

The necessary library modules required for data analysis, data visualization and machine learning are imported.

```
In [1]: import pandas as pd #library for dataframe
import numpy as np #library for vectorized computation
import os #library for interacting with our OS
import seaborn as sns #library for data visualization
import matplotlib.pyplot as plt
from sklearn.metrics import r2_score
from sklearn.metrics import accuracy_score, precision_score, recall_score, roc_auc_score, roc_curve, classification_report
from sklearn.metrics import permutation_importance
from sklearn.preprocessing import StandardScaler
from sklearn import SimpleImputer
from sklearn.utils import resample
from sklearn.svm import SVC
from sklearn.pipeline import make_pipeline

import warnings
warnings.filterwarnings('ignore')

In [2]: def load_dataset(file_name):
    """Arg: file_name- the name of the dataset to be loaded as dataframe
    Return: Dataframe of the dataset"""
    return pd.read_csv(file_name)

In [3]: dataset=load_dataset("diabetes.csv") # function call and the object is pointed to the variable called "dataset"

In [4]: dataset.head() # head method displays first 5 rows

Out[4]:
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
0	6	148	72	35	0.336	33.6	0.627	50	1
1	1	85	66	29	0.336	33.6	0.351	31	0
2	8	183	64	0	0.233	33.6	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 [5]: dataset.columns # different columns in the dataset

Out[5]: Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
        'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome'],
        dtype='object')
```

## Dataset Exploration and Visualization

```
In [6]: dataset.info() # method to display the features along with number of values in it.

Out[6]:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 768 entries, 0 to 767
Data columns (total 9 columns):
Pregnancies    768 non-null int64
Glucose        768 non-null int64
BloodPressure  768 non-null int64
SkinThickness  768 non-null int64
Insulin        768 non-null float64
BMI            768 non-null float64
DiabetesPedigreeFunction  768 non-null float64
Age           768 non-null int64
Outcome        768 non-null int64
dtypes: float64(2), int64(7)
memory usage: 54.1 KB

In [7]: dataset.dtypes # method to display only the data type of all the values present in the dataset

Out[7]:
Pregnancies    int64
Glucose        int64
BloodPressure  int64
SkinThickness  int64
Insulin        int64
BMI            float64
DiabetesPedigreeFunction  float64
Age            int64
Outcome        int64
dtype: object

In [8]: dataset.isna().any() # method to display for the presence of any null values, True means presence of null values,
# False means absence of null values

Out[8]:
Pregnancies    False
Glucose        False
BloodPressure  False
SkinThickness  False
Insulin        False
BMI            False
DiabetesPedigreeFunction  False
Age            False
Outcome        False
dtype: bool

In [9]: dataset.isna().sum() # method to display the number of missing values if it is present
# 0 indicates no missing values

Out[9]:
Pregnancies    0
Glucose        0
BloodPressure  0
SkinThickness  0
Insulin        0
BMI            0
DiabetesPedigreeFunction  0
Age            0
Outcome        0
dtype: int64

In [10]: dataset.describe() # method to display the various numerical properties of the column features.

Out[10]:
```

	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	3.845952	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348951
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.478951
min	0.000000	0.000000	0.000000	0.000000	0.000000	27.300000	0.043750	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.247500	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	32.000000	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

```
In [11]: dataset.shape # shape of the dataset

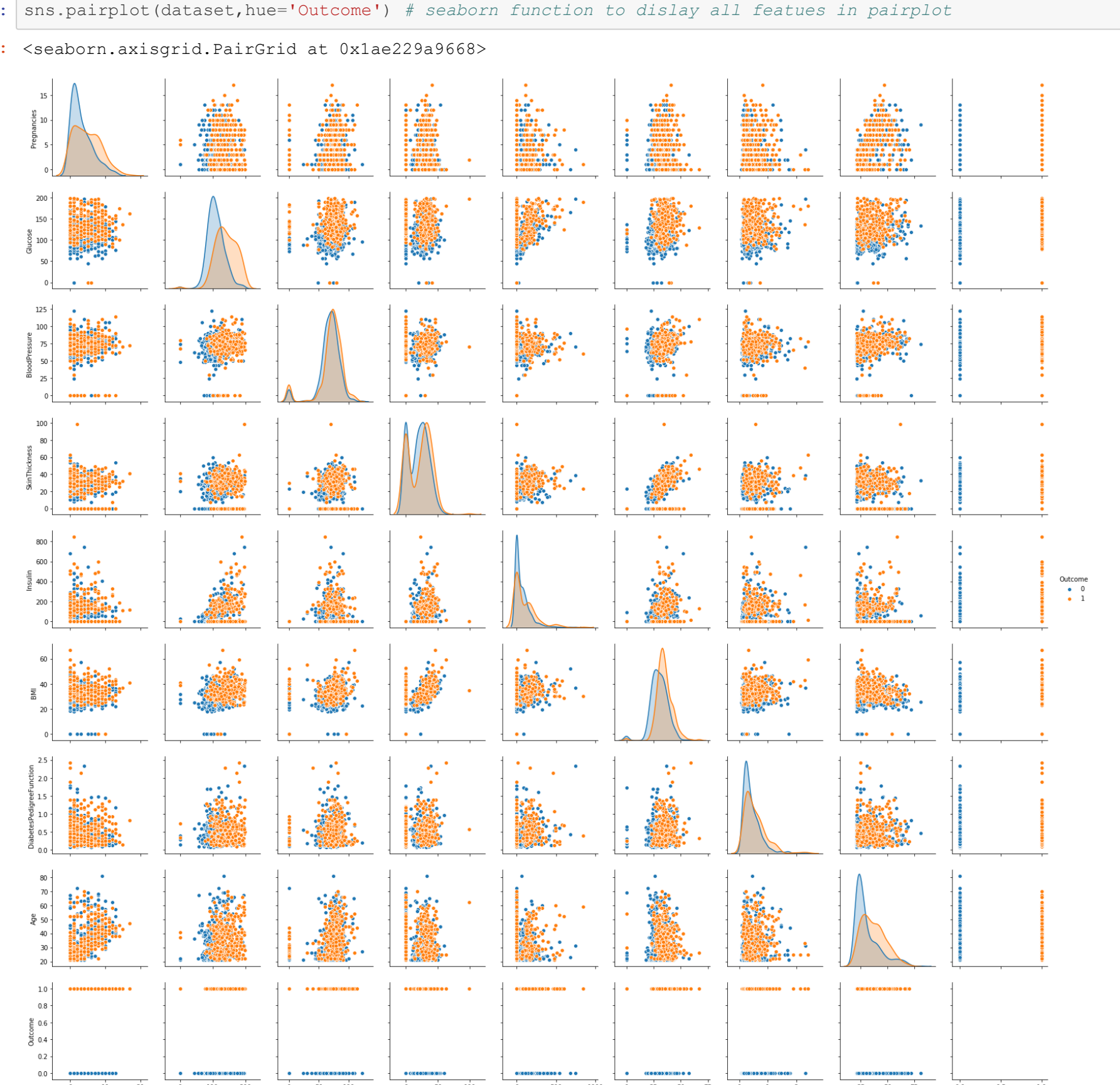
Out[11]: (768, 9)

In [12]: dataset.hist(bins=50, figsize=(20,15)); # method to draw a histogram of all features in the dataset

Out[12]:
```

```
In [13]: plt.figure(figsize=(10,8))
sns.heatmap(dataset.corr(),linecolor='white',linewidth=2); # seaborn function to draw a heatmap base d'n Feature

# correlation
```



