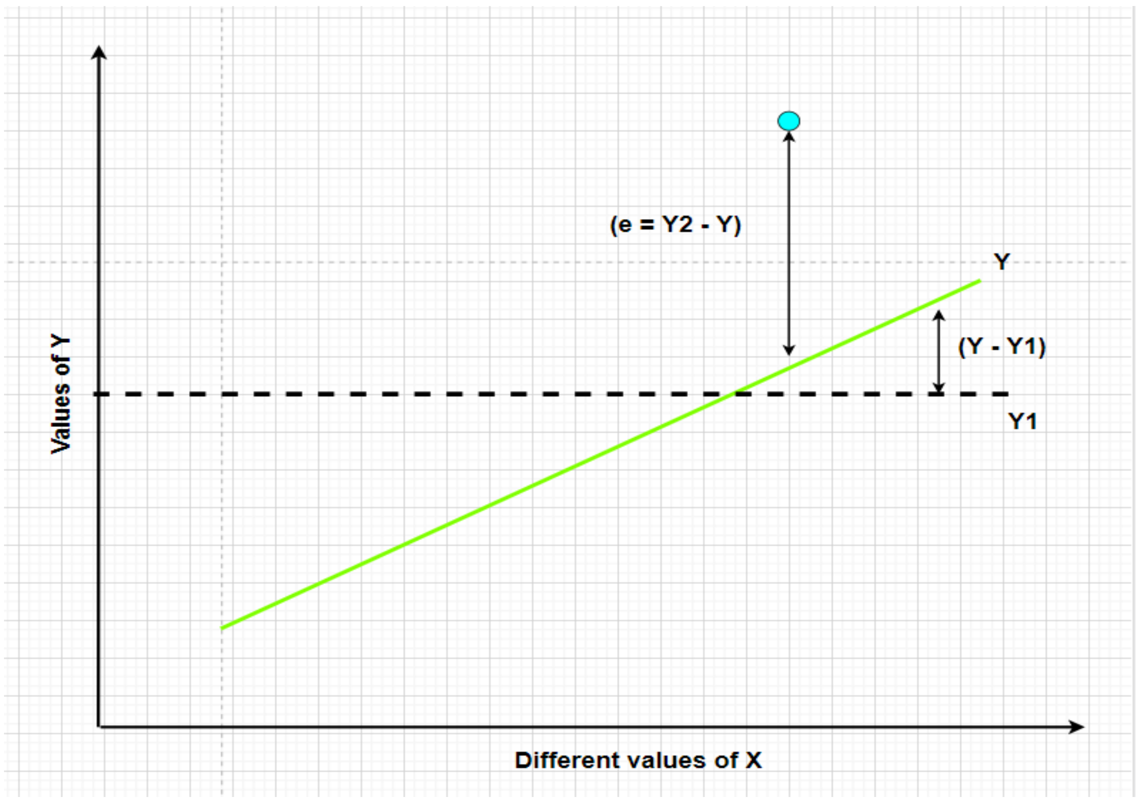
Let's, consider the blue dots as positive values and the red dots as negative values. If we calculate the sum of errors like this the result will be 0.

$$\sum e = 0$$

Because of the negative values. So, we must square all the values to solve this problem. Then minimize the sum of the squared errors (SSE) to find the proper Y line.

$$\text{Min}\sum e^2$$

Only one line can satisfy this condition and that will be the correct Y line.



Now, here Y1 line is the mean of Y values from a data set. Here, (Y – Y1) is the explained deviation from the mean. Because we expected it to be in that state based on a given data set. And e = Y2 – Y is the unexplained deviation from the mean. Since, the distance of that point is much higher than what predicted.

Now, based on this we can derive these terms:

Sum of squares due to regression (SSR) = $\sum(\boldsymbol{Y} - \boldsymbol{Y1})^2$

Sum of squares due to error (SSE) = $\sum(\boldsymbol{Y2} - \boldsymbol{Y})^2$

Sum of square total (SST) = SSR + SSE

R^2 = SSR/SST

Here, R^2 is the proportion of the variation in Y being explained by the variation in X.

Another part of linear regression is degrees of freedom. It means the minimum number of observations required to estimate a regression. The formula for degrees of freedom is:

$$\mathbf{df = n - k - 1}$$

df = Degrees of freedom

n = Number of observations

k = Number of explanatory (X) variables

In linear regression R^2 gives us the idea of the strength of relation between X and Y values. And R^2 is affected by degrees of freedom. Because as the degrees of freedom decreases the R^2 will increase. This happens when more useless variables are added in the model. Which will give us the wrong output which must be avoided.

With the help of adjusted R^2 this issue can be solved. Equation for adjusted R^2:

$$\text{adj } R^2 = 1 - \left(1 - R^2\right)\frac{n-1}{n-k-1}$$

Here it can be seen that as k increases, adj R^2 will decrease reflecting the reduced power in a model when the model have a low number of degrees of freedom. If the model is given useful variables only then the adj R^2 increase. This solves the problem faced by using only R^2.

Logistic regression

Logistic regression is a regression model similar to linear regression model. Linear regression models the data using a linear function but logistic regression models the data using the sigmoid function. Logistic Regression is used when the dependent variable is categorical. This model can work with continuous data and discrete data. Logistic regression classifies these data according to the given inputs. It doesn`t use SSE, SSR, SST and R^2 instead it uses maximum likelihood.

Maximum likelihood is a probabilistic framework that can estimate parameters of a logistic regression model. Under this framework, a probability distribution for the target variable (class label) must be assumed and then a likelihood function defined that calculates the probability of observing the outcome given the input data and the model. This function can then be optimized to find the set of parameters that results in the largest sum likelihood over the training dataset.

