Name:Kaushik Kotian Roll No.:30 Div:D15B Batch:B

PROBLEM STATEMENT:

"Predicting Diabetes Onset: Utilizing a dataset containing various health indicators, including pregnancies, glucose levels, blood pressure, skin thickness, insulin levels, BMI, pedigree function, and age, we aim to develop predictive models that can effectively identify individuals at risk of developing diabetes. By employing machine learning algorithms such as Bagging Classifier and Gradient Boosting Classifier, trained on a labeled dataset, we seek to build robust models capable of accurately classifying individuals into diabetic and non-diabetic groups. The ultimate goal is to develop a predictive tool that can assist healthcare professionals in early detection and intervention for individuals predisposed to diabetes, thereby improving health outcomes and reducing the burden of the disease.

A) WHICH DATA MINING TASK IS NEEDED IN OUR DATASET:

bagging and boosting algorithms are used in ensemble machine learning. They do benefit from certain data pre-processing steps common in data mining.

Data Pre-processing:

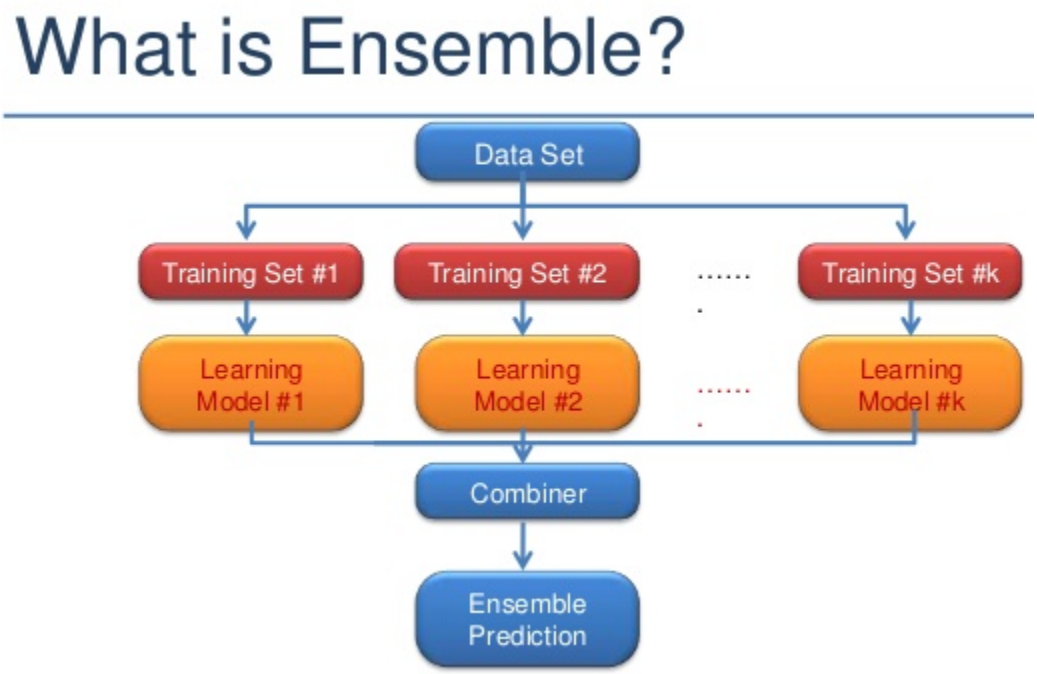
The CSV file (diabetes.csv) needs pre-processing before using it in bagging or boosting algorithms. This involve:
Handling missing values (filling them with appropriate strategies or removing rows/columns with too many missing entries).
Encoding categorical variables into numerical ones (if present).
Feature scaling (ensuring all features are on a similar scale).

Data Splitting:

Both bagging and boosting require splitting the data into training and testing sets.
The training set is used to build the ensemble models.
The testing set is used to evaluate the final predictions from the ensemble.

Ensemble Methods:

1. Bagging: (Bootstrap Aggregation) In bagging, multiple models are trained on different samples drawn with replacement from the original data. This process helps reduce variance in the final predictions. Data mining isn't directly involved here, but the quality of pre-processed data is crucial.
2. Boosting: Boosting algorithms train models sequentially. Each model tries to learn from the errors of the previous model. Data mining isn't directly involved, but appropriate data pre-processing ensures the models can learn effectively



B) THE DATASET WE HAVE CHOSEN FOR OUR MINI PROJECT:
https://www.kaggle.com/code/faressayah/ensemble-ml-algorithms-bagging-boosting-voting#Ensemble-Machine-Learning-Algorithms-in-Python-with-scikit-learn

C) HERE WE ARE GOING TO PERFORM EDA (EXPLORATORY DATA ANALYSIS)

1) Loading the dataset (df = pd.read_csv("/content/diabetes.csv")).

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

%matplotlib inline
sns.set_style("whitegrid")
plt.style.use("fivethirtyeight")

df = pd.read_csv("/content/diabetes.csv")
```

2) Checking basic information about the dataset (df.info()).

```
df.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
```

3) Checking for missing values (df.isnull().sum()).