A pairs plot allows us to see both distribution of single variables and relationships between two variables.

```
In [14]: sns.pairplot(dataset,hue='Outcome') # seaborn function to display all features in pairplot

Out[14]: <seaborn.axisgrid.PairGrid at 0x1ae229a668>
```

```
In [15]: sns.pairplot(dataset[['Insulin','BMI','Outcome']],hue='Outcome')
# only selected features

Out[15]: <seaborn.axisgrid.PairGrid at 0x1ae22b3c5f8>
```

## kNN Algorithm - Machine Learning

The Machine learning part consisting of segregating the input features into dependent variables and independent variables, training the model and testing.

```
In [16]: # Create features and Labels
features = dataset.drop(['Outcome'], axis=1) # independent variable
Labels = dataset['Outcome'] # dependent variable

In [17]: # Create training and test set
features_train, features_test, labels_train, labels_test = \
train_test_split(features, labels, test_size=0.25, random_state=15)

In [18]: # Initialize the algorithm
classifier = KNeighborsClassifier(n_neighbors=5)

In [19]: # Fit data
classifier.fit(features_train, labels_train)

Out[19]: KNeighborsClassifier()

In [20]: #prediction
pred = classifier.predict(features_test)

In [21]: accuracy = accuracy_score(labels_test, pred) # accuracy
print('Accuracy: {:.2f}'.format(accuracy))

precision = precision_score(labels_test, pred) # precision
print('Precision: {:.2f}'.format(precision))

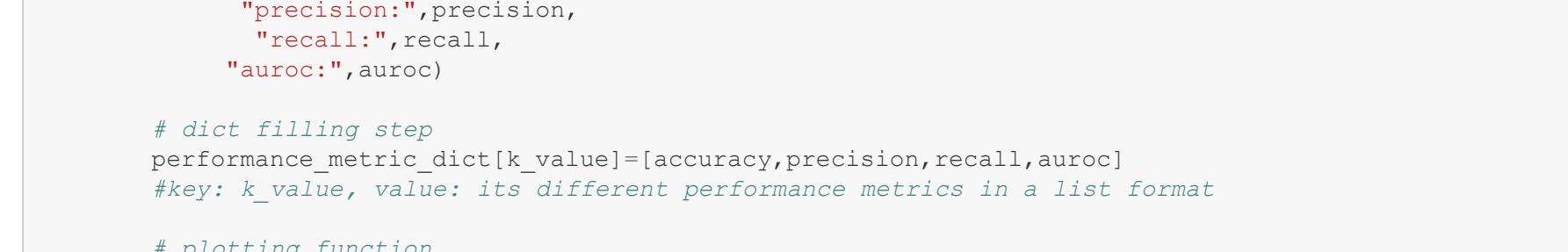
recall = recall_score(labels_test, pred) # recall
print('Recall: {:.2f}'.format(recall))

aucroc = roc_auc_score(labels_test, pred) # area under curve
print('AUROC score: {:.2f}'.format(aucroc))

Accuracy: 0.72
Precision: 0.58
Recall: 0.54
AUROC score: 0.67

In [22]: fpr, tpr, thresholds = roc_curve(labels_test, pred)
# defining a function to plot the AUROC-curve
def plot_roc_curve(fpr, tpr):
    plt.plot(fpr, tpr, color='orange', label='ROC')
    plt.plot([0, 1], [0, 1], color='darkblue', linestyle='--')
    plt.xlabel('False Positive Rate',fontsize=15,fontweight='bold')
    plt.ylabel('True Positive Rate',fontsize=15,fontweight='bold')
    plt.title('Receiver Operating Characteristic (ROC) Curve')
    plt.legend()
    plt.show()

# Calling the function to finally plot the curve
plot_roc_curve(fpr, tpr)
```



Elegant way of performing kNN algorithm using a function to collect all necessary parameters.