Now, in terms of sigmoid function it is known:

$$\textbf{Sigmoid} = \frac{\mathbf{1}}{\mathbf{1} + \boldsymbol{e}^{-\boldsymbol{z}}}$$

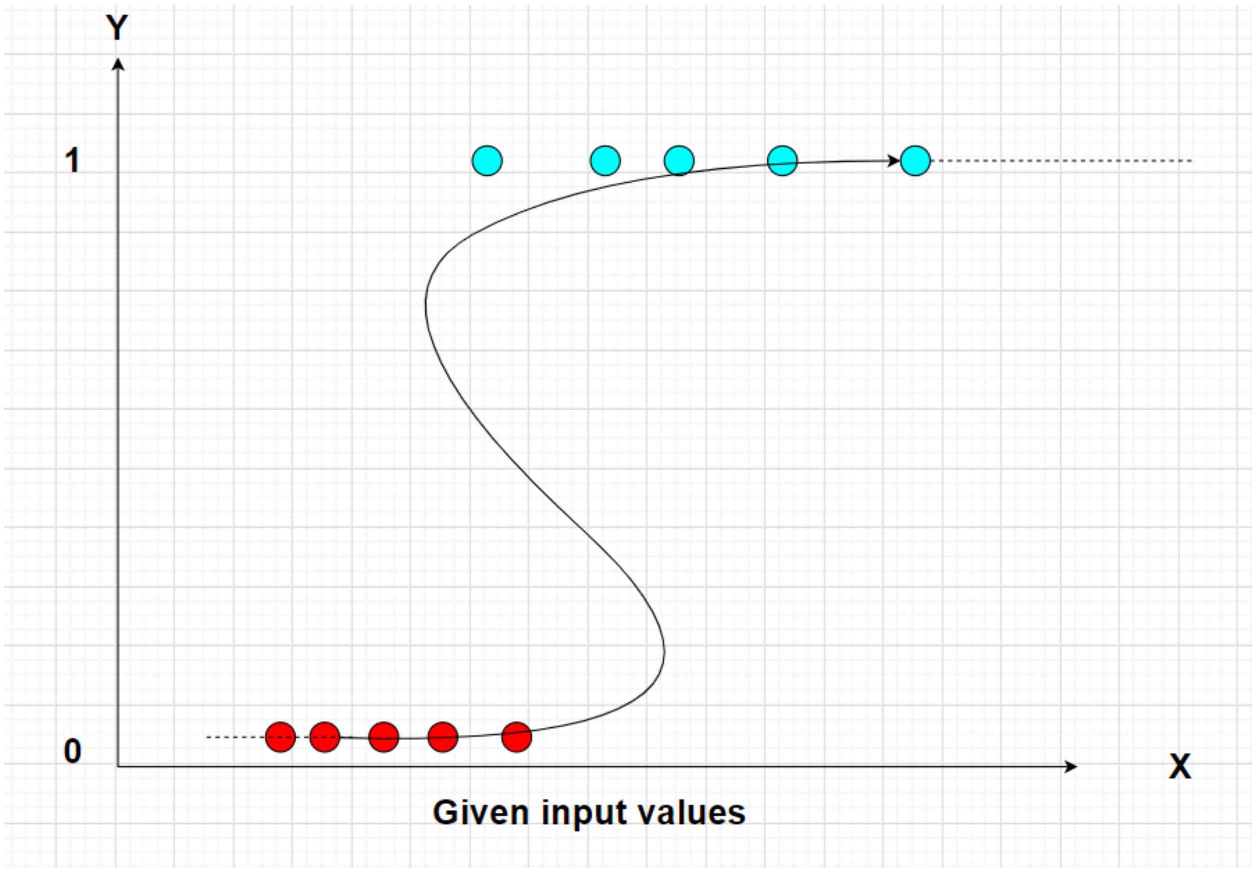
This function is converts input into a range of 0 to 1. Which helps with the classification process.

From linear regression we know that,

$$Y = \alpha + \beta . X$$

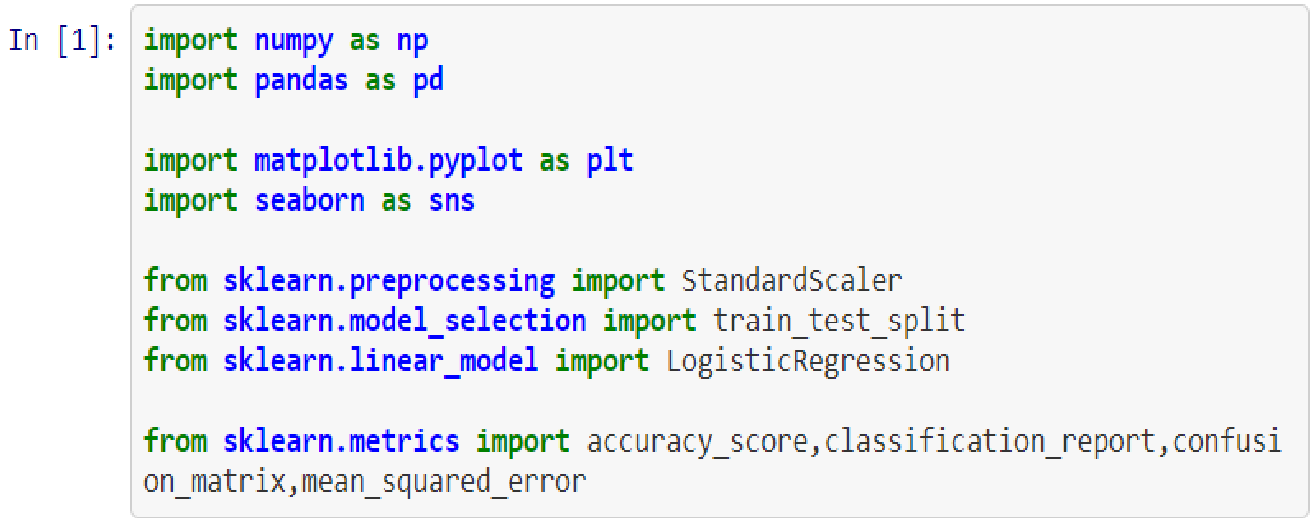
When this equation is feed to the sigmoid function we will get,

