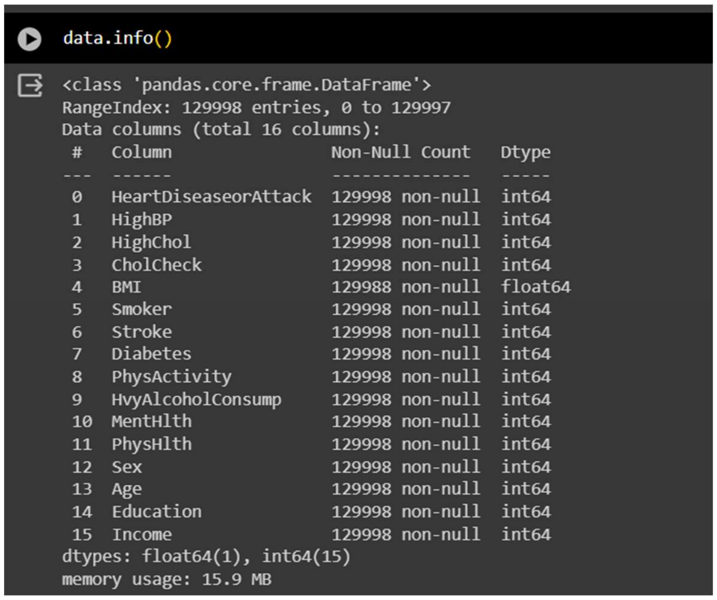
1. MODEL FOR PREDICTING HEART ATTACKS

1.1 DATA PREPARATION

The data used in this problem consists of 16 columns which all are numeric values.

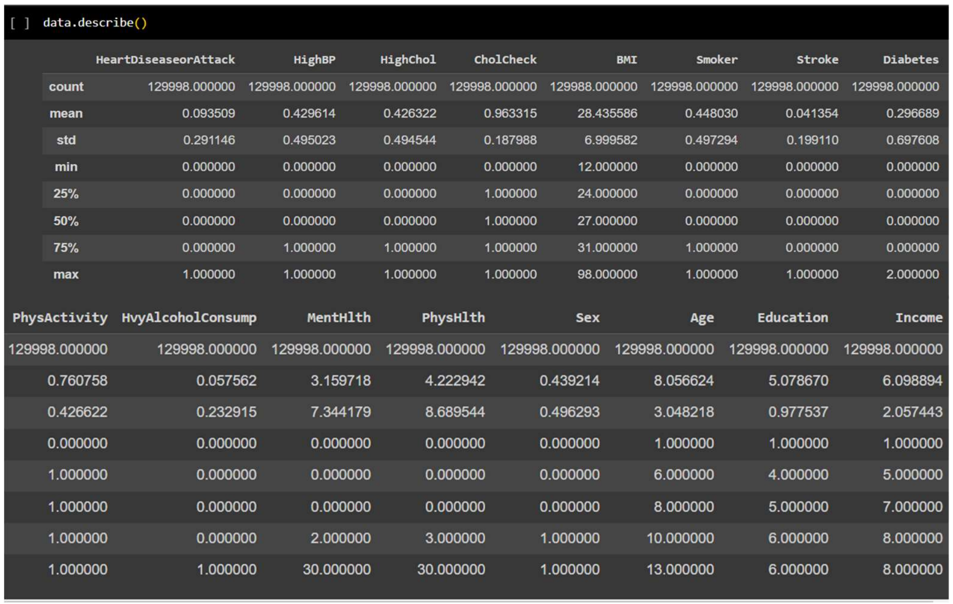


```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 129998 entries, 0 to 129997
Data columns (total 16 columns):
 #   Column                Non-Null Count   Dtype  
---  ------                --------------   -----  
 0   HeartDiseaseorAttack  129998 non-null  int64  
 1   HighBP                129998 non-null  int64  
 2   HighChol              129998 non-null  int64  
 3   CholCheck             129998 non-null  int64  
 4   BMI                   129988 non-null  float64
 5   Smoker                129998 non-null  int64  
 6   Stroke                129998 non-null  int64  
 7   Diabetes              129998 non-null  int64  
 8   PhysActivity          129998 non-null  int64  
 9   HvyAlcoholConsump     129998 non-null  int64  
 10  MentHlth              129998 non-null  int64  
 11  PhysHlth              129998 non-null  int64  
 12  Sex                   129998 non-null  int64  
 13  Age                   129998 non-null  int64  
 14  Education             129998 non-null  int64  
 15  Income                129998 non-null  int64  
dtypes: float64(1), int64(15)
memory usage: 15.9 MB
```

The target column is HeartDiseaseorAttack; the features that help us predict our target are the remaining 15 columns.

The image below shows each column's basic statistics.

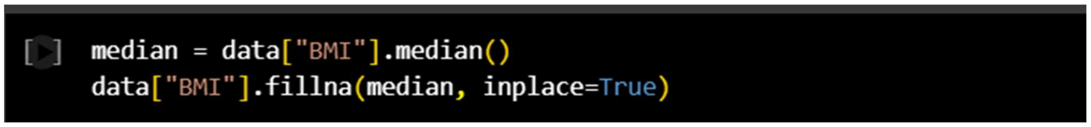


```
[ ] data.describe()
```

	HeartDiseaseorAttack	HighBP	HighChol	CholCheck	BMI	Smoker	Stroke	Diabetes
count	129998.000000	129998.000000	129998.000000	129998.000000	129988.000000	129998.000000	129998.000000	129998.000000
mean	0.093509	0.429614	0.426322	0.963315	28.435586	0.448030	0.041354	0.296689
std	0.291146	0.495023	0.494544	0.187988	6.999582	0.497294	0.199110	0.697608
min	0.000000	0.000000	0.000000	0.000000	12.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	1.000000	24.000000	0.000000	0.000000	0.000000
50%	0.000000	0.000000	0.000000	1.000000	27.000000	0.000000	0.000000	0.000000
75%	0.000000	1.000000	1.000000	1.000000	31.000000	1.000000	0.000000	0.000000
max	1.000000	1.000000	1.000000	1.000000	98.000000	1.000000	1.000000	2.000000

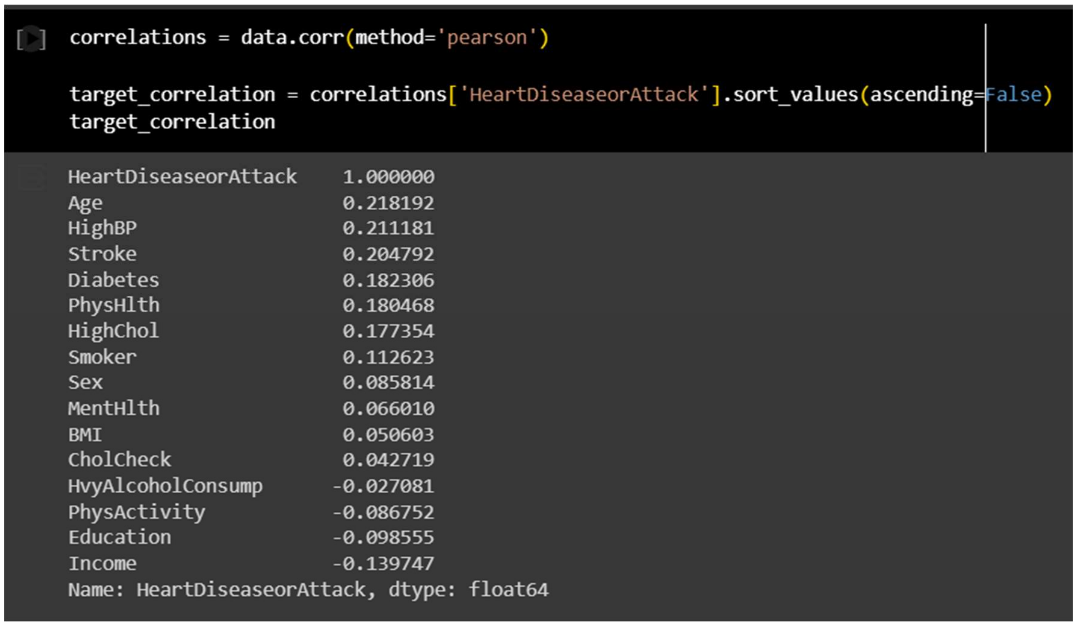
PhysActivity	HvyAlcoholConsump	MentHlth	PhysHlth	Sex	Age	Education	Income
129998.000000	129998.000000	129998.000000	129998.000000	129998.000000	129998.000000	129998.000000	129998.000000
0.760758	0.057562	3.159718	4.222942	0.439214	8.056624	5.078670	6.098894
0.426622	0.232915	7.344179	8.689544	0.496293	3.048218	0.977537	2.057443
0.000000	0.000000	0.000000	0.000000	0.000000	1.000000	1.000000	1.000000
1.000000	0.000000	0.000000	0.000000	0.000000	6.000000	4.000000	5.000000
1.000000	0.000000	0.000000	0.000000	0.000000	8.000000	5.000000	7.000000
1.000000	0.000000	2.000000	3.000000	1.000000	10.000000	6.000000	8.000000
1.000000	1.000000	30.000000	30.000000	1.000000	13.000000	6.000000	8.000000

It is evident from the first two analyses that only the BMI feature has 10 NULL values. The median value of the BMI feature is used to fill in the NULL values.



```
median = data["BMI"].median()
data["BMI"].fillna(median, inplace=True)
```