function to perform kNN algorithm - Input : k value

```
In [23]: def performing_knn(k_value): # for one k value
    """function to perform kNN algorithm for one input k value at a time
    Arg- k_value: k value to be used for the algorithm
    Returns: Different performance metrics with the AUC plot """
    print("Performing K nearest neighbor algorithm for k value :{}".format(k_value))
    classifier = KNeighborsClassifier(n_neighbors=k_value)
    classifier.fit(features_train, labels_train) # algorithm initialization
    pred = classifier.predict(features_test) # model training
    accuracy = accuracy_score(labels_test, pred) # accuracy
    precision = precision_score(labels_test, pred) # precision
    recall = recall_score(labels_test, pred) # recall
    aucroc = roc_auc_score(labels_test, pred) # aucroc
    print("accuracy",accuracy,
          "precision",precision,
          "recall",recall,
          "aucroc",aucroc)

    fpr, tpr, thresholds = roc_curve(labels_test, pred)
    plt.plot(fpr, tpr, color='orange', label='ROC')
    plt.plot([0, 1], [0, 1], color='darkblue', linestyle='--')
    plt.xlabel('False Positive Rate',fontsize=15,fontweight='bold',family='Arial')
    plt.ylabel('True Positive Rate',fontsize=15,fontweight='bold',family='Arial')
    plt.title('Receiver Operating Characteristic (ROC) Curve')
    plt.legend()
    plt.show()

    #return accuracy,precision,recall,aucroc

In [24]: [performing_knn(k) for k in [3,5]]

Performing K nearest neighbor algorithm for k value :3
accuracy: 0.6708333333333334 precision: 0.5087719298245614 recall: 0.4603174603174603 aucroc: 0.621631
5983757844

Receiver Operating Characteristic (ROC) Curve

True Positive Rate

False Positive Rate

Performing K nearest neighbor algorithm for k value :5
accuracy: 0.71875 precision: 0.57621186440678 recall: 0.5396825396825397 aucroc: 0.6729420450350682

Receiver Operating Characteristic (ROC) Curve

True Positive Rate

False Positive Rate

In [24]: [None, None]
```

function to perform kNN algorithm - Input : list of k values in a list

```
In [25]: performance_metric_dict={} # dict to hold the values of performance metrics
def performing_knn_list(k_value_list): # for a list of k values
    """function to perform kNN algorithm for a list of k as input values
    Arg- k_value_list: list with k values to be used for the algorithm
    Returns: Different performance metrics with the AUC plot """
    for k_value in k_value_list:
        print("Performing K nearest neighbor algorithm for k value :{}".format(k_value))
        classifier = KNeighborsClassifier(n_neighbors=k_value)
        classifier.fit(features_train, labels_train) # algorithm initialization
        pred = classifier.predict(features_test) # model training
        accuracy = accuracy_score(labels_test, pred) # accuracy
        precision = precision_score(labels_test, pred) # precision
        recall = recall_score(labels_test, pred) # recall
        aucroc = roc_auc_score(labels_test, pred) # aucroc
        print("accuracy",accuracy,
              "precision",precision,
              "recall",recall,
              "aucroc",aucroc)

        # dict filling step
        performance_metric_dict[k_value]=[accuracy,precision,recall,aucroc]
        #key: k_value, value: its different performance metrics in a list format

    # plotting function
    fpr, tpr, thresholds = roc_curve(labels_test, pred)
    plt.plot(fpr, tpr, color='orange', label='ROC')
    plt.plot([0, 1], [0, 1], color='darkblue', linestyle='--')
    plt.xlabel('False Positive Rate',fontsize=15,fontweight='bold',family='Arial')
    plt.ylabel('True Positive Rate',fontsize=15,fontweight='bold',family='Arial')
    plt.title('Receiver Operating Characteristic (ROC) Curve')
    plt.legend()
    plt.show()
    print("====")

    #return accuracy,precision,recall,aucroc

In [26]: performing_knn_list([3,5,7,9,15,40])

Performing K nearest neighbor algorithm for k value :3
accuracy: 0.6708333333333334 precision: 0.5087719298245614 recall: 0.4603174603174603 aucroc: 0.621631
5983757844

Receiver Operating Characteristic (ROC) Curve

True Positive Rate

False Positive Rate

Performing K nearest neighbor algorithm for k value :5
accuracy: 0.71875 precision: 0.57621186440678 recall: 0.5396825396825397 aucroc: 0.6729420450350682

Receiver Operating Characteristic (ROC) Curve

True Positive Rate

False Positive Rate

Performing K nearest neighbor algorithm for k value :7
accuracy: 0.7239583333333334 precision: 0.5833333333333334 recall: 0.5555555555555556 aucroc: 0.680878
552915763

Receiver Operating Characteristic (ROC) Curve

True Positive Rate

False Positive Rate

Performing K nearest neighbor algorithm for k value :9
accuracy: 0.7552083333333334 precision: 0.6481481481481481 recall: 0.5555555555555556 aucroc: 0.704134
3669250646

Receiver Operating Characteristic (ROC) Curve

True Positive Rate

False Positive Rate

Performing K nearest neighbor algorithm for k value :15
accuracy: 0.765625 precision: 0.6666666666666666 recall: 0.5714285714285714, 0.7159468438538206

Receiver Operating Characteristic (ROC) Curve

True Positive Rate

False Positive Rate

Performing K nearest neighbor algorithm for k value :40
accuracy: 0.71875 precision: 0.6451612903225806 recall: 0.31746031746031744 aucroc: 0.61609449981543

Receiver Operating Characteristic (ROC) Curve

True Positive Rate

False Positive Rate
```

