Netaji Subhas University of Technology

Sector – 3, Dwarka, Delhi – 110078

Practical File

Machine Learning – COCSC17

Name – Adarsh Kumar

Roll No. – 2020UCO1663

Dept. – Computer Engineering

Section – 3

INDEX

Experiment-1

Aim: Implement Linear Regression in Python using Jupyter Notebook.

1) Import necessary libraries

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

2) Load Dataset

In [2]:
```
data = pd.read_csv("car data.csv")
```

In [3]:
```
data.head() # checking the first five rows from the dataset
```

Out[3]:

	Car_Name	Year	Selling_Price	Present_Price	Kms_Driven	Fuel_Type	Seller_Type	Transmission	Owner
0	ritz	2014	3.35	5.59	27000	Petrol	Dealer	Manual	0
1	sx4	2013	4.75	9.54	43000	Diesel	Dealer	Manual	0
2	ciaz	2017	7.25	9.85	6900	Petrol	Dealer	Manual	0
3	wagon r	2011	2.85	4.15	5200	Petrol	Dealer	Manual	0
4	swift	2014	4.60	6.87	42450	Diesel	Dealer	Manual	0

In [4]:
```
data.shape ## printing the no. of columns and rows of the dataframe
```

Out[4]:
```
(301, 9)
```

In [5]:
```
data.info() # printing the summary of the dataframe
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 301 entries, 0 to 300
Data columns (total 9 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   Car_Name       301 non-null    object
 1   Year           301 non-null    int64
 2   Selling_Price  301 non-null    float64
 3   Present_Price  301 non-null    float64
 4   Kms_Driven     301 non-null    int64
 5   Fuel_Type      301 non-null    object
 6   Seller_Type    301 non-null    object
 7   Transmission   301 non-null    object
 8   Owner          301 non-null    int64
dtypes: float64(2), int64(3), object(4)
memory usage: 21.3+ KB
```

In [6]:
```
data.isnull().sum() # finding the count of missing values from different columns
```

Out[6]:
```
Car_Name         0
Year             0
Selling_Price    0
Present_Price    0
Kms_Driven       0
Fuel_Type        0
Seller_Type      0
Transmission     0
Owner            0
dtype: int64
```

In [7]:
```
data.describe() # Statistical Information
```

Out[7]:

	Year	Selling_Price	Present_Price	Kms_Driven	Owner
count	301.000000	301.000000	301.000000	301.000000	301.000000
mean	2013.627907	4.661296	7.628472	36947.205980	0.043189
std	2.891554	5.082812	8.644115	38886.883882	0.247915
min	2003.000000	0.100000	0.320000	500.000000	0.000000
25%	2012.000000	0.900000	1.200000	15000.000000	0.000000
50%	2014.000000	3.600000	6.400000	32000.000000	0.000000
75%	2016.000000	6.000000	9.900000	48767.000000	0.000000
max	2018.000000	35.000000	92.600000	500000.000000	3.000000

In [8]:
```
data= data.drop(columns = ['Car_Name'])
data.head()
```

Out[8]:

	Year	Selling_Price	Present_Price	Kms_Driven	Fuel_Type	Seller_Type	Transmission	Owner
0	2014	3.35	5.59	27000	Petrol	Dealer	Manual	0
1	2013	4.75	9.54	43000	Diesel	Dealer	Manual	0
2	2017	7.25	9.85	6900	Petrol	Dealer	Manual	0
3	2011	2.85	4.15	5200	Petrol	Dealer	Manual	0
4	2014	4.60	6.87	42450	Diesel	Dealer	Manual	0

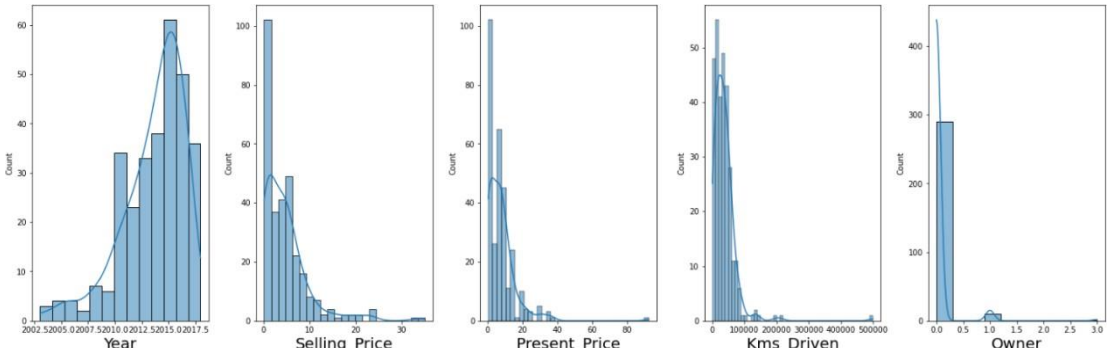
In [11]:
```
Categorical_features = [col for col in data.columns if data[col].dtype == 'O']
Categorical_features
```

Out[11]:
```
['Fuel_Type', 'Seller_Type', 'Transmission']
```

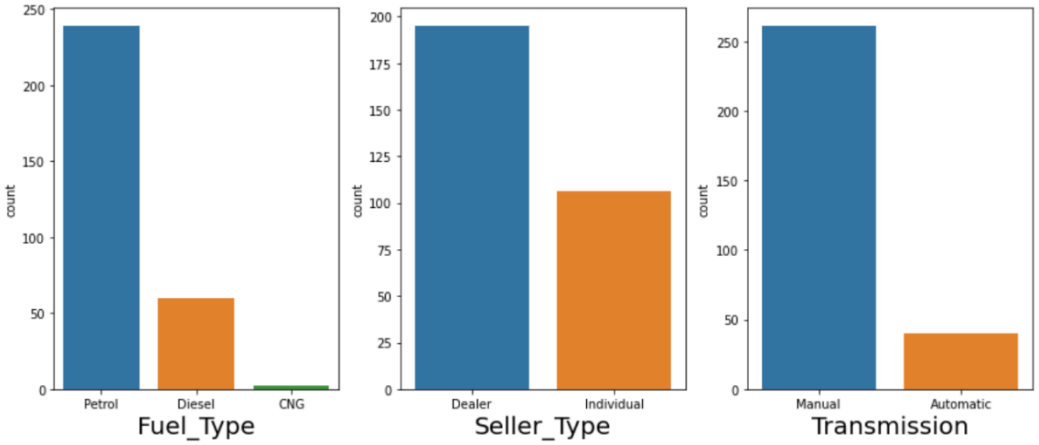
In [12]:
```
numerical_features = [col for col in data.columns if data[col].dtype != 'O']
numerical_features
```

Out[12]:
```
['Year', 'Selling_Price', 'Present_Price', 'Kms_Driven', 'Owner']
```

In [18]:
```
# let's see how numerical_features is distributed for every column
plt.figure(figsize=(20,25), facecolor='white')
plotnumber = 1

for column in numerical_features:
    if plotnumber<=5 :
        ax = plt.subplot(4,5,plotnumber)
        sns.histplot(data[column],kde = True)
        plt.xlabel(column,fontsize=20)
    plotnumber+=1
plt.tight_layout()
```



In [22]:
```
# let's see how data is distributed for every column
plt.figure(figsize=(20,25), facecolor='white')
plotnumber = 1

for column in Categorical_features:
    if plotnumber<=3 :
        ax = plt.subplot(5,5,plotnumber)
        sns.countplot(x = column,data = data)
        plt.xlabel(column,fontsize=20)
    plotnumber+=1
plt.tight_layout()
```



In [23]:
```
data = pd.get_dummies(data)
data.head()
```

Out[23]:

	Year	Selling_Price	Present_Price	Kms_Driven	Owner	Fuel_Type_CNG	Fuel_Type_Diesel	Fuel_Type_Petrol	Seller_Type_Dealer	Seller_Type_Individual	Tran
0	2014	3.35	5.59	27000	0	0	0	1	1	0	
1	2013	4.75	9.54	43000	0	0	1	0	1	0	
2	2017	7.25	9.85	6900	0	0	0	1	1	0	
3	2011	2.85	4.15	5200	0	0	0	1	1	0	
4	2014	4.60	6.87	42450	0	0	1	0	1	0	

In [24]:
```
# create X(Independent data) and y(dependent data)
feature = ["Year","Present_Price","Kms_Driven","Owner","Fuel_Type_CNG","Fuel_Type_Diesel","Seller_Type_Dealer","Transmission_
X = data[feature]
y = data.Selling_Price
```

In [25]:
```
X.head()
```

Out[25]:

	Year	Present_Price	Kms_Driven	Owner	Fuel_Type_CNG	Fuel_Type_Diesel	Seller_Type_Dealer	Transmission_Automatic
0	2014	5.59	27000	0	0	0	1	0
1	2013	9.54	43000	0	0	1	1	0
2	2017	9.85	6900	0	0	0	1	0
3	2011	4.15	5200	0	0	0	1	0
4	2014	6.87	42450	0	0	1	1	0

In [26]:
```
y.head()
```

Out[26]:
```
0    3.35
1    4.75
2    7.25
3    2.85
4    4.60
Name: Selling_Price, dtype: float64
```

In [27]:
```
from sklearn.model_selection import train_test_split
```

In [29]:
```
x_train, x_test, y_train, y_test = train_test_split(X,y,random_state=5)
```

In [30]:
```
x_train.shape , y_train.shape , x_test.shape , y_test.shape
```

Out[30]:
```
((225, 8), (225,), (76, 8), (76,))
```

Implementation of LinearRegression Model

In [31]:
```
from sklearn.linear_model import LinearRegression
```

In [32]:
```
lr = LinearRegression()
lr.fit(x_train,y_train)
```

Out[32]:
```
LinearRegression()
```

In [33]:
```
prediction = lr.predict(x_test)
```

In [35]:
```
lr.score(x_train,y_train) ## training score
```

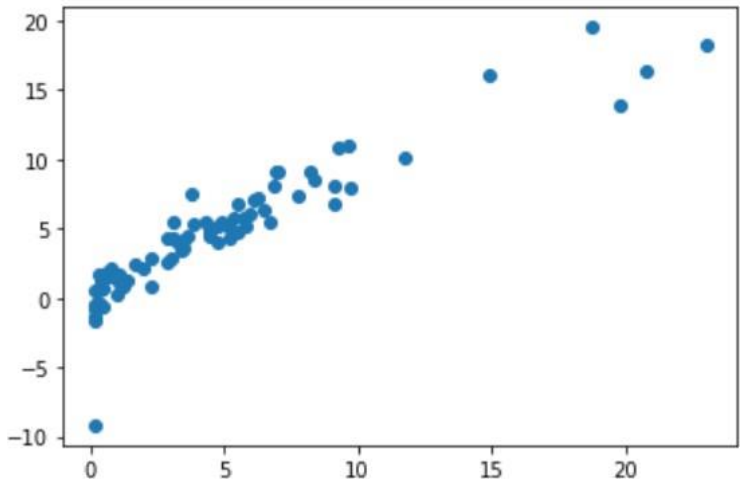
Out[35]:
```
0.8799908666232623
```

In [37]:
```
lr.score(x_test,y_test) ## test score
```

Out[37]:
```
0.8608525898495389
```

In [38]:
```
plt.scatter(y_test,prediction) ## plot y_test v/s prediction value of model
```

Out[38]:
```
<matplotlib.collections.PathCollection at 0x2ac078ce520>
```



Evolution Metrics of model

```
In [41]: from sklearn import metrics
```

```
In [42]: print('MAE : ',metrics.mean_absolute_error(y_test,prediction))
         print('MSE : ',metrics.mean_squared_error(y_test,prediction))
         print('RMSE : ',np.sqrt(metrics.mean_squared_error(y_test,prediction)))
         print('R squared : ',metrics.r2_score(y_test,prediction))
```

```
MAE :  1.1409961087856992
MSE :  3.2815963416670617
RMSE :  1.8115176901336243
R squared :  0.8608525898495389
```

Implementation of Regularization Model

1) LASSO (L1-Norm)

```
In [44]: from sklearn.linear_model  import Ridge,Lasso,RidgeCV, LassoCV, ElasticNet, ElasticNetCV
```

```
In [54]: # LassoCV will return best alpha and coefficients after performing 10 cross validations
         lassocv = LassoCV(alphas = None,cv =10, max_iter = 100000,normalize = True)
         lassocv.fit(x_train, y_train)
```

```
Out[54]: LassoCV(cv=10, max_iter=100000, normalize=True)
```

```
In [55]: # best alpha parameter
         alpha = lassocv.alpha_
         alpha
```

```
Out[55]: 0.014905096676743052
```

```
In [56]: #now that we have best parameter, let's use Lasso regression and see how well our data has fitted before
         lasso_reg = Lasso(alpha)
         lasso_reg.fit(x_train, y_train)
```

```
Out[56]: Lasso(alpha=0.014905096676743052)
```

```
In [57]: prediction = lasso_reg.predict(x_test)
```

```
In [58]: print('MAE : ',metrics.mean_absolute_error(y_test,prediction))
         print('MSE : ',metrics.mean_squared_error(y_test,prediction))
         print('RMSE : ',np.sqrt(metrics.mean_squared_error(y_test,prediction)))
         print('R squared : ',metrics.r2_score(y_test,prediction))
```

2) Ridge(L2-Norm)

In [59]:
```
# RidgeCV will return best alpha and coefficients after performing 10 cross validations. We will pass an array of

alphas = np.random.uniform(low=0, high=10, size=(50,))
ridgecv = RidgeCV(alphas = alphas,cv=10,normalize = True)
ridgecv.fit(x_train, y_train)
```

Out[59]:
```
RidgeCV(alphas=array([2.6906836 , 2.59960857, 8.12639411, 1.61598335, 4.16685595,
       5.09373017, 2.06237126, 9.93194163, 5.84637547, 6.42196261,
       3.63917991, 8.05612584, 2.88051393, 1.63675005, 9.90353569,
       8.80318691, 8.25077244, 5.88203548, 9.32860793, 1.22842734,
       8.39177915, 9.06553571, 7.1212448 , 5.92935654, 4.9152103 ,
       1.73100406, 5.39842325, 0.78988506, 5.13973145, 7.37094895,
       6.01757596, 5.3938236 , 8.98625131, 2.96162805, 3.68449987,
       2.6926045 , 3.17235751, 0.92019116, 9.19378148, 4.25682258,
       5.86813771, 1.73632283, 3.32712328, 3.60451109, 1.29938048,
       8.50864923, 9.59323106, 5.00713259, 9.75804357, 8.6867954 ]),
        cv=10, normalize=True)
```

In [60]:
```
ridgecv.alpha_
```

Out[60]:
```
0.789885059103369
```

In [61]:
```
ridge_model = Ridge(alpha=ridgecv.alpha_)
ridge_model.fit(x_train, y_train)
```

Out[61]:
```
Ridge(alpha=0.789885059103369)
```

Out[61]:
```
Ridge(alpha=0.789885059103369)
```

In [63]:
```
prediction = ridge_model.predict(x_test)
```

In [64]:
```
print('MAE : ',metrics.mean_absolute_error(y_test,prediction))
print('MSE : ',metrics.mean_squared_error(y_test,prediction))
print('RMSE : ',np.sqrt(metrics.mean_squared_error(y_test,prediction)))
print('R squared : ',metrics.r2_score(y_test,prediction))
```

```
MAE :  1.1338944072792678
MSE :  3.2712447136861806
RMSE :  1.8086582633781818
R squared :  0.8612915232448782
```

3) Elastic Net

In [71]:
```
elasticCV = ElasticNetCV(alphas = None, cv =10, max_iter = 100000,normalize = True)
elasticCV.fit(x_train, y_train)
```

Out[71]:
```
ElasticNetCV(cv=10, max_iter=100000, normalize=True)
```

In [72]:
```
elasticCV.alpha_
```

Out[72]:
```
0.0014836603477759683
```

In [73]:
```
# l1_ration gives how close the model is to L1 regularization, below value indicates we are giving equal preference to L1 a
elasticCV.l1_ratio
```

Out[73]:
```
0.5
```

In [74]:
```
elasticnet_reg = ElasticNet(alpha = elasticCV.alpha_,l1_ratio=0.5)
elasticnet_reg.fit(x_train, y_train)
```

Out[74]:
```
ElasticNet(alpha=0.0014836603477759683)
```

In [75]:
```
prediction = elasticnet_reg.predict(x_test)
```

In [76]:
```
print('MAE : ',metrics.mean_absolute_error(y_test,prediction))
print('MSE : ',metrics.mean_squared_error(y_test,prediction))
print('RMSE : ',np.sqrt(metrics.mean_squared_error(y_test,prediction)))
print('R squared : ',metrics.r2_score(y_test,prediction))
```

Prediction

In [82]:
```
Year = int(input("Enter Year of Purchase : "))
Present_Price = int(input("Enter Showroom Price of your car : "))
Kms_Driven = int(input("How many kilometer drive : "))
Owner = int(input("How much owners previously the car had? (0,1 or 2) : "))
Fuel_Type = str(input("What is fuel type? (CNG , DIESEL , PETROL) : "))
Seller_Type = str(input("Are u Dealer or Individual ? : "))
Transmission = str(input("Enter Transmission Type : (Automatic , Manual Car) : "))
```

```
Enter Year of Purchase : 2015
Enter Showroom Price of your car : 250000
How many kilometer drive : 100
How much owners previously the car had? (0,1 or 2) : 1
What is fuel type? (CNG , DIESEL , PETROL) : CNG
Are u Dealer or Individual ? : Dealer
Enter Transmission Type : (Automatic , Manual Car) : Manual Car
```

In [85]:
```
if Fuel_Type == "CNG" :
    Fuel_Type_CNG , Fuel_Type_Diesel = 1 , 0
elif Fuel_Type == "DIESEL":
    Fuel_Type_CNG , Fuel_Type_Diesel = 0 , 1
else:
    Fuel_Type_CNG , Fuel_Type_Diesel = 0 , 0
```

In [86]:
```
if Seller_Type == "Dealer":
    Seller_Type_Dealer = 1
else:
    Seller_Type_Dealer = 0
```

In [87]:
```
if Transmission == "Automatic":
    Transmission_Automatic = 1
else:
    Transmission_Automatic = 0
```

In [90]:
```
prediction_value = lr.predict([[Year,Present_Price,Kms_Driven,Owner,Fuel_Type_CNG,Fuel_Type_Diesel,Seller_Type_Dealer,Tr
print("Predicted value : ",prediction_value[0])
```

```
Predicted value :  110015.88909564455
```

Experiment-2

Aim: Implement Logistic Regression in Python using Jupyter Notebook.

