Untitled3

August 5, 2024

[1]:
```
 # Understand the dataset
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

[2]:
```
df=pd.read_csv("health care diabetes.csv")
df.head()
df.info()
df.isna().sum()
```

```
<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
```

[2]:
```
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

1 Data Exploration:

2 1. Perform descriptive analysis. Understand the variables and their corresponding values. On the columns below, a value of zero does not make sense and thus indicates missing value:

• Glucose • BloodPressure • SkinThickness • Insulin • BMI # 2. Visually explore these variables using histograms. Treat the missing values accordingly. # 3. There are integer and float data type variables in this dataset. Create a count (frequency) plot describing the data types and the count of variables.

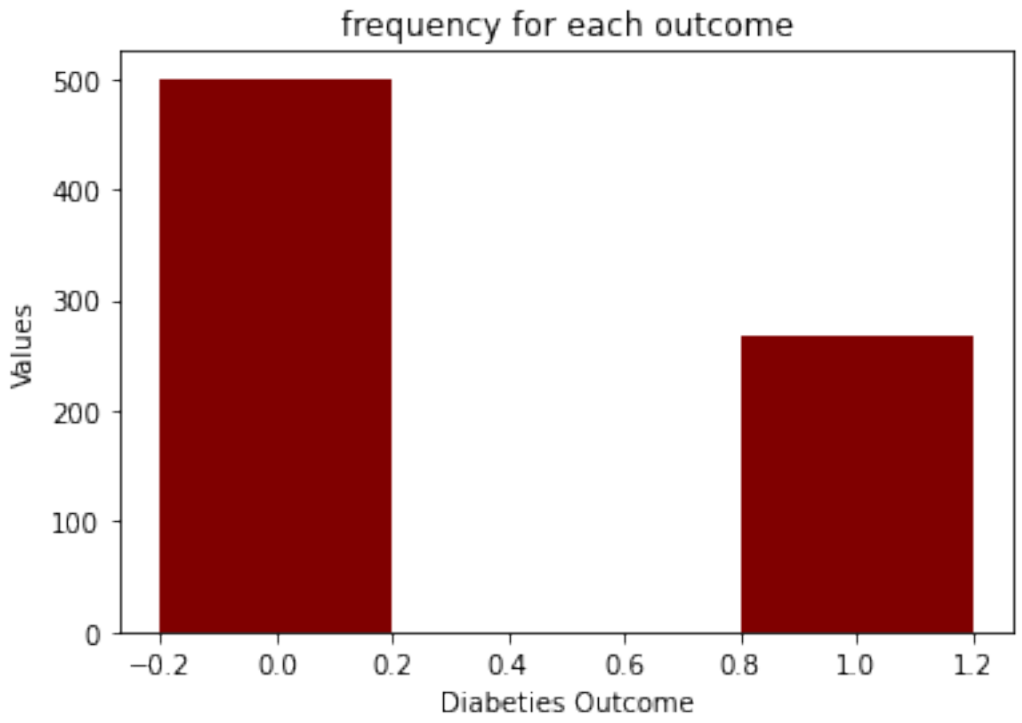
[3]:
```
# target variable is outcome
#lets understand about target variable
target=df['Outcome'].value_counts()
```

[4]:
```
target
```

[4]:
```
0    500
1    268
Name: Outcome, dtype: int64
```

[ ]:
```

```

[5]:
```
from collections import Counter
#plt.bar(target, color ='maroon', width = 0.4)
# Count the frequency of each unique value
counter = Counter(df['Outcome'])
categories = list(counter.keys())
values = list(counter.values())
plt.bar(categories,values,color='maroon', width=0.4)
plt.title("frequency for each outcome")
plt.xlabel("Diabeties Outcome")
plt.ylabel("Values")
plt.savefig('Fig1. frequency for each item.png', dpi=300, bbox_inches='tight')
plt.show()
```



[6]:
```
categories
```

[6]:
```
[1, 0]
```

[ ]:
```

```

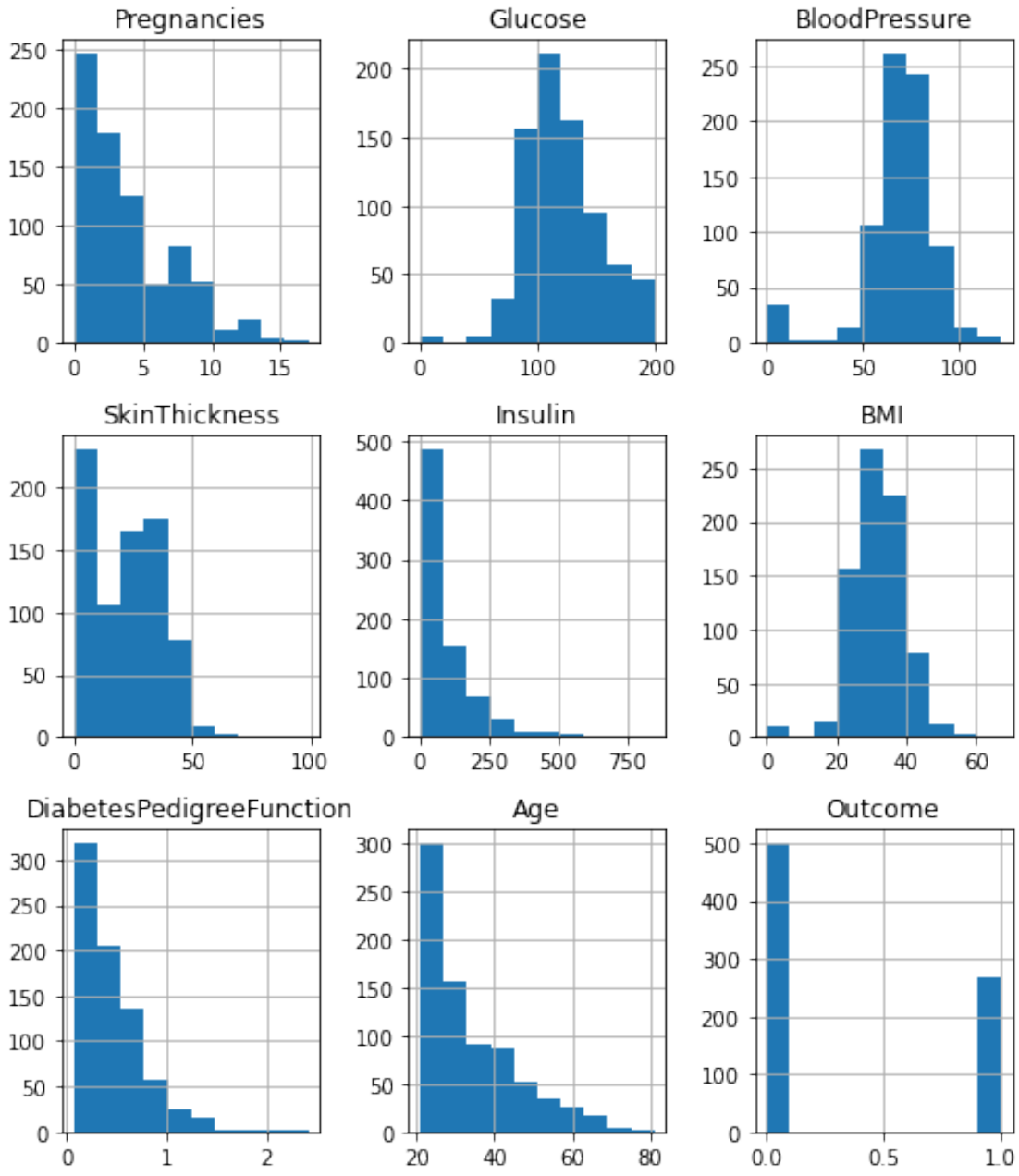
checking for duplicated record

[7]:
```
df.duplicated().sum()
```

[7]:
```
0
```

no duplicate record found

[8]:
```
# to see distributio of each variable in the dataset
df.hist(figsize=(7,8))
plt.tight_layout()
plt.savefig('Fig2. distribution of each variable.png', dpi=300,␣
 ↪bbox_inches='tight')
plt.show()
```