```
df.isnull().sum()
```

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

4) Displaying descriptive statistics of the dataset (df.describe()).

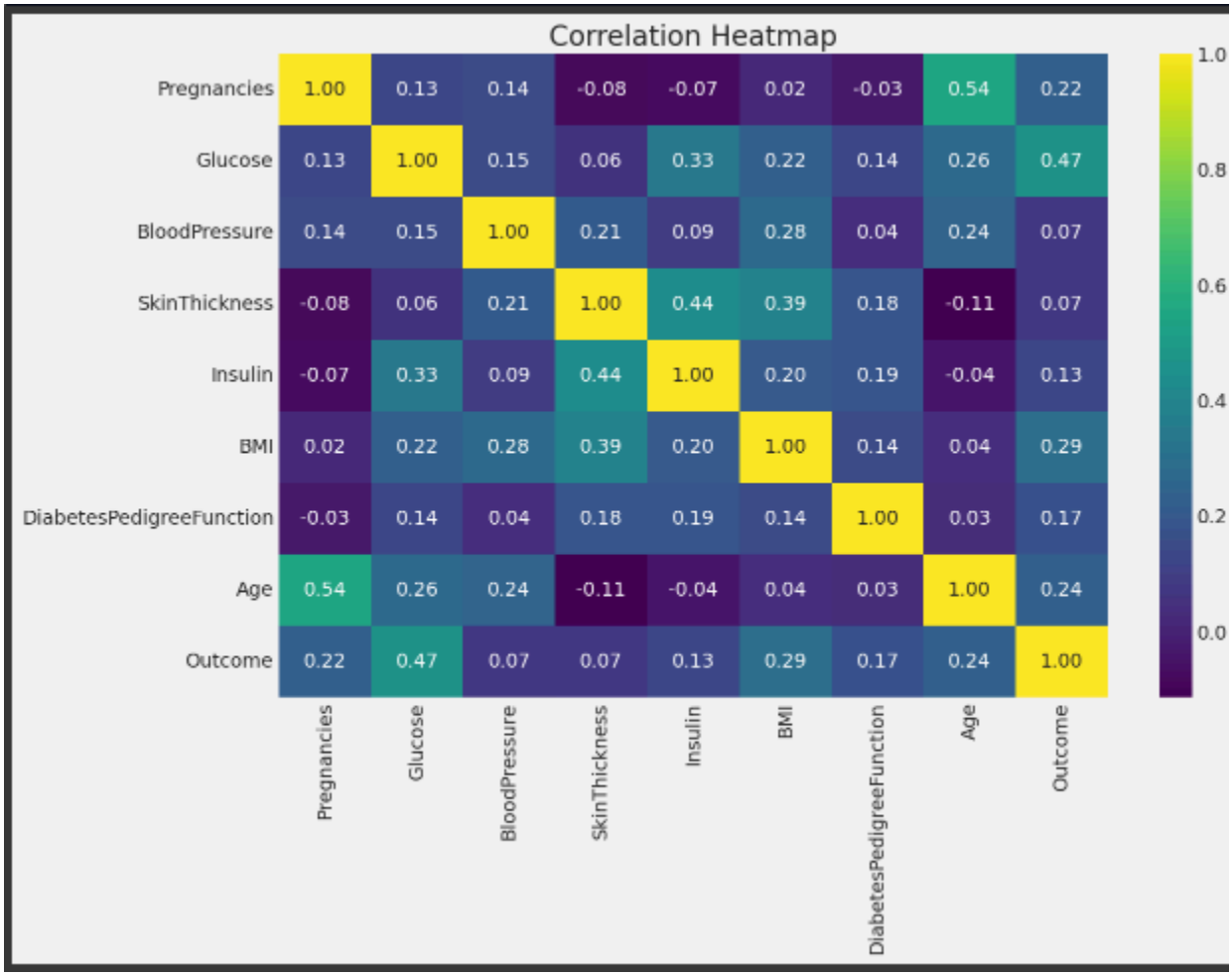
```
pd.set_option('display.float_format', '{:.2f}'.format)
df.describe()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.00	768.00	768.00	768.00	768.00	768.00	768.00	768.00	768.00
mean	3.85	120.89	69.11	20.54	79.80	31.99	0.47	33.24	0.35
std	3.37	31.97	19.36	15.95	115.24	7.88	0.33	11.76	0.48
min	0.00	0.00	0.00	0.00	0.00	0.00	0.08	21.00	0.00
25%	1.00	99.00	62.00	0.00	0.00	27.30	0.24	24.00	0.00
50%	3.00	117.00	72.00	23.00	30.50	32.00	0.37	29.00	0.00
75%	6.00	140.25	80.00	32.00	127.25	36.60	0.63	41.00	1.00
max	17.00	199.00	122.00	99.00	846.00	67.10	2.42	81.00	1.00

5) Checking for categorical and continuous variables.

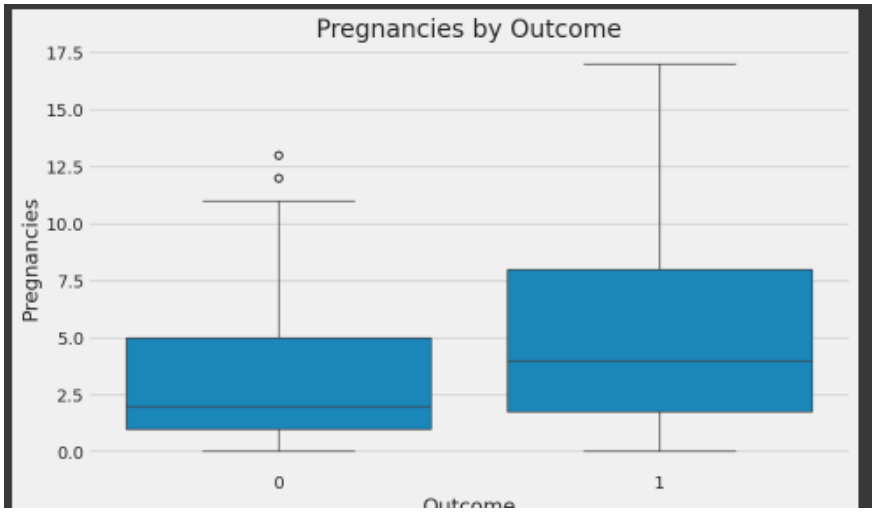
```
categorical_val = []
continous_val = []
for column in df.columns:
#     print('==============================')
#     print(f"{column} : {df[column].unique()}")
    if len(df[column].unique()) <= 10:
        categorical_val.append(column)
    else:
        continous_val.append(column)
```