$$\mathbf{Y} = \frac{\mathbf{1}}{\mathbf{1}+ \boldsymbol{e}^{-(\boldsymbol{\alpha}+ \boldsymbol{\beta}.\boldsymbol{X})}}$$



As we can see in the graph that after using the sigmoid function we get an S shaped line. This line is the regression line for logistic regression. When raw data is given, the value of X is passed to the sigmoid function for logistic regression. This way the output value stays within the range of 0 and 1. Then conditions are added such as, if Y > 0.5 then categorize as 1 otherwise 0. This way all the categorical data are classified. This is how the logistic regression works.

Predicting client default by using logistic regression code explanation



```
In [1]: import numpy as np
        import pandas as pd

        import matplotlib.pyplot as plt
        import seaborn as sns

        from sklearn.preprocessing import StandardScaler
        from sklearn.model_selection import train_test_split
        from sklearn.linear_model import LogisticRegression

        from sklearn.metrics import accuracy_score,classification_report,confusi
        on_matrix,mean_squared_error
```

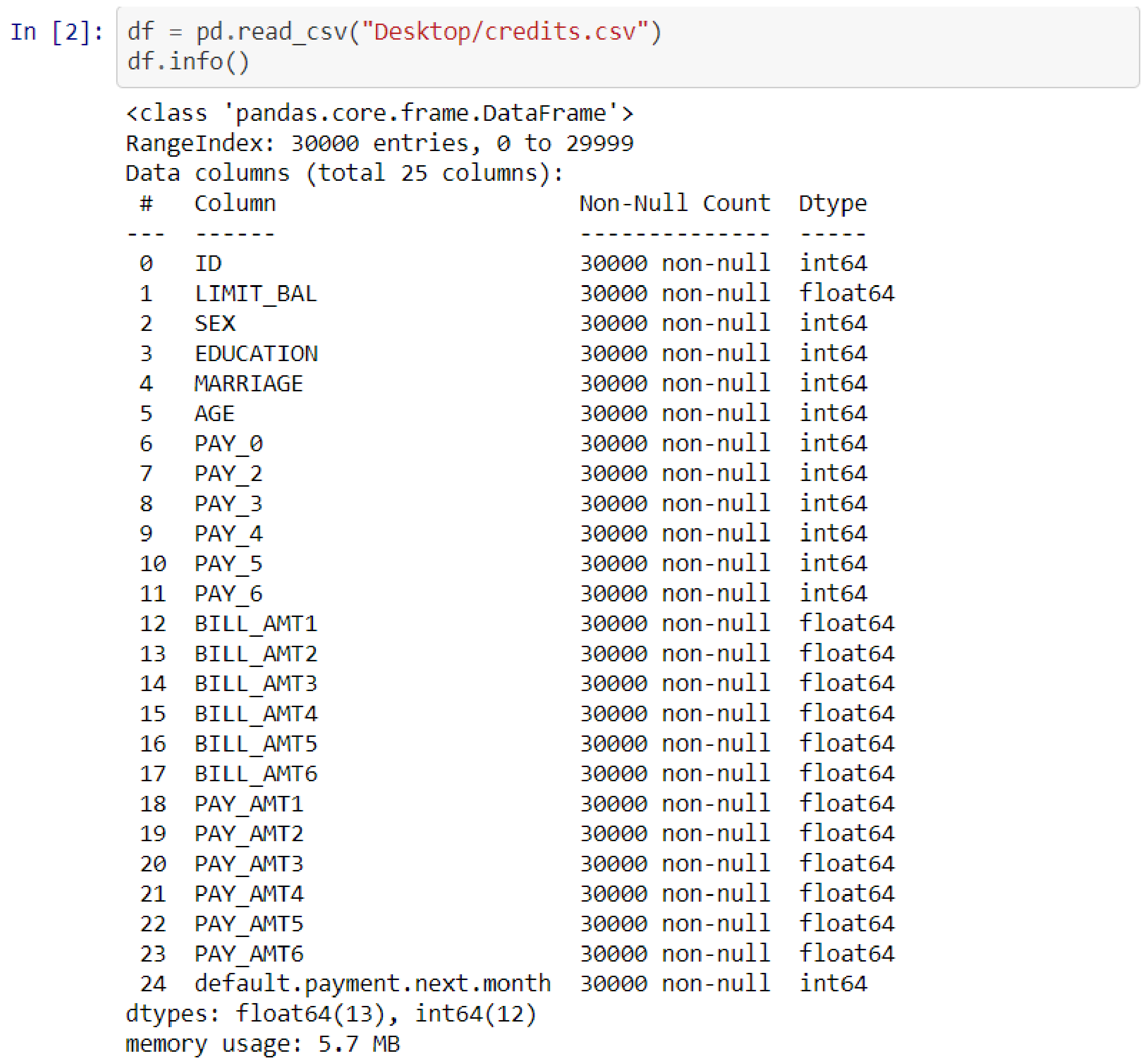
First all the necessary libraries must be imported:

numpy is used for mathematical operations.

pandas is used for data manipulation and analysis.

matplotlib and **seaborn** are used for data visualization.