The performance\_metric\_dict holds the k values as keys and different evaluation metrics enclosed in a list as their values.

```
In [27]: performance_metric_dict

Out[27]: {3: [0.6708333333333334,
0.5087719298245614,
0.4603174603174603,
0.6216315983757844],
5: [0.71875,
0.57621186440678,
0.5396825396825397,
0.6729420450350682],
7: [0.7239583333333334,
0.5833333333333334,
0.5555555555555556,
0.680878552915763],
9: [0.7552083333333334,
0.6481481481481481,
0.5555555555555556,
0.7041343669250646],
15: [0.765625,
0.6666666666666666,
0.5714285714285714,
0.7159468438538206],
40: [0.71875,
0.6451612903225806,
0.31746031746031744,
0.61609449981543]}

In [28]: #plotting different performance metrics in a line plot
def plotting_performance_metrics():
    plt.figure(figsize=(10,8))
    plt.plot(list(performance_metric_dict.keys()),[i] for i in performance_metric_dict.values()),"b",
            "color='blue'",
            linewidth=3,markersize=8,label='Accuracy')
    plt.plot(list(performance_metric_dict.keys()),[i] for i in performance_metric_dict.values()),"r",
            "color='red'",
            linewidth=3,markersize=8,label='Precision')
    plt.plot(list(performance_metric_dict.keys()),[i] for i in performance_metric_dict.values()),"g",
            "color='cyan'",
            linewidth=3,markersize=10,label='Recall')
    plt.plot(list(performance_metric_dict.keys()),[i] for i in performance_metric_dict.values()),"m",
            "color='magenta'",
            linewidth=3,markersize=8,label='AUC')
    plt.legend()
    plt.xlabel("Different k values in the algorithm",fontsize=15)
    plt.ylabel("Different Performance Metrics",fontsize=15);

In [29]: plotting_performance_metrics()

Out[29]:
```

On observing this line plot, the accuracy, precision, recall, AUC was high in k value = 15. Therefore, the best k is 15.

```
In [30]: !pip install --upgrade scikit-learn
```

## Feature Importance using kNN classifier

```
In [31]: results = permutation_importance(classifier, features_train, labels_train, scoring='accuracy', random_state=15)
# get importance
importance = results.importances_mean

# summarize feature importance
feature_importance_dict={}
print("Different features with their importances:")
print()
for i in range(len(importance)):
    print(features_train.columns[i],":",importance[i])
    feature_importance_dict[features_train.columns[i]]=importance[i]

# plot feature importance
plt.bar(features_train.columns,importance)
plt.xticks(rotation=90)
plt.xlabel("Different features in the dataset",fontsize=15)
plt.ylabel("Feature Importance Measures",fontsize=15)
plt.show()

#https://machinelearningmastery.com/calculate-feature-importance-with-python/
#https://scikit-learn.org/stable/modules/generated/sklearn.inspection.permutation_importance.html

Different features with their importances:
Pregnancies : 0.005208333333333337
Glucose : 0.15555555555555556
BloodPressure : 0.015972222222222256
SkinThickness : 0.014583333333333337
Insulin : 0.05833333333333333
BMI : 0.004861111111111111
DiabetesPedigreeFunction : 0.0003472222222222221
Age : 0.020833333333333334
```



```
In [32]: #[k: v for k, v in sorted(feature_importance_dict.items(), key=lambda item: item[1],reverse=True)]
print("The names of the features with decreasing strength of feature importance are as follows:")
[k for k, v in sorted(feature_importance_dict.items(), key=lambda item: item[1],reverse=True)]

The names of the features with decreasing strength of feature importance are as follows:
Out[32]: ['Glucose',
'Insulin',
'Age',
'BloodPressure',
'SkinThickness',
'Pregnancies',
'BMI',
'DiabetesPedigreeFunction']
```

## Scaling

### Standardization

First, Standardization step was performed.

```
In [33]: x_sc=StandardScaler()
features_train_scaled = x_sc.fit_transform(features_train)
features_test_scaled = x_sc.transform(features_test)
```



In [34]: features\_train\_scaled, columns=features.columns).describe()

Out [34]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
count	576.000000	576.000000	576.000000	576.000000	576.000000	576.000000	576.000000	576.000000
mean	3.860206	120.618956	69.319444	20.26164	80.3611	31.864028	0.471340	33.260417
std	3.387170	32.112952	19.114357	15.702044	111.294999	7.836445	0.322216	11.623841
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.084000	21.000000
25%	1.000000	100.000000	62.000000	0.000000	0.000000	27.370000	0.243500	24.000000
50%	3.000000	117.000000	72.000000	23.000000	36.000000	32.000000	0.378000	29.000000
75%	6.000000	139.250000	80.000000	32.000000	130.000000	36.500000	0.637000	40.250000
max	17.000000	199.000000	122.000000	60.000000	660.000000	67.100000	2.288000	81.000000

In [35]: pd.DataFrame(features\_train\_scaled, columns=features.columns).describe()

Out [35]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
count	576.000000+02	576.000000+02	576.000000+02	576.000000+02	576.000000+02	576.000000+02	576.000000+02	576.000000+02
mean	-2.637184e-17	-0.112761e+00	6.129335e-17	-5.629214e-17	6.553400e-18	4.257782e-16	1.937100e-16	2.328384e-16
std	1.0000889e+00	1.000889e+00	1.000889e+00	1.000889e+00	1.000889e+00	1.000889e+00	1.000889e+00	1.000889e+00
min	-1.146573e+00	-3.759321e+00	-3.629716e+00	-1.276115e+00	-7.226833e-01	-0.722222e+00	-1.203150e+00	-1.055678e-00
25%	-2.801005e-01	-0.426060e-01	-3.832620e-01	-1.276115e+00	-7.226833e-01	-5.758921e-02	-7.084902e-01	-7.973639e-00
50%	-8.059815e-01	-1.127604e-01	-1.403597e-01	-1.276115e+00	-3.989371e-01	1.481195e-02	-2.899320e-01	-3.866841e-00
75%	6.263820e-01	5.807046e-01	5.652570e-01	7.626092e-01	4.463999e-01	5.895511e-01	5.145694e-01	6.016349e-00
max	3.8768030e+00	2.442942e+00	2.758460e+00	2.548588e+00	5.392521e+00	4.497775e+00	5.642878e+00	4.110956e+00