[9]:
```
# convert 0 values in the columns [• Glucose • BloodPressure • SkinThickness •␣
↪Insulin • BMI] to NAN
```

[10]:
```
zero_col=['Glucose', 'BloodPressure',  'SkinThickness',  'Insulin', 'BMI']
df[zero_col]=df[zero_col].replace(0,np.nan)
df.head()
```

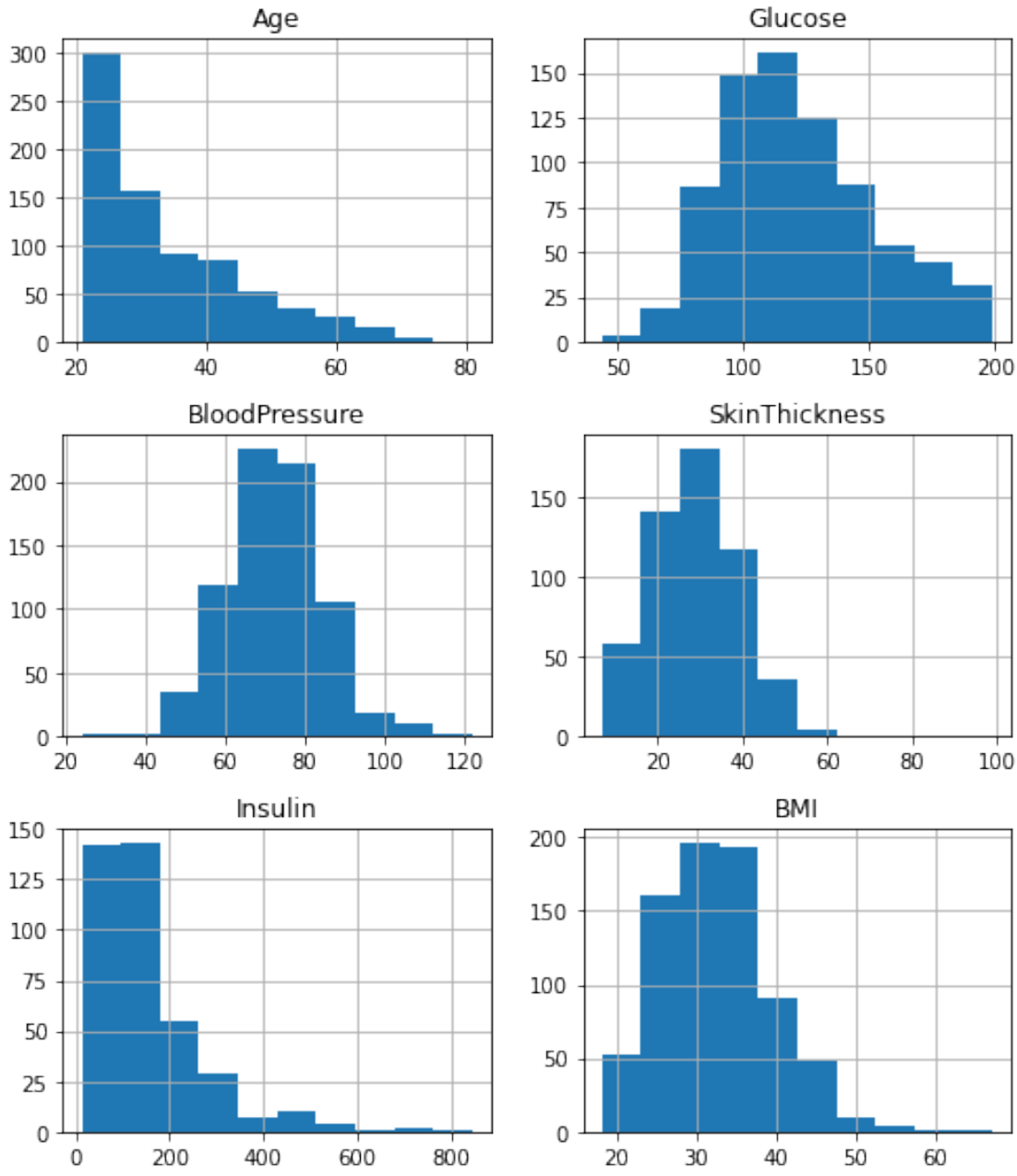
[10]:
```
   Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
0            6    148.0           72.0           35.0      NaN  33.6
1            1     85.0           66.0           29.0      NaN  26.6
2            8    183.0           64.0            NaN      NaN  23.3
3            1     89.0           66.0           23.0     94.0  28.1
4            0    137.0           40.0           35.0    168.0  43.1

   DiabetesPedigreeFunction  Age  Outcome
0                     0.627   50        1
1                     0.351   31        0
2                     0.672   32        1
3                     0.167   21        0
4                     2.288   33        1
```

[11]:
```
df.isna().sum()
```

[11]:
```
Pregnancies                 0
Glucose                     5
BloodPressure              35
SkinThickness             227
Insulin                   374
BMI                        11
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
dtype: int64
```

[12]:
```
#df.describe()
```

[13]:
```
# plot histogram to see the distribution selected columns Age, Glcose,␣
 ↪BloodPressure, SkinThickness, Insulin, BMI
df[['Age', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI']].
 ↪hist(figsize=(7,8))
plt.tight_layout()
plt.savefig('Fig3. histogram to distribution of Age, Glu, BP, ST, Insulin, BMI.
 ↪png', dpi=300, bbox_inches='tight')
plt.show()
```



[14]:
```
# it is seen that insulin data is highly left skewed and insulin values depends␣
 ↪on the age group. so nan value in the insulin column is filled based on age␣
 ↪group.
# creating new column for age groups
age_bins=[20,30,40, 50,60,float('inf')]
labels=['21-30','31-40','41-50', '51-60','above 60']
df['Age_Group']=pd.cut(df['Age'], bins=age_bins, labels=labels,␣
 ↪include_lowest=True)
```

[15]:
```
df.head()
```

[15]:
```
   Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
0            6    148.0           72.0           35.0      NaN  33.6
1            1     85.0           66.0           29.0      NaN  26.6
2            8    183.0           64.0            NaN      NaN  23.3
3            1     89.0           66.0           23.0     94.0  28.1
4            0    137.0           40.0           35.0    168.0  43.1

   DiabetesPedigreeFunction  Age  Outcome Age_Group
0                     0.627   50        1     41-50
1                     0.351   31        0     31-40
2                     0.672   32        1     31-40
3                     0.167   21        0     21-30
4                     2.288   33        1     31-40
```

[16]:
```
# compute median of insulin values based on age_group
insulin_median_age_group=df.groupby('Age_Group')['Insulin'].median()
```

[17]:
```
print(insulin_median_age_group)
```

```
Age_Group
21-30       105.0
31-40       140.0
41-50       131.0
51-60       207.0
above 60    180.0
Name: Insulin, dtype: float64
```

[18]:
```
insulin_median_age_group['21-30'] #example
```

[18]:
```
105.0
```

[19]:
```
#fill nan insulin values with the median value of the respective age group
def fill_insulin(row):
    if pd.isna(row['Insulin']):
        return insulin_median_age_group[row['Age_Group']]
    else:
              return row['Insulin']

df['Insulin']=df.apply(fill_insulin, axis=1)
```

[20]:
```
df.head()
```

[20]:
```
   Pregnancies  Glucose  BloodPressure  SkinThickness  Insulin   BMI  \
0            6    148.0           72.0           35.0    131.0  33.6
1            1     85.0           66.0           29.0    140.0  26.6
```

```
2            8    183.0           64.0            NaN    140.0  23.3
3            1     89.0           66.0           23.0     94.0  28.1
4            0    137.0           40.0           35.0    168.0  43.1

   DiabetesPedigreeFunction  Age  Outcome Age_Group
0                     0.627   50        1     41-50
1                     0.351   31        0     31-40
2                     0.672   32        1     31-40
3                     0.167   21        0     21-30
4                     2.288   33        1     31-40
```