Python Implementation

In [1]:
```
#Let's start with importing necessary libraries

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

In [2]:
```
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model  import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix, roc_curve, roc_auc_score
```

In [3]:
```
data = pd.read_csv("diabetes.csv") # Reading the Data
data.head() # top 5 rows
```

Out[3]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

In [4]:
```
data.tail() ## Last 5 rows
```

Out[4]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
763	10	101	76	48	180	32.9	0.171	63	0
764	2	122	70	27	0	36.8	0.340	27	0
765	5	121	72	23	112	26.2	0.245	30	0
766	1	126	60	0	0	30.1	0.349	47	1
767	1	93	70	31	0	30.4	0.315	23	0

Data Profiling :-

In [5]:
```
## Size of DataSet
data.shape
```

Out[5]:
```
(768, 9)
```

There are 768 no. of rows and 9 no. of columns.

In [6]:
```
## Number of missing values per column
data.isnull().sum()
```

Out[6]:
```
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

In [7]:
```
## Dataset Information
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
 #   Column                    Non-Null Count  Dtype  
---  ------                    --------------  -----  
 0   Pregnancies               768 non-null    int64  
 1   Glucose                   768 non-null    int64  
 2   BloodPressure             768 non-null    int64  
 3   SkinThickness             768 non-null    int64  
 4   Insulin                   768 non-null    int64  
 5   BMI                       768 non-null    float64
 6   DiabetesPedigreeFunction  768 non-null    float64
 7   Age                       768 non-null    int64  
 8   Outcome                   768 non-null    int64  
dtypes: float64(2), int64(7)
memory usage: 54.1 KB
```

In [4]:
```
## Number of categorical columns and their list
categorical = [column for column in data.columns if data[column].dtype == 'O']
len(categorical)
```

Out[4]: 0

In [5]:
```
## number of duplicate rows
duplicate_data = data[data.duplicated()]
duplicate_data.shape
```

Out[5]: (0, 9)

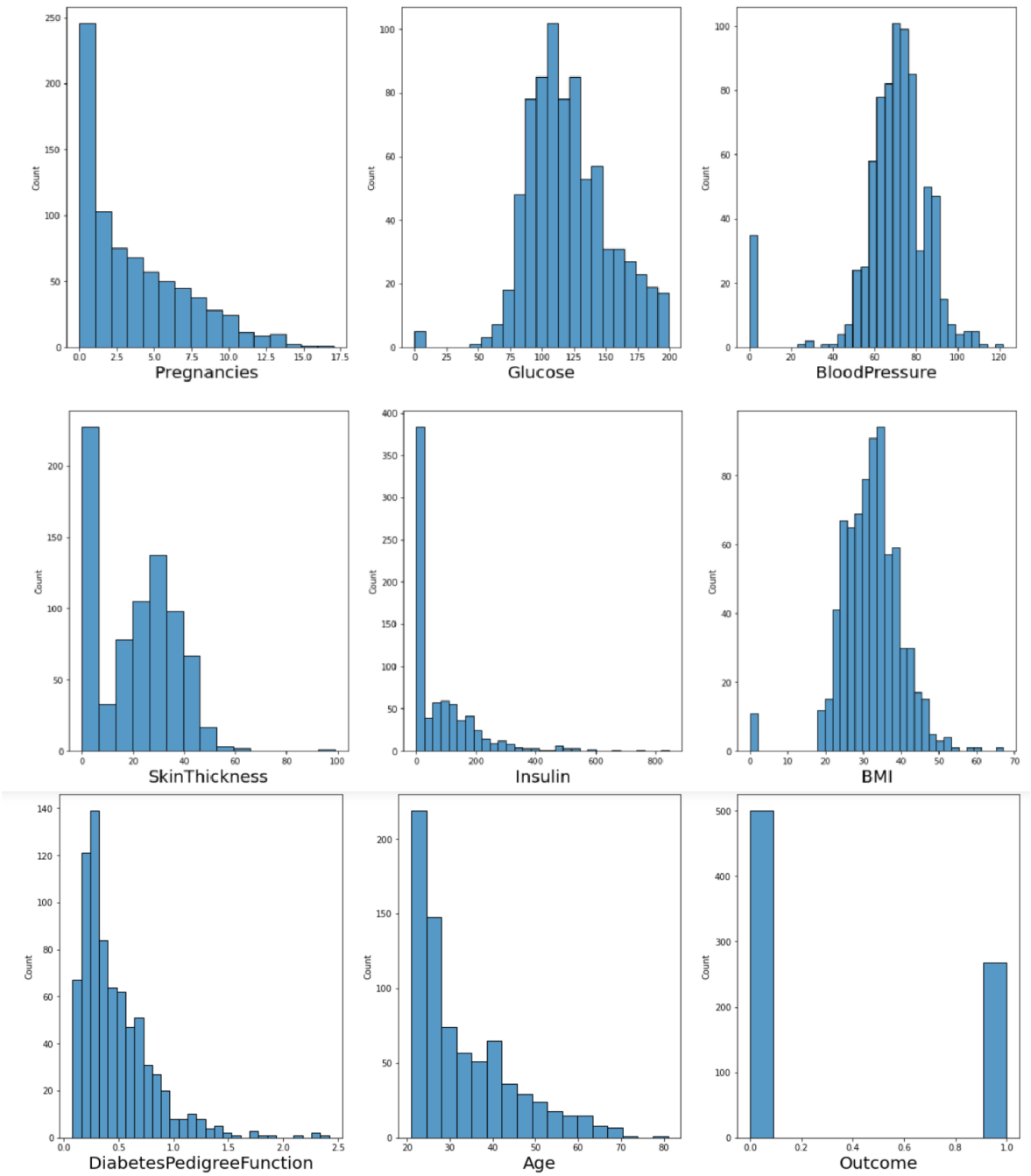
In [12]:
```
data.describe() ## Statistical Information from dataset
```

Out[12]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

Explotary Data Analysis :-

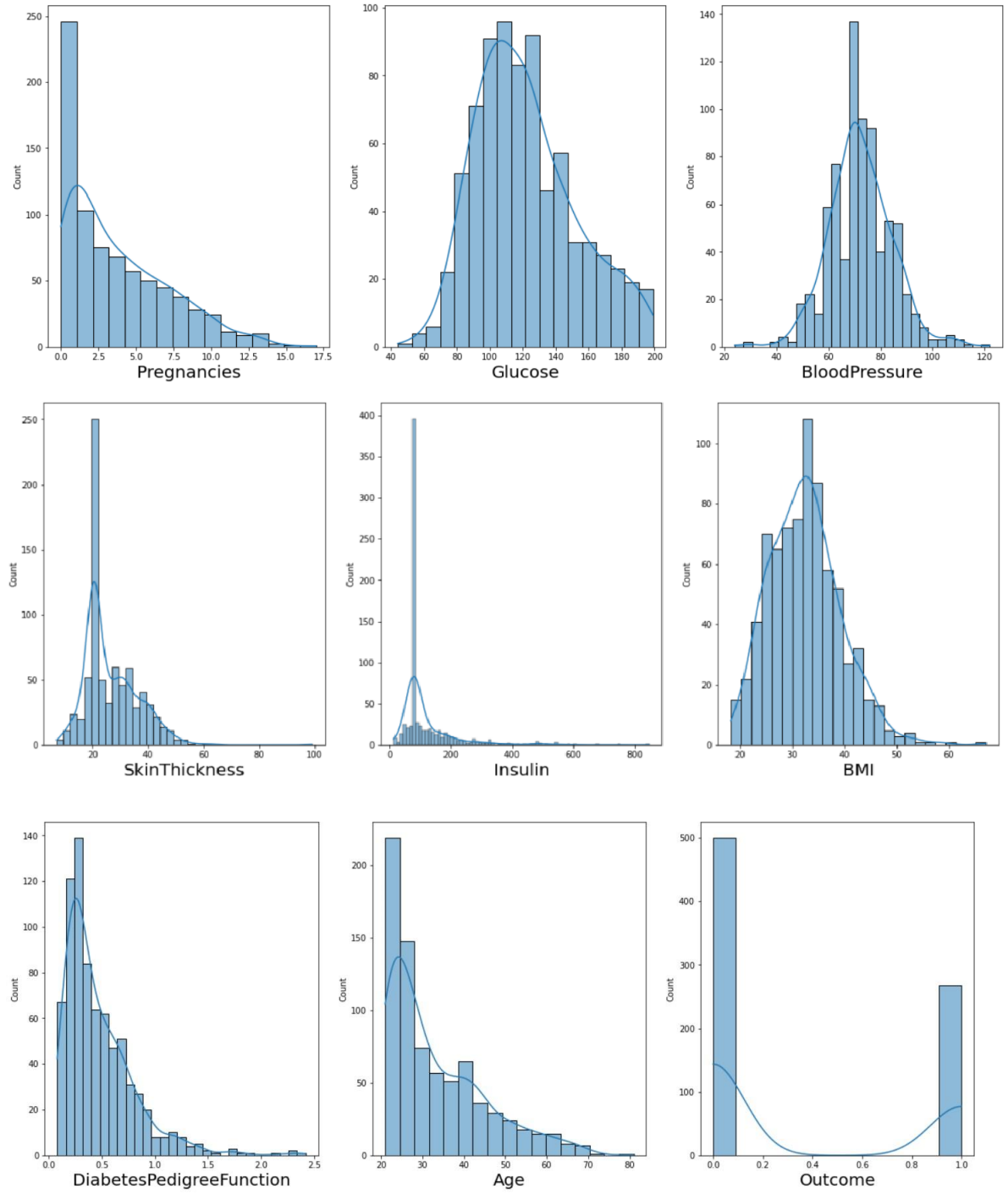
In [14]:
```
# let's see how data is distributed for every column
plt.figure(figsize=(20,25), facecolor='white')
plotnumber = 1

for column in data:
    if plotnumber<=9 :     # as there are 9 columns in the data
        ax = plt.subplot(3,3,plotnumber)
        sns.histplot(data[column])
        plt.xlabel(column,fontsize=20)
    plotnumber+=1
plt.show()
```



We can see there is some skewness in the data, let's deal with data.

In [15]:
```
# replacing zero values with the mean of the column
data['BMI'] = data['BMI'].replace(0,data['BMI'].mean())
data['BloodPressure'] = data['BloodPressure'].replace(0,data['BloodPressure'].mean())
data['Glucose'] = data['Glucose'].replace(0,data['Glucose'].mean())
data['Insulin'] = data['Insulin'].replace(0,data['Insulin'].mean())
data['SkinThickness'] = data['SkinThickness'].replace(0,data['SkinThickness'].mean())
```

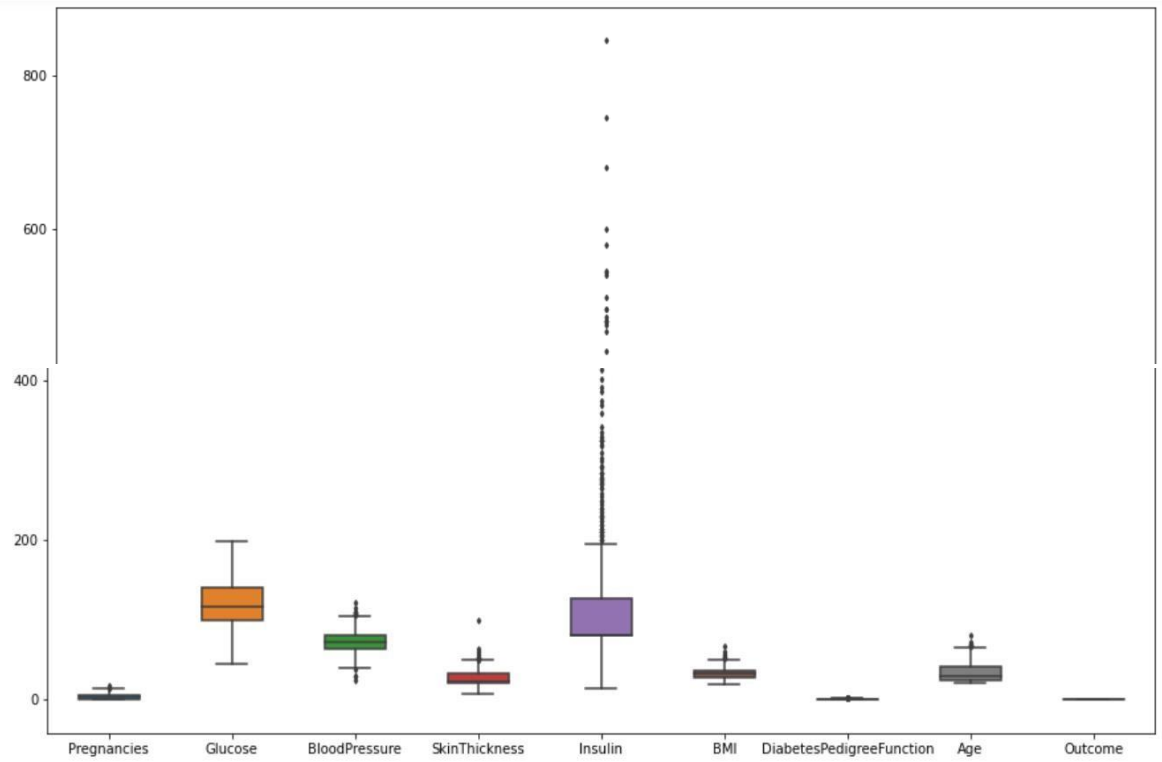
In [16]:
```
# let's see how data is distributed for every column
plt.figure(figsize=(20,25), facecolor='white')
plotnumber = 1

for column in data:
    if plotnumber<=9 :
        ax = plt.subplot(3,3,plotnumber)
        sns.histplot(data[column],kde=True)
        plt.xlabel(column,fontsize=20)
    plotnumber+=1
plt.show()
```



Now we have dealt with the 0 values and data looks better. But, there still are outliers present in some columns. Let's deal with them.