1.2 CORRELATION ANALYSIS OF FEATURES WITH TARGET

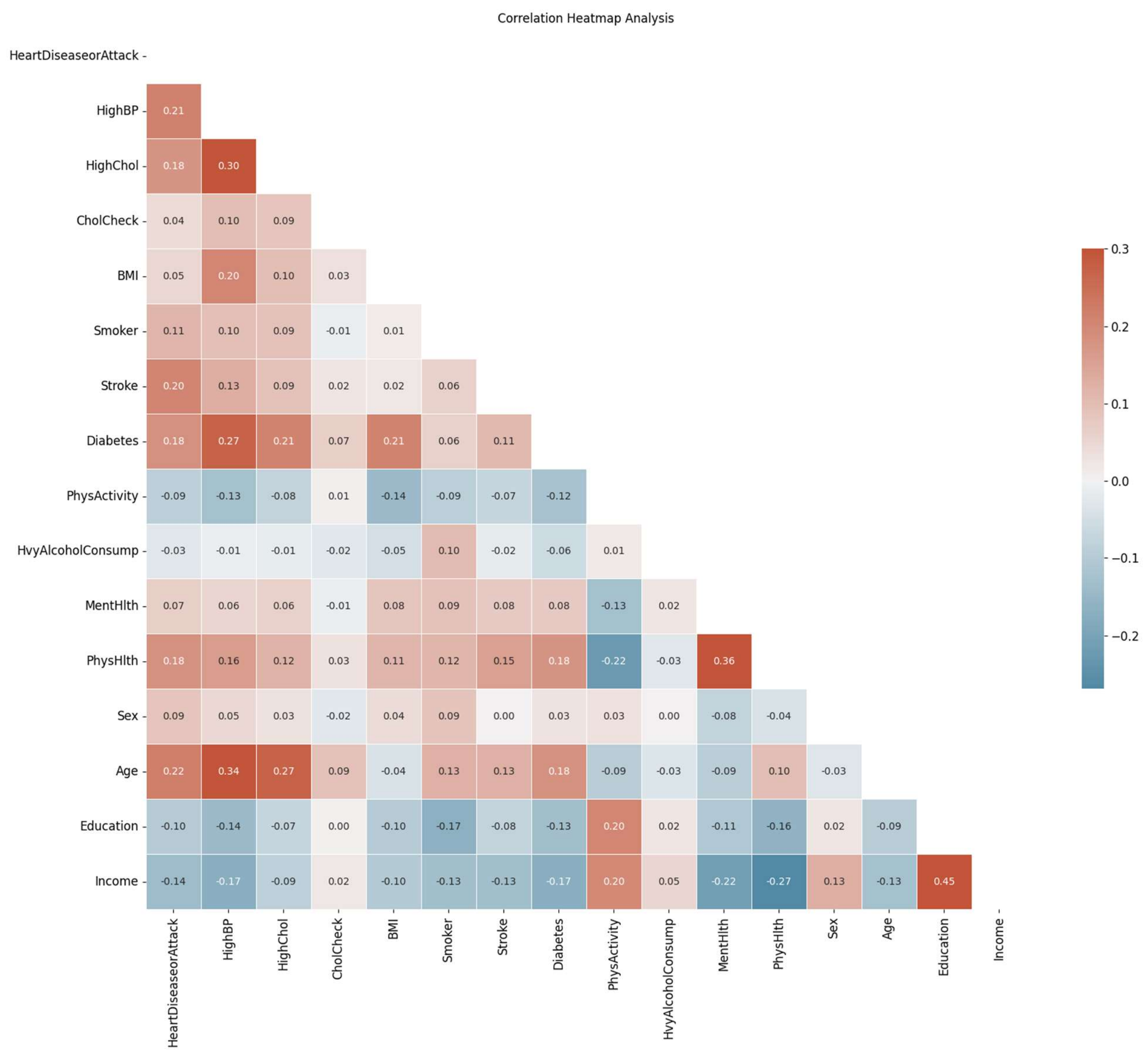


```
correlations = data.corr(method='pearson')

target_correlation = correlations['HeartDiseaseorAttack'].sort_values(ascending=False)
target_correlation
```

```
HeartDiseaseorAttack    1.000000
Age                     0.218192
HighBP                  0.211181
Stroke                  0.204792
Diabetes                0.182306
PhysHlth                0.180468
HighChol                0.177354
Smoker                  0.112623
Sex                     0.085814
MentHlth                0.066010
BMI                     0.050603
CholCheck               0.042719
HvyAlcoholConsump      -0.027081
PhysActivity           -0.086752
Education              -0.098555
Income                 -0.139747
Name: HeartDiseaseorAttack, dtype: float64
```

The Age feature has the strongest positive correlation with our target variable, followed by HighBP and Stroke, according to the Pearson correlation method. The feature with the highest negative correlation is income, meaning that the likelihood of a heart attack falls as income rises and vice versa.

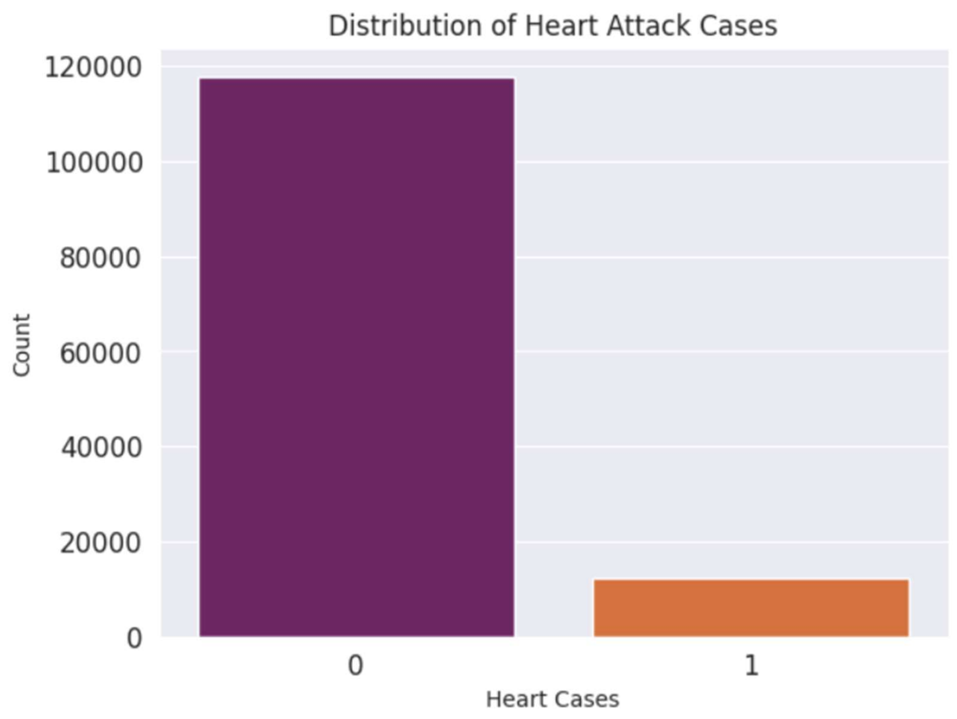
We can see how each feature correlates with every other feature in the correlation heatmap below.



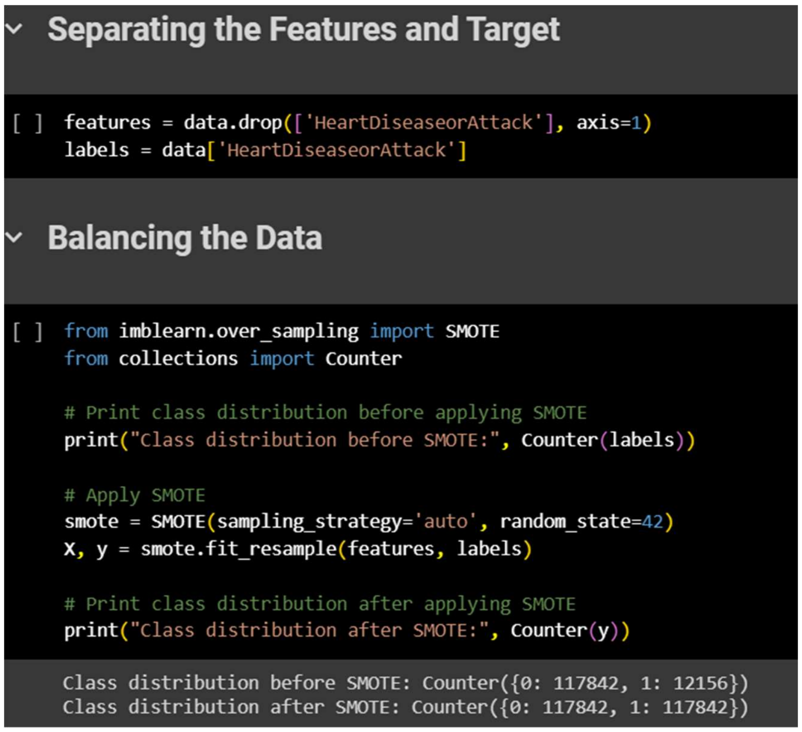
1.3 BALANCING THE CLASS DISTRIBUTION

To prevent the model from becoming biassed towards predicting the most frequent class, we should make sure that the class distribution of our target is balanced before feeding the data into our machine learning models.

As the bar chart below shows, there are significantly more heart attack cases (represented as 1) than non-heart attack cases (represented as 0), indicating a major imbalance in the distribution of our target class.



The data can be balanced by using the SMOTE method, which employs the over-sampling technique (increasing the minority class data by introducing synthetic samples to balance the count of the majority class). However, first, our target is distinguished from other features.