[21]:
```
df['Insulin'].isna().sum()
```

[21]:
```
0
```

[22]:
```
# no null value in the Insulain column now
# since the other variables follows symmetrical distri so the nan value in␣
 ↪those variables column can be replaced with mean
nan_mean=['Glucose','BloodPressure', 'SkinThickness','BMI']
df_mean=df[nan_mean].mean()
df[nan_mean]=df[nan_mean].fillna(df_mean)
```

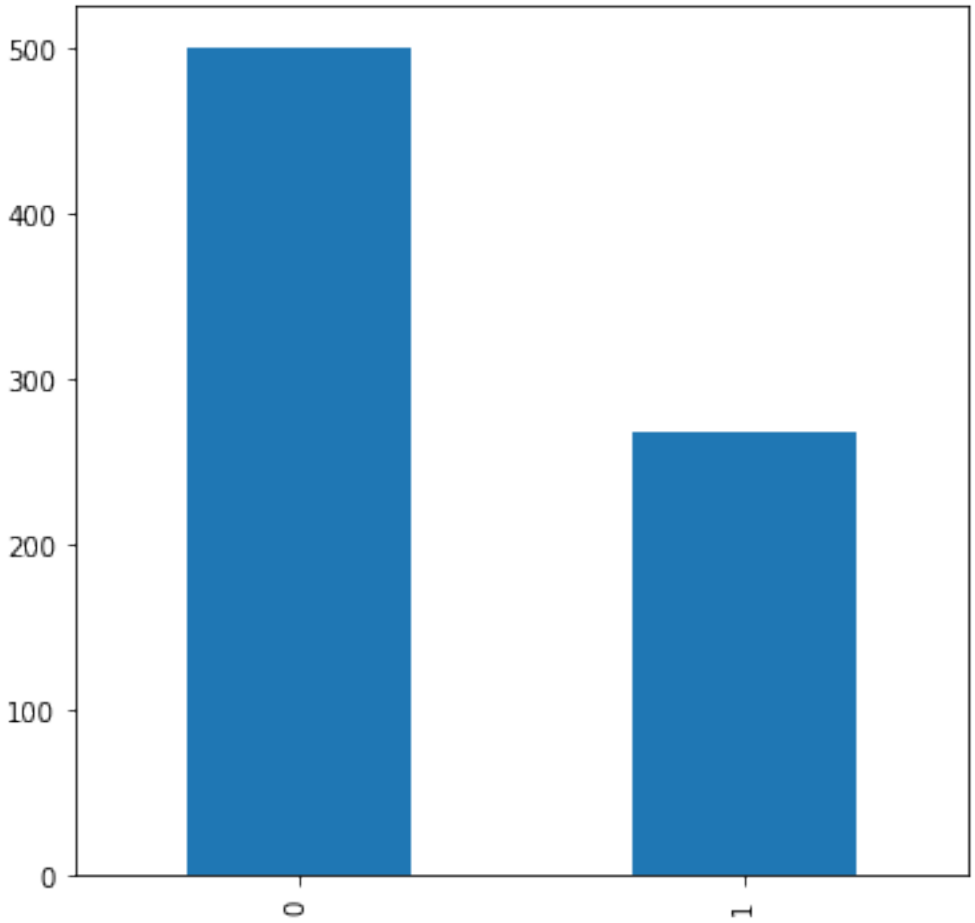
[23]:
```
df.isna().sum()
```

[23]:
```
Pregnancies                 0
Glucose                     0
BloodPressure               0
SkinThickness               0
Insulin                     0
BMI                         0
DiabetesPedigreeFunction    0
Age                         0
Outcome                     0
Age_Group                   0
dtype: int64
```

[24]:
```
# no nan value in any of the column reported now
# so the data is clean for further processing
```

[25]:
```
# Data Exploration:
# 4.        Check the balance of the data by plotting the count of outcomes by␣
 ↪their value. Describe your findings and plan future course of action.
# 5.        Create scatter charts between the pair of variables to understand␣
 ↪the relationships. Describe your findings.
# 6.        Perform correlation analysis. Visually explore it using a heat map.
```

[26]:
```
# 4. lets look at the target variable
plt.figure(figsize=(6,6))
df['Outcome'].value_counts().plot(kind='bar')
plt.savefig('Fig4. target variable.png', dpi=300, bbox_inches='tight')
```



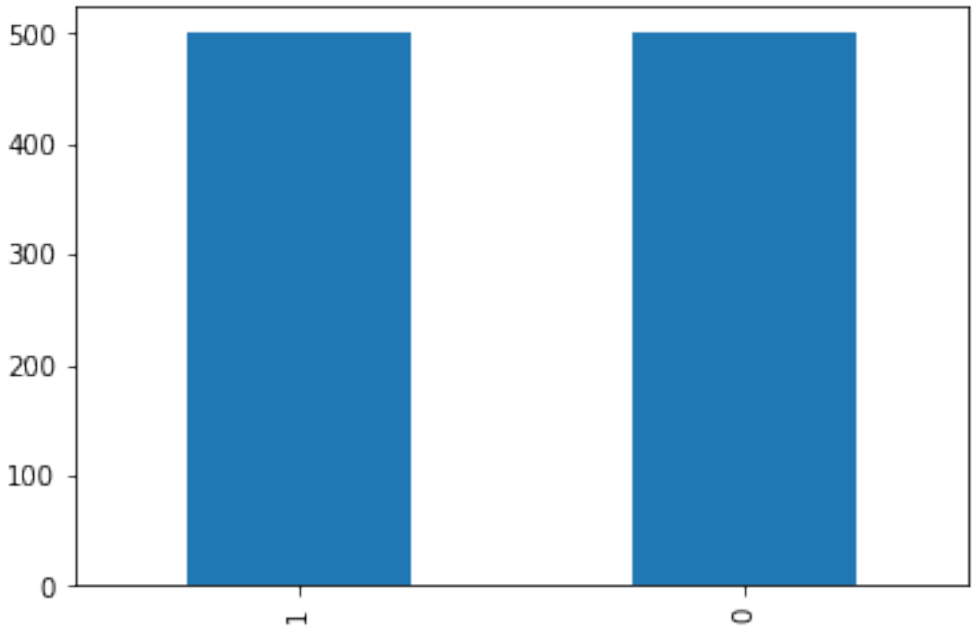
[27]:
```
# Above bar graph reveals that outcome "0" which means no diabeties are more␣
 ↪compared to number of people with diabeties.
# It seems that the Outcome data is more uneven and biased we need to use SMOTE␣
 ↪technique for resampling data and make it balance.
# Import the SMOTE class from the imblearn.over_sampling module

from imblearn.over_sampling import SMOTE
```

[28]:
```
# Extract the feature columns and target column by dropping the 'Outcome' and␣
 ↪'Age Group' columns from the diabetes_data DataFrame

data_X = df.drop(['Outcome','Age_Group'], axis=1)
data_y = df['Outcome']
```

```
# Apply SMOTE oversampling technique to balance the classes by creating␣
 ↪synthetic samples

X_resampled, y_resampled = SMOTE(random_state=100).fit_resample(data_X, data_y)
print(X_resampled.shape, y_resampled.shape)

# Plot a bar chart to visualize the class distribution after oversampling
y_resampled.value_counts().plot(kind='bar')
```

```
(1000, 8) (1000,)
```