In [17]:
```
fig, ax = plt.subplots(figsize=(15,10))
sns.boxplot(data=data, width= 0.5,ax=ax,  fliersize=3)
```

Out[17]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x2adb3c914f0>
```



"BMI","Pregnancies","BloodPressure","SkinThickness","Insulin","Age" columns contain Outilers.

In [18]:
```
## Display percentage of Outilers
for k, v in data.items():
        q1 = v.quantile(0.25)
        q3 = v.quantile(0.75)
        inter_q = q3 - q1
        v_col = v[(v <= q1 - 1.5 * inter_q) | (v >= q3 + 1.5 * inter_q)]
        perc = np.shape(v_col)[0] * 100.0 / np.shape(data)[0]
        print("Column %s outliers = %.2f%%" % (k, perc))
```

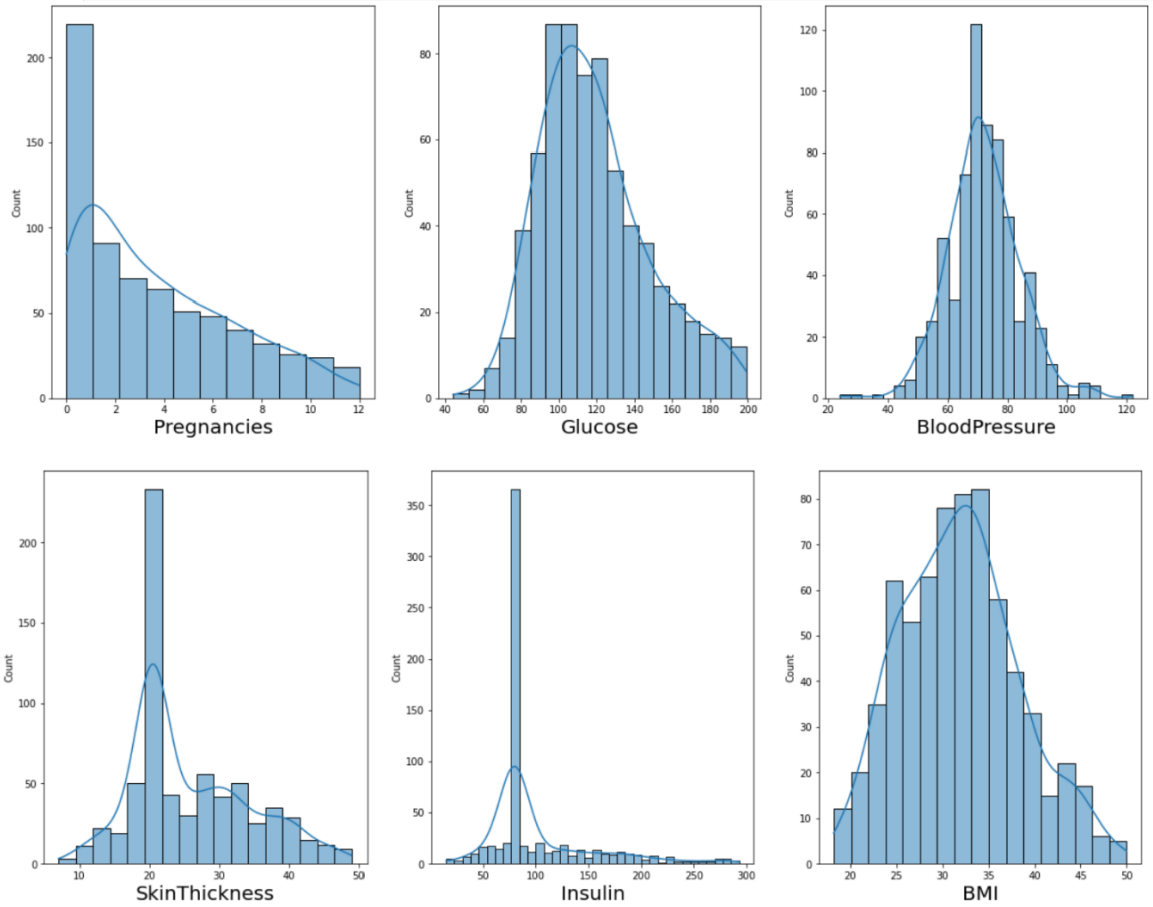
```
Column Pregnancies outliers = 0.52%
Column Glucose outliers = 0.00%
Column BloodPressure outliers = 2.21%
Column SkinThickness outliers = 1.56%
Column Insulin outliers = 11.59%
Column BMI outliers = 1.04%
Column DiabetesPedigreeFunction outliers = 3.78%
Column Age outliers = 1.17%
Column Outcome outliers = 0.00%
```

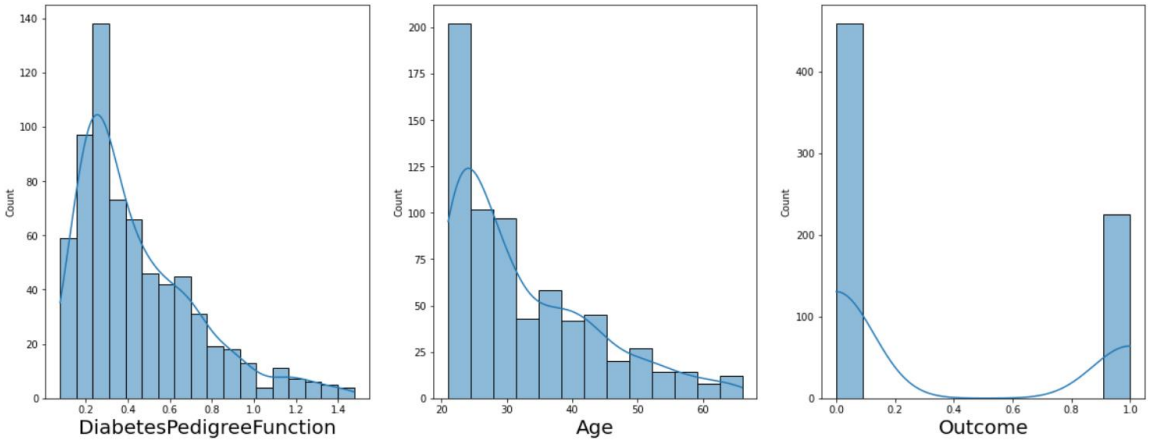
In [19]:
```
q = data['Pregnancies'].quantile(0.99)
# we are removing the top 1% data from the Pregnancies column
data_cleaned = data[data['Pregnancies']<q]

q = data_cleaned['BMI'].quantile(0.99)
# we are removing the top 1% data from the BMI column
data_cleaned  = data_cleaned[data_cleaned['BMI']<q]
```

```
q = data_cleaned['SkinThickness'].quantile(0.99)
# we are removing the top 1% data from the SkinThickness column
data_cleaned  = data_cleaned[data_cleaned['SkinThickness']<q]

q = data_cleaned['Insulin'].quantile(0.95)
# we are removing the top 5% data from the Insulin column
data_cleaned  = data_cleaned[data_cleaned['Insulin']<q]

q = data_cleaned['DiabetesPedigreeFunction'].quantile(0.99)
# we are removing the top 1% data from the DiabetesPedigreeFunction column
data_cleaned  = data_cleaned[data_cleaned['DiabetesPedigreeFunction']<q]

q = data_cleaned['Age'].quantile(0.99)
# we are removing the top 1% data from the Age column
data_cleaned  = data_cleaned[data_cleaned['Age']<q]
```

In [22]:

```
# let's see how data is distributed for every column
plt.figure(figsize=(20,25), facecolor='white')
plotnumber = 1

for column in data_cleaned:
    if plotnumber<=9 :
        ax = plt.subplot(3,3,plotnumber)
        sns.histplot(data_cleaned[column],kde = True)
        plt.xlabel(column,fontsize=20)
    plotnumber+=1
```

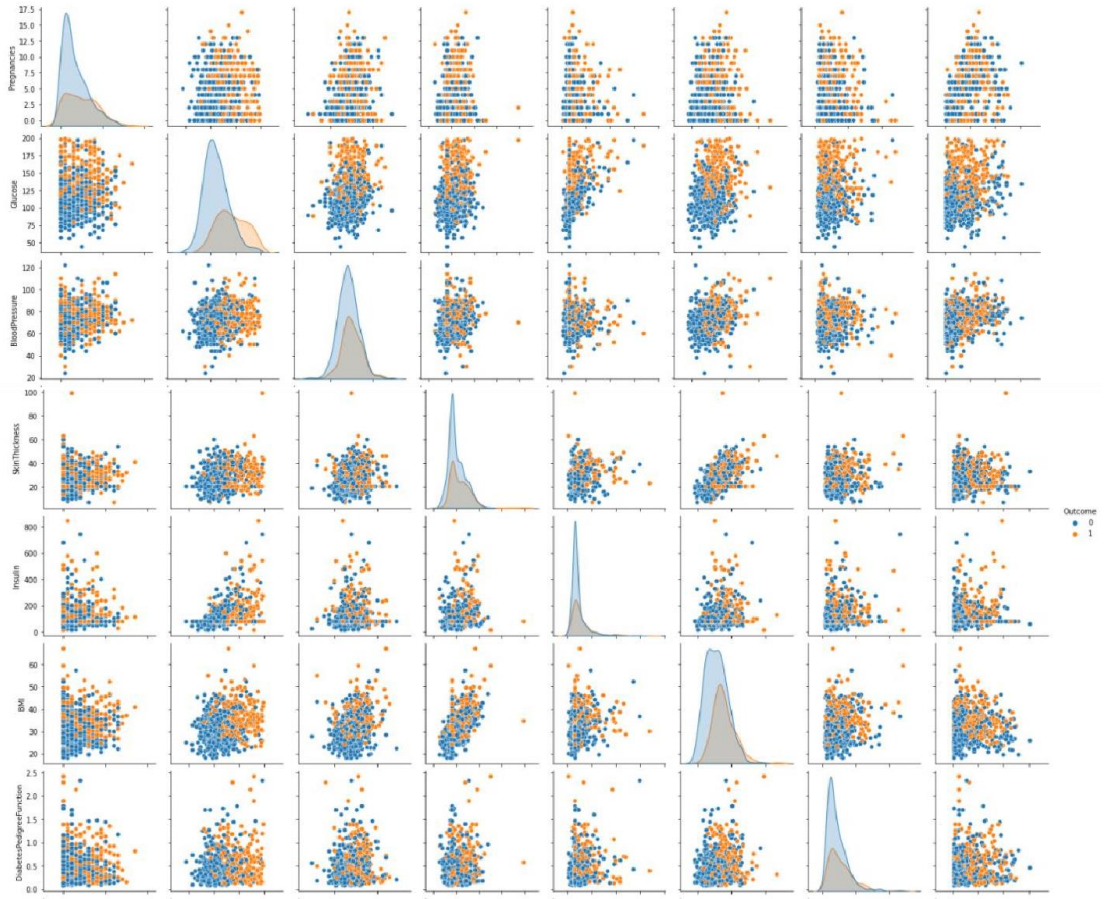




The data looks much better now than before. We will start our analysis with this data now as we don't want to lose important information. If our model doesn't work with accuracy, we will come back for more preprocessing.