In [36]: pd.DataFrame(features\_test\_scaled, columns=features.columns).describe()

Out [36]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age
count	192.000000	192.000000	192.000000	192.000000	192.000000	192.000000	192.000000	192.000000
mean	-0.941554	0.034648	-0.944917	0.129193	-0.022023	0.055456	0.006960	-0.006727
std	0.981657	0.085357	0.125492	1.058181	1.138954	1.025649	1.112492	1.049702
min	-1.146573	-1.854255	-3.629716	-1.276115	-0.722683	-0.472222	-1.221787	-1.055678
25%	-0.851081	-0.775066	-0.304719	-1.276115	-0.722683	-0.649331	-0.706937	-0.767364
50%	-0.260095	-0.117264	0.035635	-1.892855	-0.659733	0.059514	-0.320994	-0.452946
75%	0.626382	0.705373	0.586438	0.263782	0.277782	0.634263	0.392652	0.666413
max	2.694832	2.336088	2.025398	5.031435	6.885350	3.514333	6.052894	3.335654

In [37]: from sklearn.preprocessing import MinMaxScaler

In [38]: x\_mm = MinMaxScaler()  
features\_train\_minmaxscaled = x\_mm.fit\_transform(features\_train)  
features\_test\_minmaxscaled = x\_mm.transform(features\_test)

## kNN algorithm for Transformed data

The kNN algorithm was applied for these scaled/normalized data.

In [39]:

```
def performing_knn_list_scaling(k_value_list, feature_train, feature_test, feature_test_scaled):  
    """function to perform kNN algorithm for a list of k as input values  
    Arg: k_value_list: list with k values to be used for the algorithm  
    Returns: Different performance metrics with the AUC plot  
    """  
    performance_metric_dict = {} # dict to hold the values of performance metrics  
    for k_value in k_value_list:  
        #print("Performing K nearest neighbor algorithm for k value: {}".format(k_value))  
        classifier = KNeighborsClassifier(n_neighbors=k_value) # algorithm initialization  
        labels_train = classifier.fit(feature_train, labels_train) # model training  
        pred = classifier.predict(feature_test_scaled) # model testing  
        accuracy = accuracy_score(labels_test, pred) # accuracy  
        precision = precision_score(labels_test, pred) # precision  
        recall = recall_score(labels_test, pred) # recall  
        auroc = roc_auc_score(labels_test, pred) # auroc  
        # dict filling step  
        performance_metric_dict[k_value] = [accuracy, precision, recall, auroc]  
        # Key: k_value, value: its different performance metrics in a list format  
    return performance_metric_dict
```

Out [39]:

k_value	accuracy	precision	recall	auroc
1	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
5	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
7	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
9	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
15	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
20	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886

## kNN for Standardized data

In [40]: standardized\_knn\_performing\_knn\_list\_scaling([3, 5, 7, 9, 15, 20], features\_train\_scaled, features\_test\_scaled)

Out [40]:

k_value	accuracy	precision	recall	auroc
3	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
5	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
7	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
9	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
15	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
20	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886

## kNN for Normalized data

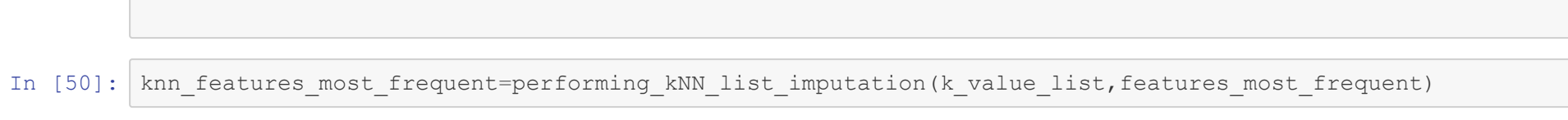
In [41]: normalized\_knn\_performing\_knn\_list\_scaling([3, 5, 7, 9, 15, 20], features\_train\_minmaxscaled, features\_test\_minmaxscaled)

Out [41]:

k_value	accuracy	precision	recall	auroc
3	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
5	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
7	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
9	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
15	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
20	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886

In [42]:

```
# Barplot to observe the performance of standardized and normalized data.  
rcParams['figure.figsize'] = 10, 10  
bar_width = 0.5  
width = 0.2  
for i in normalized_knn.keys():  
    index = range(len(k_values))  
    rects = plt.bar(index, list([i] for i in normalized_knn.values()))  
    color = 'blue'  
    align = 'center'  
    label = 'Standardization', width = 1*width  
    plt.bar(index, list([i] for i in standardized_knn.values()))  
    color = 'green'  
    align = 'center'  
    label = 'Standardization', width = 1*width  
plt.xlabel('k values in the kNN algorithm', fontsize=15)  
plt.ylabel('Accuracy measures', fontsize=15)  
plt.title('Accuracy comparison across the standardized and normalized data')  
plt.xticks(index, k_values)  
plt.legend()  
plt.show()
```



On Observing, the barplot it can be concluded that the standardized data performed well and hence the standardized data will be used for further analysis.