Separating the Features and Target

```
features = data.drop(['HeartDiseaseorAttack'], axis=1)
labels = data['HeartDiseaseorAttack']
```

Balancing the Data

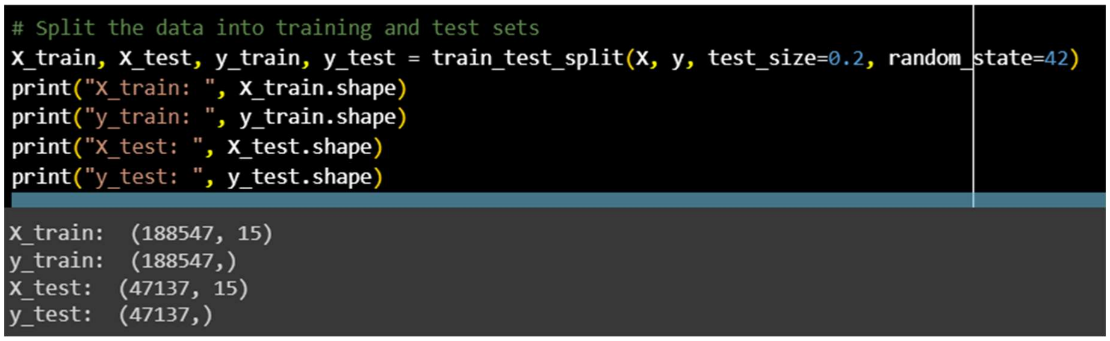
```
from imblearn.over_sampling import SMOTE
from collections import Counter

# Print class distribution before applying SMOTE
print("Class distribution before SMOTE:", Counter(labels))

# Apply SMOTE
smote = SMOTE(sampling_strategy='auto', random_state=42)
X, y = smote.fit_resample(features, labels)

# Print class distribution after applying SMOTE
print("Class distribution after SMOTE:", Counter(y))
```

```
Class distribution before SMOTE: Counter({0: 117842, 1: 12156})
Class distribution after SMOTE: Counter({0: 117842, 1: 117842})
```

The image above shows that there are 117842 data points for both classes, which can now be incorporated into our models.

1.4 SPLITTING AND STANDARDIZING THE DATA

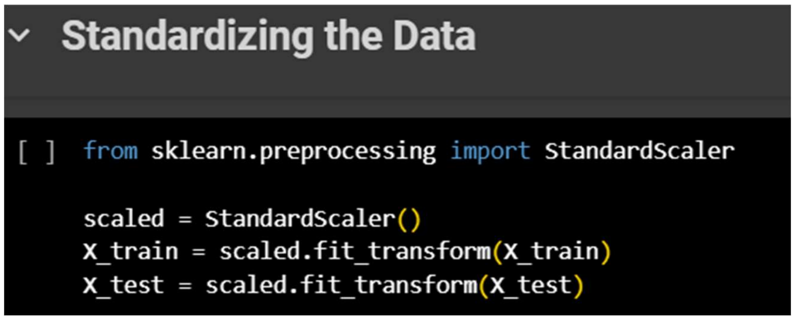
The features and targets are divided into 80% and 20% segments for training and testing the models, respectively. The structure of each segment is illustrated in the figure below.



```
# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
print("X_train: ", X_train.shape)
print("y_train: ", y_train.shape)
print("X_test: ", X_test.shape)
print("y_test: ", y_test.shape)
```

```
X_train:  (188547, 15)
y_train:  (188547,)
X_test:  (47137, 15)
y_test:  (47137,)
```

Standard Scaler is used to standardize the features x_train and x_test, guaranteeing that the values have a mean of 0 and a standard deviation of 1.



Standardizing the Data

```
from sklearn.preprocessing import StandardScaler

scaled = StandardScaler()
X_train = scaled.fit_transform(X_train)
X_test = scaled.fit_transform(X_test)
```

1.5 TRAINING AND EVALUATION OF MODELS

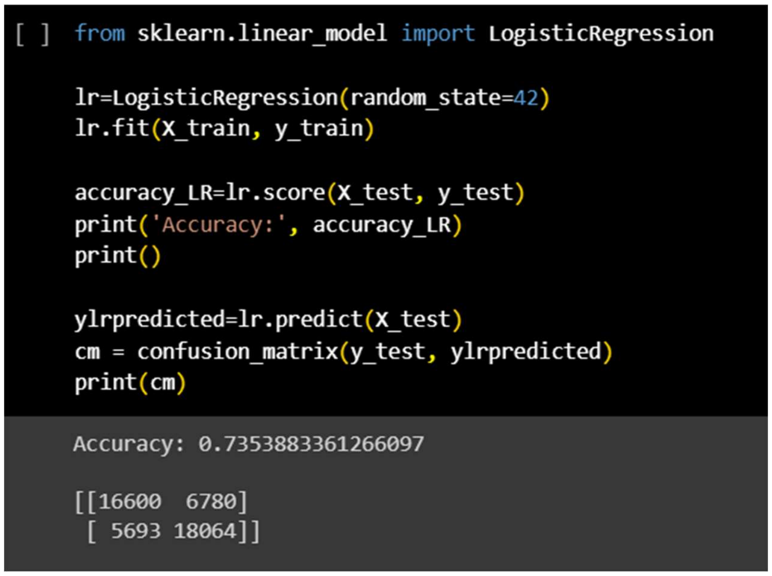
Our target variable, for the purpose of heart attack prediction, is composed of two classes: 0 denotes non-heart attack cases, while 1 denotes heart attack cases. Thus, a binary classification problem is what we have.

Our objective is to build and select up to five models that are appropriate for this binary classification. Then, we will assess several performance metrics, including Precision, Recall, F1-score, AUC-ROC, and Accuracy, to determine which model is best suited for this task.

We will also attempt to enhance the model's performance through a variety of techniques after the optimal model has been chosen, making our model a stronger contender for heart attack prediction.

Logistic Regression, Decision Tree, K-Nearest Neighbors, Random Forest, and an ANN model are the models selected for this purpose. The performance of each model on our data is shown in the figures below.

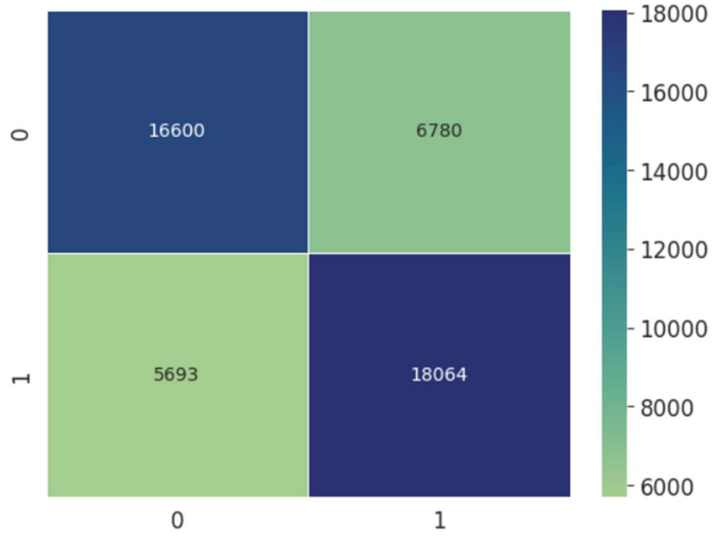
LOGISTIC REGRESSION MODEL



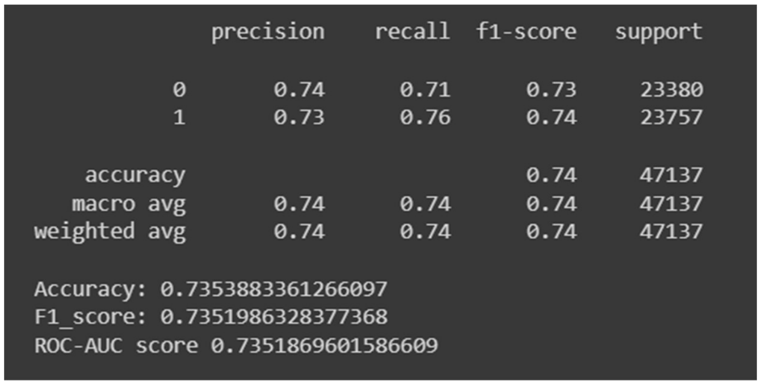
```
[ ]  from sklearn.linear_model import LogisticRegression

     lr=LogisticRegression(random_state=42)
     lr.fit(X_train, y_train)

     accuracy_LR=lr.score(X_test, y_test)
     print('Accuracy:', accuracy_LR)
     print()

     ylrpredicted=lr.predict(X_test)
     cm = confusion_matrix(y_test, ylrpredicted)
     print(cm)
```

```
Accuracy: 0.7353883361266097

[[16600  6780]
 [ 5693 18064]]
```