[28]: <AxesSubplot: >



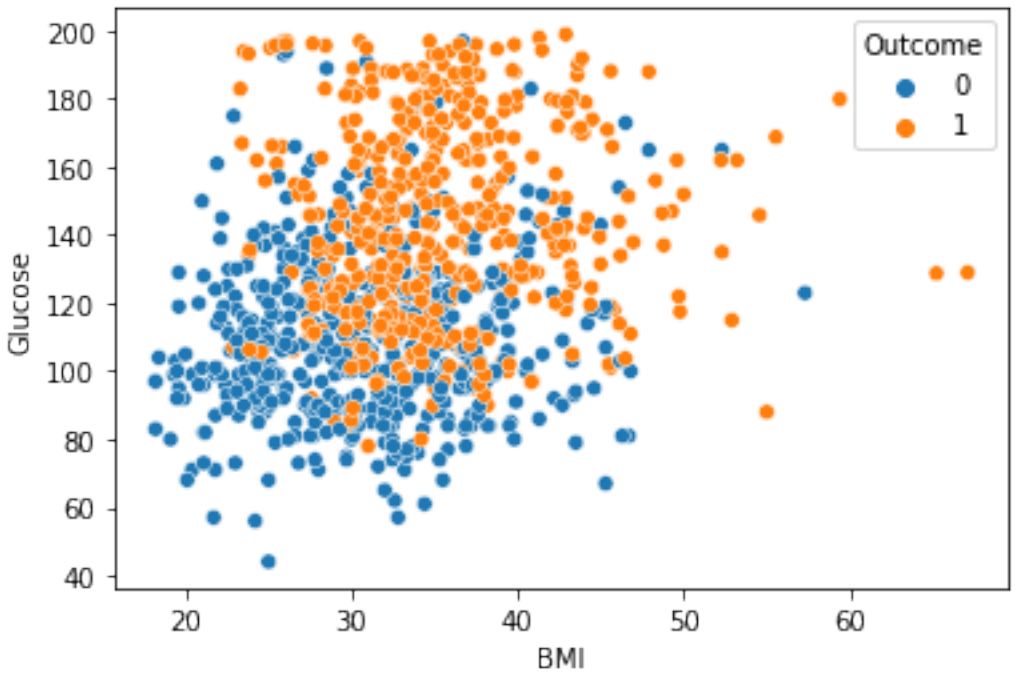
[29]:
```
#above bar chart shows the data is now unbiased
```

[30]:
```
# Concatenate X_resampled and y_resampled along axis 1 to create the resampled␣
 ↪data

data_resampled = pd.concat([X_resampled, y_resampled],axis=1)
data_resampled.shape
```

[30]: (1000, 9)

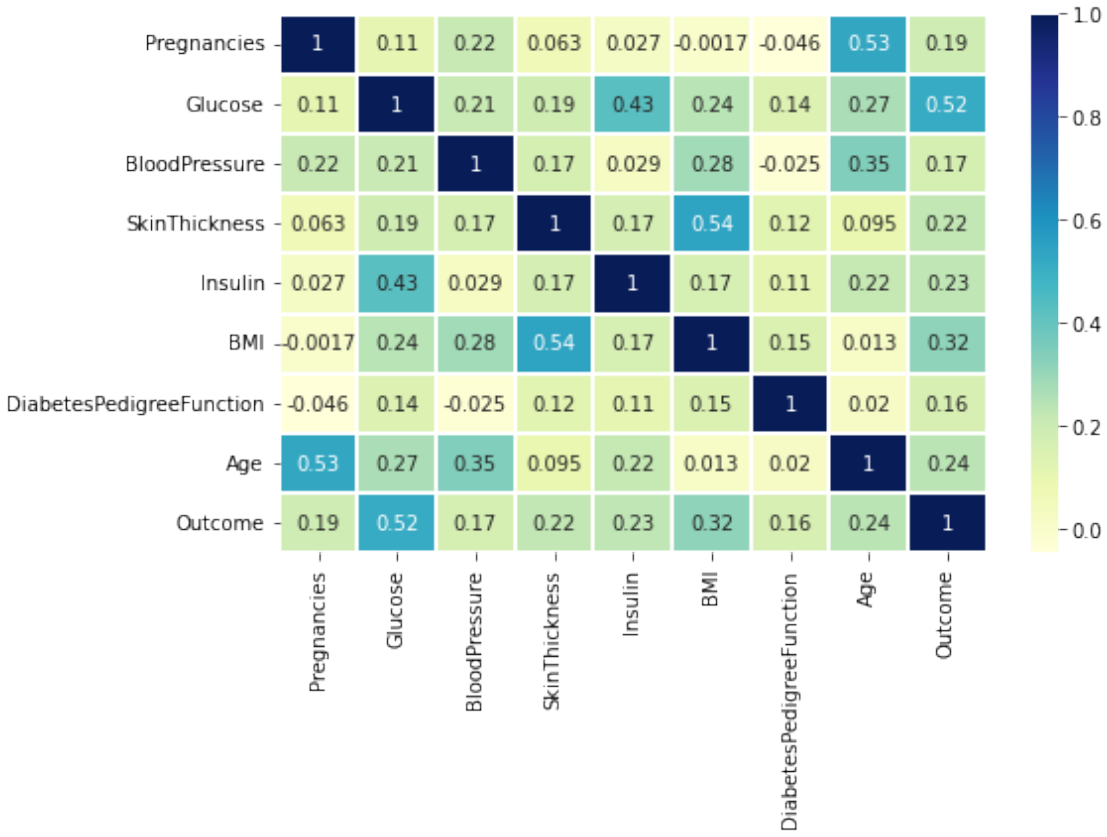
[31]:
```
# Create a scatter plot of 'BMI' vs 'Glucose' using the resampled data, with␣
 ↪'Outcome' as the hue
```

```
sns.scatterplot(x="BMI", y="Glucose", data=data_resampled, hue="Outcome");
plt.savefig('Fig5. BMI and Glucose.png', dpi=300, bbox_inches='tight')
```



Individual with high glucose level tend to have high BMI as per the graph shown The orange points (Outcome = 1) are more dispersed and seem to occupy higher ranges of BMI and Glucose values, suggesting that individuals with the condition are more likely to have higher BMI and Glucose levels. This scatter plot indicates a relationship between BMI and Glucose levels, with a tendency for individuals with a higher Glucose level and higher BMI to be classified as 1.

[32]:
```
# Create a heatmap of the correlation matrix of the resampled data

sns.heatmap(data_resampled.corr(),annot=True, cmap='YlGnBu', linewidths=0.1)
fig=plt.gcf()
fig.set_size_inches(8,5)
plt.savefig('Fig6. heat map.png', dpi=300, bbox_inches='tight')
plt.show()
```



Outcome:

Glucose (0.52): There is a moderate positive correlation between Glucose and Outcome, indicating that higher Glucose levels are associated with the positive outcome (likely presence of a condition).

BMI (0.32): There is a moderate positive correlation between BMI and Outcome, suggesting that higher BMI is also associated with the positive outcome.

Age (0.24): There is a weak positive correlation between Age and Outcome.

Insulin (0.23): A weak positive correlation with Outcome.

BloodPressure (0.17): A weak positive correlation with Outcome.

SkinThickness (0.16): A very weak positive correlation with Outcome.

Pregnancies (0.19): A weak positive correlation with Outcome.

DiabetesPedigreeFunction (0.16): A very weak positive correlation with Outcome.

- The heatmap reveals that Glucose, BMI, and Age are moderately correlated with the Outcome, suggesting that these features are important for predicting the outcome. Other features like BloodPressure, SkinThickness, Insulin, Pregnancies, and DiabetesPedigreeFunction show weaker correlations with the Outcome. Understanding these correlations can help in feature selection and improving predictive models.

[33]:
```
data_resampled.columns
```

[33]:
```
Index(['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin',
       'BMI', 'DiabetesPedigreeFunction', 'Age', 'Outcome'],
      dtype='object')
```

3 Next Step:- Data Modeling:

4 1. Devise strategies for model building. It is important to decide the right validation framework. Express your thought process.