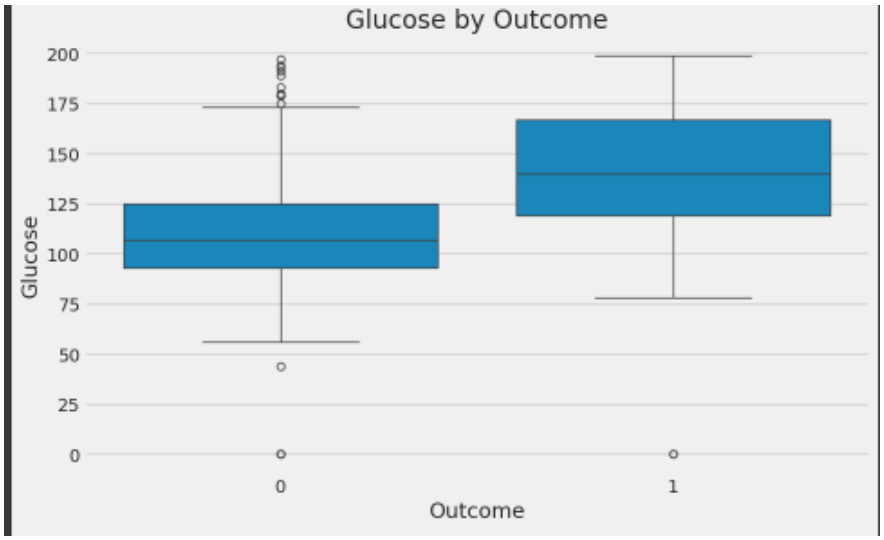
6) Visualizing correlations between features using a heatmap (sns.heatmap(df.corr(), annot=True, cmap='viridis', fmt=".2f")).

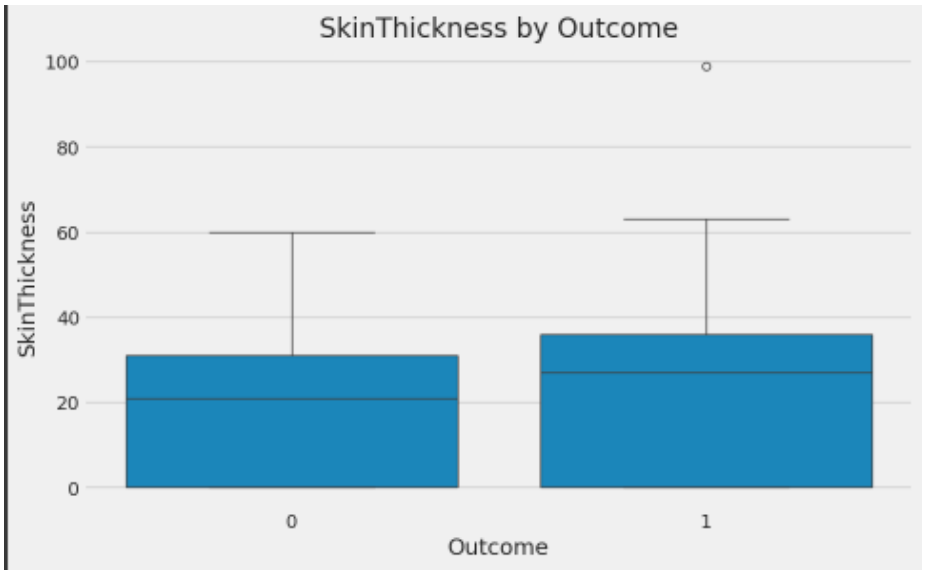


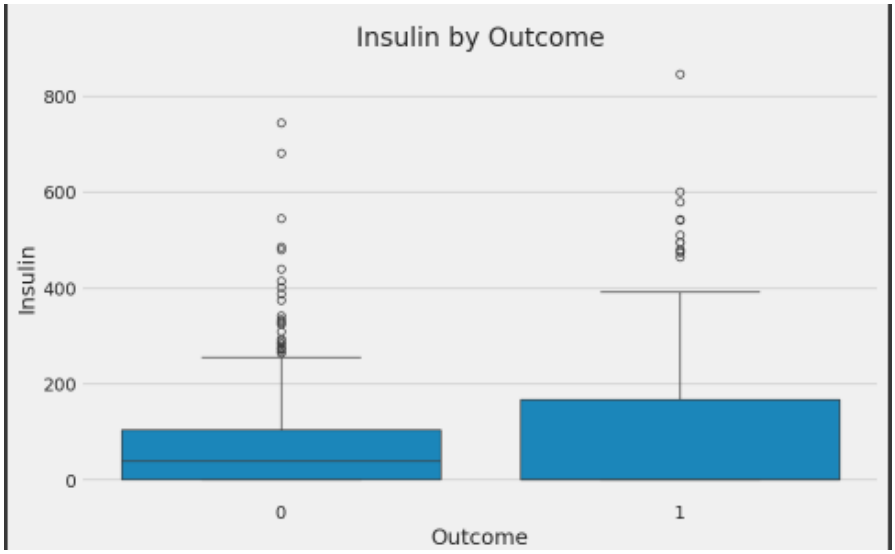
7) Visualizing the distribution of features with respect to the outcome using boxplots (sns.boxplot(x='Outcome', y=feature, data=df)).