Finally, multiple functions from the **sklearn** library. These functions will help us to standardize data, train and test the data, preform logistic regression and evaluate the final machine learning model.

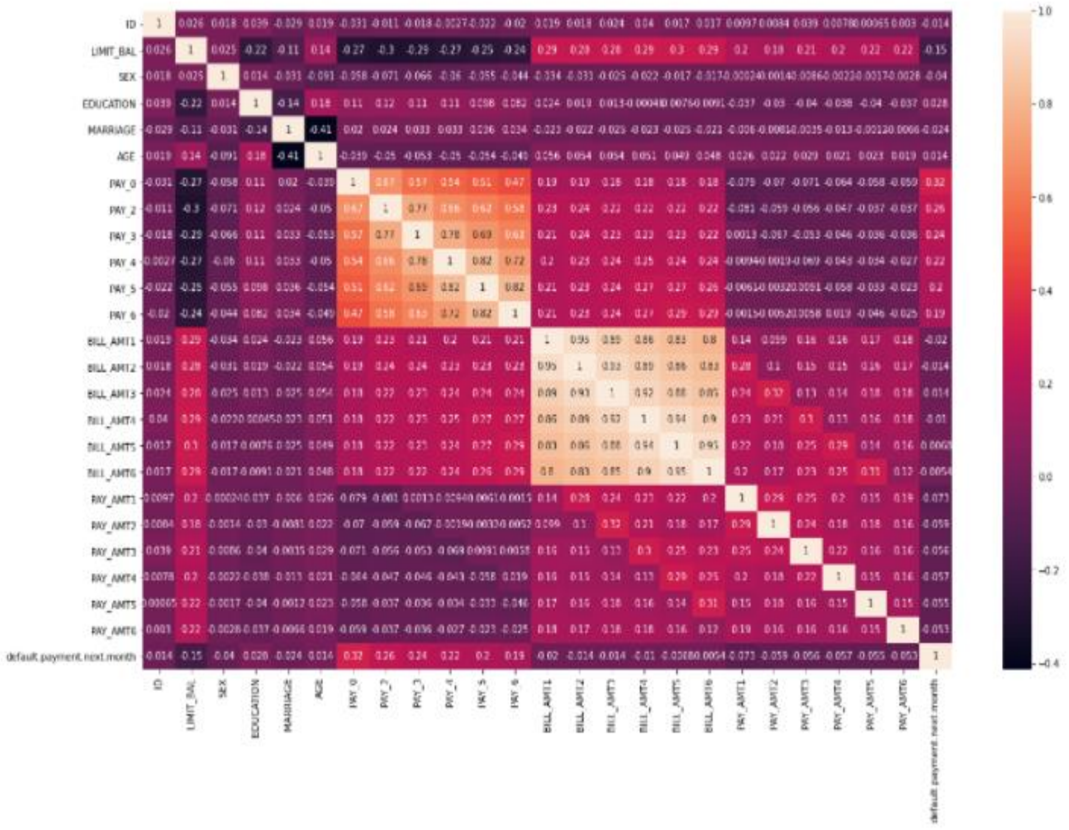


```
In [2]: df = pd.read_csv("Desktop/credits.csv")
        df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30000 entries, 0 to 29999
Data columns (total 25 columns):
 #   Column                      Non-Null Count  Dtype  
---  ------                      --------------  -----  
 0   ID                          30000 non-null  int64  
 1   LIMIT_BAL                   30000 non-null  float64
 2   SEX                         30000 non-null  int64  
 3   EDUCATION                   30000 non-null  int64  
 4   MARRIAGE                    30000 non-null  int64  
 5   AGE                         30000 non-null  int64  
 6   PAY_0                       30000 non-null  int64  
 7   PAY_2                       30000 non-null  int64  
 8   PAY_3                       30000 non-null  int64  
 9   PAY_4                       30000 non-null  int64  
 10  PAY_5                       30000 non-null  int64  
 11  PAY_6                       30000 non-null  int64  
 12  BILL_AMT1                   30000 non-null  float64
 13  BILL_AMT2                   30000 non-null  float64
 14  BILL_AMT3                   30000 non-null  float64
 15  BILL_AMT4                   30000 non-null  float64
 16  BILL_AMT5                   30000 non-null  float64
 17  BILL_AMT6                   30000 non-null  float64
 18  PAY_AMT1                    30000 non-null  float64
 19  PAY_AMT2                    30000 non-null  float64
 20  PAY_AMT3                    30000 non-null  float64
 21  PAY_AMT4                    30000 non-null  float64
 22  PAY_AMT5                    30000 non-null  float64
 23  PAY_AMT6                    30000 non-null  float64
 24  default.payment.next.month  30000 non-null  int64  
dtypes: float64(13), int64(12)
memory usage: 5.7 MB
```

With **pd.read_csv** and **df.info()** we can read the raw dataset. Also we get the overall idea about the dataset. Like the number of columns, rows, types of data etc.