5 2. Apply an appropriate classification algorithm to build a model.

6 3. Compare various models with the results from KNN algorithm.

7 4. Create a classification report by analyzing sensitivity, specificity, AUC (ROC curve), etc.

8 Please be descriptive to explain what values of these parameter you have used.

Convert categorical variables to numeric using encoding techniques like one-hot encoding.

feature scaling

Validation includes:

Train-Test Split: Initially split the data into training and testing sets (e.g., 80-20 split) to evaluate model performance on unseen data.

Cross-Validation: Use k-fold cross-validation (e.g., 5-fold) to ensure that the model's performance is consistent across different subsets of the data. This helps in mitigating the risk of overfitting.

Stratification: Ensure that the train-test split and cross-validation are stratified so that the class distribution in each fold is similar to the overall distribution.

[34]:
```
data_resampled.head()
```

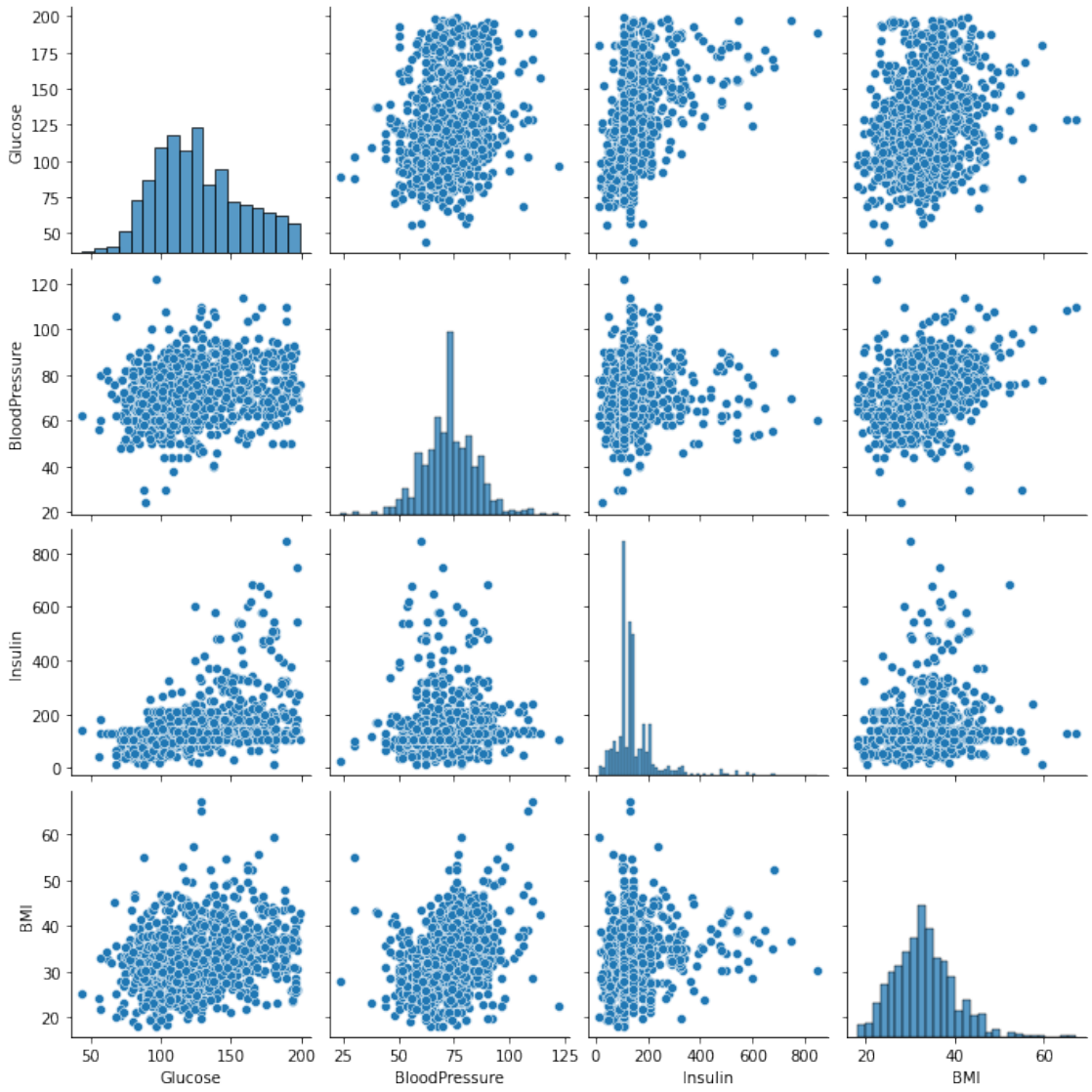
[34]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	\
0	6	148.0	72.0	35.00000	131.0	33.6	
1	1	85.0	66.0	29.00000	140.0	26.6	
2	8	183.0	64.0	29.15342	140.0	23.3	
3	1	89.0	66.0	23.00000	94.0	28.1	
4	0	137.0	40.0	35.00000	168.0	43.1	

	DiabetesPedigreeFunction	Age	Outcome
0	0.627	50	1
1	0.351	31	0
2	0.672	32	1
3	0.167	21	0
4	2.288	33	1

[35]:
```
# since 'data' variale is already defined for only numerical continous features␣
 ↪in above code. No encoding require
select_data=data_resampled.loc[:,['Glucose','BloodPressure','Insulin','BMI']]
sns.pairplot(select_data)

#A baseline model to predict the risk of diabetes using a various machine␣
 ↪learning models
#Feature scaling: Standardize or normalize the features to ensure they have a␣
 ↪similar scale, which is particularly important for algorithms like KNN and␣
 ↪logistic regression.
```

[35]: <seaborn.axisgrid.PairGrid at 0x7f5c2b13ffa0>



[36]:
```
# Import the StandardScaler from sklearn.preprocessing

from sklearn.preprocessing import StandardScaler
sc = StandardScaler()

# Get the column names of the resampled data (excluding the target column)

columns = data_resampled.columns[:-1]
scaled_data = sc.fit_transform(data_resampled[columns])
diabetes_data_sc = pd.DataFrame(scaled_data, columns= columns)
diabetes_data_sc.head()
```

[36]:
```
   Pregnancies   Glucose  BloodPressure  SkinThickness   Insulin       BMI  \
0     0.636169  0.674491      -0.076432       0.648324 -0.210513  0.081380
1    -0.897507 -1.336990      -0.589807      -0.076048 -0.115817 -0.959876
2     1.249639  1.791981      -0.760932      -0.057525 -0.115817 -1.450754
3    -0.897507 -1.209277      -0.589807      -0.800419 -0.599819 -0.736750
4    -1.204242  0.323280      -2.814432       0.648324  0.178792  1.494513

   DiabetesPedigreeFunction       Age
0                  0.448508  1.402507
1                 -0.403452 -0.253654
2                  0.587414 -0.166488
3                 -0.971425 -1.125318
4                  5.575700 -0.079321
```

[37]:
```
# Create empty lists to store models and evaluation metrics

models = []
model_accuracy = []
model_f1_score = []
model_auc_score = []
```

[38]:
```
# 1) Logistic Regression

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix

# Assigning the feature data to X
X = diabetes_data_sc

# Assigning the target variable to y
y = data_resampled['Outcome']

# Splitting the data into training and testing sets (80 and 20 ratio)
# Splitting the data into training and testing sets using train_test_split␣
 ↪function

X_train, X_test, y_train, y_test = train_test_split (X, y, test_size = 0.2,␣
 ↪random_state = 100)
```