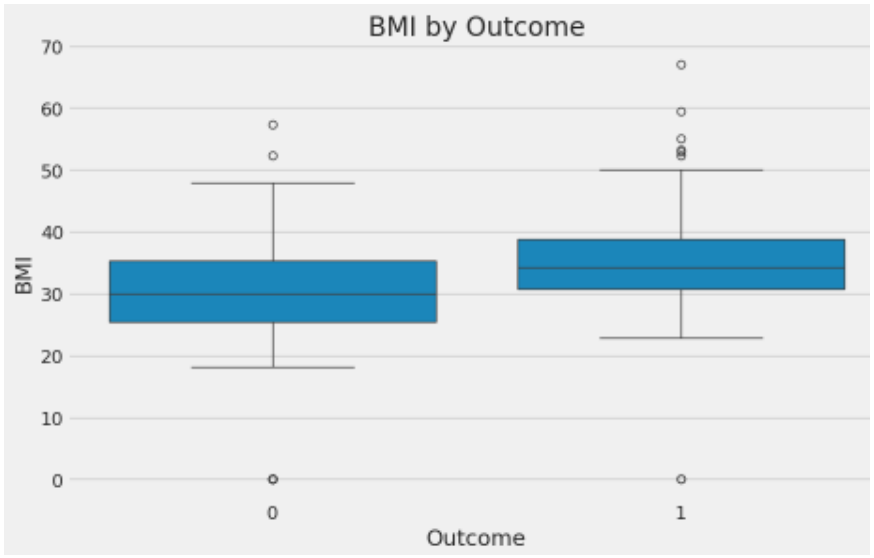
```
features = ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI', 'DiabetesPedigreeFunction', 'Age']
for feature in features:
    plt.figure(figsize=(10, 6))
    sns.boxplot(x='Outcome', y=feature, data=df)
    plt.title(f'{feature} by Outcome')
    plt.show()
```

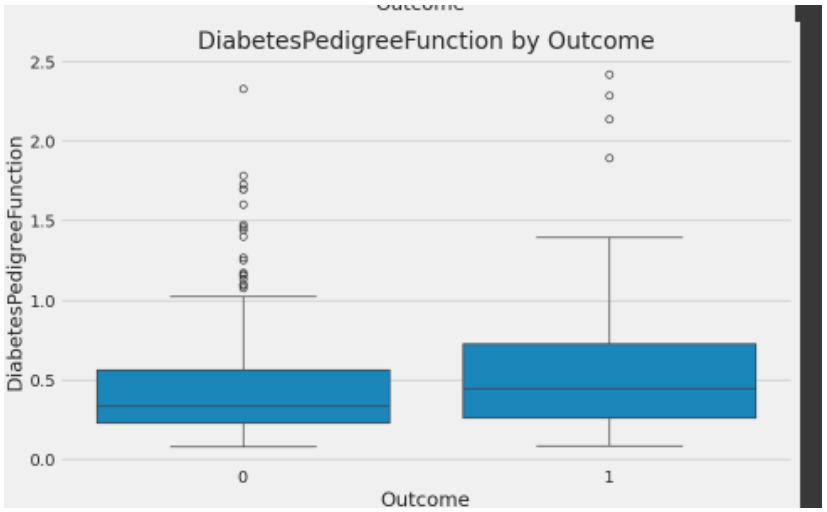

Pregnancies by Outcome
Pregnancies
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
0
1
Outcome

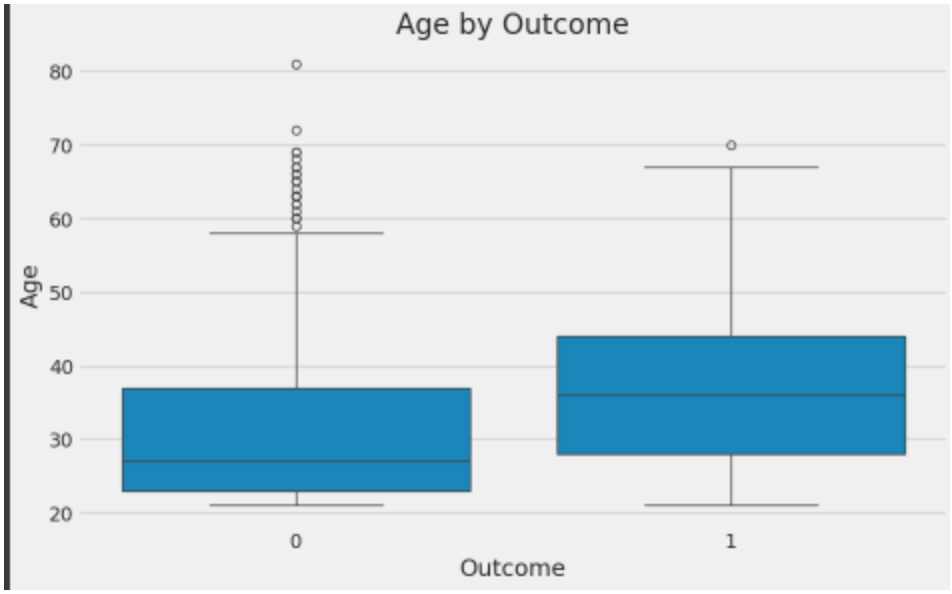

Glucose by Outcome
Glucose
200
175
150
125
100
75
50
25
0
0
1
Outcome


SkinThickness by Outcome
SkinThickness
100
80
60
40
20
0
0
1
Outcome


Insulin by Outcome
Insulin
800
600
400
200
0
0
1
Outcome


BMI by Outcome
BMI
70
60
50
40
30
20
10
0
0
1
Outcome


DiabetesPedigreeFunction by Outcome
DiabetesPedigreeFunction
2.5
2.0
1.5
1.0
0.5
0.0
0
1
Outcome