```
In [23]: sns.pairplot(data = data,hue = "Outcome")
```

```
Out[23]: <seaborn.axisgrid.PairGrid at 0x2adb3c4d3a0>
```



In [27]:
```
#correlation value of "price" column with other columns
data.corr()['Outcome'][:-1].sort_values(ascending=False)
```

Out[27]:
```
Glucose                     0.492908
BMI                         0.312254
Age                         0.238356
Pregnancies                 0.221898
Insulin                     0.179185
SkinThickness               0.175026
DiabetesPedigreeFunction    0.173844
BloodPressure               0.162986
Name: Outcome, dtype: float64
```

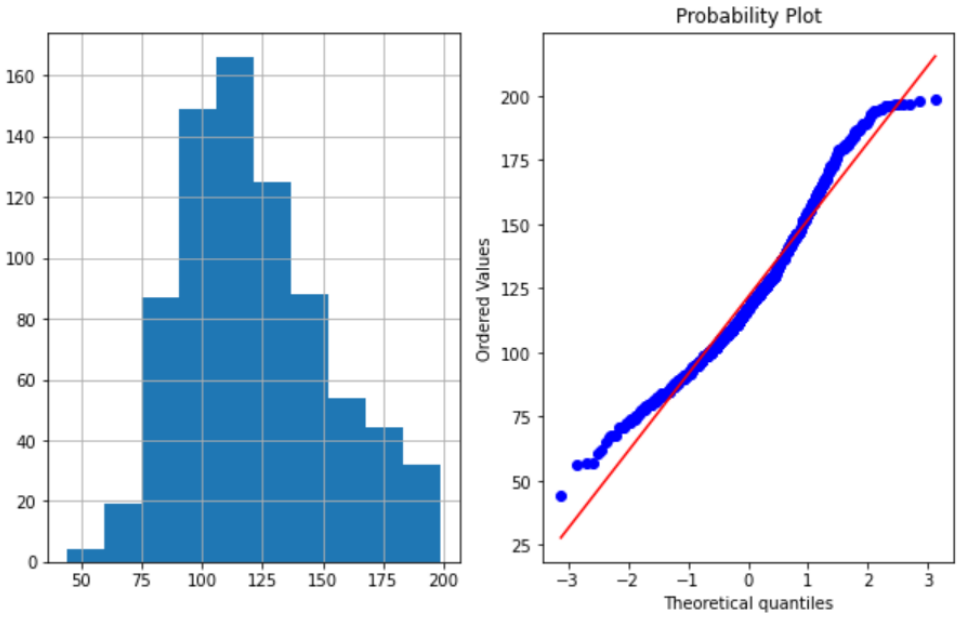
In [28]:
```
import scipy.stats as stat
import pylab
import matplotlib.pyplot as plt
```

In [29]:
```
## Function to check normal distribution
def plot_data(d,feature) :
    plt.figure(figsize = (10,6))
    plt.subplot(1,2,1)
    d[feature].hist()
    plt.subplot(1,2,2)
    stat.probplot(d[feature],dist = 'norm',plot=pylab)
    plt.show()
```

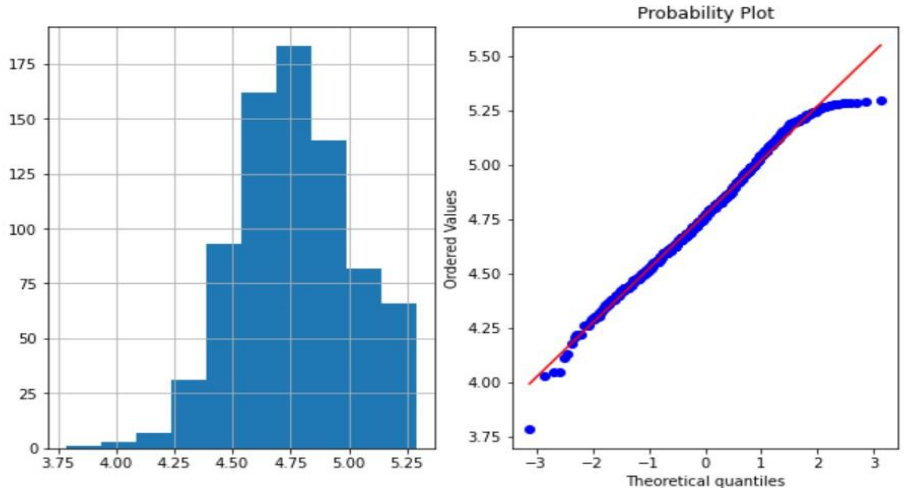
If our data is not normally distributed then we transform it.

1) for "Glucose" Column

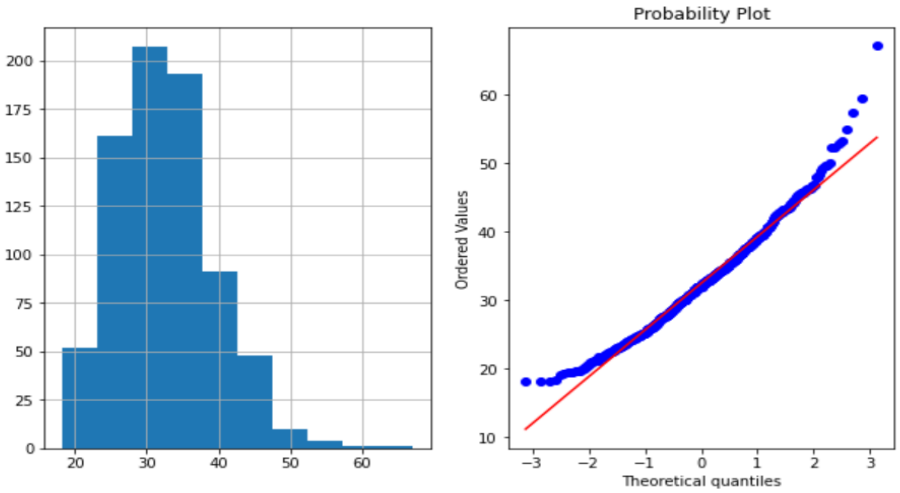
In [30]:
```
plot_data(data, "Glucose")
```



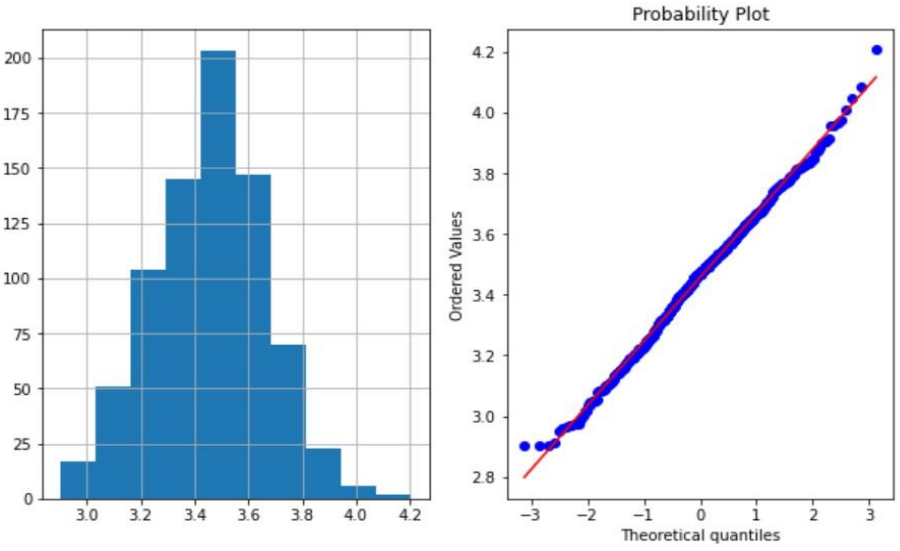
In [32]:
```
# Using Logarithmic Transformation
data['log_Glucose'] = np.log(data['Glucose'])
plot_data(data,'log_Glucose')
```



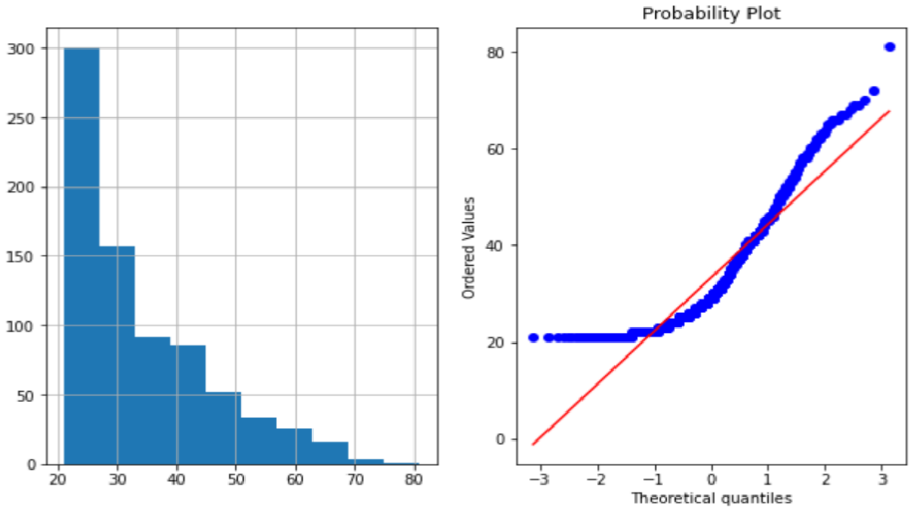
In [33]:
```
plot_data(data, "BMI")
```



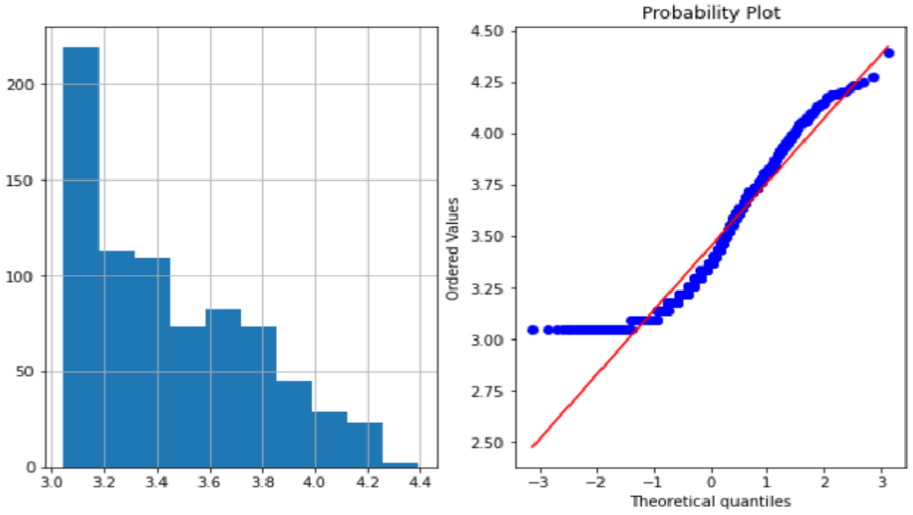
In [35]:
```
# Using Logarithmic Transformation
data['log_BMI'] = np.log(data['BMI'])
plot_data(data,'log_BMI')
```



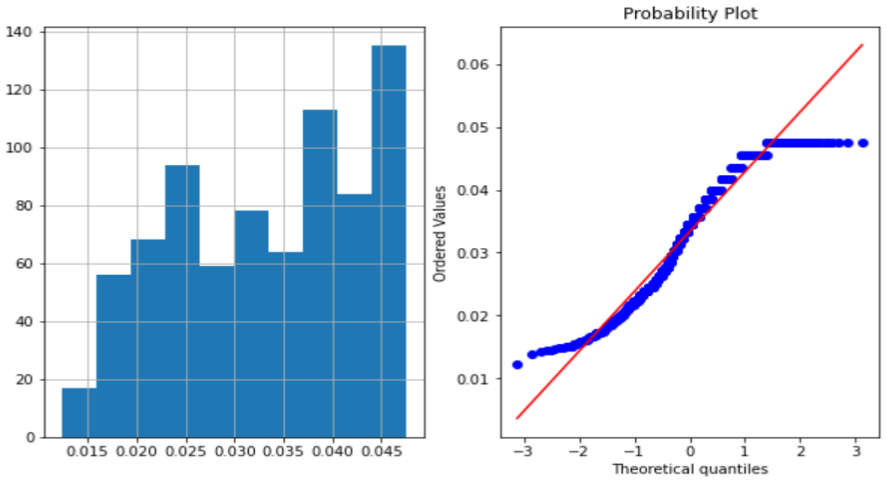
In [36]:
```
plot_data(data, "Age")
```



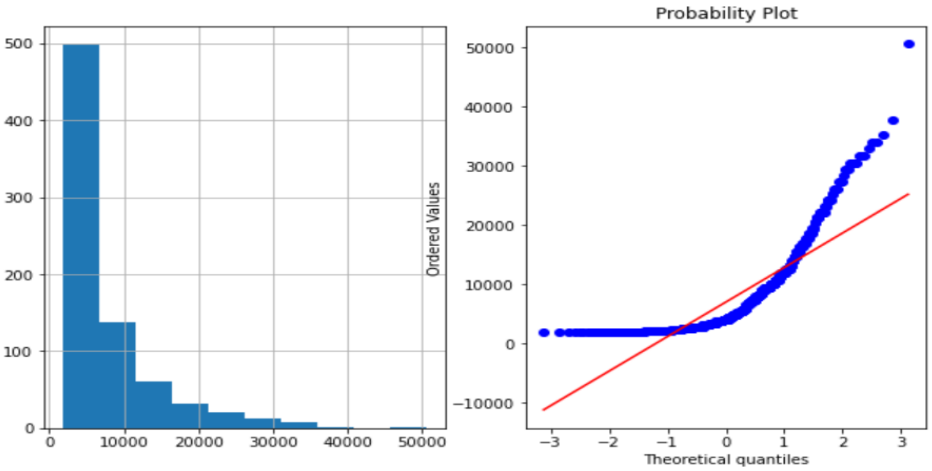
In [37]:
```
# Using Logarithmic Transformation
data['log_Age'] = np.log(data['Age'])
plot_data(data,'log_Age')
```



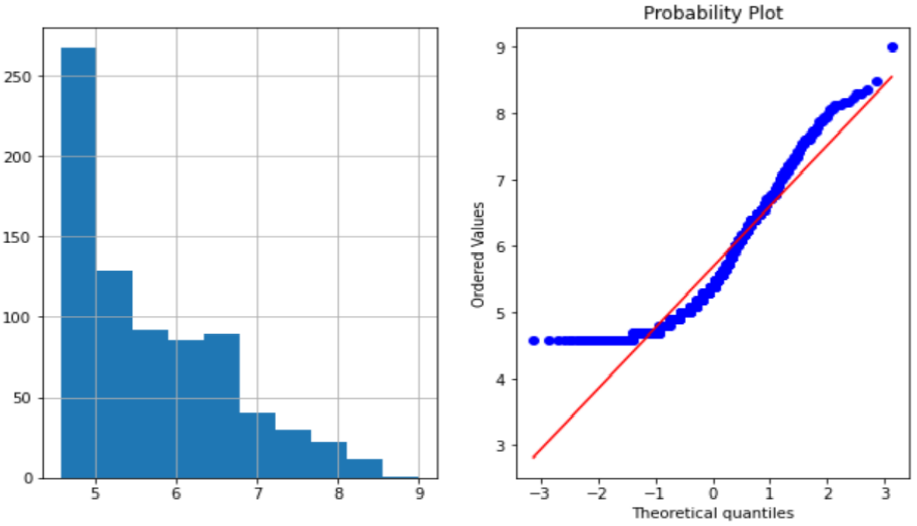
In [38]:
```
# Using Reciprocal Transformation
data['Reci_Age'] = 1/data['Age']
plot_data(data,'Reci_Age')
```



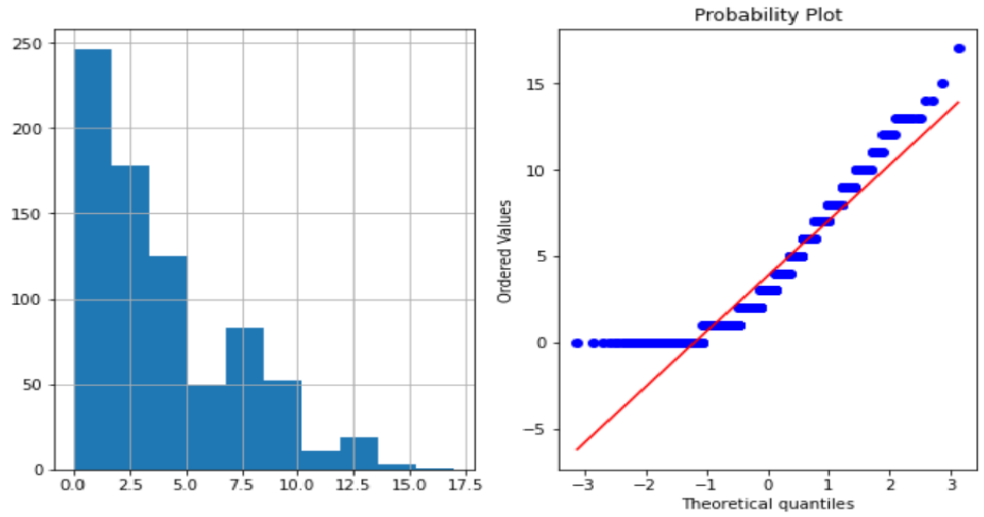
In [43]:
```
# Using Exponential  Transformation
e = 2.465
data['Exp_Age'] = data['Age']**e
plot_data(data,'Exp_Age')
```



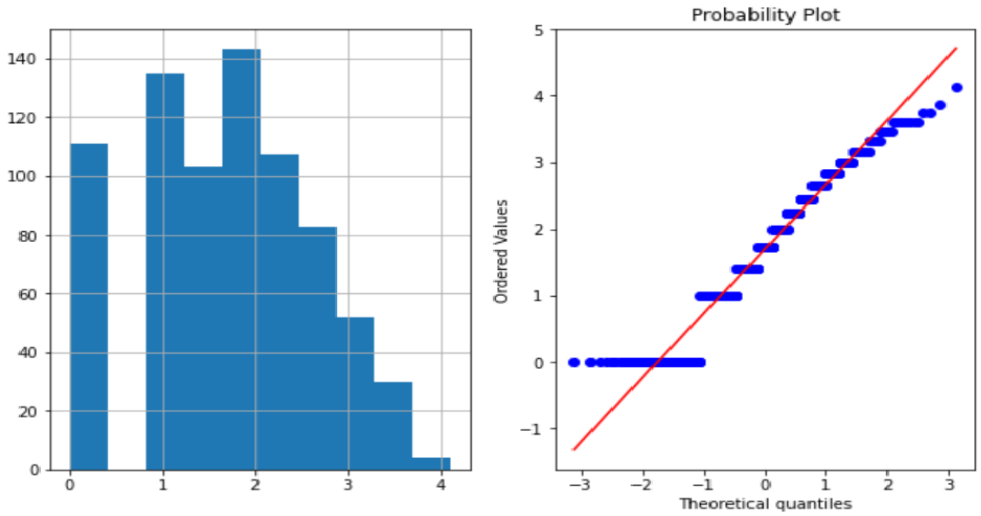
In [45]:
```
# Using Square root Transformation
data['Sq_Age'] = np.sqrt(data['Age'])
plot_data(data,'Sq_Age')
```



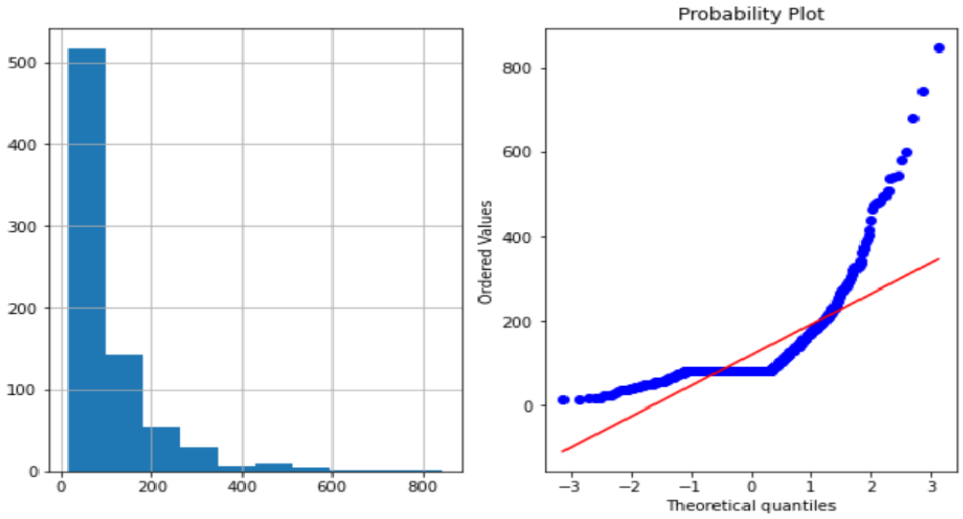
In [46]:
```
plot_data(data,'Pregnancies')
```



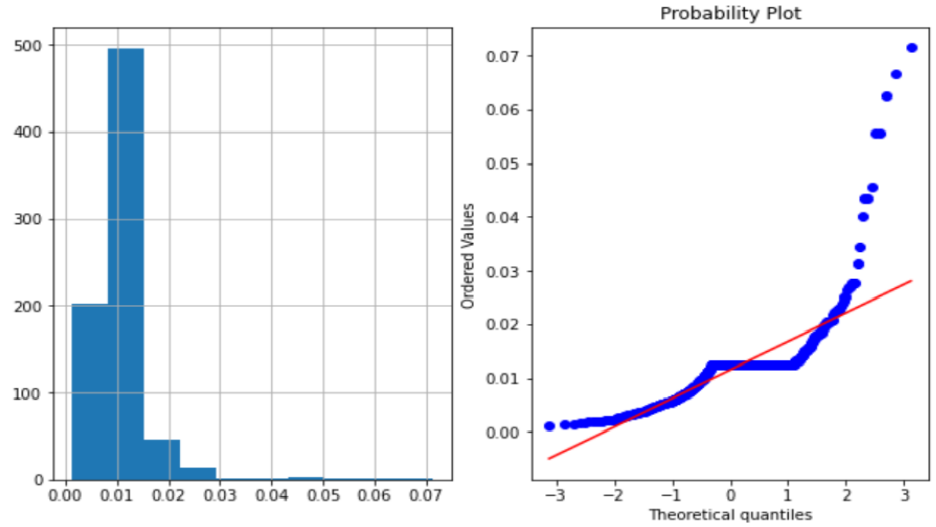
In [47]:
```
# Using Square root Transformation
data['Sq_Pregnancies'] = np.sqrt(data['Pregnancies'])
plot_data(data,'Sq_Pregnancies')
```



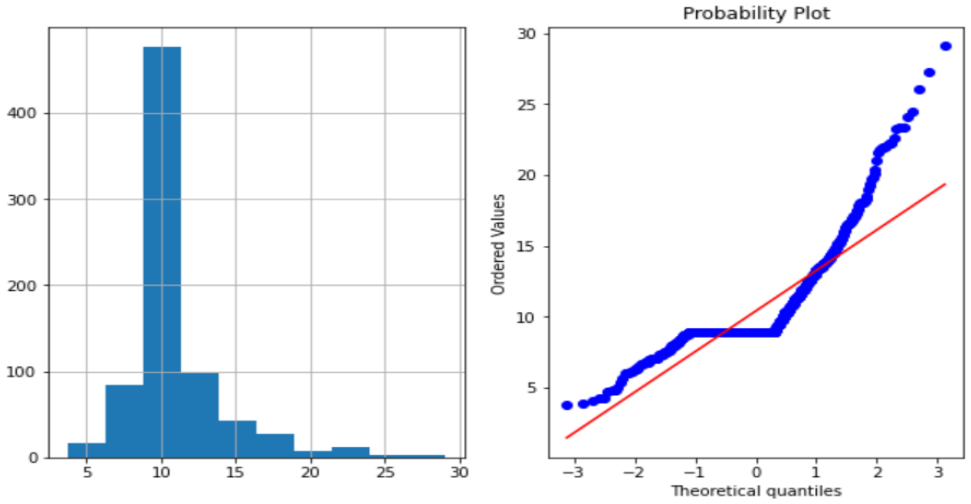
In [48]:
```
plot_data(data,'Insulin')
```



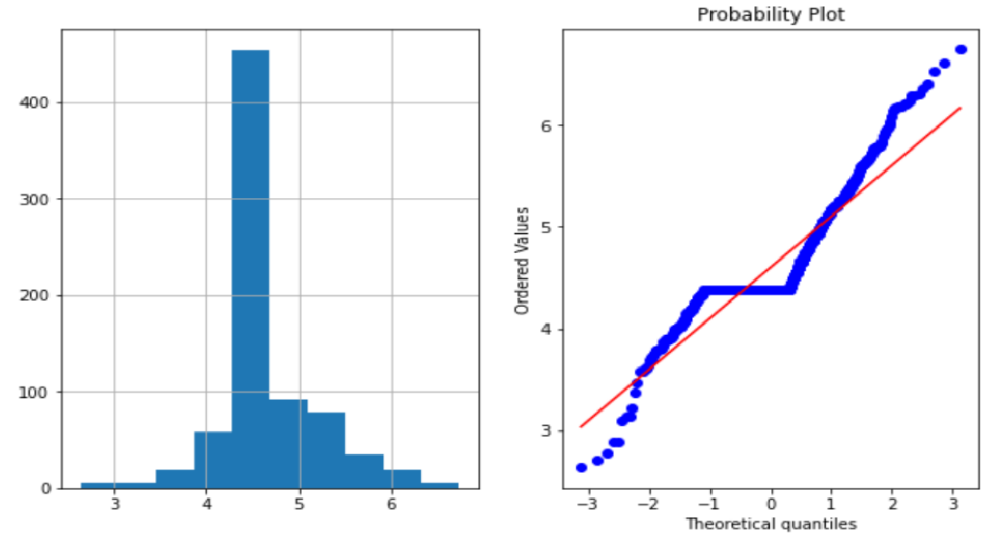
In [49]:
```
# Using Reciprocal Transformation
data['Reci_Insulin'] = 1/data['Insulin']
plot_data(data,'Reci_Insulin')
```



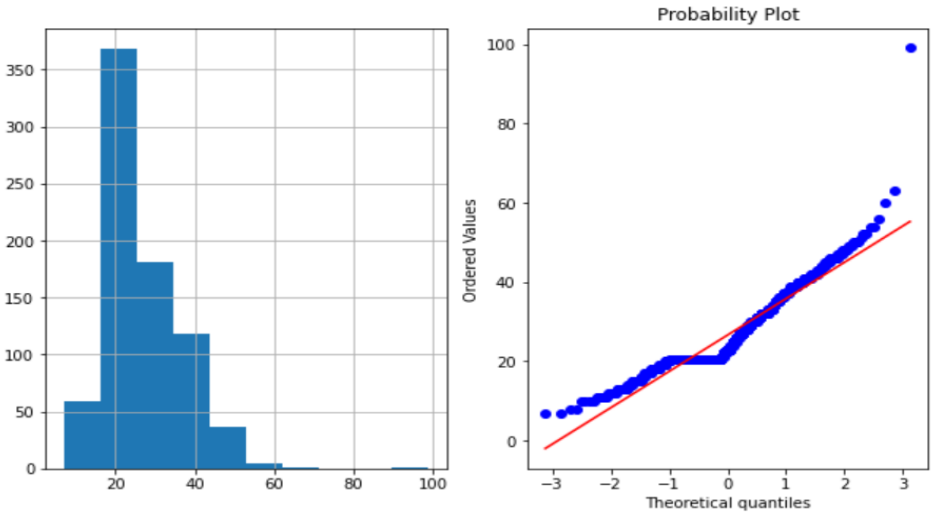
In [50]:
```
# Using Square root Transformation
data['Sq_Insulin'] = np.sqrt(data['Insulin'])
plot_data(data,'Sq_Insulin')
```



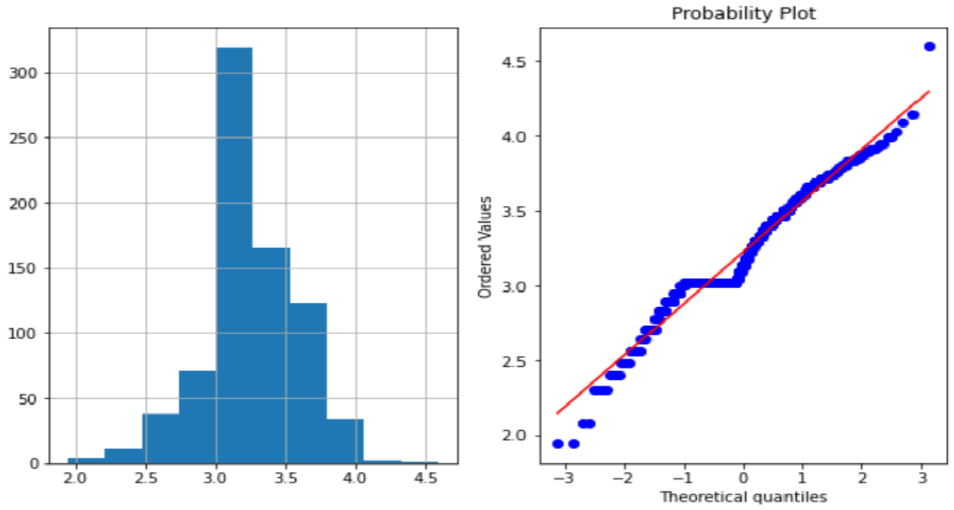
In [51]:
```
# Using Logarithmic Transformation
data['log_Insulin'] = np.log(data['Insulin'])
plot_data(data,'log_Insulin')
```



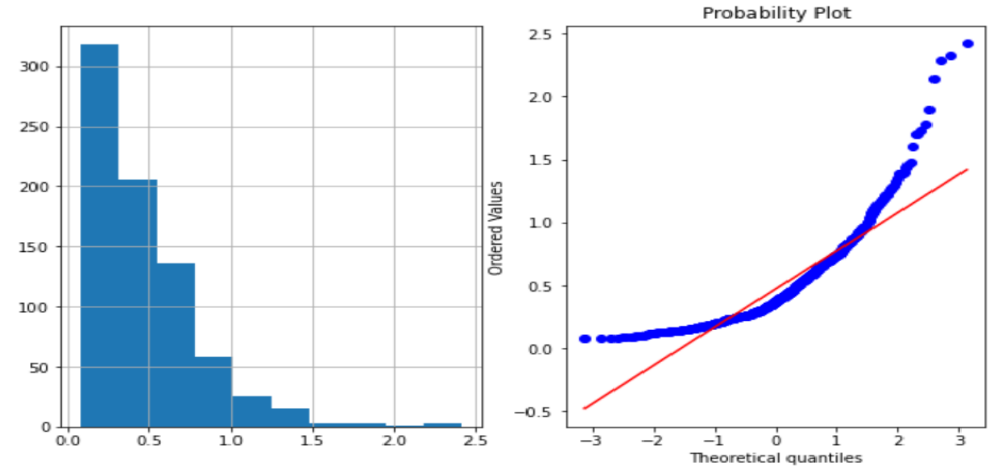
In [52]:
```
plot_data(data,'SkinThickness')
```



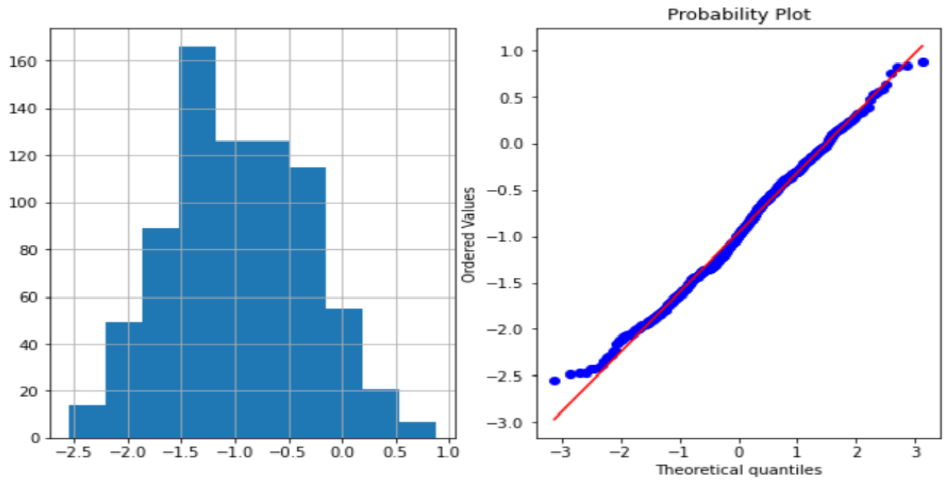
In [53]:
```
# Using Logarithmic Transformation
data['log_SkinThickness'] = np.log(data['SkinThickness'])
plot_data(data,'log_SkinThickness')
```



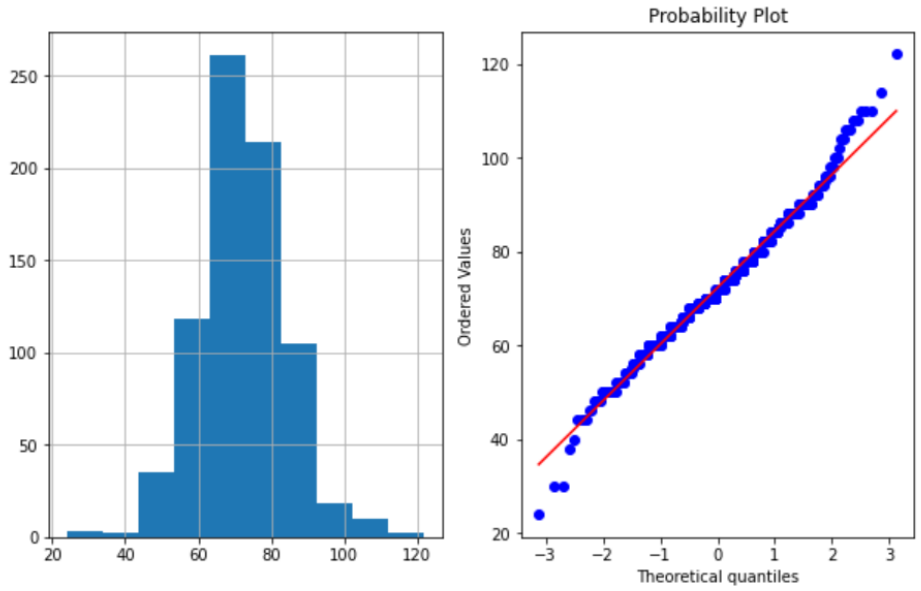
In [54]:
```
plot_data(data,'DiabetesPedigreeFunction')
```



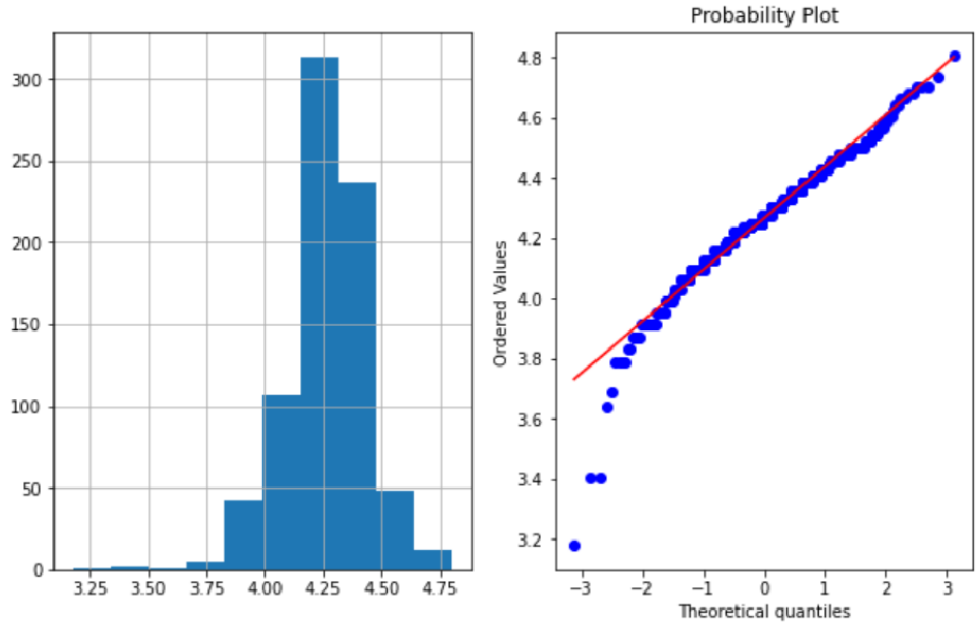
In [55]:
```
# Using Logarithmic Transformation
data['log_DiabetesPedigreeFunction'] = np.log(data['DiabetesPedigreeFunction'])
plot_data(data,'log_DiabetesPedigreeFunction')
```



In [56]:
```
plot_data(data,'BloodPressure')
```



In [57]:
```
# Using Logarithmic Transformation
data['log_BloodPressure'] = np.log(data['BloodPressure'])
plot_data(data,'log_BloodPressure')
```



Data set after Normalization

In [58]:
```
data.head()
```

Out[58]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome	log_Glucose	...	Reci_Age	Exp_Age
0	6	148.0	72.0	35.000000	79.799479	33.6	0.627	50	1	4.997212	...	0.020000	15415.622286
1	1	85.0	66.0	29.000000	79.799479	26.6	0.351	31	0	4.442651	...	0.032258	4744.676155
2	8	183.0	64.0	20.536458	79.799479	23.3	0.672	32	1	5.209486	...	0.031250	5130.913541
3	1	89.0	66.0	23.000000	94.000000	28.1	0.167	21	0	4.488636	...	0.047619	1816.647065
4	0	137.0	40.0	35.000000	168.000000	43.1	2.288	33	1	4.919981	...	0.030303	5535.245172

5 rows × 22 columns

In [59]:
```
dataset = data[["log_BMI","log_Glucose","log_Age","Sq_Pregnancies","Reci_Insulin","log_SkinThickness","log_DiabetesPedigreeFunct
dataset.head()
```

Out[59]:

	log_BMI	log_Glucose	log_Age	Sq_Pregnancies	Reci_Insulin	log_SkinThickness	log_DiabetesPedigreeFunction	log_BloodPressure	Outcome
0	3.514526	4.997212	3.912023	2.449490	0.012531	3.555348	-0.466809	4.276666	1
1	3.280911	4.442651	3.433987	1.000000	0.012531	3.367296	-1.046969	4.189655	0
2	3.148453	5.209486	3.465736	2.828427	0.012531	3.022202	-0.397497	4.158883	1
3	3.335770	4.488636	3.044522	1.000000	0.010638	3.135494	-1.789761	4.189655	0
4	3.763523	4.919981	3.496508	0.000000	0.005952	3.555348	0.827678	3.688879	1

In [60]:
```
X = dataset.drop(columns = ['Outcome'])
y = dataset['Outcome']
```

In [61]:
```
scalar = StandardScaler()
X_scaled = scalar.fit_transform(X)
```

In [62]:
```
X_scaled
```

Out[62]:
```
array([[ 0.27009922,  0.90856252,  1.43637931, ...,  0.95720591,
         0.76584846,  0.06379277],
       [-0.83876157, -1.31367612, -0.04593931, ...,  0.42067547,
        -0.13515874, -0.43414786],
       [-1.46747726,  1.75918707,  0.05250873, ..., -0.56390989,
         0.87349186, -0.61024505],
       ...,
       [-0.91068013,  0.10142474, -0.14761572, ..., -0.24067602,
        -0.69351704,  0.06379277],
       [-0.25202375,  0.26368185,  1.2445128 , ..., -0.56390989,
        -0.14403323, -0.97958011],
       [-0.2049503 , -0.95323526, -0.97152211, ...,  0.61095214,
        -0.30321784, -0.09742091]])
```

In [64]:
```
dataset_scaled = pd.DataFrame(X_scaled,columns=dataset.columns[:-1])
dataset_scaled.head()
```

Out[64]:

	log_BMI	log_Glucose	log_Age	Sq_Pregnancies	Reci_Insulin	log_SkinThickness	log_DiabetesPedigreeFunction	log_BloodPressure
0	0.270099	0.908563	1.436379	0.765422	0.158418	0.957206	0.765848	0.063793
1	-0.838762	-1.313676	-0.045939	-0.706001	0.158418	0.420675	-0.135159	-0.434148
2	-1.467477	1.759187	0.052509	1.150094	0.158418	-0.563910	0.873492	-0.610245
3	-0.578375	-1.129404	-1.253612	-0.706001	-0.131039	-0.240676	-1.288739	-0.434148
4	1.451972	0.599081	0.147927	-1.721134	-0.847514	0.957206	2.776227	-3.299938

Splitting Of Data Into Train And test

In [65]:
```
x_train,x_test,y_train,y_test = train_test_split(dataset_scaled,y, test_size= 0.25, random_state = 355)
```

In [66]:
```
x_train.shape , x_test.shape
```

Out[66]:
```
((576, 8), (192, 8))
```

Implementation of Logistic Regression Algorithm

In [67]:
```
log_reg = LogisticRegression()