[39]:
```
# Logistic regression
model_lr = LogisticRegression(random_state=100)  # Create a logistic regression␣
 ↪model
model_lr.fit(X_train, y_train)  # Fit the model to the training data
y_pred = model_lr.predict(X_test)  # Predict the target variable for the test␣
 ↪data
```

```
accuracy_lr = accuracy_score(y_test, y_pred)  # Calculate the accuracy of the␣
 ↪model
print('Accuracy of Logistic Regression= %.3f' % accuracy_lr)  # Print the␣
 ↪accuracy of the model
```

```
Accuracy of Logistic Regression= 0.735
```

lets try to hypertune parameter using gridsaerchCV and RandomizedSearchCV to update model and check accuracy

[40]:
```
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV,␣
 ↪cross_val_score
parameters = {'C': np.logspace(0, 5, 50)}  # Define the parameter grid for grid␣
 ↪search

gs_lr = GridSearchCV(model_lr, param_grid=parameters, cv=5, verbose=0)  #␣
 ↪Perform grid search with cross-validation
gs_lr.fit(X_train, y_train)  # Fit the grid search model to the training data

lr_best_param = gs_lr.best_params_  # Get the best parameters found by grid␣
 ↪search
lr_best_param
```

[40]:
```
{'C': 2.023589647725157}
```

[41]:
```
# Logistic regression
model_lr_1 = LogisticRegression(C=2.02, random_state=100)  # Create a logistic␣
 ↪regression model with best parameters
model_lr_1.fit(X_train, y_train)  # Fit the model to the training data with␣
 ↪best parameters
y_pred_lr = model_lr_1.predict(X_test)  # Predict the target variable for the␣
 ↪test data using the updated model
accuracy_lr = accuracy_score(y_test, y_pred_lr)  # Calculate the accuracy of␣
 ↪the updated model
print('Accuracy of Logistic Regression= %.3f' % accuracy_lr)  # Print the␣
 ↪accuracy of the updated model
```

```
Accuracy of Logistic Regression= 0.730
```

[ ]:
```
# Define the parameter grid for RandomizedSearchCV
param_dist = {
    'C': np.logspace(0, 5, 50),  # C ranges from 10^0 to 10^5
    'penalty': ['l1', 'l2'],     # Penalty type (L1 or L2)
    'solver': ['liblinear', 'saga']  # Solver type
}

# Perform Randomized Search CV
```

```
random_search = RandomizedSearchCV(model_lr, param_distributions=param_dist,␣
 ↪n_iter=100, cv=5, scoring='accuracy', random_state=42)
random_search.fit(X_train, y_train)

# Get the best parameters and the best score
best_params_random = random_search.best_params_


# Fit the model with the best parameters
best_model_random = random_search.best_estimator_
best_model_random.fit(X_train, y_train)

# Predict using the best model
y_pred_random = best_model_random.predict(X_test)

# Calculate accuracy
accuracy_lr_random = accuracy_score(y_test, y_pred_random)
print(f'Accuracy on test set with updated Logistic Regression from␣
 ↪RandomizedSearchCV: {accuracy_lr_random:.3f}')
```

it can be seen that here accuaracy reduces in logistic regression by hypertuning parametres

[ ]:
```
from sklearn.metrics import roc_auc_score, roc_curve

probs = model_lr.predict_proba(X_test)  # Get predicted probabilities for the␣
 ↪test data
probs = probs[:, 1]  # Extract probabilities of the positive class
auc_lr = roc_auc_score(y_test, probs)  # Calculate the AUC-ROC score
print('AUC:', auc_lr)  # Print the AUC-ROC score

fpr, tpr, thresholds = roc_curve(y_test, probs)  # Calculate ROC curve metrics
plt.plot(fpr, tpr, marker='.')  # Plot ROC curve
plt.plot([0, 1], [0, 1], linestyle='--')  # Plot diagonal line
plt.xlabel('False Positive Rate')  # Set x-axis label
plt.ylabel('True Positive Rate')  # Set y-axis label
plt.title('ROC curve - Logistic Regression')  # Set title

#Append model name, model accuracy and AUC.

models.append('LR')

model_accuracy.append(accuracy_lr)
model_auc_score.append(auc_lr)
```

[ ]:
```
#2) Decision Tree:

from sklearn.tree import DecisionTreeClassifier
```

```
model_dt = DecisionTreeClassifier(random_state=100)

# Define the parameters for grid search
parameters = {
    'max_depth': [1, 2, 3, 4, 5, 6, None]
}
# Create a GridSearchCV object with DecisionTreeClassifier and parameters
gs_dt = GridSearchCV(model_dt, param_grid=parameters, cv=5, verbose=0)

gs_dt.fit(X_train, y_train)  # Fit the GridSearchCV object to the training data

gs_dt.best_params_  # Get the best parameters found by grid search
print(gs_dt.best_params_)

# Get the best score found by grid search
gs_dt.best_score_
print(gs_dt.best_score_)

model_dt = DecisionTreeClassifier(max_depth = 3)
model_dt.fit(X_train, y_train)
accuracy_dt = model_dt.score(X_test, y_test)
print('Accuracy of Decision Tree= %.3f' %accuracy_dt)
```

[ ]:
```
model_dt.feature_importances_

plt.figure(figsize=(8,3))  # Create a figure with a specific size
columns = X_train.columns  # Get the column names of X_train
sns.barplot(y=columns, x=model_dt.feature_importances_)  # Create a bar plot of␣
 ↪feature importance
plt.title("Feature Importance in Model")  # Set the title of the plot

probs = model_dt.predict_proba(X_test)  # Get the predicted probabilities from␣
 ↪the model
probs = probs[:,1]  # Extract the probabilities for the positive class
auc_dt = roc_auc_score(y_test, probs)  # Calculate the AUC score
print('AUC:', auc_dt)  # Print the AUC score

fpr, tpr, thresholds = roc_curve(y_test, probs)  # Calculate the ROC curve
plt.plot(fpr, tpr, marker='.')  # Plot the ROC curve
plt.plot([0,1], [0,1], linestyle='--')  # Plot the diagonal line
plt.xlabel('False Positive Rate')  # Set the x-axis label
plt.ylabel('True Positive Rate')  # Set the y-axis label
plt.title('ROC curve - Decision Tree')  # Set the title of the plot

models.append('DT')  # Add the model name to the list of models
model_accuracy.append(accuracy_dt)  # Add the model accuracy to the list of␣
 ↪accuracies
```

```
model_auc_score.append(auc_dt)  # Add the AUC score to the list of AUC scores
```

[ ]:
```
#3) RandomForest Classifier:

from sklearn.ensemble import RandomForestClassifier
rf=RandomForestClassifier(random_state=100)  # Create a Random Forest classifier

rf.fit(X_train, y_train)  # Fit the model to the training data
y_pred = rf.predict(X_test)  # Predict the target variable for the test data
accuracy_rf = accuracy_score(y_test, y_pred)  # Calculate the accuracy of the␣
 ↪model
print('Accuracy of random forest= %.3f' % accuracy_rf)  # Print the accuracy of␣
 ↪the model

#hyprttuning

parameters = {
    'n_estimators' : [10,50,100,150],  # Define the number of trees in the␣
 ↪forest
    'max_depth' : [None,1,3,5,7,9],  # Define the maximum depth of the tree
    'min_samples_leaf' : [1,3,5,7,9],  # Define the minimum number of samples␣
 ↪required to be at a leaf node
    'min_samples_split': [1,2,3,4,5],  # Define the minimum number of samples␣
 ↪required to split an internal node
    'bootstrap': [True, False]
}
```

[ ]:
```
gs_rf = GridSearchCV(estimator=rf,param_grid=parameters,cv=5,verbose=0)  #␣
 ↪Perform grid search to find the best hyperparameters
gs_rf.fit(X_train, y_train)  # Fit the model with training data

print(gs_rf.best_score_)  # Print the best score achieved during grid search

gs_rf.best_params_  # Print the best hyperparameters found during grid search
print(gs_rf.best_params_)

model_rf =␣
 ↪RandomForestClassifier(n_estimators=100,max_depth=None,min_samples_leaf=1,min_samples_split
 ↪ # Create a Random Forest classifier with specific hyperparameters
model_rf.fit(X_train, y_train)  # Fit the model with training data
accuracy_rf = model_rf.score(X_test, y_test)  # Calculate the accuracy of the␣
 ↪model on test data
print('Accuracy of Random Forest= %.3f' %accuracy_rf)  # Print the accuracy
```