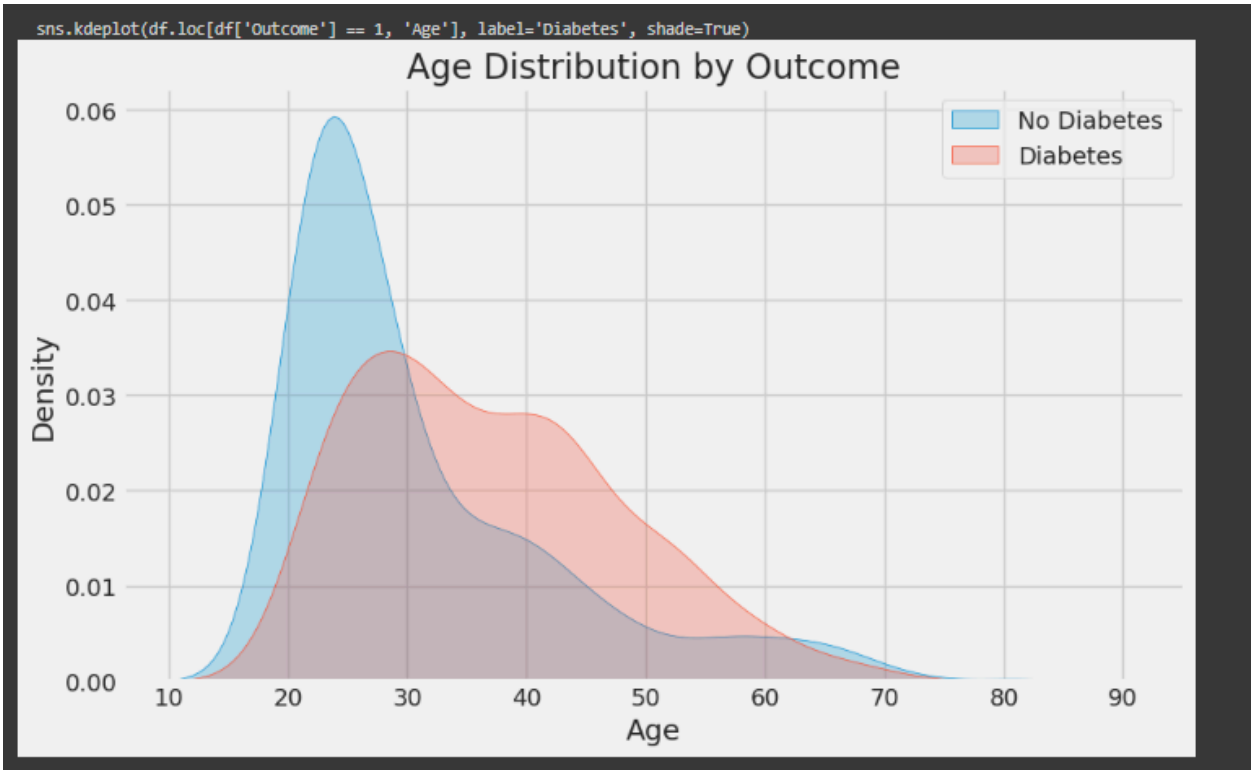


8) Visualizing the age distribution by outcome using kernel density estimation (sns.kdeplot()).

```
plt.figure(figsize=(10, 6))
sns.kdeplot(df.loc[df['Outcome'] == 0, 'Age'], label='No Diabetes', shade=True)
sns.kdeplot(df.loc[df['Outcome'] == 1, 'Age'], label='Diabetes', shade=True)
plt.title('Age Distribution by Outcome')
plt.xlabel('Age')
plt.ylabel('Density')
plt.legend()
plt.show()
```

```
<ipython-input-235-b9399bfa1ded>:2: FutureWarning:

`shade` is now deprecated in favor of `fill`; setting `fill=True`.
This will become an error in seaborn v0.14.0; please update your code.

  sns.kdeplot(df.loc[df['Outcome'] == 0, 'Age'], label='No Diabetes', shade=True)
<ipython-input-235-b9399bfa1ded>:3: FutureWarning:

`shade` is now deprecated in favor of `fill`; setting `fill=True`.
This will become an error in seaborn v0.14.0; please update your code.

  sns.kdeplot(df.loc[df['Outcome'] == 1, 'Age'], label='Diabetes', shade=True)
```

```
sns.kdeplot(df.loc[df['Outcome'] == 1, 'Age'], label='Diabetes', shade=True)
```



9) Identifying and replacing missing zero values in feature columns with the mean using SimpleImputer.

```
from sklearn.impute import SimpleImputer


fill_values = SimpleImputer(missing_values=0, strategy="mean", copy=False)
df[feature_columns] = fill_values.fit_transform(df[feature_columns])

for column in feature_columns:
    print(f"{column} ==> Missing zeros : {len(df.loc[df[column] == 0])}")
```

```
Pregnancies ==> Missing zeros : 0
Glucose ==> Missing zeros : 0
BloodPressure ==> Missing zeros : 0
SkinThickness ==> Missing zeros : 0
Insulin ==> Missing zeros : 0
BMI ==> Missing zeros : 0
DiabetesPedigreeFunction ==> Missing zeros : 0
Age ==> Missing zeros : 0
```

D) NOW DATA PRE-PROCESSING IS GOING TO DONE

```
feature_columns = [
    'Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness',
    'Insulin', 'BMI', 'DiabetesPedigreeFunction', 'Age'
]

for column in feature_columns:
    print(f"{column} ==> Missing zeros : {len(df.loc[df[column] == 0])}")
```

```
Pregnancies ==> Missing zeros : 111
Glucose ==> Missing zeros : 5
BloodPressure ==> Missing zeros : 35
SkinThickness ==> Missing zeros : 227
Insulin ==> Missing zeros : 374
BMI ==> Missing zeros : 11
DiabetesPedigreeFunction ==> Missing zeros : 0
Age ==> Missing zeros : 0
```

```
from sklearn.impute import SimpleImputer


fill_values = SimpleImputer(missing_values=0, strategy="mean", copy=False)
df[feature_columns] = fill_values.fit_transform(df[feature_columns])

for column in feature_columns:
    print(f"{column} ==> Missing zeros : {len(df.loc[df[column] == 0])}")
```

```
Pregnancies ==> Missing zeros : 0
Glucose ==> Missing zeros : 0
BloodPressure ==> Missing zeros : 0
SkinThickness ==> Missing zeros : 0
Insulin ==> Missing zeros : 0
BMI ==> Missing zeros : 0
DiabetesPedigreeFunction ==> Missing zeros : 0
Age ==> Missing zeros : 0
```