log_reg.fit(x_train,y_train)
```

Out[67]:
```
LogisticRegression()
```

Let's see how well our model performs on the test data set.

In [68]:
```
y_pred = log_reg.predict(x_test)
```

In [69]:
```
accuracy = accuracy_score(y_test,y_pred)
accuracy
```

Out[69]:
```
0.75
```

In [70]:
```
# Confusion Matrix
conf_mat = confusion_matrix(y_test,y_pred)
conf_mat
```

Out[70]:
```
array([[107,  18],
       [ 30,  37]], dtype=int64)
```

In [71]:
```
true_positive = conf_mat[0][0]
false_positive = conf_mat[0][1]
false_negative = conf_mat[1][0]
true_negative = conf_mat[1][1]
```

In [72]:
```
# Breaking down the formula for Accuracy
Accuracy = (true_positive + true_negative) / (true_positive +false_positive + false_negative + true_negative)
Accuracy
```

Out[72]:
```
0.75
```

In [74]:
```
# Precison
Precision = true_positive/(true_positive+false_positive)
Precision
```

Out[74]:
```
0.856
```

In [75]:
```
# Recall
Recall = true_positive/(true_positive+false_negative)
Recall
```

Out[75]:
```
0.781021897810219
```

In [76]:
```
# F1 Score
F1_Score = 2*(Recall * Precision) / (Recall + Precision)
F1_Score
```

Out[76]: 0.816793893129771

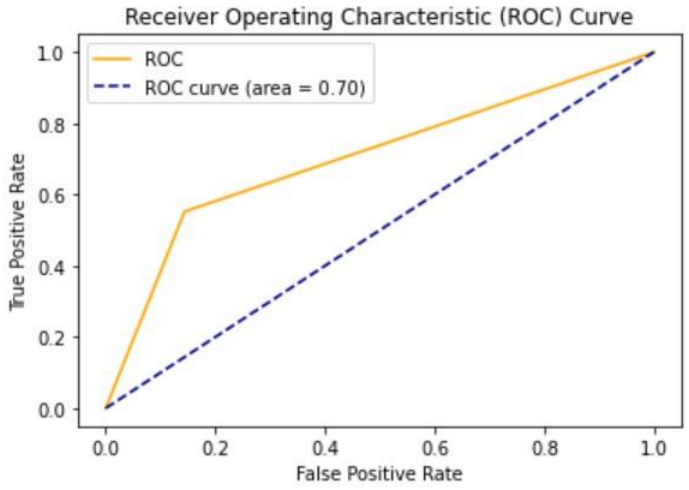
In [77]:
```
# Area Under Curve
auc = roc_auc_score(y_test, y_pred)
auc
```

Out[77]: 0.7041194029850747

ROC

In [78]:
```
fpr, tpr, thresholds = roc_curve(y_test, y_pred)
```

In [79]:
```
plt.plot(fpr, tpr, color='orange', label='ROC')
plt.plot([0, 1], [0, 1], color='darkblue', linestyle='--',label='ROC curve (area = %0.2f)' % auc)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend()
plt.show()
```



In [73]:
```
import pickle
# Writing different model files to file
with open( 'modelForPrediction.sav', 'wb') as f:
    pickle.dump(log_reg,f)