[ ]:
```
# Create and train the initial Random Forest classifier
rf = RandomForestClassifier(random_state=100)
```

```
rf.fit(X_train, y_train)

# Predict the target variable for the test data
y_pred = rf.predict(X_test)

# Calculate and print the accuracy of the initial model
accuracy_rf = accuracy_score(y_test, y_pred)
print('Accuracy of initial random forest= %.3f' % accuracy_rf)

# Define the parameter grid for hyperparameter tuning
parameters = {
    'n_estimators': [10, 50, 100, 150],
    'max_depth': [None, 1, 3, 5, 7, 9],
    'min_samples_leaf': [1, 3, 5, 7, 9],
    'min_samples_split': [2, 3, 4, 5],
    'bootstrap': [True, False]
}

# Perform grid search to find the best hyperparameters
gs_rf = GridSearchCV(estimator=rf, param_grid=parameters, cv=5, verbose=0)
gs_rf.fit(X_train, y_train)

# Print the best score and best parameters found during grid search
print('Best score during grid search: %.3f' % gs_rf.best_score_)
print('Best hyperparameters:', gs_rf.best_params_)

# Create a new Random Forest classifier with the best hyperparameters
model_rf = RandomForestClassifier(
    n_estimators=gs_rf.best_params_['n_estimators'],
    max_depth=gs_rf.best_params_['max_depth'],
    min_samples_leaf=gs_rf.best_params_['min_samples_leaf'],
    min_samples_split=gs_rf.best_params_['min_samples_split'],
    bootstrap=gs_rf.best_params_['bootstrap'],
    random_state=100
)

# Fit the model with training data
model_rf.fit(X_train, y_train)

# Calculate and print the accuracy of the tuned model on test data
accuracy_rf_tuned = model_rf.score(X_test, y_test)
print('Accuracy of tuned Random Forest= %.3f' % accuracy_rf_tuned)
```

[ ]:
```
plt.figure(figsize=(8,3))
sns.barplot(y=columns, x=model_rf.feature_importances_)  # Plot the feature␣
 ↪importance in the model
plt.title("Feature Importance in Model")
```

```
probs = model_rf.predict_proba(X_test)  # Get predicted probabilities from the␣
 ↪model
probs = probs [:,1]  # Extract the probabilities for the positive class
auc_rf = roc_auc_score(y_test, probs)  # Calculate the AUC score
print('AUC:', auc_rf)  # Print the AUC score

fpr, tpr, thresholds = roc_curve(y_test, probs)  # Calculate the ROC curve
plt.plot(fpr,tpr,marker='.')
plt.plot([0,1],[0,1],linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC curve - Random Forest');

models.append('RF')  # Add the model name to a list of models
model_accuracy.append(accuracy_rf)  # Add the model accuracy to a list
model_auc_score.append(auc_rf)  # Add the model AUC score to a list
```

[ ]:
```
#4) K-Nearest Neighbour (KNN):

from sklearn.neighbors import KNeighborsClassifier
model_knn = KNeighborsClassifier()  # Create KNN classifier

knn_neighbors = [i for i in range(2,20)]  # List of neighbors to test
parameters = {
    'n_neighbors': knn_neighbors
}

gs_knn = GridSearchCV(estimator=model_knn,param_grid=parameters,cv=5,verbose=0)␣
 ↪ # Perform grid search for best parameters
gs_knn.fit(X_train, y_train)  # Fit the model with training data

gs_knn.best_params_  # Print the best parameters found by grid search

gs_knn.best_score_  # Print the best score achieved by the model

model_knn = KNeighborsClassifier(n_neighbors=3, p=2)  # Create KNN model with␣
 ↪specified parameters
model_knn.fit(X_train,y_train)  # Fit the model with training data
model_knn.score(X_train,y_train)  # Calculate the accuracy score on training␣
 ↪data

accuracy_knn = model_knn.score(X_test, y_test)  # Calculate the accuracy score␣
 ↪on test data
print('Accuracy of KNN= %.3f' %accuracy_knn)
```

[ ]:
```
pred_y_knn = model_knn.predict(X_test)  # Make predictions on test data
accuracy_score(y_test,pred_y_knn)  # Calculate accuracy score using predicted␣
 ↪and true labels

probs = model_knn.predict_proba(X_test)  # Get class probabilities for test data
probs = probs [:,1]  # Extract probabilities for positive class
auc_knn = roc_auc_score(y_test, probs)  # Calculate AUC score
print('AUC:', auc_knn)

fpr, tpr, thresholds = roc_curve(y_test, probs)  # Calculate ROC curve
plt.plot(fpr,tpr,marker='.')  # Plot ROC curve
plt.plot([0,1],[0,1],linestyle='--')  # Plot diagonal line
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC curve - KNN');

models.append('KNN')  # Add model name to list
model_accuracy.append(accuracy_knn)  # Add model accuracy to list
model_auc_score.append(auc_knn)  # Add model AUC score to list

gs_knn.cv_results_['mean_test_score']  # Print mean test scores for different␣
 ↪parameter values

plt.figure(figsize=(6,4))
sns.barplot(x=knn_neighbors, y=gs_knn.cv_results_['mean_test_score'])  # Plot␣
 ↪bar chart of test accuracy vs. number of neighbors
plt.xlabel("N_Neighbors")
plt.ylabel("Test Accuracy")
plt.title("Test Accuracy vs. N_Neighbors");
```

[ ]:
```
# 5) Support Vector Machine (SVM):

from sklearn.svm import SVC
model_svm = SVC(kernel='rbf', random_state=100, verbose=0)  # Create an SVM␣
 ↪model with RBF kernel

parameters = {
    'C': [1, 5, 10, 15, 20, 25]  # Define a grid of C values for hyperparameter␣
 ↪tuning
}

gs_svm = GridSearchCV(estimator=model_svm, param_grid=parameters, cv=5,␣
 ↪verbose=5)  # Perform grid search with cross-validation
gs_svm.fit(X, y)  # Fit the model to the training data

gs_svm.best_score_
```

```
gs_svm.best_params_

gs_svm.best_estimator_

model_svm_1 = SVC(probability=True, C=5, kernel='rbf', random_state=100,␣
 ↪verbose=0)  # Create a new SVM model with optimized hyperparameters

model_svm_1.fit(X_train,y_train)

model_svm_1.score(X_train,y_train)

accuracy_svm = model_svm_1.score(X_test, y_test)  # Calculate the accuracy of␣
 ↪the SVM model
print('Accuracy of SVM = %.3f' % accuracy_svm)

probs = model_svm_1.predict_proba(X_test)  # Get the predicted probabilities␣
 ↪from the SVM model
probs = probs[:, 1]  # Select the probabilities for the positive class
auc_svm = roc_auc_score(y_test, probs)  # Calculate the AUC score
print('AUC: %.3f' % auc_svm)

fpr, tpr, thresholds = roc_curve(y_test, probs)  # Calculate the ROC curve␣
 ↪values
plt.plot(fpr, tpr, marker='.')  # Plot the ROC curve
plt.plot([0, 1], [0, 1], linestyle='--')  # Plot the diagonal line
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC curve - SVM');

models.append('SVM')
model_accuracy.append(accuracy_svm)
model_auc_score.append(auc_svm)
print(accuracy_svm, '%.3f' % auc_svm)  # Print the accuracy and AUC score
```