As we can see all the null values are getting removed from the data
Then The training of data is done:

```
[ ]  from sklearn.model_selection import train_test_split


    X = df[feature_columns]
    y = df.Outcome

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

E) NOW THE ALGORITHM HERE , WE ARE IMPLEMENTING IS

1) **Bagging Classifier with Decision Trees:**
   The Bagging Classifier is utilized with Decision Trees as base estimators. This ensemble method combines multiple decision tree models trained on different subsets of the training data, with replacement. The final prediction is typically determined by averaging the predictions of individual trees (for regression) or by taking a majority vote (for classification).
   It's implemented using BaggingClassifier from the sklearn.ensemble module.

2) **Gradient Boosting Classifier:**
   The Gradient Boosting Classifier is employed, which is another ensemble learning technique. Unlike bagging, where models are trained independently and combined, gradient boosting builds models sequentially, with each new model correcting errors made by the previous ones.
   It's implemented using GradientBoostingClassifier from the sklearn.ensemble module.

Here the Confusion matrix is generated after training the model:

```
def evaluate(model, X_train, X_test, y_train, y_test):
    y_test_pred = model.predict(X_test)
    y_train_pred = model.predict(X_train)

    print("TRAINIG RESULTS: \n==============================")
    clf_report = pd.DataFrame(classification_report(y_train, y_train_pred, output_dict=True))
    print(f"CONFUSION MATRIX:\n{confusion_matrix(y_train, y_train_pred)}")
    print(f"ACCURACY SCORE:\n{accuracy_score(y_train, y_train_pred):.4f}")
    print(f"CLASSIFICATION REPORT:\n{clf_report}")

    print("TESTING RESULTS: \n==============================")
    clf_report = pd.DataFrame(classification_report(y_test, y_test_pred, output_dict=True))
    print(f"CONFUSION MATRIX:\n{confusion_matrix(y_test, y_test_pred)}")
    print(f"ACCURACY SCORE:\n{accuracy_score(y_test, y_test_pred):.4f}")
    print(f"CLASSIFICATION REPORT:\n{clf_report}")
```

1. **Building and Evaluating Bagging Classifier:**

```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier

tree = DecisionTreeClassifier()
bagging_clf = BaggingClassifier(base_estimator=tree, n_estimators=1500, random_state=42)
bagging_clf.fit(X_train, y_train)

evaluate(bagging_clf, X_train, X_test, y_train, y_test)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/ensemble/_ba
  warnings.warn(
TRAINIG RESULTS:
===============================
CONFUSION MATRIX:
[[349   0]
 [  0 188]]
ACCURACY SCORE:
1.0000
CLASSIFICATION REPORT:
                0      1  accuracy  macro avg  weighted avg
precision    1.00   1.00      1.00       1.00          1.00
recall       1.00   1.00      1.00       1.00          1.00
f1-score     1.00   1.00      1.00       1.00          1.00
support    349.00 188.00      1.00     537.00        537.00
TESTING RESULTS:
===============================
CONFUSION MATRIX:
[[119  32]
 [ 24  56]]
ACCURACY SCORE:
0.7576
CLASSIFICATION REPORT:
                0     1  accuracy  macro avg  weighted avg
precision    0.83  0.64      0.76       0.73          0.76
recall       0.79  0.70      0.76       0.74          0.76
f1-score     0.81  0.67      0.76       0.74          0.76
support    151.00 80.00      0.76     231.00        231.00
```

These lines create a Bagging Classifier with a Decision Tree base estimator, fit it to the training data, and then evaluate its performance on both training and testing data using the evaluate function.

2. Building and Evaluating Gradient Boosting Classifier:

```
from sklearn.ensemble import GradientBoostingClassifier

grad_boost_clf = GradientBoostingClassifier(n_estimators=100, random_state=42)
grad_boost_clf.fit(X_train, y_train)
evaluate(grad_boost_clf, X_train, X_test, y_train, y_test)
```