## Missing Value Imputation

In [43]: imputer = SimpleImputer(missing\_values = 0, strategy = 'mean') # mean imputation  
imputer = imputer.fit(features)  
features\_mean = imputer.transform(features)

In [44]: imputer = SimpleImputer(missing\_values = 0, strategy = 'median') # median imputation  
imputer = imputer.fit(features)  
features\_median = imputer.transform(features)

In [45]: imputer = SimpleImputer(missing\_values = 0, strategy = 'most\_frequent') # mode imputation  
imputer = imputer.fit(features)  
features\_most\_frequent = imputer.transform(features)

In [46]:

```
def performing_knn_list_imputation(k_value_list, features): # for a list of k values  
    """function to perform kNN algorithm for a list of k as input values  
    Arg: k_value_list: list with k values to be used for the algorithm  
    Returns: Different performance metrics with the AUC plot  
    """  
    # Create training and test set  
    features_train, features_test, labels_train, labels_test = \\\n        train_test_split(features, labels_train, labels_test, random_state=15)  
    performance_metric_dict = {} # dict to hold the values of performance metrics  
    for k_value in k_value_list:  
        classifier = KNeighborsClassifier(n_neighbors=k_value) # algorithm initialization  
        labels_train = classifier.fit(features_train, labels_train) # model training  
        pred = classifier.predict(features_test) # model testing  
        accuracy = accuracy_score(labels_test, pred) # accuracy  
        precision = precision_score(labels_test, pred) # precision  
        recall = recall_score(labels_test, pred) # recall  
        auroc = roc_auc_score(labels_test, pred) # auroc  
        # dict filling step  
        performance_metric_dict[k_value] = [accuracy, precision, recall, auroc]  
        # Key: k_value, value: its different performance metrics in a list format  
    return performance_metric_dict
```

In [47]: k\_value\_list = [3, 5, 7, 9, 15, 20]

In [48]: knn\_mean\_performing\_knn\_list\_imputation(k\_value\_list, features\_mean)

Out [48]:

k_value	accuracy	precision	recall	auroc
3	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
5	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
7	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
9	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
15	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
20	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886

In [49]: knn\_median\_performing\_knn\_list\_imputation(k\_value\_list, features\_median)

In [50]: knn\_features\_most\_frequent\_performing\_knn\_list\_imputation(k\_value\_list, features\_most\_frequent)

In [51]:

```
# plotting different performance metrics in a line plot  
def plotting_imputation(k):  
    """function to plot different evaluation metrics for different imputed data  
    Args: k = value from [3, 5, 7, 9, 15, 20]  
    Return: line plot indicating all evaluation metrics"""  
    measure = ["accuracy", "precision", "recall", "auroc"]  
    plt.figure(figsize=(10, 5))  
    plt.plot(list(knn_mean.keys()), list([i] for i in knn_mean.values(i)), "--", color="blue", \\\n            linewidth=3, markersize=10, label="Mean Imputation")  
    plt.plot(list(knn_median.keys()), list([i] for i in knn_median.values(i)), "--", color="red", \\\n            linewidth=3, markersize=10, label="Median Imputation")  
    plt.plot(list(knn_features_most_frequent.keys()), list([i] for i in knn_features_most_frequent.values(i)), \\\n            color="green", \\\n            linewidth=3, markersize=10, label="Most Frequent Imputation")  
    plt.plot(list(standardized_knn.keys()), list([i] for i in standardized_knn.values(i)), "--", color="black", \\\n            linewidth=3, markersize=10, label="Standardized kNN")  
    plt.plot(list(normalized_knn.keys()), list([i] for i in normalized_knn.values(i)), "--", color="yellow", \\\n            linewidth=3, markersize=10, label="Normalized kNN")  
    plt.legend()  
    plt.xlabel("Different k values in the algorithm", fontsize=15)  
    plt.ylabel("Measures of ( ) across different k values", format(measure[k]), fontsize=15)  
    plt.title("Comparison of ( ) across different k values ( )", format(measure[k], list(knn_mean.keys(i))), \\\n            fontsize=15)
```



In [52]: plotting\_imputation(0)

Out [52]:

k_value	accuracy	precision	recall	auroc
3	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
5	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
7	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
9	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
15	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
20	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886

In [53]: plotting\_imputation(1)

Out [53]:

k_value	accuracy	precision	recall	auroc
3	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
5	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
7	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
9	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
15	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
20	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886

In [54]: plotting\_imputation(2)

Out [54]:

k_value	accuracy	precision	recall	auroc
3	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
5	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
7	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
9	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
15	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
20	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886

In [55]: plotting\_imputation(3)

Out [55]:

k_value	accuracy	precision	recall	auroc
3	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
5	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
7	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
9	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
15	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886
20	0.71875	0.584905603773585	0.4920634926349204	0.6607604282022886

On Observing all the evaluation metrics plots, the standardized kNN showed a better performance. Also, the imputation had no significant effect on the improvement of the performance metrics is the important message. The best k value is 5.

## Treating imbalanced data.

The number of diabetic vs non-diabetic patients in the dataset.

In [56]: dataset['Outcome'].value\_counts()

Out [56]:

Outcome	count
0	500
1	268

In [57]:

```
# Upsample minority class  
df_majority = dataset[dataset.Outcome==0]  
df_minority = dataset[dataset.Outcome==1]  
df_minority_upsampled = resample(df_minority, \\\n                                  replace=True, \\\n                                  n_samples=500, \\\n                                  random_state=123) # to match majority class  
# Combine majority class with upsampled minority class  
df_upsampled = pd.concat([df_majority, df_minority_upsampled])  
# Display new class counts  
df_upsampled.Outcome.value_counts()

Out [57]:



| Outcome | count |
|---------|-------|
| 0       | 500   |
| 1       | 500   |


```

In [58]:

```
# Create features and labels  
features_up = df_upsampled.drop(['Outcome'], axis=1) # independent variable  
labels_up = df_upsampled['Outcome'] # dependent variable  
# kNN model  
clf_1 = KNeighborsClassifier().fit(features_up, labels_up)  
pred_y_1 = clf_1.predict(features_up)  
print(np.unique(pred_y_1))  
[0 1]  
0.819
```