[ ]:
```
# 6) Naive Bayes Algorithm:

from sklearn.naive_bayes import GaussianNB
model_gnb = GaussianNB()  # Create Gaussian Naive Bayes model

model_gnb.fit(X_train, y_train)  # Train the model

accuracy_gnb = model_gnb.score(X_test, y_test)  # Calculate accuracy score
accuracy_gnb

probs = model_gnb.predict_proba(X_test)  # Get predicted probabilities
probs = probs[:, 1]  # Select probabilities for positive class
auc_gnb = roc_auc_score(y_test, probs)  # Calculate AUC score
```

```
print('AUC: %.3f' % auc_gnb)

fpr, tpr, thresholds = roc_curve(y_test, probs)  # Calculate ROC curve
plt.plot(fpr, tpr, marker='.')  # Plot ROC curve
plt.plot([0, 1], [0, 1], linestyle='--')  # Add diagonal reference line
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC curve - GNB');

models.append('GNB')  # Add model name to list
model_accuracy.append(accuracy_gnb)  # Add accuracy score to list
model_auc_score.append(auc_gnb)  # Add AUC score to list
print(accuracy_gnb, '%.3f' % auc_gnb)  # Print accuracy score and AUC score
```

[ ]:
```
#7) Ensembler Learning --> Adaptive Boosting

from sklearn.ensemble import AdaBoostClassifier
model_ada = AdaBoostClassifier(random_state=100)  # Initialize AdaBoost␣
 ↪classifier

parameters = {
    'n_estimators': [10,100,500,1000]  # Set parameter grid for grid search
}
gs_ada = GridSearchCV(model_ada,param_grid=parameters,cv=5,verbose=0)  #␣
 ↪Perform grid search
gs_ada.fit(X,y)  # Fit grid search to data

gs_ada.best_params_  # Print the best parameters found by grid search

gs_ada.best_score_  # Print the best score found by grid search

model_ada = AdaBoostClassifier(n_estimators=100,random_state=100)  # Initialize␣
 ↪AdaBoost classifier with best parameters
model_ada.fit(X_train,y_train)  # Fit the model to the training data
accuracy_ada = model_ada.score(X_test,y_test)  # Calculate accuracy on the test␣
 ↪data
accuracy_ada  # Print the accuracy

probs = model_ada.predict_proba(X_test)  # Get predicted probabilities
probs = probs [:,1]  # Extract probabilities for positive class
auc_ada = roc_auc_score(y_test, probs)  # Calculate AUC score
print('AUC: %.3f' %auc_ada)  # Print the AUC score

fpr, tpr, thresholds = roc_curve(y_test, probs)  # Calculate ROC curve values
plt.plot(fpr,tpr,marker='.')  # Plot ROC curve
plt.plot([0,1],[0,1],linestyle='--')  # Plot diagonal line
plt.xlabel('False Positive Rate')  # Set x-axis label
```

```
plt.ylabel('True Positive Rate')  # Set y-axis label
plt.title('ROC curve - ADA');  # Set title for the plot

models.append('ADA')  # Append model name to a list
model_accuracy.append(accuracy_ada)  # Append accuracy to a list
model_auc_score.append(auc_ada)  # Append AUC score to a list
print(accuracy_ada, '%.3f' % auc_ada)  # Print accuracy and AUC score
```

[ ]:
```
# 8) Ensembler Learning --> Gradient Boosting

!pip install xgboost  # Install XGBoost library
from xgboost import XGBClassifier  # Import XGBoost classifier
xgb = XGBClassifier()  # Initialize XGBoost classifier

parameters = {
    'n_estimators': range(2, 10, 1),  # Define range of values for number of␣
 ↪estimators
    'max_depth': range(10, 250, 50),  # Define range of values for maximum depth
    'learning_rate': [0.1, 0.01, 0.05]  # Define learning rates to be tested
}

gs_xgb = GridSearchCV(xgb, param_grid=parameters, cv=5, verbose=0)  # Perform␣
 ↪grid search with cross-validation
gs_xgb.fit(X, y)  # Fit the model with the best parameters

gs_xgb.best_params_  # Display the best parameters found by grid search

gs_xgb.best_score_  # Display the best score obtained by grid search

model_xgb = XGBClassifier(n_estimators=8, learning_rate=0.1, max_depth=10)  #␣
 ↪Create XGBoost classifier with specified parameters
model_xgb.fit(X_train, y_train)  # Fit the XGBoost model to the training data
accuracy_xgb = model_xgb.score(X_test, y_test)  # Calculate the accuracy of the␣
 ↪model on the test data
accuracy_xgb  # Display the accuracy of the model on the test data

model_xgb.score(X_train, y_train)  # Calculate the accuracy of the model on the␣
 ↪training data

probs = model_xgb.predict_proba(X_test)  # Calculate the predicted␣
 ↪probabilities for each class on the test data
probs = probs[:, 1]  # Keep the probabilities of the positive class
auc_xgb = roc_auc_score(y_test, probs)  # Calculate the AUC score using the␣
 ↪predicted probabilities
print('AUC: %.3f' % auc_xgb)  # Display the AUC score
```

```
fpr, tpr, thresholds = roc_curve(y_test, probs)  # Calculate the ROC curve
plt.plot(fpr, tpr, marker='.')  # Plot the ROC curve
plt.plot([0, 1], [0, 1], linestyle='--')  # Plot the diagonal line
plt.xlabel('False Positive Rate')  # Set x-axis label
plt.ylabel('True Positive Rate')  # Set y-axis label
plt.title('ROC curve - XGBoost');  # Set title for the plot

plt.figure(figsize=(8, 3))  # Create a new figure with specified size
sns.barplot(y=columns, x=model_xgb.feature_importances_)  # Create a bar plot␣
 ↪for feature importance
plt.title("Feature Importance in Model");  # Set title for the plot
plt.savefig('Fig7. Feature important.png', dpi=300, bbox_inches='tight')
plt.show()

models.append('XGBoost')  # Add model name to the list of models
model_accuracy.append(accuracy_xgb)  # Add model accuracy to the list
model_auc_score.append(auc_xgb)  # Add AUC score to the list
print(accuracy_xgb, '%.3f' % auc_xgb)  # Display accuracy and AUC score
```

[ ]:

[ ]:
```
from sklearn.metrics import classification_report, confusion_matrix
# Creating a dataframe to summarize model performance
model_summary = pd.
 ↪DataFrame(zip(models,model_accuracy,model_auc_score),columns=␣
 ↪['Model','Accuracy','AUC'])
model_summary = model_summary.set_index('Model')

# Displaying the model summary table
model_summary
print(model_summary)

# Plotting a bar chart to compare different classification models
model_summary.plot(figsize=(10,7),kind='bar')
plt.xlabel('Different classification models')
plt.yticks(np.arange(0, 1.2, step=0.2))
plt.title ("Comparison of different classification Algorithms");
plt.savefig('Fig8. Compariosn of different ML algorithms.png', dpi=300,␣
 ↪bbox_inches='tight')
plt.show()
```

Obs: As Random Forest Model showed highest accuracy in our data, we will set Random Forest as our Final Model Data Modeling:

[ ]:
```
# Creating a Classification report for Random Forest model
# Initializing the best model with specific hyperparameters
```

```
best_model =␣
 ↪RandomForestClassifier(n_estimators=100,max_depth=None,min_samples_leaf=1,min_samples_split

# Fitting the best model on the training data
best_model.fit(X_train,y_train)

# Predicting the target variable using the best model on the test data
y_predict_rf = best_model.predict(X_test)

# Generating the classification report
report_RF = classification_report(y_test, y_predict_rf)
print(report_RF)

# Generating the confusion matrix
CF_matrix = confusion_matrix(y_test,y_predict_rf)
print('Confusion Matrix:\n',CF_matrix)

# Creating a heatmap of the confusion matrix
sns.heatmap(CF_matrix/np.sum(CF_matrix),annot=True,fmt='.2%')

model_score = best_model.score(X_test, y_test)
print ('Accuracy of Random Forest: %.3f' % model_score)
```

With this, it is concluded that in given dataset, Random forest provides the best accuracy of 84% compared to all other machine learning algorithms