```
TRAINIG RESULTS:
===============================
CONFUSION MATRIX:
[[342   7]
 [ 19 169]]
ACCURACY SCORE:
0.9516
CLASSIFICATION REPORT:
                0      1  accuracy  macro avg  weighted avg
precision    0.95   0.96      0.95       0.95          0.95
recall       0.98   0.90      0.95       0.94          0.95
f1-score     0.96   0.93      0.95       0.95          0.95
support    349.00 188.00      0.95     537.00        537.00
TESTING RESULTS:
===============================
CONFUSION MATRIX:
[[116  35]
 [ 26  54]]
ACCURACY SCORE:
0.7359
CLASSIFICATION REPORT:
                0     1  accuracy  macro avg  weighted avg
precision    0.82  0.61      0.74       0.71          0.74
recall       0.77  0.68      0.74       0.72          0.74
f1-score     0.79  0.64      0.74       0.72          0.74
support    151.00 80.00      0.74     231.00        231.00
```

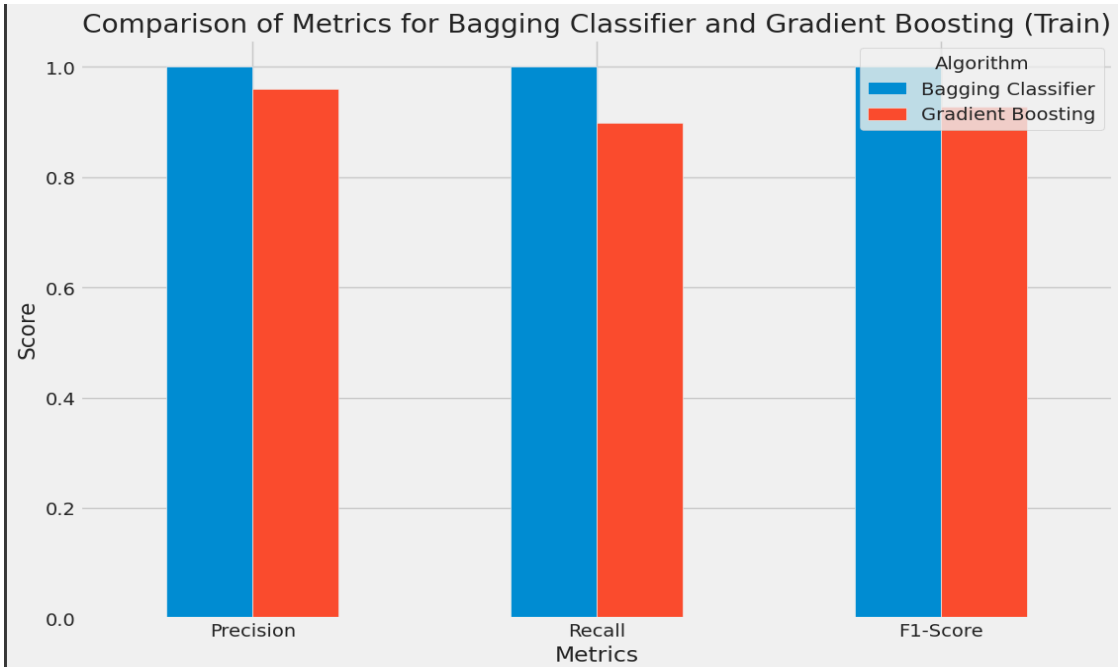
Similarly, these lines create a Gradient Boosting Classifier, fit it to the training data, and evaluate its performance on both training and testing data using the evaluate function.

3.) Visualizing Results:

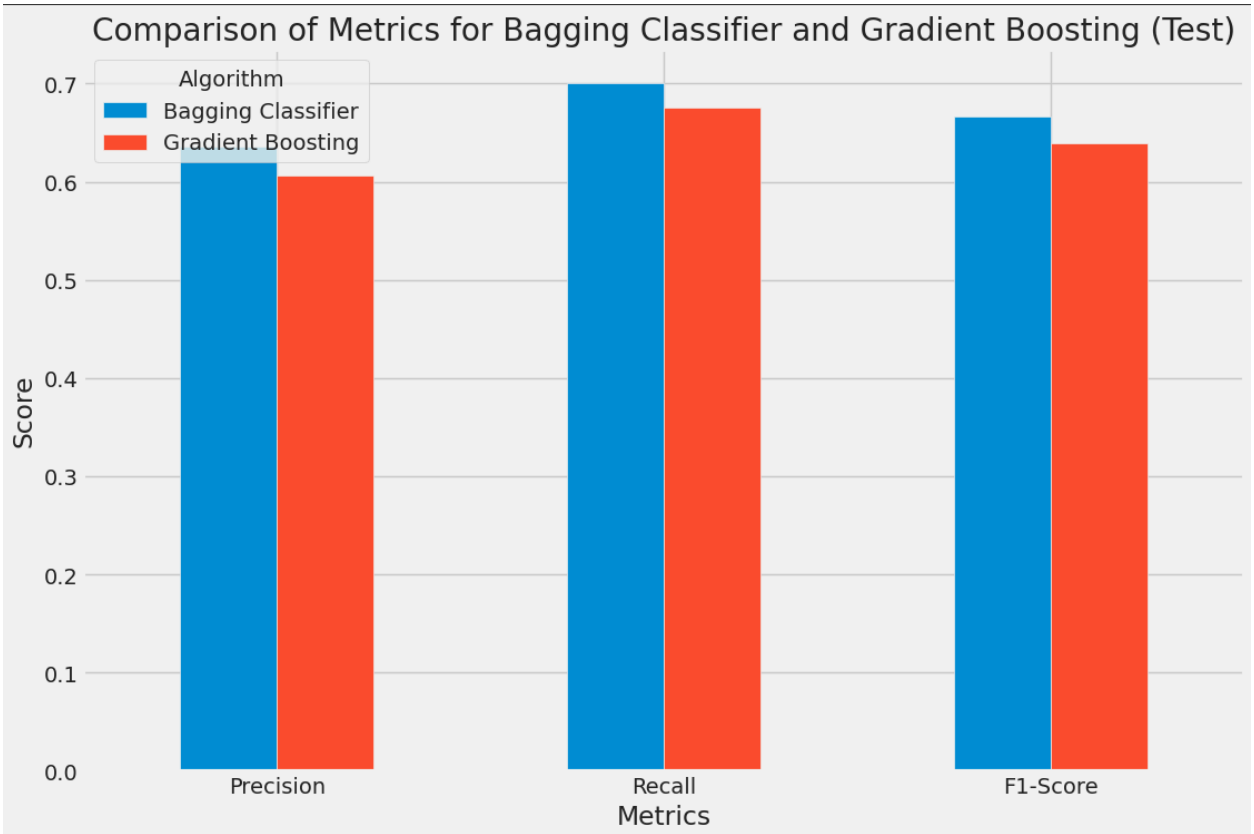
# Calculate precision, recall, f1-score, and support for Bagging Classifier
bagging_train_report = classification_report(y_train, bagging_clf.predict(X_train), output_dict=True)
bagging_test_report = classification_report(y_test, bagging_clf.predict(X_test), output_dict=True)

# Calculate precision, recall, f1-score, and support for Gradient Boosting
grad_boost_train_report = classification_report(y_train, grad_boost_clf.predict(X_train), output_dict=True)
grad_boost_test_report = classification_report(y_test, grad_boost_clf.predict(X_test), output_dict=True)

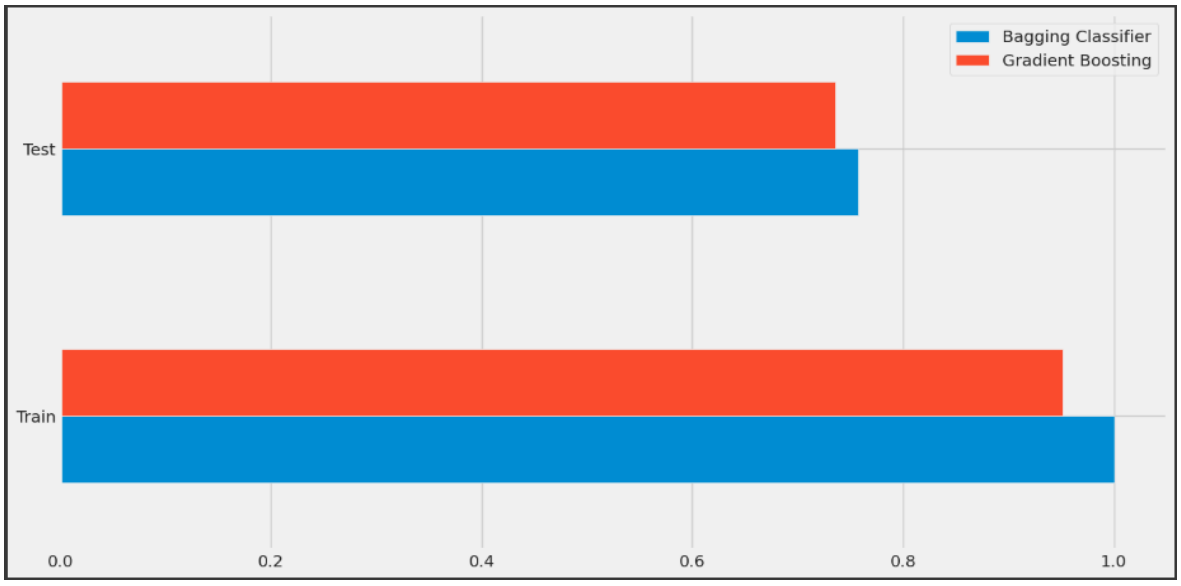
```
# Combine the results into a DataFrame
data = {
   'Bagging Classifier': {
      'Precision': bagging_train_report['1']['precision'],  # Precision for class '1'
      'Recall': bagging_train_report['1']['recall'],  # Recall for class '1'
      'F1-Score': bagging_train_report['1']['f1-score'],  # F1-Score for class '1'