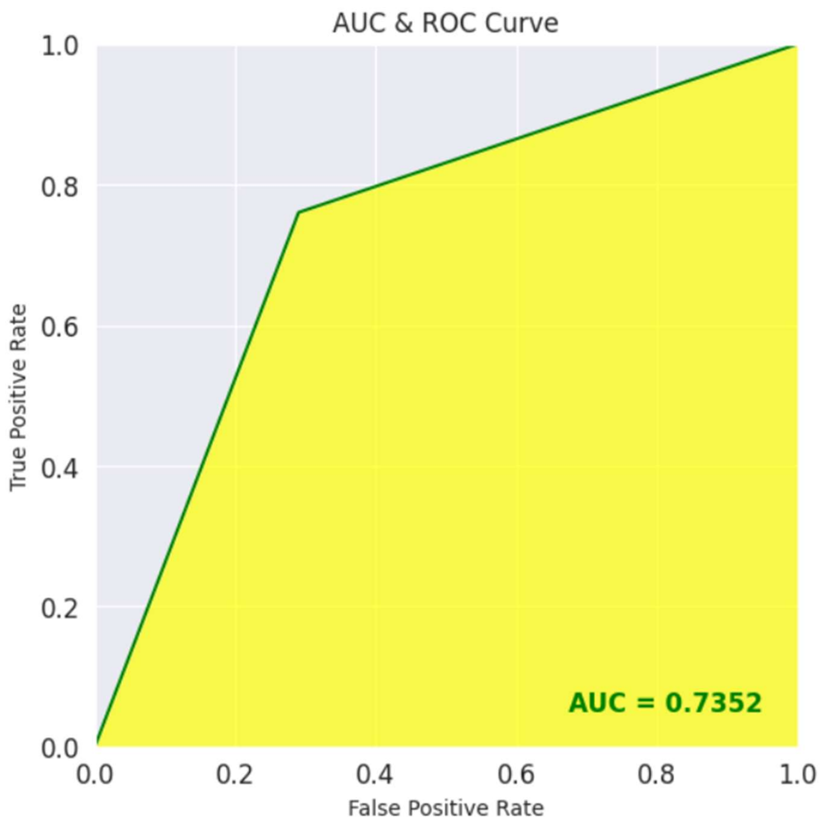
CONFUSION MATRIX



PERFORMANCE METRICS

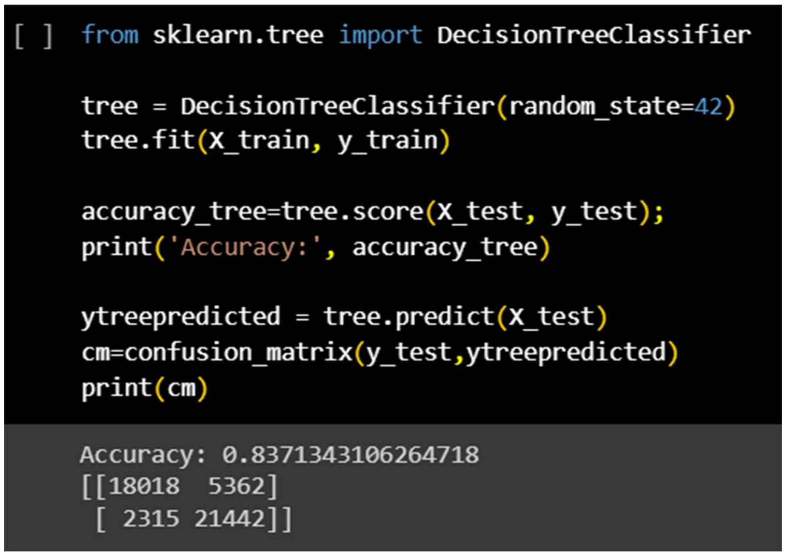


```
              precision    recall  f1-score   support

           0       0.74      0.71      0.73     23380
           1       0.73      0.76      0.74     23757

    accuracy                           0.74     47137
   macro avg       0.74      0.74      0.74     47137
weighted avg       0.74      0.74      0.74     47137

Accuracy: 0.7353883361266097
F1_score: 0.7351986328377368
ROC-AUC score 0.7351869601586609
```



With an accuracy rate of 73.53%, the Logistic Regression successfully identified 18064 heart attack cases out of 23757, or 76% of all heart attack cases. Additionally, 16600 out of 23380 cases—or 71%—had been correctly classified by the model as non-heart attack cases.

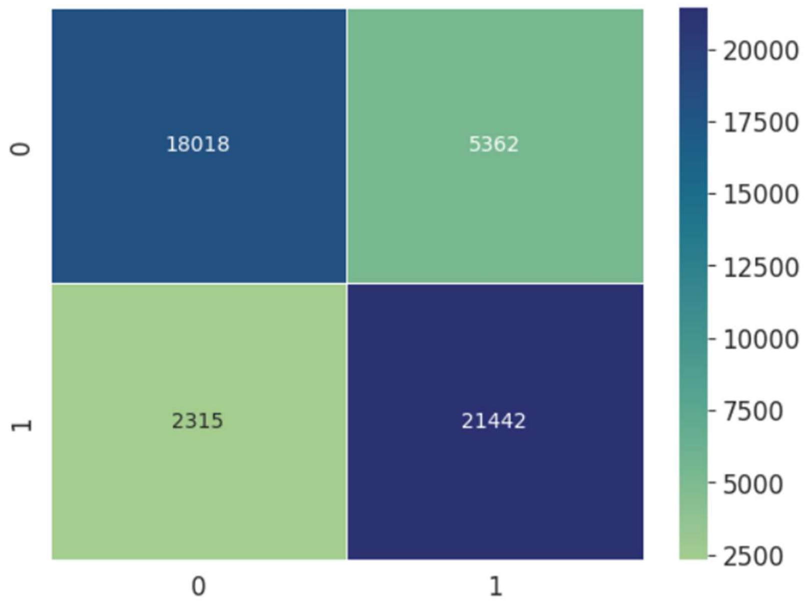
DECISION TREE CLASSIFIER



```
[ ]  from sklearn.tree import DecisionTreeClassifier

     tree = DecisionTreeClassifier(random_state=42)
     tree.fit(X_train, y_train)

     accuracy_tree=tree.score(X_test, y_test);
     print('Accuracy:', accuracy_tree)

     ytreepredicted = tree.predict(X_test)
     cm=confusion_matrix(y_test,ytreepredicted)
     print(cm)
```

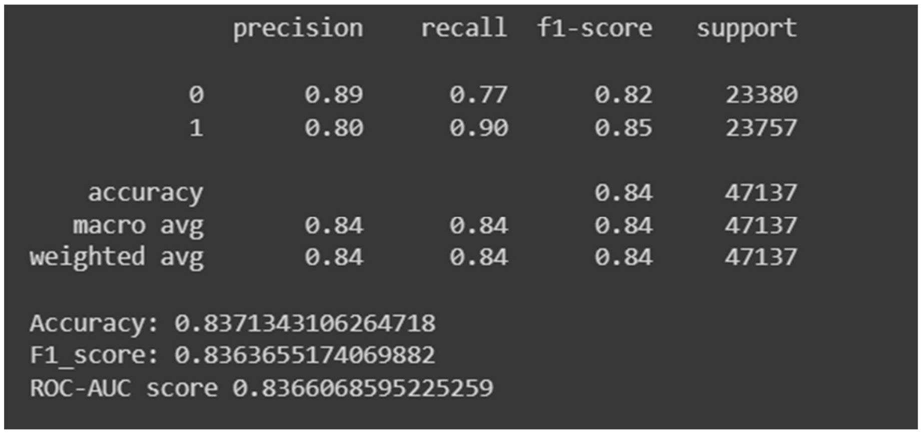
```
Accuracy: 0.8371343106264718
[[18018  5362]
 [ 2315 21442]]
```

CONFUSION MATRIX

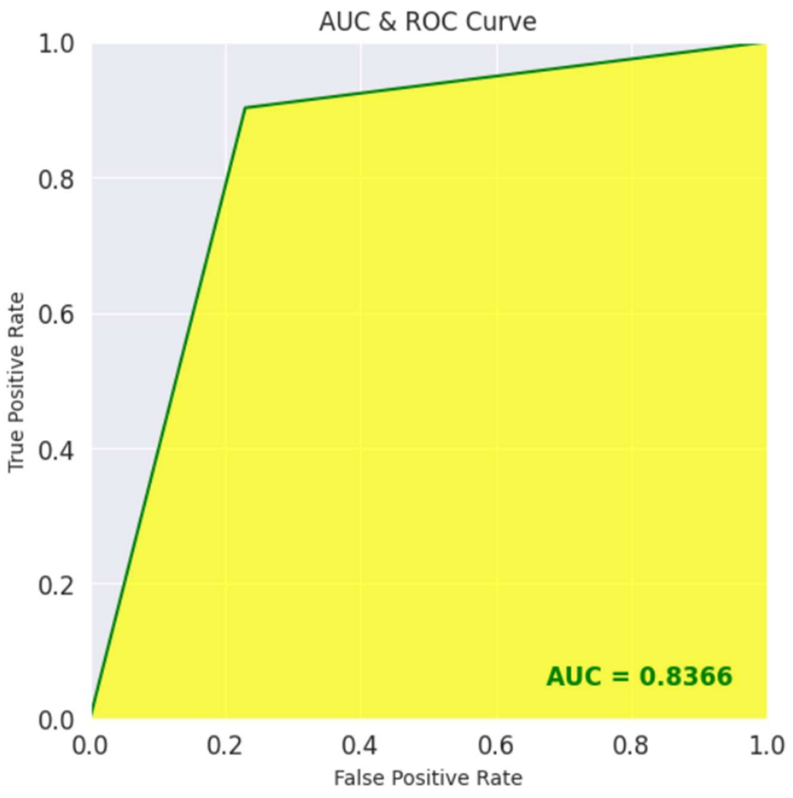


The Decision Tree Model successfully identified 21442 heart attack cases out of 23757 cases and correctly predicted 18018 non-heart attack cases out of 23380 cases, as demonstrated by the above figures, which show the model's overall accuracy of 83.71%.

Below is a display of the Decision Tree model's various performance metrics scores.