In [59]:

```
# Downsample majority class  
df_majority_downsampled = resample(df_majority, \\\n                                    replace=False, \\\n                                    n_samples=268, \\\n                                    random_state=123) # sample without replacement  
# Combine majority class with downsampled majority class  
df_downsampled = pd.concat([df_majority_downsampled, df_minority])  
# Display new class counts  
df_downsampled.Outcome.value_counts()

Out [59]:



| Outcome | count |
|---------|-------|
| 0       | 268   |
| 1       | 268   |


```

In [60]:

```
# Create features and labels  
features_down = df_downsampled.drop(['Outcome'], axis=1) # independent variable  
labels_down = df_downsampled['Outcome'] # dependent variable  
# kNN model  
clf_2 = KNeighborsClassifier().fit(features_down, labels_down)  
pred_y_2 = clf_2.predict(features_down)  
print(np.unique(pred_y_2))  
[0 1]  
0.793477611940298
```

The upsampling and downsampling had an accuracy of 0.82 and 0.79 respectively. Therefore, the upsampling suited this dataset apt.

## SVM

In [61]:

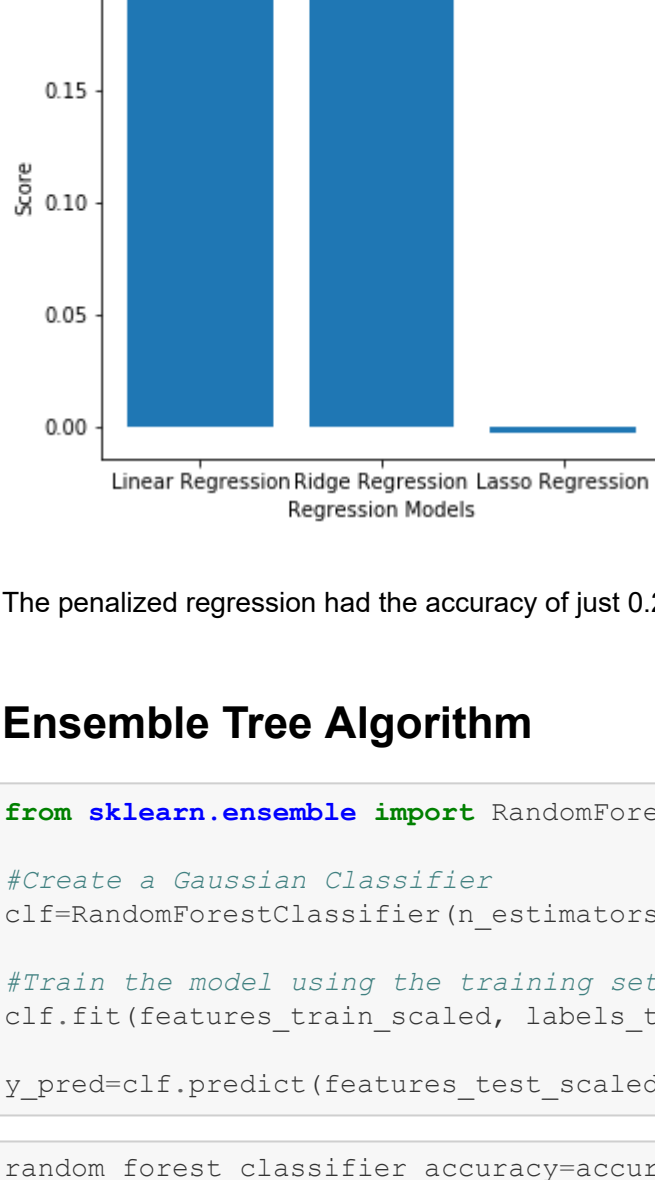
```
from sklearn.svm import SVC  
clf_3 = SVC(kernel='linear', class_weight='balanced', # penalize  
            probability=True)  
clf_3.fit(features, labels)  
pred_y_3 = clf_3.predict(features)  
print(np.unique(pred_y_3))  
[0 1]  
0.7482291666666666
```

In [62]:

```
prob_y_3 = clf_3.predict_proba(features)  
prob_y_3 = [p[1] for p in prob_y_3]  
print(roc_auc_score(labels,
```



```
[80]: plt.bar(model, scores)
plt.xlabel('Regression Models')
plt.ylabel('Score')
plt.show()
```



The penalized regression had the accuracy of just 0.20 which is extremely poor in performance.

## Ensemble Tree Algorithm

```
In [81]: from sklearn.ensemble import RandomForestClassifier

# Create a Gaussian Classifier
clf=RandomForestClassifier(n_estimators=2)

# Train the model using the training sets y_pred=clf.predict(X_test)
clf.fit(features_train_scaled, labels_train)

y_pred=clf.predict(features_test_scaled)

In [82]: random_forest_classifier_accuracy=accuracy_score(labels_test, y_pred)
random_forest_classifier_accuracy

Out[82]: 0.7291666666666666
```