with open('sandardScalar.sav', 'wb') as f:
    pickle.dump(scalar,f)
```

Experiment-3

Aim: Implement Decision tree regression in Python using Jupyter Notebook.

Importing the required modules

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

Defining Utility Functions for Entropy and GINI calculation

In [2]:
```
def calcEntropy(pop, target):
    counts = target.value_counts(normalize = True)
    counts *= np.log2(counts)
    entropy = -counts.sum()
    return entropy
```

In [3]:
```
def calcGini(pop, target):
    counts = target.value_counts(normalize = True)
    counts = np.square(counts)
    gini = 1 - np.sum(counts)
    return gini
```

Defining Utility Functions to Generate Subpopulations and Find the Optimal Split

In [4]:
```
def createSubpopulation(pop, target, feature, feature_val):
    return (pop[pop[feature] == feature_val], target[pop[feature] == feature_val])
```

In [5]:
```
def minEntropySplit(pop, target, features):
    minfeature = -1
    classes = target.value_counts()
    minEntropy = np.log2(len(classes)) + 1 #maximum possible entropy + 1
    n = len(pop)
    for f in features:
        # split using this feature and calculate entropy
        unique_vals = set(pop.loc[:,f])
        totalEntropy = 0
        for u in unique_vals:
            # split where pop[f] = u
            subpop, subtarget = createSubpopulation(pop, target, f, u)
            ni = len(subpop)
            entropy = calcEntropy(subpop, subtarget)
            totalEntropy += (ni/n)*entropy
        if(totalEntropy < minEntropy):
            minfeature = f
            minEntropy = totalEntropy
    return (minfeature, minEntropy)
```

In [6]:
```
def minGiniSplit(pop, target, features):
    minfeature = -1
    classes = target.value_counts()
    minGini = 1 #maximum possible gini (for n-> inf)
    n = len(pop)
    for f in features:
        # split using this feature and calculate Gini
        unique_vals = set(pop.loc[:, f])
        totalGini = 0
        for u in unique_vales:
            #split where pop[j][f] = u
            subpop, subtarget = createSubpopulation(pop, target, f, u)
            ni = len(subpop)
            gini = calcGini(subpop, subtarget)
            totalGini += (ni/n)*gini
        if(totalGini < minGini):
            minGini = totalGini
            minfeature = f
    return (minfeature, minGini)
```

Defining the data

In [7]:
```
X = pd.DataFrame({
    'outlook': ['sunny', 'sunny', 'overcast', 'rain', 'rain', 'rain', 'overcast', 'sunny', 'sunny', 'rain', 'sunny', 'ove
    'temp':['hot', 'hot', 'hot', 'mild', 'cool', 'cool', 'cool', 'mild', 'cool', 'mild', 'mild', 'mild', 'hot', 'mild'],
    'humidity':['high', 'high', 'high', 'high', 'normal', 'normal', 'normal', 'high', 'normal', 'normal', 'normal', 'high
    'wind':['strong', 'strong', 'weak', 'weak', 'weak', 'strong', 'weak', 'weak', 'weak', 'weak', 'strong', 'strong', 'we
})
y = pd.Series([0, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0], name = "Play tennis")
```

In [8]:
```
X.head()
```

Out[8]:

	outlook	temp	humidity	wind
0	sunny	hot	high	strong
1	sunny	hot	high	strong
2	overcast	hot	high	weak
3	rain	mild	high	weak
4	rain	cool	normal	weak

In [9]:
```
y.head()
```

Out[9]:
```
0    0
1    0
2    1
3    1
4    1
Name: Play tennis, dtype: int64
```

Generating the training and testing sets from the data

The data is split into features and lables and 30% of the data is used as a testing set and 70% is used as training set

In [10]:
```
from sklearn.model_selection import train_test_split
np.random.seed(42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3)
```

In [11]:
```
X_train.head()
```

Out[11]:

	outlook	temp	humidity	wind
8	sunny	cool	normal	weak
2	overcast	hot	high	weak
1	sunny	hot	high	strong
13	rain	mild	high	strong

In [12]:
```
X_test.head()
```

Out[12]:

	outlook	temp	humidity	wind
9	rain	mild	normal	weak
11	overcast	mild	high	strong
0	sunny	hot	high	strong
12	overcast	hot	normal	weak
5	rain	cool	normal	strong

In [13]:
```
y_train.head()
```

Out[13]:
```
8     1
2     1
1     0
13    0
4     1
Name: Play tennis, dtype: int64
```

In [14]:
```
y_test.head()
```

Out[14]:
```
9     1
11    1
0     0
12    1
```

In [15]:
```
## Information Gain and gini index value of train dataset
calcEntropy(X_train, y_train), calcGini(X_train, y_train)
```

Out[15]:
```
(0.9182958340544896, 0.4444444444444444)
```

In [16]:
```
### Information Gain and gini index value of test dataset
calcEntropy(X_test, y_test), calcGini(X_test, y_test)
```

Out[16]:
```
(0.9709505944546686, 0.48)
```

Implementation of Decision Tree Class without Pruning

In [17]:
```
class DecisionTree:

        def __init__(self, pop, target, criteria = 'entropy'):
            self.pop = pop
            self.target = target
            self.criteria = criteria
            self.children = {}
            self.sel_feature = None
            self.decision = None
            self.isLeaf = False
            if(criteria == 'entropy'):
                self.dec_param = calcEntropy(self.pop, self.target)
            elif(criteria == 'gini'):
                self.dec_param = calcGini(self.pop, self.target)

        def fit(self, features):

        # if the decision parameter of self is 0, make a decision, rather than learning
            if(self.dec_param == 0):
                # already a pure population, take the most popular decision
                self.isLeaf = True
                self.decision = self.target.value_counts().index[0]
                return
```

```
    # find the optimal split
        if(self.criteria == 'entropy'):
            f, dec_param = minEntropySplit(self.pop, self.target, features)
        elif(self.criteria == 'gini'):
            f, dec_param = minGiniSplit(self, pop, self.target, features)

    # remove the selected feature from the list of remaining features
        features.remove(f)
        self.sel_feature = f

        unique_vals = set(self.pop.loc[:, f])
        for u in unique_vals:
            #for each possible value of the feature
            #create a child node
            subpop, subtarget = createSubpopulation(self.pop, self.target, self.sel_feature, u)
            self.children[u] = DecisionTree(subpop, subtarget, self.criteria)
            #train that child
            self.children[u].fit(features)

    def predict_row(self, row):
        if(self.isLeaf):
            return self.decision
        else:
            return self.children[row.loc[self.sel_feature]].predict_row(row)
```

```
        def predict(self, test_pop):
            preds = []
            for i in range(len(test_pop)):
                preds.append(self.predict_row(test_pop.iloc[i,:]))
            return pd.Series(data = preds, name="preds", index = test_pop.index)


        def visualize(self, level=0):
            if(level == 0): print("root ", end = "")
            print("-> ", end = "")
            if(self.isLeaf):
                print("~"+str(self.decision)+"~", end = "")
                print("")
                return
            print(self.sel_feature)
            for i in self.children.keys():
                print("\t"*(2*level+1), "==", i, end = " ")
                self.children[i].visualize(level+1)
            print("")
```

In [18]:
```
dt = DecisionTree(X_train, y_train)
dt.fit(list(X_train.columns))
```

In [21]:
```
y_test
```

Out[21]:
```
9     1
11    1
0     0
12    1
5     0
Name: Play tennis, dtype: int64
```

As we clearly see index no. 5 give wrong prediction output.

In [19]:
```
dt.visualize()
```

```
root -> humidity
         == high -> outlook
                         == sunny -> ~0~
                         == rain -> wind
                                         == strong -> ~0~
                                         == weak -> ~1~

                         == overcast -> ~1~

         == normal -> ~1~
```

In [20]:
```
dt.predict(X_test)
```

Out[20]:
```
9     1
11    1
0     0
12    1
5     1
Name: preds, dtype: int64
```

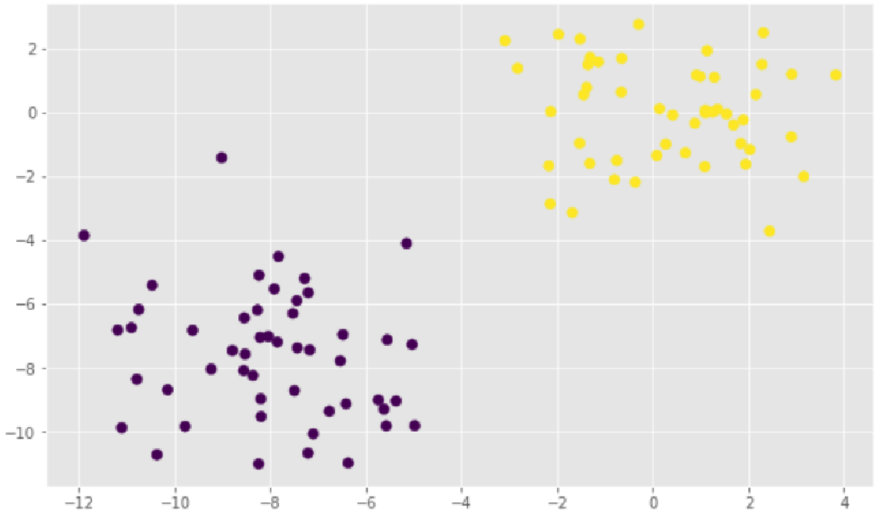
Experiment-4

Aim: Implement Bayesian Learning in Python using Jupyter Notebook.

In [3]:
```
from sklearn import datasets
X,y = datasets.make_blobs(100,2,centers=2,random_state=1701,cluster_std=2)
```

In [4]:
```
import matplotlib.pyplot as plt
plt.style.use('ggplot')
%matplotlib inline
```

In [5]:
```
plt.figure(figsize=(10,6))
plt.scatter(X[:,0],X[:,1],c=y,s=50);
```



In [7]:
```
import numpy as np
from sklearn import model_selection as ms
X_train, X_test,y_train,y_test = ms.train_test_split(
X.astype(np.float32),y,test_size=0.1
)
```

In [8]:
```
%pip install opencv-python
```

```
Collecting opencv-python
  Downloading opencv_python-4.6.0.66-cp36-abi3-win_amd64.whl (35.6 MB)
Requirement already satisfied: numpy>=1.19.3 in c:\users\91995\anaconda3\lib\site-packages (from opencv-python) (1.21.5)
Installing collected packages: opencv-python
Successfully installed opencv-python-4.6.0.66
Note: you may need to restart the kernel to use updated packages.
```

In [12]:
```
import cv2
model_norm = cv2.ml.NormalBayesClassifier_create()
```

In [13]:
```
model_norm.train(X_train,cv2.ml.ROW_SAMPLE,y_train)
```

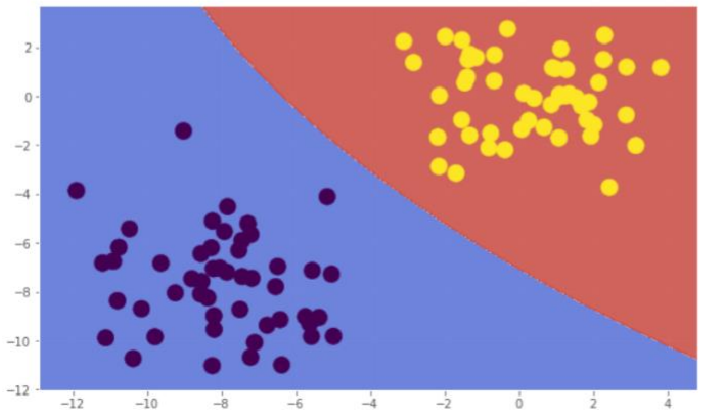
Out[13]:
```
True
```

In [14]:
```
_,y_pred = model_norm.predict(X_test)
```

In [15]:
```
from sklearn import metrics
metrics.accuracy_score(y_test,y_pred)
```

Out[15]:
```
1.0
```

In [18]:
```
def plot_decision_boundary(model,X_test,y_test):
    #create a mesh to plot in
    h = 0.02 # step size in mesh
    x_min,x_max = X_test[:,0].min()-1,X_test[:,0].max()+1
    y_min,y_max = X_test[:,1].min()-2,X_test[:,1].max()+1
    xx,yy = np.meshgrid(np.arrange(x_min, x_max,h),
                        np.arrange(y_min,y_max,h))
    X_hypo = np.column_stack((xx.ravel().astype(np.float32),
                              yy.ravel().astype(np.float32)))
    ret = model.predict(X_hypo)
    if isinstance(ret,tuple):
        zz = ret[1]
    else:
        zz = ret
    zz = zz.reshape(xx.shape)