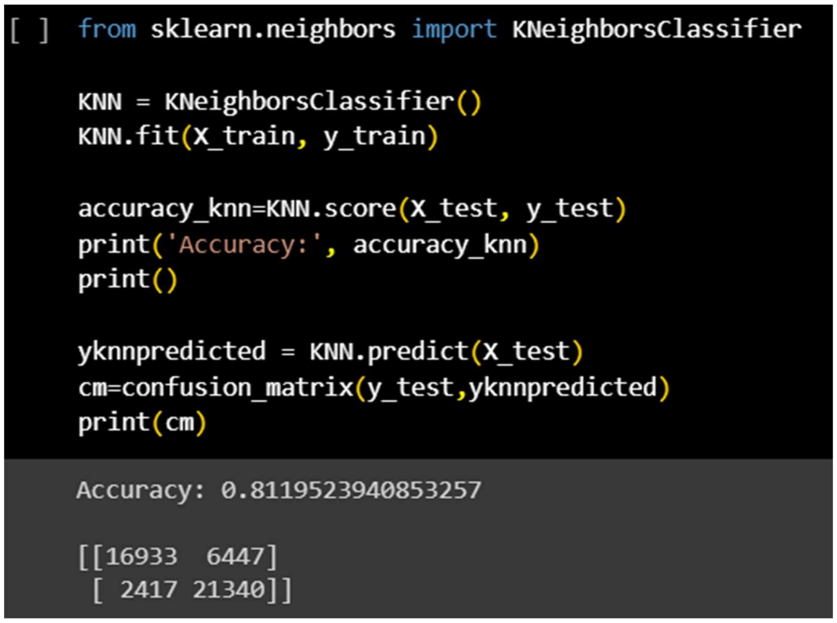


```
              precision    recall  f1-score   support

           0       0.89      0.77      0.82     23380
           1       0.80      0.90      0.85     23757

    accuracy                           0.84     47137
   macro avg       0.84      0.84      0.84     47137
weighted avg       0.84      0.84      0.84     47137

Accuracy: 0.8371343106264718
F1_score: 0.8363655174069882
ROC-AUC score 0.8366068595225259
```



As can be seen from the recall score of the positive class, out of all actual heart attack cases, 90% of heart attack cases are correctly predicted by the model; however, the recall of the negative class is only 77%, suggesting that this model is not a good fit for non-heart attack cases but rather for heart attack cases exclusively.

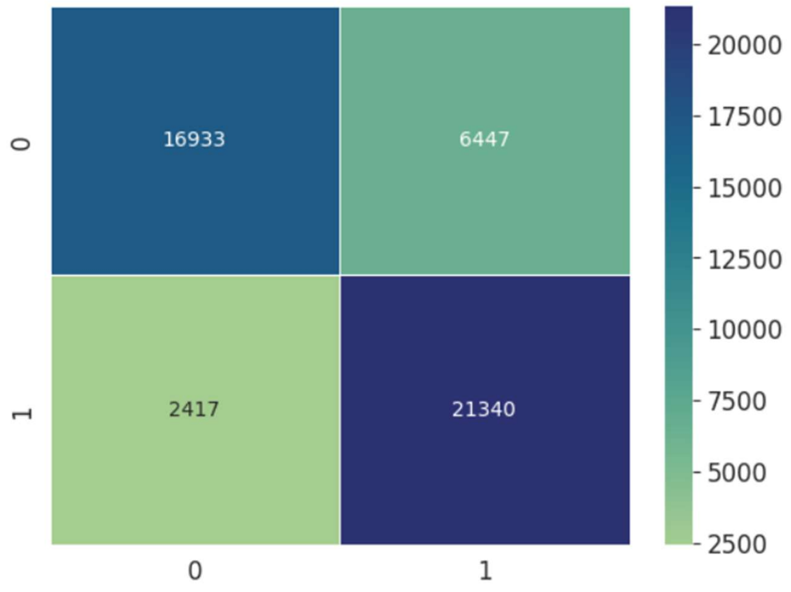
K-NEAREST NEIGHBORS CLASSIFIER



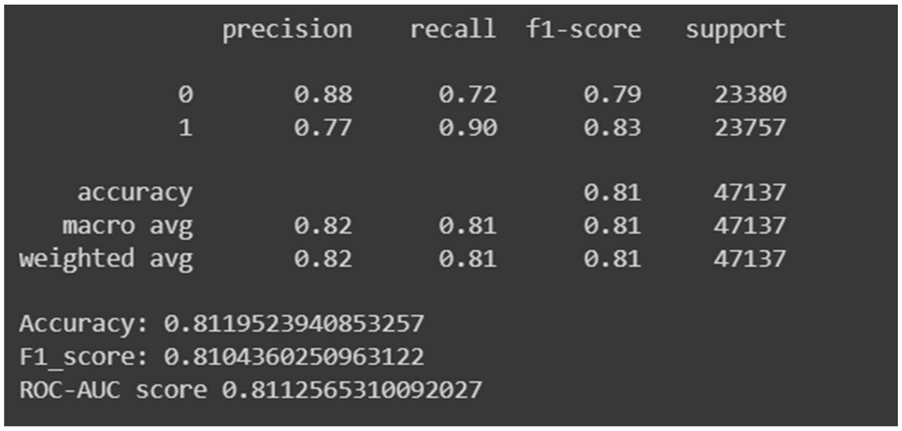
```
[ ]  from sklearn.neighbors import KNeighborsClassifier

     KNN = KNeighborsClassifier()
     KNN.fit(X_train, y_train)

     accuracy_knn=KNN.score(X_test, y_test)
     print('Accuracy:', accuracy_knn)
     print()

     yknnpredicted = KNN.predict(X_test)
     cm=confusion_matrix(y_test,yknnpredicted)
     print(cm)
```

```
Accuracy: 0.8119523940853257

[[16933  6447]
 [ 2417 21340]]
```

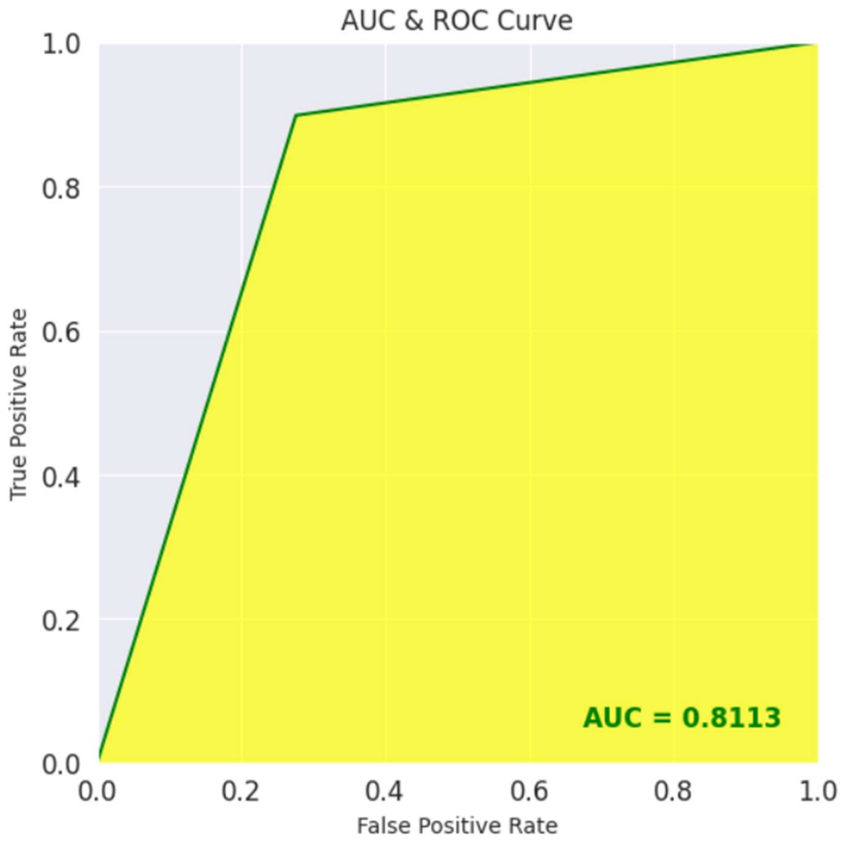
CONFUSION MATRIX



The KNN model has an overall accuracy of 81.19%, and it is clear from looking at the confusion matrix that the model performs poorly for the negative class and is biassed towards the positive class prediction.

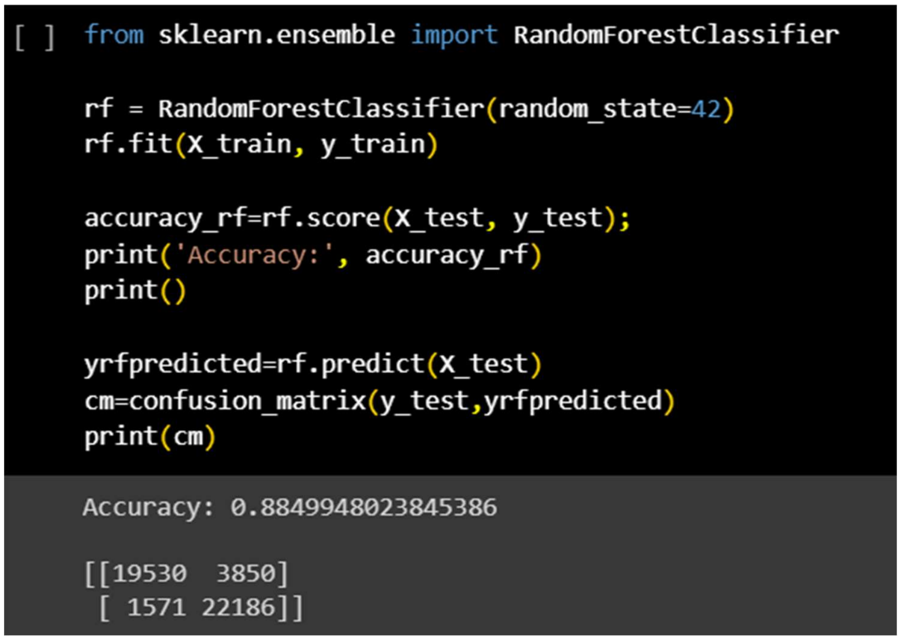


```
              precision    recall  f1-score   support

           0       0.88      0.72      0.79     23380
           1       0.77      0.90      0.83     23757

    accuracy                           0.81     47137
   macro avg       0.82      0.81      0.81     47137
weighted avg       0.82      0.81      0.81     47137

Accuracy: 0.8119523940853257
F1_score: 0.8104360250963122
ROC-AUC score 0.8112565310092027
```



There is a significant discrepancy between the positive class's precision and recall scores: 77% of the former shows that only 77% of the positive classes the model predicted are in fact positive, while 90% of the recall score indicates that 90% of the actual positive classes are correctly classified.