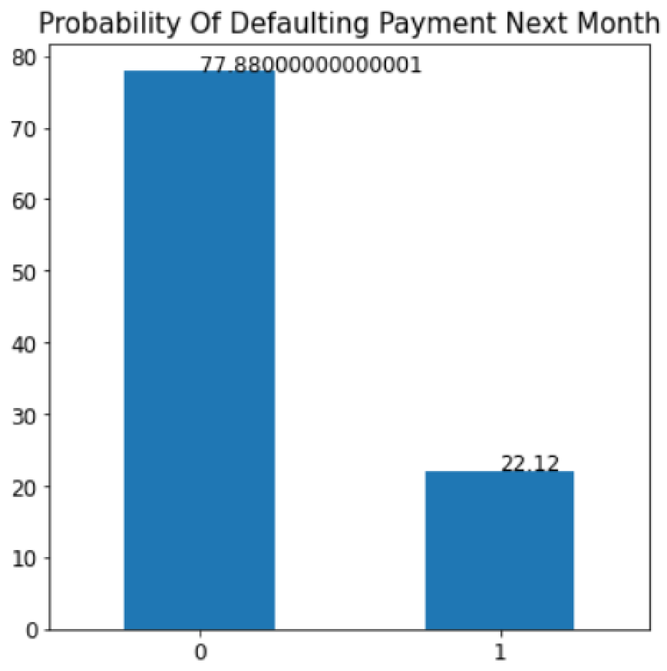
```
In [3]: corr = df.corr() # co relations columns

        plt.figure(figsize = (20, 12))
        sns.heatmap(corr, annot=True)
        plt.show()
```



This is known as a heat map. It visualizes the correlation between different attributes in the dataset. The **df.corr()** is used to find the correlations and the rest of the code is for heat map visualization.

```
In [4]: df.rename(columns={'default.payment.next.month':'def_pay'}, inplace=Tru
        e)
        df.rename(columns={'PAY_0':'PAY_1'}, inplace=True)
```

```
In [5]: def_cnt = (df.def_pay.value_counts(normalize=True)*100)
        def_cnt.plot.bar(figsize=(6,6))
        plt.xticks(fontsize=12, rotation=0)
        plt.yticks(fontsize=12)
        plt.title("Probability Of Defaulting Payment Next Month", fontsize=15)
        for x,y in zip([0,1],def_cnt):
            plt.text(x,y,y,fontsize=12)
        plt.show()
```



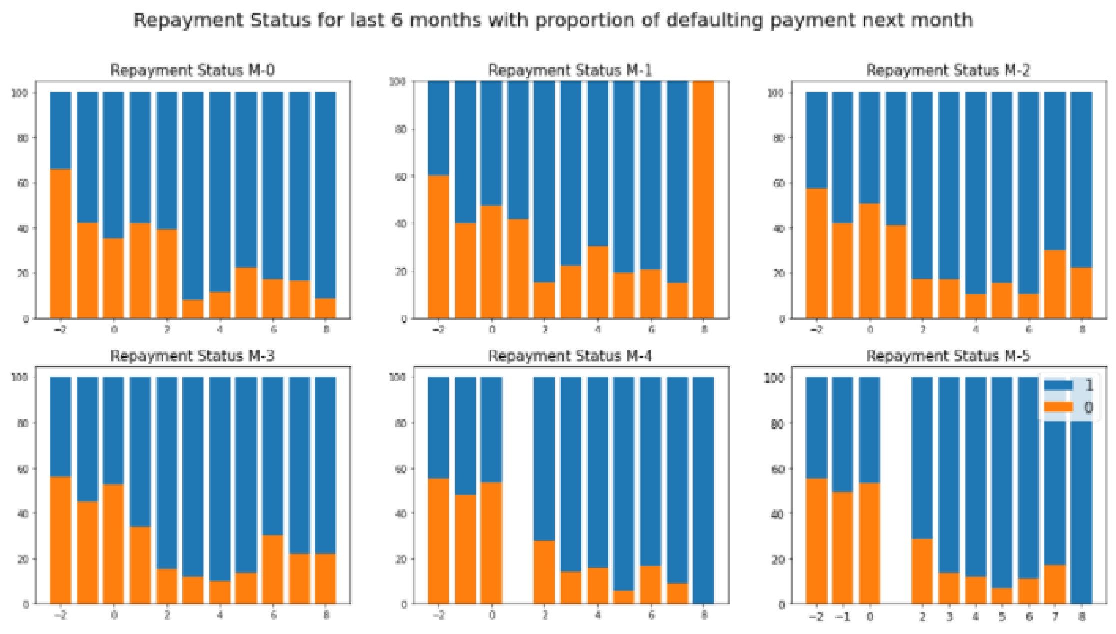
With **df.rename** function we can change the names of any column in the dataset. This is helpful in situations where the column name is too long or without proper indications.

The next part of the code is also for visualization. This visualization shows the percentage of different outcome of the raw data. Like above we can see that about 78% of the client probably will not default payment for next month and 22% will.