    plt.contourf(xx,yy,zz,cmap=plt.cm.coolwarm,alpha=0.8)
    plt.scatter(X_test[:,0],X_test[:,1],c=y_test,s=200)
```



In [22]:
```
ret,y_pred, y_proba = model_norm.predictProb(X_test)
```

In [23]:
```
y_proba.round(2)
```

Out[23]:
```
array([[0.  , 0.18],
       [0.11, 0.  ],
       [0.05, 0.  ],
       [0.  , 0.23],
       [0.25, 0.  ],
       [0.  , 0.2 ],
       [0.26, 0.  ],
       [0.  , 0.28],
       [0.18, 0.  ],
       [0.  , 0.02]], dtype=float32)
```

Experiment-5

Aim: Implement KNN in Python using Jupyter Notebook.

Import Libraries

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

Import the Data

In [5]:
```
df = pd.read_csv("Classified Data",index_col=0)
```

In [6]:
```
df.head() # display top 5 observation
```

Out[6]:

	WTT	PTI	EQW	SBI	LQE	QWG	FDJ	PJF	HQE	NXJ	TARGET CLASS
0	0.913917	1.162073	0.567946	0.755464	0.780862	0.352608	0.759697	0.643798	0.879422	1.231409	1
1	0.635632	1.003722	0.535342	0.825645	0.924109	0.648450	0.675334	1.013546	0.621552	1.492702	0
2	0.721360	1.201493	0.921990	0.855595	1.526629	0.720781	1.626351	1.154483	0.957877	1.285597	0
3	1.234204	1.386726	0.653046	0.825624	1.142504	0.875128	1.409708	1.380003	1.522692	1.153093	1
4	1.279491	0.949750	0.627280	0.668976	1.232537	0.703727	1.115596	0.646691	1.463812	1.419167	1

In [4]:
```
df.tail() #display last 5 observation
```

Out[4]:

	WTT	PTI	EQW	SBI	LQE	QWG	FDJ	PJF	HQE	NXJ	TARGET CLASS
995	1.010953	1.034006	0.853116	0.622460	1.036610	0.586240	0.746811	0.319752	1.117340	1.348517	1
996	0.575529	0.955786	0.941835	0.792882	1.414277	1.269540	1.055928	0.713193	0.958684	1.663489	0
997	1.135470	0.982462	0.781905	0.916738	0.901031	0.884738	0.386802	0.389584	0.919191	1.385504	1
998	1.084894	0.861769	0.407158	0.665696	1.608612	0.943859	0.855806	1.061338	1.277456	1.188063	1
999	0.837460	0.961184	0.417006	0.799784	0.934399	0.424762	0.778234	0.907962	1.257190	1.364837	1

Data Profiling

In [7]:
```
## size of the dataset
df.shape
```

Out[7]:
```
(1000, 11)
```

There are 1000 no. of rows and 11 no. of columns in the dataset.

In [6]:
```
## Check NA value in the dataset
df.isna().sum()
```

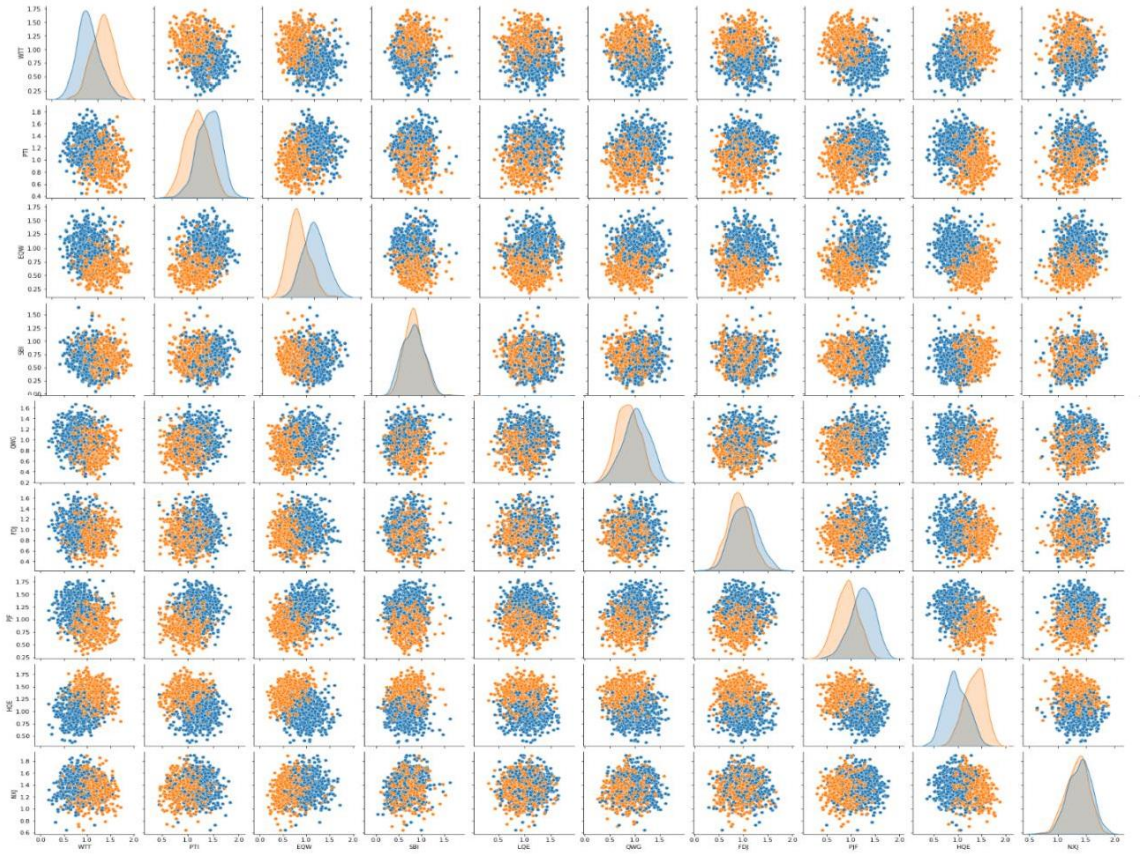
Out[6]:
```
WTT             0
PTI             0
EQW             0
SBI             0
LQE             0
QWG             0
FDJ             0
PJF             0
HQE             0
NXJ             0
TARGET CLASS    0
dtype: int64
```

In [7]:
```
## data types of columns
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 1000 entries, 0 to 999
Data columns (total 11 columns):
 #   Column         Non-Null Count  Dtype  
---  ------         --------------  -----  
 0   WTT            1000 non-null   float64
 1   PTI            1000 non-null   float64
 2   EQW            1000 non-null   float64
 3   SBI            1000 non-null   float64
```

In [8]:
```
sns.pairplot(df,hue='TARGET CLASS')
```

Out[8]:
```
<seaborn.axisgrid.PairGrid at 0x283cd0c54c0>
```



Standardize the Variables

In [10]:
```
from sklearn.preprocessing import StandardScaler
```

In [11]:
```
scaler = StandardScaler()
```

In [12]:
```
scaler.fit(df.drop('TARGET CLASS',axis=1)) # drop the target class from the dataset
```

Out[12]:
```
StandardScaler()
```

In [13]:
```
scaled_features = scaler.transform(df.drop('TARGET CLASS',axis=1))
```

In [14]:
```
## create the dataframe of scaled features
data = pd.DataFrame(scaled_features,columns=df.columns[:-1])
data.head()
```

Out[14]:

	WTT	PTI	EQW	SBI	LQE	QWG	FDJ	PJF	HQE	NXJ
0	-0.123542	0.185907	-0.913431	0.319629	-1.033637	-2.308375	-0.798951	-1.482368	-0.949719	-0.643314
1	-1.084836	-0.430348	-1.025313	0.625388	-0.444847	-1.152706	-1.129797	-0.202240	-1.828051	0.636759
2	-0.788702	0.339318	0.301511	0.755873	2.031693	-0.870156	2.599818	0.285707	-0.682494	-0.377850
3	0.982841	1.060193	-0.621399	0.625299	0.452820	-0.267220	1.750208	1.066491	1.241325	-1.026987
4	1.139275	-0.640392	-0.709819	-0.057175	0.822886	-0.936773	0.596782	-1.472352	1.040772	0.276510

Split the dataset into train and test

In [15]:
```
from sklearn.model_selection import train_test_split
```

In [16]:
```
## divided the dataset into 70% of train data and 30% of test data
X_train, X_test, y_train, y_test = train_test_split(data,df['TARGET CLASS'],test_size=0.30)
```

In [17]:
```
X_train.shape , X_test.shape , y_train.shape , y_test.shape
```

Out[17]:
```
((700, 10), (300, 10), (700,), (300,))
```

Implement KNN Model

In [18]:
```
from sklearn.neighbors import KNeighborsClassifier
```

In [19]:
```
knn = KNeighborsClassifier(n_neighbors=3) ## choose k = 3
```

In [20]:
```
knn.fit(X_train,y_train)
```

Out[20]:
```
KNeighborsClassifier(n_neighbors=3)
```

In [21]:
```
pred = knn.predict(X_test) ## prediction of test data
```

In [22]:
```
pred
```

Out[22]:
```
array([1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1,
       1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1,
       0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1,
       0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0,
       0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 1,
       0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 1,
       1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 0,
       1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1,
       1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1,
       1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0,
       1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1,
       1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1,
       0, 1, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0,
       0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0], dtype=int64)
```

Predictions and Evaluations of KNN model

In [23]:
```
from sklearn.metrics import classification_report,confusion_matrix , plot_confusion_matrix
from sklearn.model_selection import cross_val_score
```

In [24]:
```
print(confusion_matrix(y_test,pred)) ## confusion matrix
```

```
[[135  18]
 [  6 141]]
```

Choosing a K Value by using the elbow method to pick a good K Value

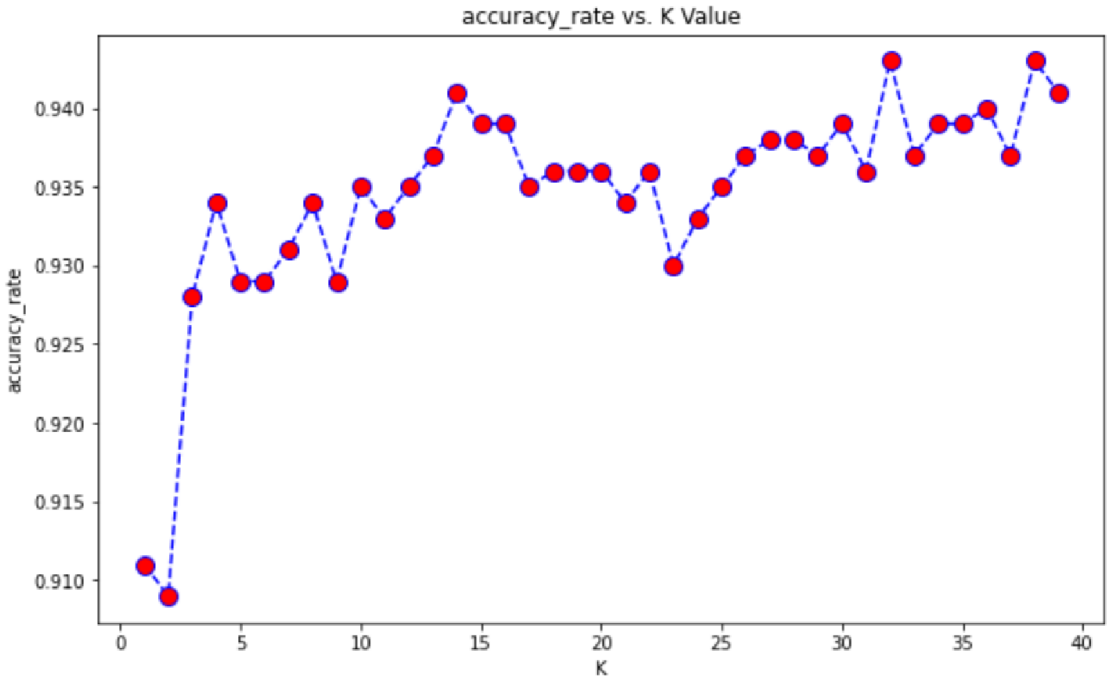
In [27]:
```
accuracy_rate = []
for i in range(1,40):

    knn = KNeighborsClassifier(n_neighbors=i)
    score=cross_val_score(knn,data,df['TARGET CLASS'],cv=10)
    accuracy_rate.append(score.mean())
```

In [28]:
```
accuracy_rate
```

Out[28]:
```
[0.9109999999999999,
 0.909,
 0.9280000000000002,
 0.9339999999999999,
 0.9289999999999999,
 0.929,
 0.9310000000000003,
 0.9340000000000002,
 0.9289999999999999,
 0.9350000000000002,
 0.9329999999999998,
 0.9350000000000002,
 0.937,
 0.9410000000000001,
 0.9390000000000001,
 0.9390000000000001,
```

In [29]:
```
plt.figure(figsize=(10,6))
plt.plot(range(1,40),accuracy_rate,color='blue', linestyle='dashed', marker='o',
         markerfacecolor='red', markersize=10)
plt.title('accuracy_rate vs. K Value')
plt.xlabel('K')
plt.ylabel('accuracy_rate')
```

Out[29]:
```
Text(0, 0.5, 'accuracy_rate')
```



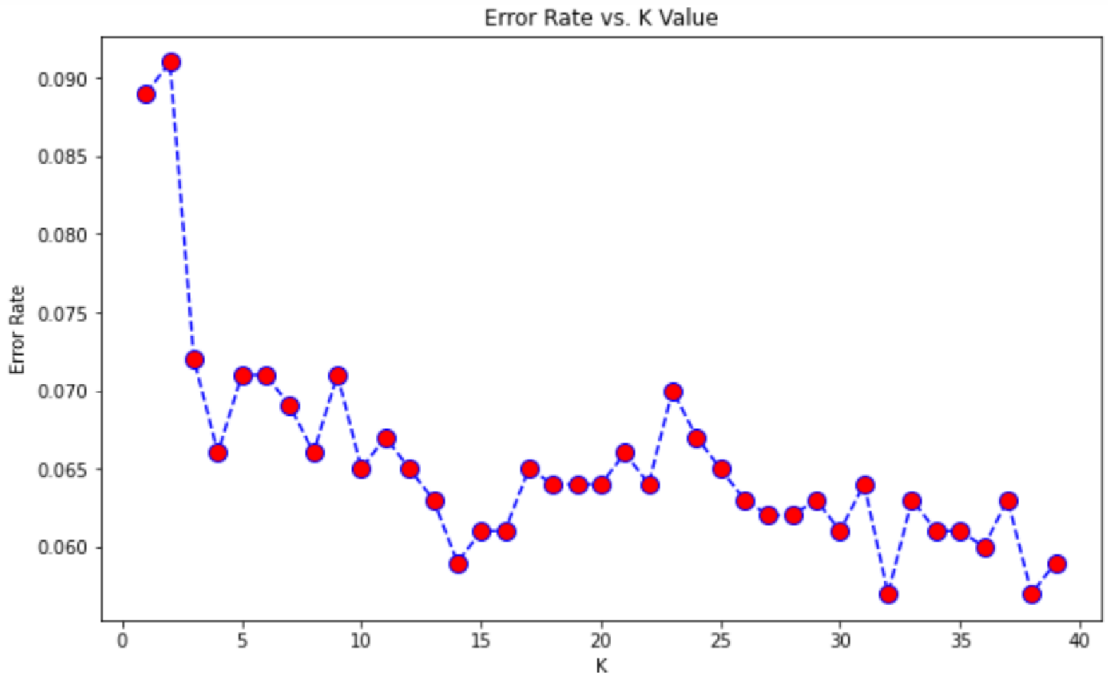
In [30]:
```
error_rate = []
for i in range(1,40):
    knn = KNeighborsClassifier(n_neighbors=i)
    score=cross_val_score(knn,data,df['TARGET CLASS'],cv=10)
    error_rate.append(1-score.mean())
```

In [31]:
```
error_rate
```

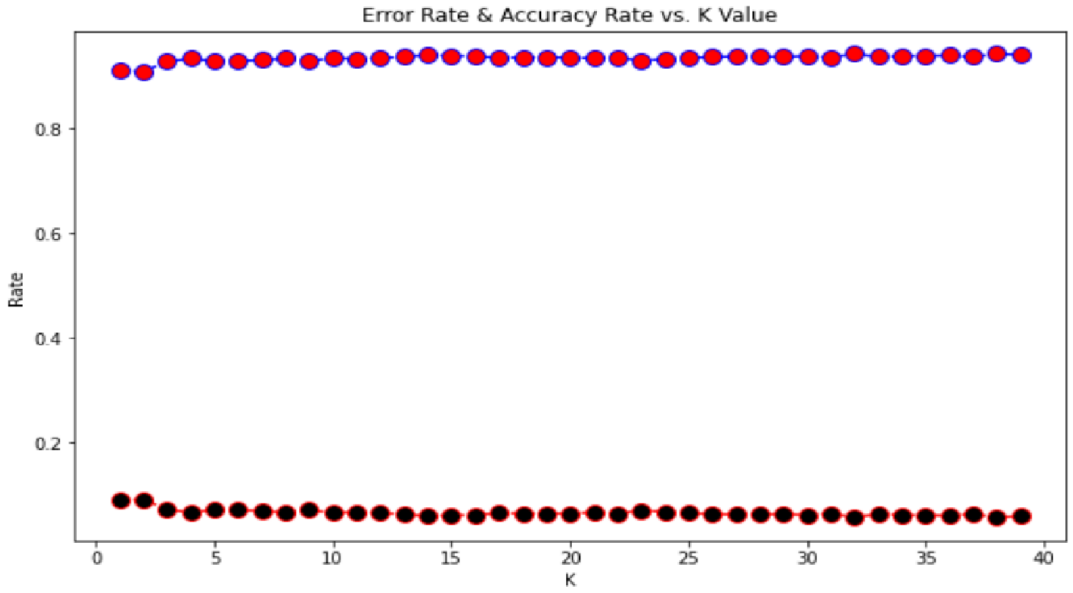
Out[31]:
```
[0.08900000000000008,
 0.09099999999999997,
 0.07199999999999984,
 0.06600000000000006,
 0.07100000000000006,
 0.07099999999999995,
 0.06899999999999973,
 0.06599999999999984,
 0.07100000000000006,
 0.06499999999999984,
 0.0670000000000017,
 0.06499999999999984,
 0.06299999999999994,
 0.05899999999999994,
 0.06099999999999994,
 0.06099999999999994,
 0.06500000000000006,
 0.06399999999999983,
 0.06399999999999983,
 0.06399999999999983,
```

In [32]:
```
plt.figure(figsize=(10,6))
plt.plot(range(1,40),error_rate,color='blue', linestyle='dashed', marker='o' ,markerfacecolor='red', markersize=10)
plt.title('Error Rate vs. K Value')
plt.xlabel('K')
plt.ylabel('Error Rate')
```

Out[32]:
```
Text(0, 0.5, 'Error Rate')
```



In [33]:
```
plt.figure(figsize=(10,6))
plt.plot(range(1,40),error_rate,color='red', linestyle='dashed', marker='o',markerfacecolor='black', markersize=10)
plt.plot(range(1,40),accuracy_rate,color='blue', linestyle='dashed', marker='o',markerfacecolor='red', markersize=10)
plt.title('Error Rate & Accuracy Rate vs. K Value')
plt.xlabel('K')
plt.ylabel('Rate')
plt.show()
```



Here we can see that that after around K>23 the error rate just tends to hover around 0.06-0.05 Let retrain the model with that and check the classification report of model.

In [34]:
```
# WITH K=23
knn = KNeighborsClassifier(n_neighbors=23)
knn.fit(X_train,y_train)
pred = knn.predict(X_test)
```

In [35]:
```
print('WITH K=23')
print('\n')
print(classification_report(y_test,pred))
```