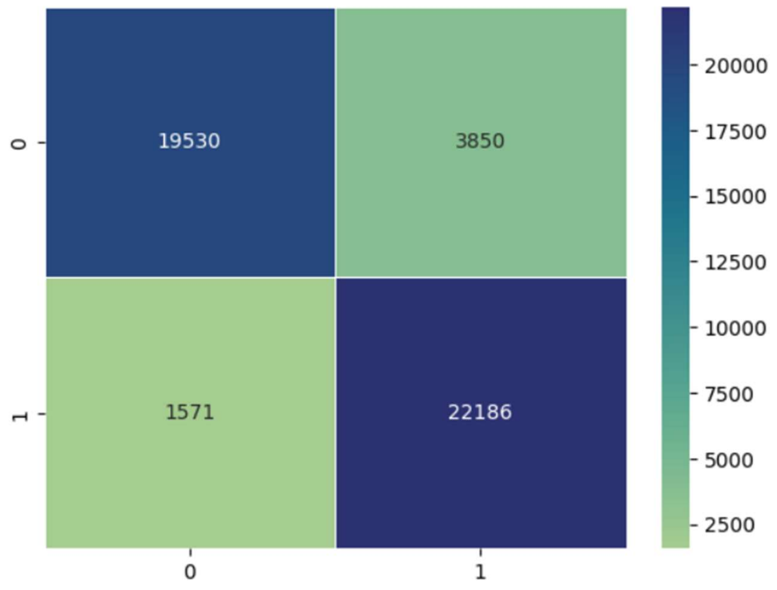
RANDOM FOREST CLASSIFIER



```
[ ] from sklearn.ensemble import RandomForestClassifier

    rf = RandomForestClassifier(random_state=42)
    rf.fit(X_train, y_train)

    accuracy_rf=rf.score(X_test, y_test);
    print('Accuracy:', accuracy_rf)
    print()

    yrfpredicted=rf.predict(X_test)
    cm=confusion_matrix(y_test,yrfpredicted)
    print(cm)
```

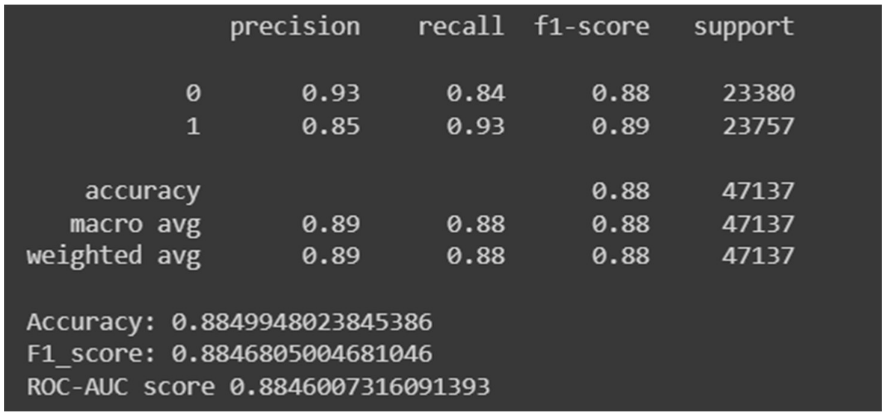
```
Accuracy: 0.8849948023845386

[[19530  3850]
 [ 1571 22186]]
```

CONFUSION MATRIX

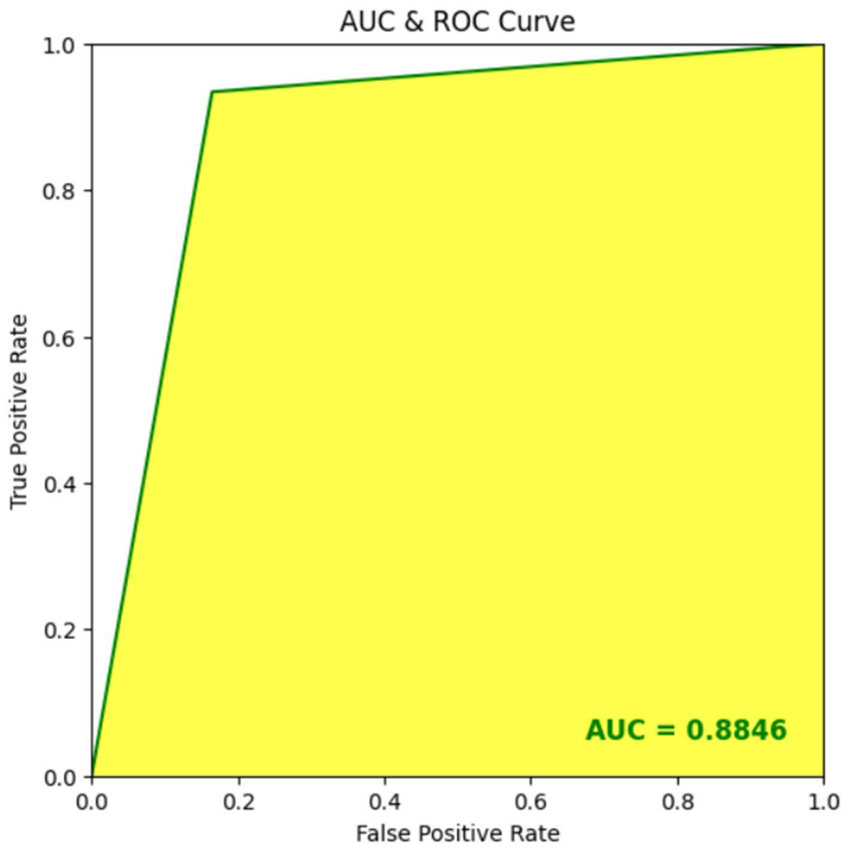


According to the results above, the Random Forest Classifier correctly classified 22186 positive class samples out of 23757 and 19530 negative class samples out of 23380, achieving a good overall accuracy of 88.49%.

The image below displays the model's performance metrics.



```
              precision    recall  f1-score   support

           0       0.93      0.84      0.88     23380
           1       0.85      0.93      0.89     23757

    accuracy                           0.88     47137
   macro avg       0.89      0.88      0.88     47137
weighted avg       0.89      0.88      0.88     47137

Accuracy: 0.8849948023845386
F1_score: 0.8846805004681046
ROC-AUC score 0.8846007316091393
```



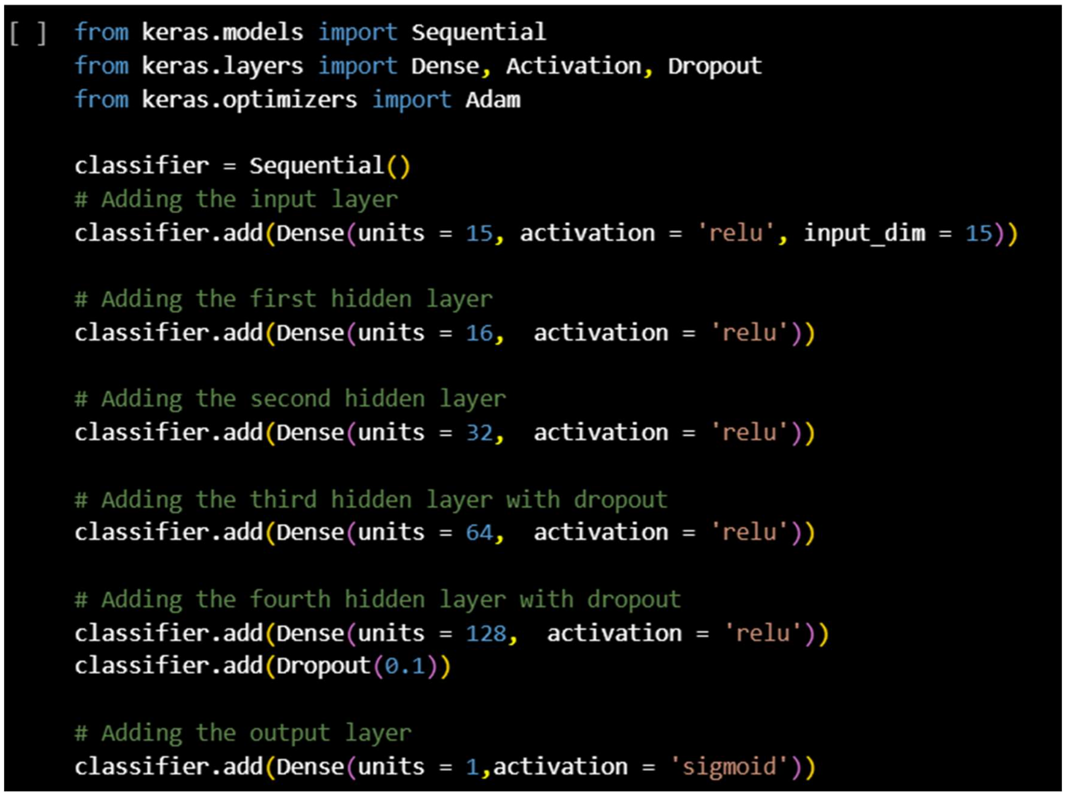
It is evident from this analysis that the Random Forest Model performs well across the board, correctly classifying 84% of the actual negative class samples and 93% of the actual positive class samples. The model's good performance in terms of both True Positive Rate and True Negative Rate is indicated by its AUC-ROC Score of 0.8846.