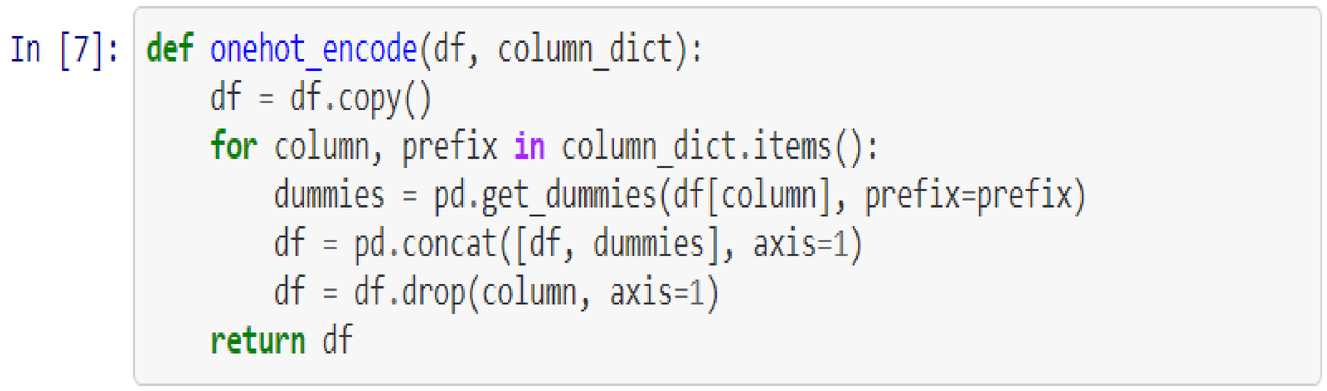
```
In [6]: plt.subplots(figsize=(20,10))

        ind = sorted(df.PAY_1.unique())
        pay_0 = (df.PAY_1[df['def_pay'] == 0].value_counts(normalize=True))
        pay_1 = (df.PAY_1[df['def_pay'] == 1].value_counts(normalize=True))
        total = pay_0.values+pay_1.values
        pay_0_prop = np.true_divide(pay_0, total)*100
        pay_1_prop = np.true_divide(pay_1, total)*100
        plt.subplot(231)
        plt.bar(ind, pay_1_prop, bottom=pay_0_prop, label='1')
        plt.bar(ind, pay_0_prop, label='0')
        plt.title("Repayment Status M-0", fontsize=15)

        ind = sorted(df.PAY_2.unique())
        pay_0 = (df.PAY_2[df['def_pay'] == 0].value_counts(normalize=True))
        pay_1 = (df.PAY_2[df['def_pay'] == 1].value_counts(normalize=True))
        for i in pay_0.index:
            if i not in pay_1.index:
                pay_1[i]=0
        total = pay_0.values+pay_1.values
        pay_0_prop = np.true_divide(pay_0, total)*100
        pay_1_prop = np.true_divide(pay_1, total)*100
        plt.subplot(232)
        plt.bar(ind, pay_1_prop, bottom=pay_0_prop, label='1')
        plt.bar(ind, pay_0_prop, label='0')
        plt.title("Repayment Status M-1", fontsize=15)

        ind = sorted(df.PAY_3.unique())
        pay_0 = (df.PAY_3[df['def_pay'] == 0].value_counts(normalize=True))
        pay_1 = (df.PAY_3[df['def_pay'] == 1].value_counts(normalize=True))
        for i in pay_0.index:
            if i not in pay_1.index:
                pay_1[i]=0
        total = pay_0.values+pay_1.values
        pay_0_prop = np.true_divide(pay_0, total)*100
        pay_1_prop = np.true_divide(pay_1, total)*100
        plt.subplot(233)
        plt.bar(ind, pay_1_prop, bottom=pay_0_prop, label='1')
        plt.bar(ind, pay_0_prop, label='0')
        plt.title("Repayment Status M-2", fontsize=15)
```

```
ind = sorted(df.PAY_4.unique())
pay_0 = (df.PAY_4[df['def_pay'] == 0].value_counts(normalize=True))
pay_1 = (df.PAY_4[df['def_pay'] == 1].value_counts(normalize=True))
for i in pay_0.index:
    if i not in pay_1.index:
        pay_1[i]=0
total = pay_0.values+pay_1.values
pay_0_prop = np.true_divide(pay_0, total)*100
pay_1_prop = np.true_divide(pay_1, total)*100
plt.subplot(234)
plt.bar(ind, pay_1_prop, bottom=pay_0_prop, label='1')
plt.bar(ind, pay_0_prop, label='0')
plt.title("Repayment Status M-3", fontsize=15)

ind = sorted(df.PAY_5.unique())
pay_0 = (df.PAY_5[df['def_pay'] == 0].value_counts(normalize=True))
pay_1 = (df.PAY_5[df['def_pay'] == 1].value_counts(normalize=True))
for i in pay_0.index:
    if i not in pay_1.index:
        pay_1[i]=0
for i in pay_1.index:
    if i not in pay_0.index:
        pay_0[i]=0
total = pay_0.values+pay_1.values
pay_0_prop = np.true_divide(pay_0, total)*100
pay_1_prop = np.true_divide(pay_1, total)*100
plt.subplot(235)
plt.bar(ind, pay_1_prop, bottom=pay_0_prop, label='1')
plt.bar(ind, pay_0_prop, label='0')
plt.title("Repayment Status M-4", fontsize=15)
```

```
ind = sorted(df.PAY_6.unique())
pay_0 = (df.PAY_6[df['def_pay'] == 0].value_counts(normalize=True))
pay_1 = (df.PAY_6[df['def_pay'] == 1].value_counts(normalize=True))
for i in pay_0.index:
    if i not in pay_1.index:
        pay_1[i]=0
for i in pay_1.index:
    if i not in pay_0.index:
        pay_0[i]=0
total = pay_0.values+pay_1.values
pay_0_prop = np.true_divide(pay_0, total)*100
pay_1_prop = np.true_divide(pay_1, total)*100
plt.subplot(236)
plt.bar(ind, pay_1_prop, bottom=pay_0_prop, label='1')
plt.bar(ind, pay_0_prop, label='0')
plt.title("Repayment Status M-5", fontsize=15)

plt.xticks(ind, fontsize=12)
plt.yticks(fontsize=12)
plt.legend(loc="upper right", fontsize=15)
plt.suptitle("Repayment Status for last 6 months with proportion of defa
ulting payment next month", fontsize=20)

plt.show()
```