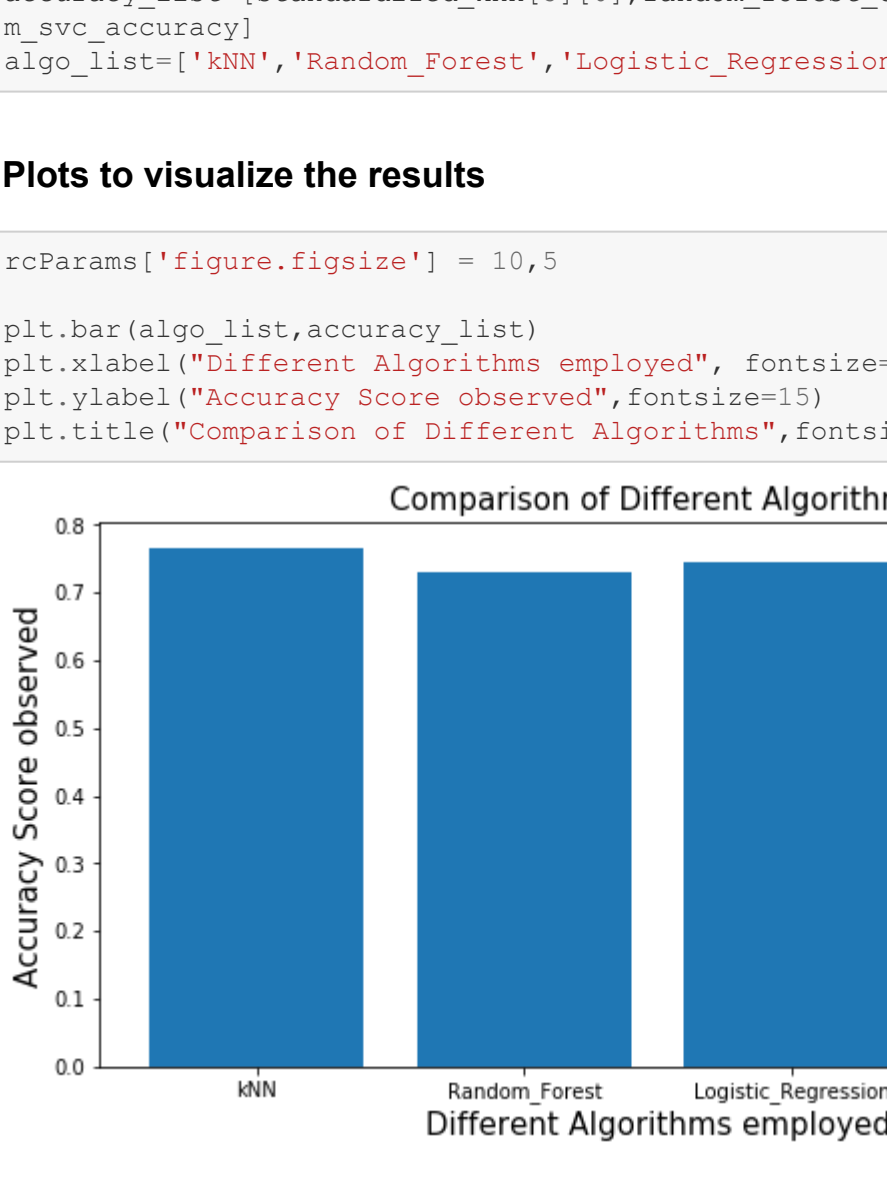
## feature importance by Random tree classifier

```
In [83]: feature_imp = pd.Series(clf.feature_importances_,index=dataset.columns[:-1]).sort_values(ascending=False)

In [84]: feature_imp

Out[84]: Glucose      0.243339
BMI      0.178864
Age      0.158452
DiabetesPedigreeFunction  0.141230
Pregnancies  0.097641
BloodPressure  0.090816
SkinThickness  0.044935
Insulin      0.044722
dtype: float64
```

```
In [85]: # Creating a bar plot
sns.barplot(x=feature_imp, y=feature_imp.index)
# Add labels to your graph
plt.xlabel('Feature Importance Score')
plt.ylabel('Features')
plt.title("Visualizing Important Features")
plt.show()
#https://www.datacamp.com/community/tutorials/random-forests-classifier-python
```



from the plot, the Glucose, BMI, Age were the most prominent features and Insulin had less significance.

## Discussion about Features Importance Ranking

Top 3 features obtained are:

1. The kNN classifier feature importance ranking suggest that Glucose, Insulin, Age
2. The Logistic Regression feature importance ranking suggest that Glucose, BMI, Diabetes Pedigree function
3. The Random tree classifier feature importance ranking suggest that Glucose, BMI, Age

All the 3 models predicted Glucose to be the most prominent feature. The scientific literature Risk Factors for Type 2 Diabetes Mellitus, Published in The Journal of cardiovascular nursing suggests that Family history of diabetes (pedigree), Age is a contributing factor as predicted by Logistic regression and random tree classifier respectively. The article also points that BMI as an important cause for diabetes and predicted by logistic regression.

## Summary - Finding Best Model

```
In [86]: standardized_knn[5][0] # KNN accuracy of k=5

Out[86]: 0.765625

In [87]: random_forest_classifier_accuracy # random forest accuracy

Out[87]: 0.7291666666666666

In [88]: logistic_regression_accuracy # logistic regression accuracy

Out[88]: 0.7447916666666666

In [89]: svm_svc_accuracy # svm best model accuracy

Out[89]: 0.7447916666666666

In [90]: accuracy_list=[standardized_knn[5][0],random_forest_classifier_accuracy,logistic_regression_accuracy,svm_svc_accuracy]
algo_list=['kNN','Random_Forest','Logistic_Regression','SVM']
```

## Plots to visualize the results

```
In [91]: rcParams['figure.figsize']=10,5

plt.bar(algo_list,accuracy_list)
plt.xlabel("Different Algorithms employed", fontsize=15)
plt.ylabel("Accuracy Score observed",fontsize=15)
plt.title("Comparison of Different Algorithms",fontsize=15);
```



## Conclusion:

The kNN, Random forest, Logistic Regression, SVM seemed to perform in a similar way. The Penalized Regression resulted in a worst performance metric.

For the diabetics prediction dataset, the kNN algorithm seemed to perform better in comparison to the other machine learning algorithms.