ARTIFICIAL NEURAL NETWORK (ANN) DEEP MODEL

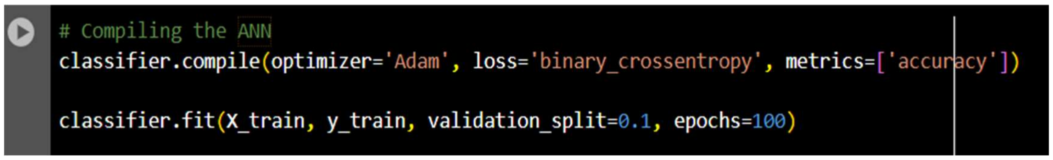
The Artificial Neural Network (ANN) architecture that is build, follows a Sequential model structure. The input layer is a Dense (fully connected) layer with 15 neurons and ReLU activation. This layer takes input from 15 features.

The architecture includes four hidden layers, all using Dense layers with ReLU activation. The first hidden layer has 16 neurons, the second has 32 neurons, and the third has 64 neurons. The fourth hidden layer consists of 128 neurons with a dropout rate of 0.1 to prevent overfitting.

Since this is a binary classification problem, the model's output layer consists of a single neuron with a Sigmoid activation function.



```
[ ]  from keras.models import Sequential
     from keras.layers import Dense, Activation, Dropout
     from keras.optimizers import Adam

     classifier = Sequential()
     # Adding the input layer
     classifier.add(Dense(units = 15, activation = 'relu', input_dim = 15))

     # Adding the first hidden layer
     classifier.add(Dense(units = 16,  activation = 'relu'))

     # Adding the second hidden layer
     classifier.add(Dense(units = 32,  activation = 'relu'))

     # Adding the third hidden layer with dropout
     classifier.add(Dense(units = 64,  activation = 'relu'))

     # Adding the fourth hidden layer with dropout
     classifier.add(Dense(units = 128,  activation = 'relu'))
     classifier.add(Dropout(0.1))

     # Adding the output layer
     classifier.add(Dense(units = 1,activation = 'sigmoid'))
```

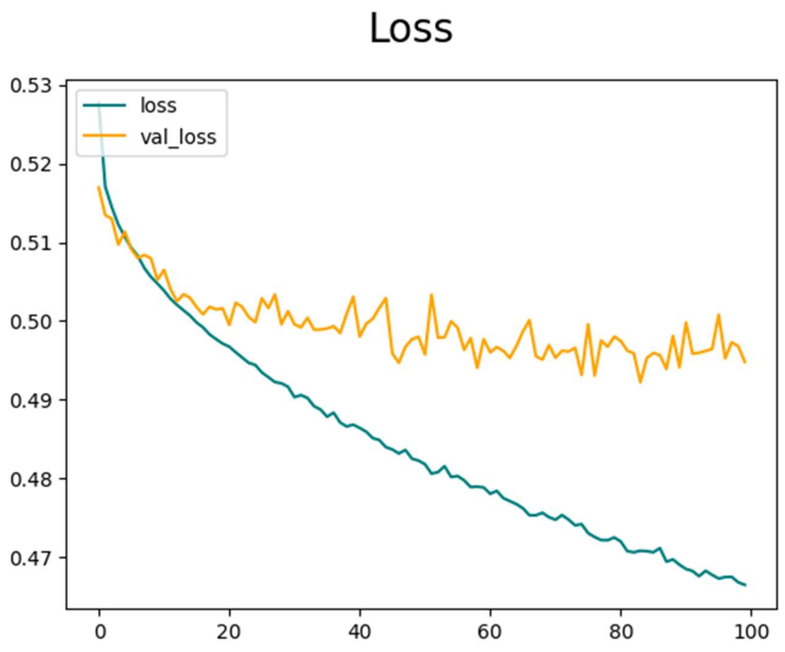


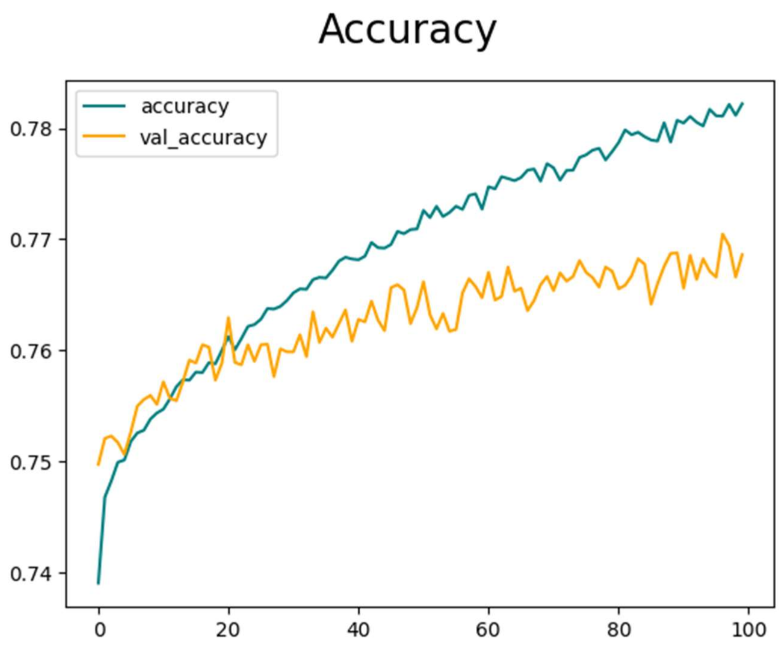
```
# Compiling the ANN
classifier.compile(optimizer='Adam', loss='binary_crossentropy', metrics=['accuracy'])

classifier.fit(X_train, y_train, validation_split=0.1, epochs=100)
```

Binary cross entropy loss, a frequently used loss function for binary classification problems, is used by the Adam optimizer to build the model. Accuracy is the metric used to assess the model's performance during training and validation.

The training data and the validation split of 0.1, or 10% of the training data, are used to fit the model and validate its performance over 100 epochs.

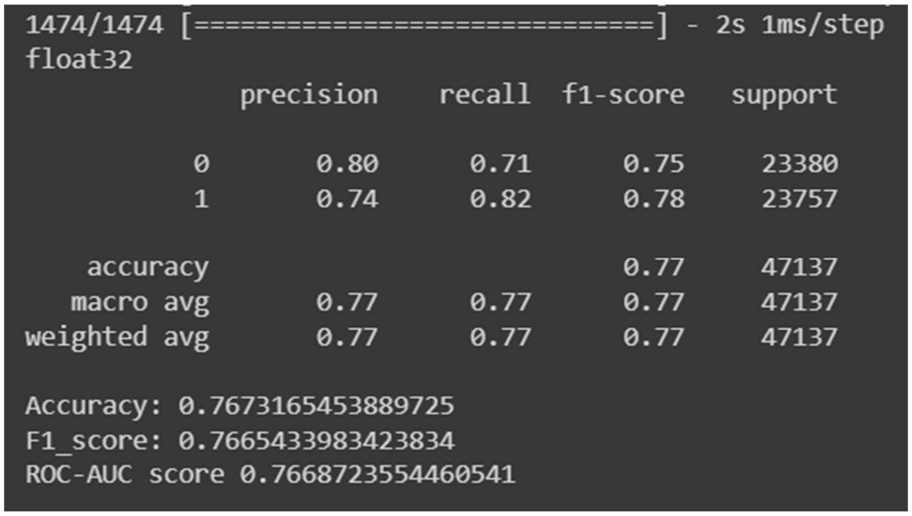
The image below displays the results.





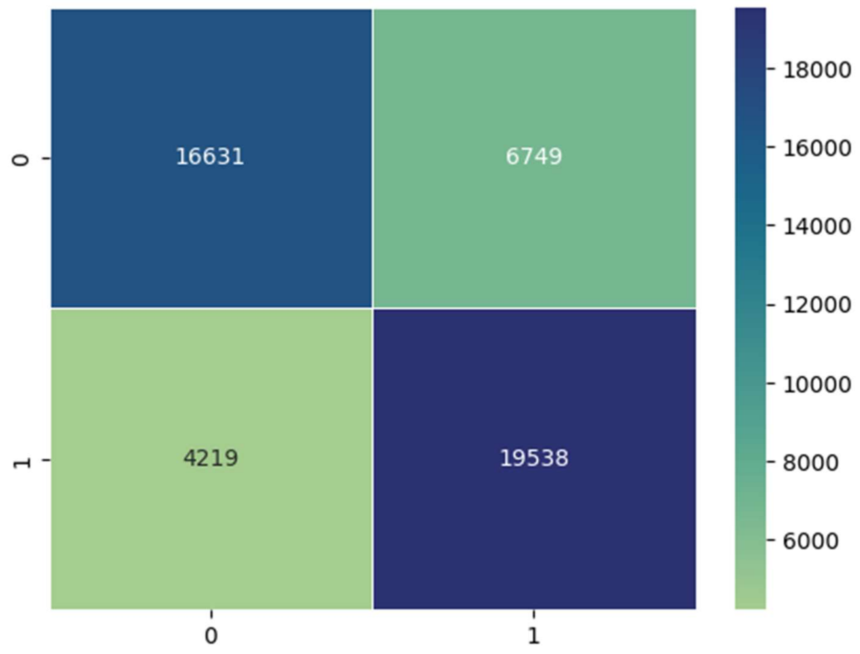
The aforementioned analysis shows that the validation loss gradually drops to 0.50 before beginning to decline very slowly. Our accuracy also shows a similar pattern. The reason for the decline in performance in the validation data is that the model began to overfit the training data.