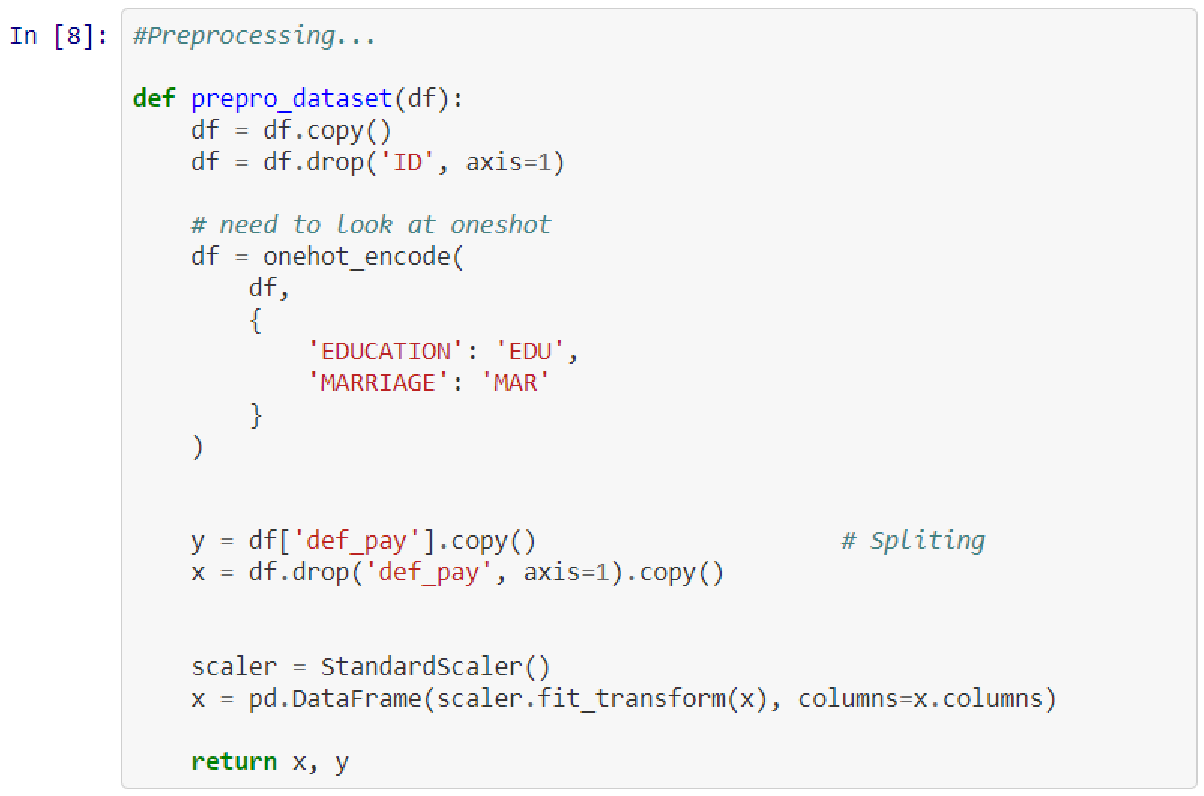


This is a visualization of client default payment proposition compared with the repayment status of last 6 months. These are some of the ways of visualization. This is necessary to analyze the dataset and make decisions like- which attributes are useful and which are not, which training model to use, is standardization necessary or not etc.



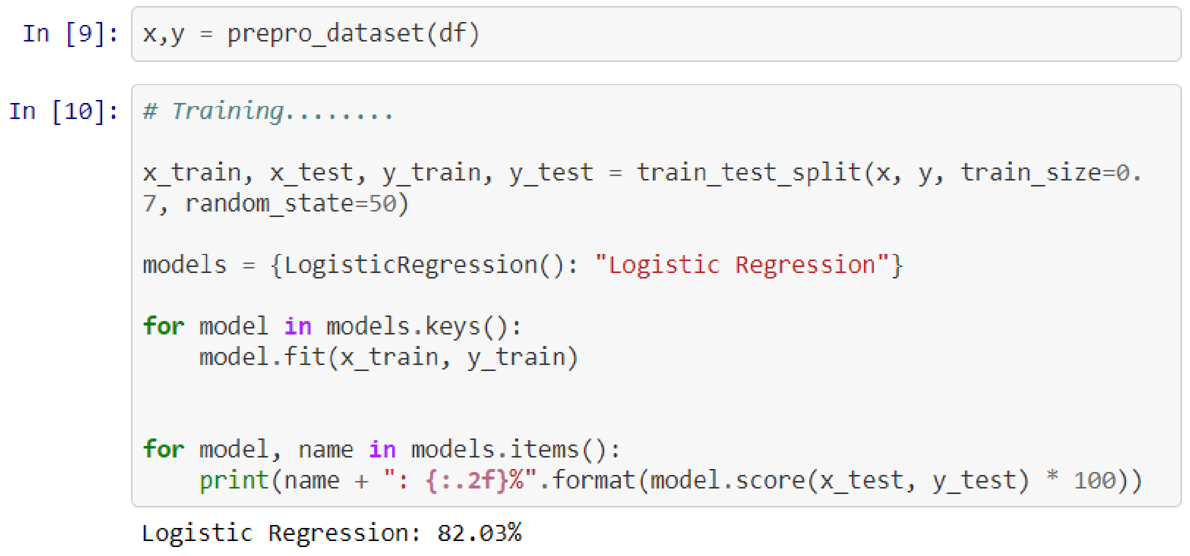
```
In [7]: def onehot_encode(df, column_dict):
            df = df.copy()
            for column, prefix in column_dict.items():
                dummies = pd.get_dummies(df[column], prefix=prefix)
                df = pd.concat([df, dummies], axis=1)
                df = df.drop(column, axis=1)
            return df
```