```
WITH K=23


              precision    recall  f1-score   support

           0       0.95      0.91      0.93       153
           1       0.91      0.95      0.93       147

    accuracy                           0.93       300
   macro avg       0.93      0.93      0.93       300
weighted avg       0.93      0.93      0.93       300
```

Experiment-6

Aim: Implement K-Mean Clustering in Python using Jupyter Notebook.

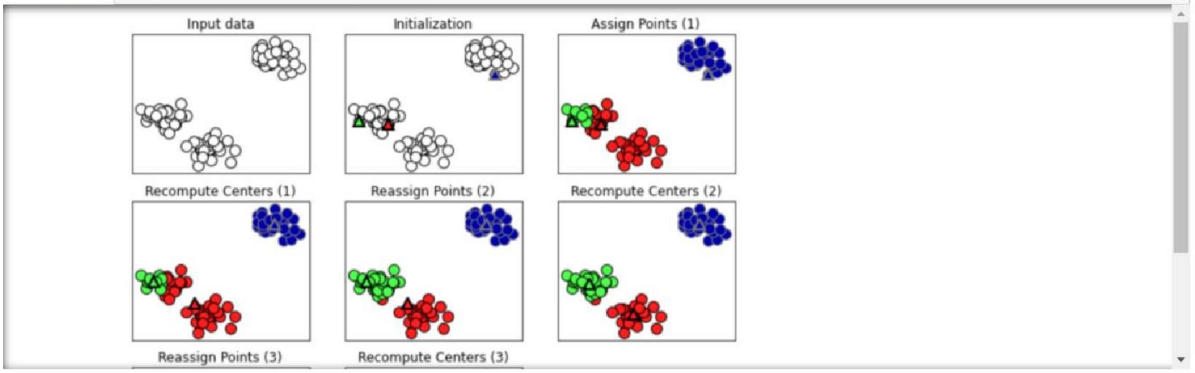
K Means Clustering

K Means Clustering is an unsupervised learning algorithm which groups the unlabeled dataset into different clusters. Here, K denotes the number of clusters that need to be created in the process.

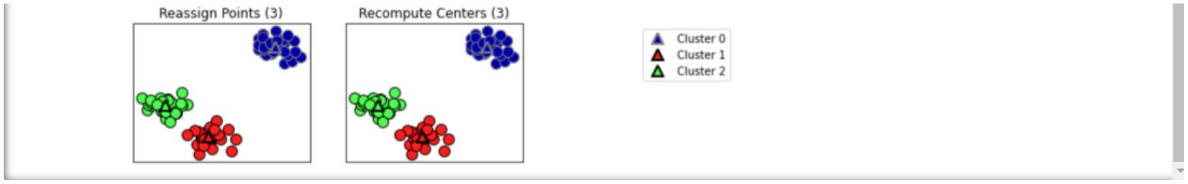
In [1]:
```
import mglearn
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import cdist
from sklearn.datasets import make_blobs
from sklearn.datasets import make_moons
from sklearn.cluster import KMeans
from sklearn.model_selection import train_test_split
from sklearn.decomposition import NMF
from sklearn.decomposition import PCA
from sklearn.datasets import fetch_lfw_people
```

Steps of a K Means clustering algorithm

In [2]:
```
mglearn.plots.plot_kmeans_algorithm()
```



First, initialization takes place where randonly 3 points are selected and marked as cluster centres. After this, the points are alloted to its data center. After this the data center updates itself and assign itself the mean of updated points. This process repeats two times after which the center does not change and the algorithm stops.

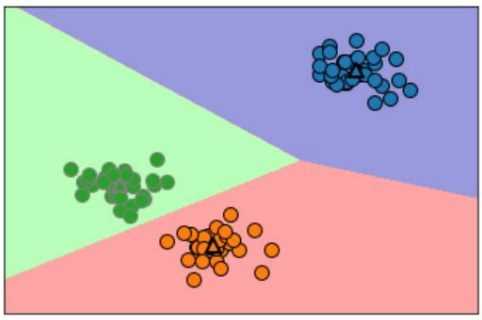


First, initialization takes place where randonly 3 points are selected and marked as cluster centres. After this, the points are alloted to its data center. After this the data center updates itself and assign itself the mean of updated points. This process repeats two times after which the center does not change and the algorithm stops.

It has the following three steps :

- Initialization - Randomly 3 points are selected and marked as cluster centers.
- Assigning - The points closest to a cluster centre are assigned those clusters.
- Computation - The cluster centres are found again and readjusted and the process repeats from assigning.

In [3]:
```
mglearn.plots.plot_kmeans_boundaries()
```



Applying K Means clustering on make_blobs() dataset

In [4]:
```
x, y = make_blobs(random_state=1)
```

In [5]:
```
kmeans = KMeans(n_clusters=3)
kmeans.fit(x)
```

Out[5]:
```
KMeans(n_clusters=3)
```

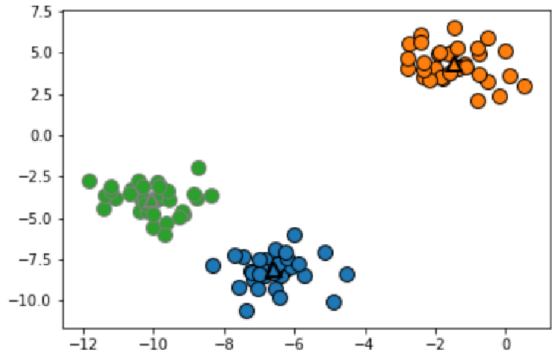
In [6]:
```
print("\nCluster labels {}".format(kmeans.labels_))

## During the fitting of data in KMeans, the clustering takes place and each x value is assigned a label. Here, we asked for 3 la
```

```
Cluster labels [1 2 2 2 0 0 0 2 1 1 2 2 0 1 0 0 0 1 2 2 0 2 0 1 2 0 0 1 1 0 1 1 0 1 2 0 2
 2 2 0 0 2 1 2 2 0 1 1 1 1 2 0 0 0 1 0 2 2 1 1 2 0 0 2 2 0 1 0 1 2 2 2 0 1
 1 2 0 0 1 2 1 2 2 0 1 1 1 1 2 1 0 1 1 2 2 0 0 1 0 1]
```

In [7]:
```
# print(kmeans.predict(x))

## predict() method is used to assign cluster labels to new points (for testing purposes). Here, since the labels_ and predict(x)
```

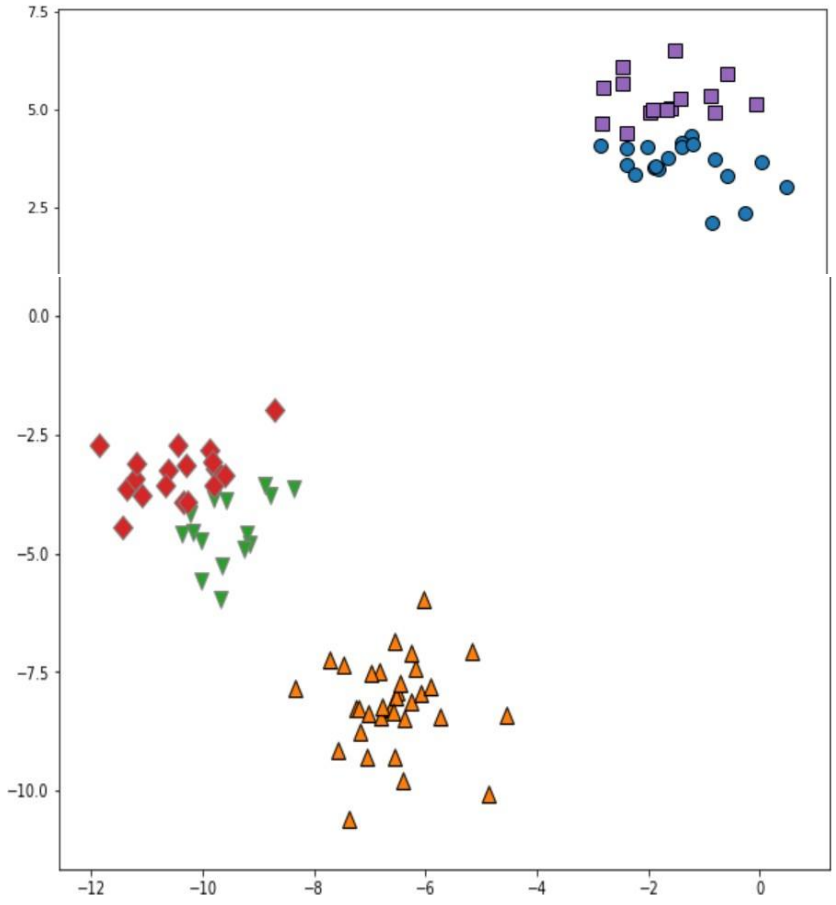
In [8]:
```
mglearn.discrete_scatter(x[:, 0], x[:, 1], kmeans.labels_, markers='o')
mglearn.discrete_scatter(kmeans.cluster_centers_[:, 0],
                         kmeans.cluster_centers_[:, 1], [0, 1, 2],
                         markers='^', markeredgewidth=2)
```

Out[8]:
```
[<matplotlib.lines.Line2D at 0x270f7ac4790>,
 <matplotlib.lines.Line2D at 0x270f7ac4a60>,
 <matplotlib.lines.Line2D at 0x270f7ac4d30>]
```



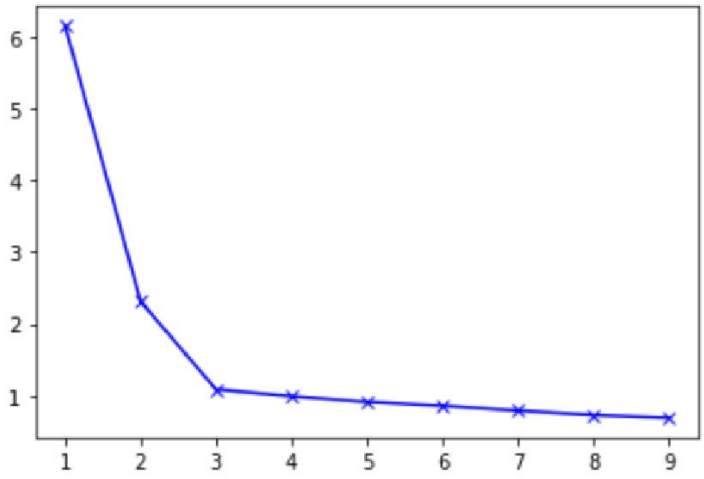
In [9]:
```
# Using 5 cluster centres

fig, axes = plt.subplots(1, 1, figsize=(10, 10))
kmeans = KMeans(n_clusters=5)
kmeans.fit(x)
assignments = kmeans.labels_

mglearn.discrete_scatter(x[:, 0], x[:, 1], assignments, ax=axes)
```

Out[9]:
```
[<matplotlib.lines.Line2D at 0x270f7b41c70>,
 <matplotlib.lines.Line2D at 0x270f7b41f40>,
 <matplotlib.lines.Line2D at 0x270f7b51250>,
 <matplotlib.lines.Line2D at 0x270f7b51520>,
 <matplotlib.lines.Line2D at 0x270f7b517f0>]
```



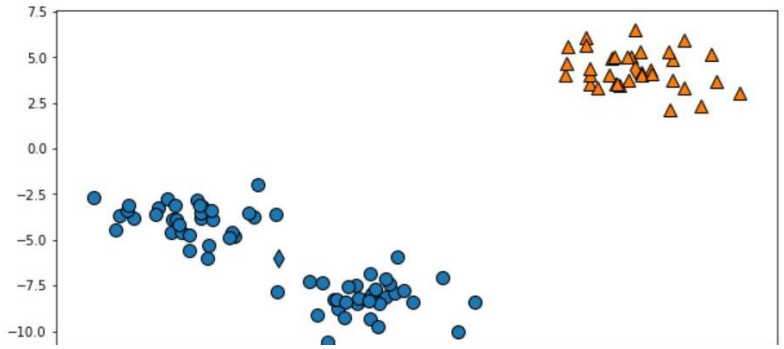
Elbow plot using distortions

In [10]:
```
K = range(1, 10)
distortions = []

for k in K:
    kmodel = KMeans(n_clusters=k).fit(x)
    kmodel.fit(x)

    distortions.append(sum(np.min(cdist(x, kmodel.cluster_centers_, 'euclidean'), axis=1)) / x.shape[0])
plt.plot(K, distortions, 'bx-')
plt.show()
```



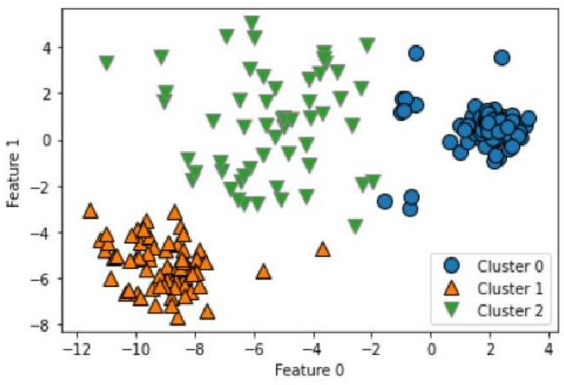
In [11]:
```
# Using 2 cluster centers

fig, axes = plt.subplots(1, 1, figsize=(10, 5))
kmeans = KMeans(n_clusters=2)
kmeans.fit(x)
assignments = kmeans.labels_

mglearn.discrete_scatter(x[:, 0], x[:, 1], assignments, ax=axes)
mglearn.discrete_scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], [0, 1], markers='d')
```

Out[11]:
```
[<matplotlib.lines.Line2D at 0x270f7570bb0>,
 <matplotlib.lines.Line2D at 0x270f7570d00>]
```



In [12]:
```
x_varied, y_varied = make_blobs(n_samples=200, cluster_std=[1.0, 2.5, 0.5],
                                random_state=170)
y_pred = KMeans(n_clusters=3, random_state=0).fit_predict(x_varied)
mglearn.discrete_scatter(x_varied[:, 0], x_varied[:, 1], y_pred)
plt.legend(["Cluster 0", "Cluster 1", "Cluster 2"])
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

Out[12]:
```
Text(0, 0.5, 'Feature 1')
```



Experiment-7

Aim: Implement CNN in Python using Jupyter Notebook.

In [2]:
```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D
from tensorflow.keras.layers import MaxPool2D
from tensorflow.keras.layers import Flatten
from tensorflow.keras.layers import Dropout
from tensorflow.keras.layers import Dense
```

In [4]:
```
(X_train,y_train),(X_test,y_test)=mnist.load_data()
#reshaping data
X_train = X_train.reshape((X_train.shape[0],X_train.shape[1],X_train.shape[2],1))
X_test = X_test.reshape((X_test.shape[0],X_test.shape[1],X_test.shape[2],1))
#checking the shape after reshaping
print(X_train.shape)
print(X_test.shape)
#normalizing the pixel values
X_train=X_train/255
X_test=X_test/255
```

```
(60000, 28, 28, 1)
(10000, 28, 28, 1)
```

In [5]:
```
model=Sequential()
#adding convolution layer
model.add(Conv2D(32,(3,3),activation='relu',input_shape=(28,28,1)))
#adding pooling layer
model.add(MaxPool2D(2,2))
#adding fully connected layer
model.add(Flatten())
model.add(Dense(100,activation='softmax'))
#compiling the model
model.compile(loss='sparse_categorical_crossentropy',optimizer='adam',metrics=['accuracy'])
#fitting the model
model.fit(X_train,y_train,epochs=10)
```

```
Epoch 1/10
1875/1875 [==============================] - 26s 13ms/step - loss: 0.2415 - accuracy: 0.9322
Epoch 2/10
1875/1875 [==============================] - 24s 13ms/step - loss: 0.0813 - accuracy: 0.9765
Epoch 3/10
1875/1875 [==============================] - 26s 14ms/step - loss: 0.0609 - accuracy: 0.9818
Epoch 4/10
1875/1875 [==============================] - 26s 14ms/step - loss: 0.0496 - accuracy: 0.9855
Epoch 5/10
1875/1875 [==============================] - 26s 14ms/step - loss: 0.0427 - accuracy: 0.9871
Epoch 6/10
1875/1875 [==============================] - 26s 14ms/step - loss: 0.0360 - accuracy: 0.9887
 Epoch 7/10
 1875/1875 [==============================] - 26s 14ms/step - loss: 0.0300 - accuracy: 0.9908
 Epoch 8/10
 1875/1875 [==============================] - 26s 14ms/step - loss: 0.0260 - accuracy: 0.9921
 Epoch 9/10
 1875/1875 [==============================] - 26s 14ms/step - loss: 0.0227 - accuracy: 0.9930
 Epoch 10/10
 1875/1875 [==============================] - 25s 13ms/step - loss: 0.0182 - accuracy: 0.9946
```

Out[5]:
```
<keras.callbacks.History at 0x1fe02f93d60>
```

In [6]:
```
model.evaluate(X_test,y_test)
```

```
313/313 [==============================] - 2s 5ms/step - loss: 0.0597 - accuracy: 0.9820
```

Out[6]:
```
[0.05966240167617798, 0.9819999933242798]
```

In [7]:
```
model.evaluate(X_test,y_test)
```

```
313/313 [==============================] - 2s 5ms/step - loss: 0.0597 - accuracy: 0.9820
```

Out[7]:
```
[0.05966240167617798, 0.9819999933242798]
```