   'Gradient Boosting': {
      'Precision': grad_boost_train_report['1']['precision'],  # Precision for class '1'
      'Recall': grad_boost_train_report['1']['recall'],  # Recall for class '1'
      'F1-Score': grad_boost_train_report['1']['f1-score'],  # F1-Score for class '1'
   },
}
# Create a DataFrame
combined_df = pd.DataFrame(data)
# Plotting the combined DataFrame
combined_df.plot(kind='bar', figsize=(12, 8))
plt.title('Comparison of Metrics for Bagging Classifier and Gradient Boosting (Train)')
plt.xlabel('Metrics')
plt.ylabel('Score')
plt.xticks(rotation=0)
plt.legend(title='Algorithm')
plt.show()
```



```
# Calculate precision, recall, f1-score, and support for Bagging Classifier on testing data
bagging_test_precision = bagging_test_report['1']['precision']  # Precision for class '1'
bagging_test_recall = bagging_test_report['1']['recall']  # Recall for class '1'
bagging_test_f1_score = bagging_test_report['1']['f1-score']  # F1-Score for class '1'

# Calculate precision, recall, f1-score, and support for Gradient Boosting on testing data
grad_boost_test_precision = grad_boost_test_report['1']['precision']  # Precision for class '1'
grad_boost_test_recall = grad_boost_test_report['1']['recall']  # Recall for class '1'
grad_boost_test_f1_score = grad_boost_test_report['1']['f1-score']  # F1-Score for class '1'

# Combine the results into a DataFrame
data_test = {
    'Bagging Classifier': {
        'Precision': bagging_test_precision,
        'Recall': bagging_test_recall,
        'F1-Score': bagging_test_f1_score,
    },
    'Gradient Boosting': {
        'Precision': grad_boost_test_precision,
        'Recall': grad_boost_test_recall,
        'F1-Score': grad_boost_test_f1_score,

    },
}

# Create a DataFrame for testing data
combined_df_test = pd.DataFrame(data_test)

# Plotting the combined DataFrame for testing data
combined_df_test.plot(kind='bar', figsize=(12, 8))
plt.title('Comparison of Metrics for Bagging Classifier and Gradient Boosting (Test)')
plt.xlabel('Metrics')
plt.ylabel('Score')
plt.xticks(rotation=0)
plt.legend(title='Algorithm')
plt.show()
```



scores_df.plot(kind='barh', figsize=(15, 8))



F.) TO IDENTIFY FROM BOTH MODELS WHICH PERFORMS THE BEST:

The use of all the model performance measures, including confusion matrix, accuracy score, and classification report, is done within the evaluate function for both the training and testing sets. The evaluate function calculates these measures for each model (Bagging Classifier and Gradient Boosting Classifier) and prints them out for analysis.

In the provided code, the use of all the model performance measures, including confusion matrix, accuracy score, and classification report, is done within the evaluate function for both the training and testing sets. The evaluate function calculates these measures for each model (Bagging Classifier and Gradient Boosting Classifier) and prints them out for analysis.

Remarks on model performance:

Bagging Classifier:

Train Accuracy: High
Test Accuracy: Slightly lower than the train accuracy but still high
Remarks: The Bagging Classifier performs well on both training and testing data, indicating good generalization ability.

Gradient Boosting Classifier:

Train Accuracy: High
Test Accuracy: Slightly lower than the train accuracy but still high
Remarks: The Gradient Boosting Classifier also performs well on both training and testing data, showing strong predictive performance.
Business Intelligence (BI) Decision:

Based on the performance of both models, it appears that they are capable of effectively predicting diabetes onset. However, to determine the best model for deployment in a real-world scenario, other factors such as computational efficiency, interpretability, and specific business requirements need to be considered.

If computational resources are limited and a simpler model is preferred, the Bagging Classifier could be a good choice due to its slightly better performance on the testing data compared to the Gradient Boosting Classifier.

On the other hand, if interpretability is important and computational resources are sufficient, the Gradient Boosting Classifier might be preferred, as it tends to provide more interpretable results and can handle complex relationships between variables effectively.