The **onehot_encode** function helps us to create dummy data for categorical data. With the help of **dp.get_dummies()** each category of the categorical data gets their own column where the data of that category is presented as 1 and others are presented as 0. With **pd.concat()** these new columns are added to the main data set copy created by **df.copy()**. Then with **df.drop()** is used to drop the original categorical data column. The for loop is adding dummies with prefix and **items()** method is used to return the list with all dictionary keys with values. When all the data are processed and added to **df** the **onehot_encode** function returns **df**.



```
In [8]: #Preprocessing...

        def prepro_dataset(df):
            df = df.copy()
            df = df.drop('ID', axis=1)

            # need to look at oneshot
            df = onehot_encode(
                df,
                {
                    'EDUCATION': 'EDU',
                    'MARRIAGE': 'MAR'
                }
            )


            y = df['def_pay'].copy()                      # Spliting
            x = df.drop('def_pay', axis=1).copy()


            scaler = StandardScaler()
            x = pd.DataFrame(scaler.fit_transform(x), columns=x.columns)

            return x, y
```

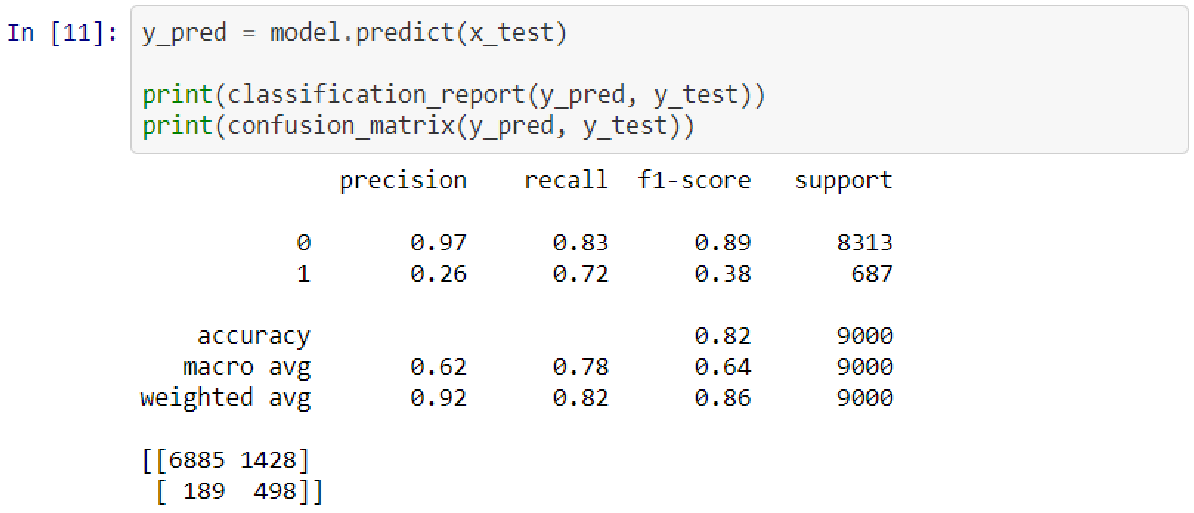
Prepro_dataset function takes a data frame as input and processes it before training phase. In this function we drop all the unnecessary attributes. Then **onehot_encode** function is called and a data frame with a dictionary containing the categorical data column and prefixes. Then X and Y are selected. Then with the help of **StandarScaler()** all values in the data set is transformed into similar types of values which makes the training and testing phase easy.



```
In [9]: x,y = prepro_dataset(df)

In [10]: # Training........

         x_train, x_test, y_train, y_test = train_test_split(x, y, train_size=0.
         7, random_state=50)

         models = {LogisticRegression(): "Logistic Regression"}

         for model in models.keys():
             model.fit(x_train, y_train)


         for model, name in models.items():
             print(name + ": {:.2f}%".format(model.score(x_test, y_test) * 100))

         Logistic Regression: 82.03%
```

Then we call the **Prepro_dataset** function and pass the data frame. This function retunes X and Y which are preprocessed and ready for training. Then with **train_test_split** we split the data into training set and testing set. Then model is created by calling **LogisticRegression()** and with **model.fit(x_train,y_train)** the model is trained.

By passing the test set to **model.score()** we can see the accuracy rate of the model.



```
In [11]: y_pred = model.predict(x_test)

         print(classification_report(y_pred, y_test))
         print(confusion_matrix(y_pred, y_test))
```

```
              precision    recall  f1-score   support

           0       0.97      0.83      0.89      8313
           1       0.26      0.72      0.38       687

    accuracy                           0.82      9000
   macro avg       0.62      0.78      0.64      9000
weighted avg       0.92      0.82      0.86      9000

[[6885 1428]
 [ 189  498]]
```

Finally with **model.predict()** and **classification_report()** we can see the precision , recall f1-score and support. And with **confusion_matrix()** we can calculate the confusion matrix.

Here,

precision = How many are correctly classified among that class.

recall = How many of this class you find over the whole number of element of this class. The recall is intuitively the ability of the classifier to find all the positive samples.

f1-score = The harmonic mean between precision & recall.

support = The number of occurrence of the given class in the dataset.

accuracy = Number of correct predictions among the total number of predictions.

For the confusion matrix:

	Predicted NO	**Predicted YES**
Actual NO	**TN**	**FP**
Actual YES	**FN**	**TP**

Here,

True positives (TP): Model predicted yes and in reality it was yes.

True negatives (TN): Model predicted no and in reality it was no.

False positives (FP): Model predicted yes and in reality it was no.

False negatives (FN): Model predicted no and in reality it was yes.