The model's performance on the test set is displayed in the figure below.



```
1474/1474 [==============================] - 2s 1ms/step
float32
              precision    recall  f1-score   support

           0       0.80      0.71      0.75     23380
           1       0.74      0.82      0.78     23757

    accuracy                           0.77     47137
   macro avg       0.77      0.77      0.77     47137
weighted avg       0.77      0.77      0.77     47137

Accuracy: 0.7673165453889725
F1_score: 0.7665433983423834
ROC-AUC score 0.7668723554460541
```

CONFUSION MATRIX



With an overall accuracy of 76.73%, the ANN model outperformed the Logistic Regression model but not the other models. 19538 positive class samples and 16631 negative class samples were correctly classified.

PERFORMANCE COMPARISION OF MODELS

S. NO	MODELS		PRECISION	RECALL	F1-SCORE	AUC-ROC	ACCURACY
1.	LOGISTIC REGRESSION	0	0.74	0.71	0.73	0.735	0.735
		1	0.73	0.76	0.74		
2.	DECISION TREE CLASSIFIER	0	0.89	0.77	0.82	0.836	0.837
		1	0.80	0.90	0.85		
3.	K-NEAREST NEIGHBORS CLASSIFIER	0	0.88	0.72	0.79	0.811	0.812
		1	0.77	0.90	0.83		
4.	RANDOM FOREST CLASSIFIER	0	0.93	0.84	0.88	0.885	0.885
		1	0.85	0.93	0.89		
5.	ANN DEEP MODEL	0	0.80	0.71	0.75	0.767	0.767
		1	0.74	0.82	0.78		

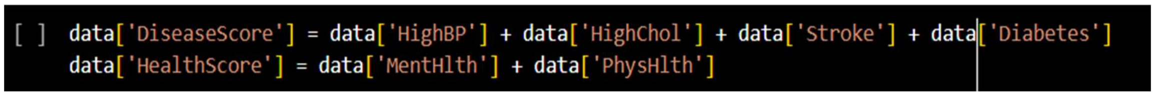
1.6 IMPROVING THE ACCURACY OF THE BEST MODEL

The table presented above illustrates how well the Random Forest Classifier compares to other models across all metrics. For our heart attack prediction problem, Random Forest is the best model. We will therefore attempt to enhance its performance through feature engineering and hyperparameter tuning.

1.6.1 FEATURE CREATION

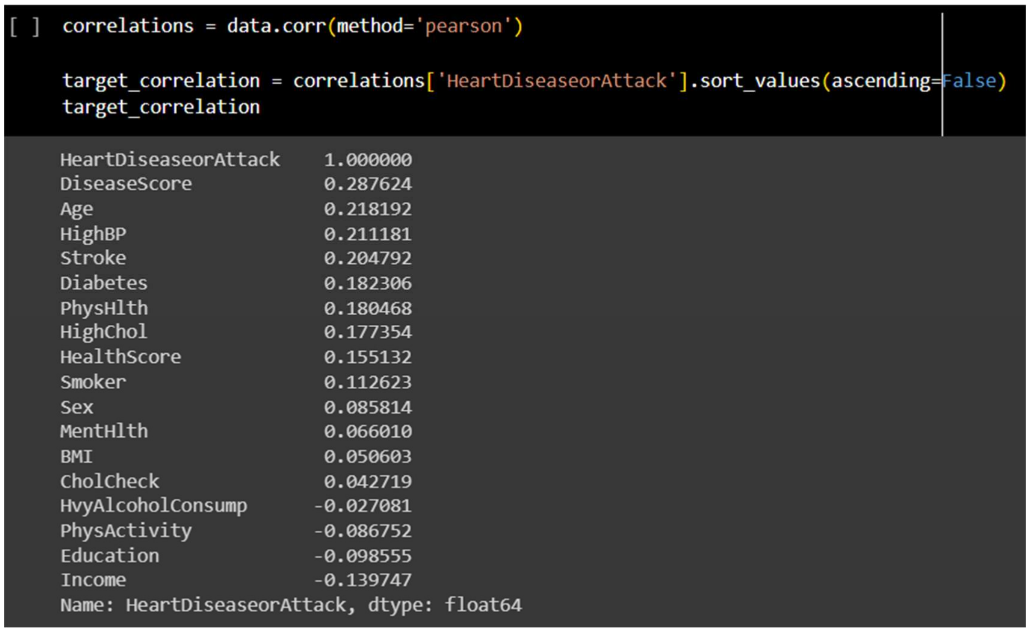
The best way to improve a model's accuracy is to combine existing features to create new ones that can be more helpful for predicting our goal.

To achieve this, two new features have been developed. The first is called DiseaseScore, and it is the outcome of combining the features that represent a person's diseases—Stroke, Diabetes, HighBP, and HighCol. The second one is called HealthScore, and it combines an individual's score for both physical and mental health.



```
[ ]  data['DiseaseScore'] = data['HighBP'] + data['HighChol'] + data['Stroke'] + data['Diabetes']
     data['HealthScore'] = data['MentHlth'] + data['PhysHlth']
```

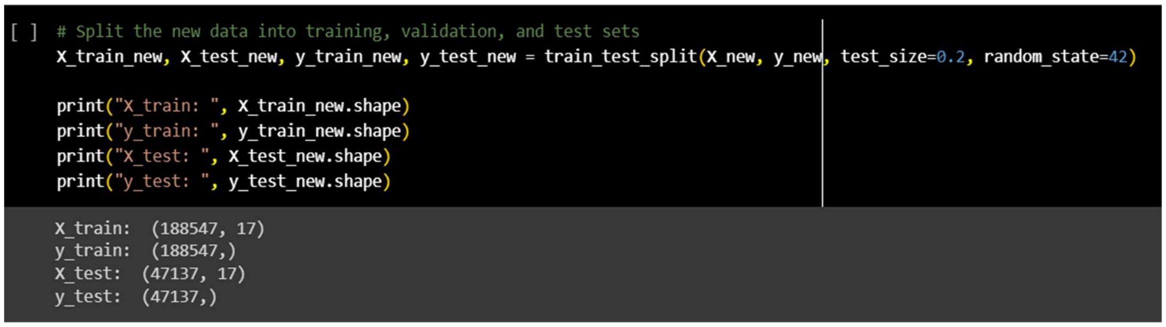
By looking at how these new features correlate with the target, we can determine whether these features are a good fit for our model.



```
[ ]  correlations = data.corr(method='pearson')

     target_correlation = correlations['HeartDiseaseorAttack'].sort_values(ascending=False)
     target_correlation

     HeartDiseaseorAttack    1.000000
     DiseaseScore            0.287624
     Age                     0.218192
     HighBP                  0.211181
     Stroke                  0.204792
     Diabetes                0.182306
     PhysHlth                0.180468
     HighChol                0.177354
     HealthScore             0.155132
     Smoker                  0.112623
     Sex                     0.085814
     MentHlth                0.066010
     BMI                     0.050603
     CholCheck               0.042719
     HvyAlcoholConsump      -0.027081
     PhysActivity           -0.086752
     Education              -0.098555
     Income                 -0.139747
     Name: HeartDiseaseorAttack, dtype: float64
```

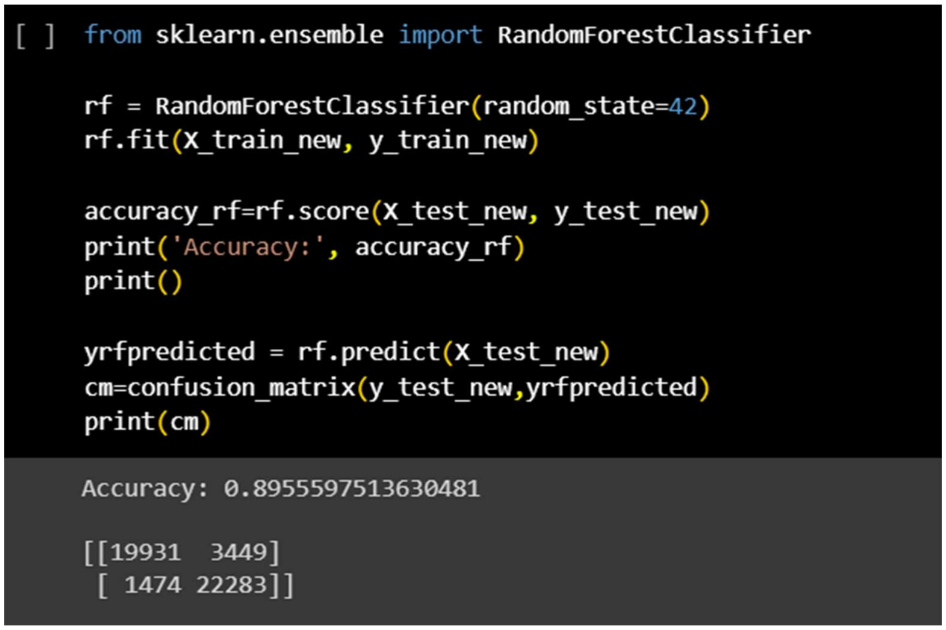
The aforementioned figure illustrates how our recently developed feature, DiseaseScore, has the strongest correlation with the target. Additionally, the HealthScore feature has surpassed features such as Sex, BMI, CholCheck, and Smoker. This suggests that our model will function more effectively with the additional data.

Using our updated data set, let's now train and evaluate the Random Forest Classifier.



```
[ ] # Split the new data into training, validation, and test sets
    X_train_new, X_test_new, y_train_new, y_test_new = train_test_split(X_new, y_new, test_size=0.2, random_state=42)

    print("X_train: ", X_train_new.shape)
    print("y_train: ", y_train_new.shape)
    print("X_test: ", X_test_new.shape)
    print("y_test: ", y_test_new.shape)
```

```
X_train:  (188547, 17)
y_train:  (188547,)
X_test:  (47137, 17)
y_test:  (47137,)
```

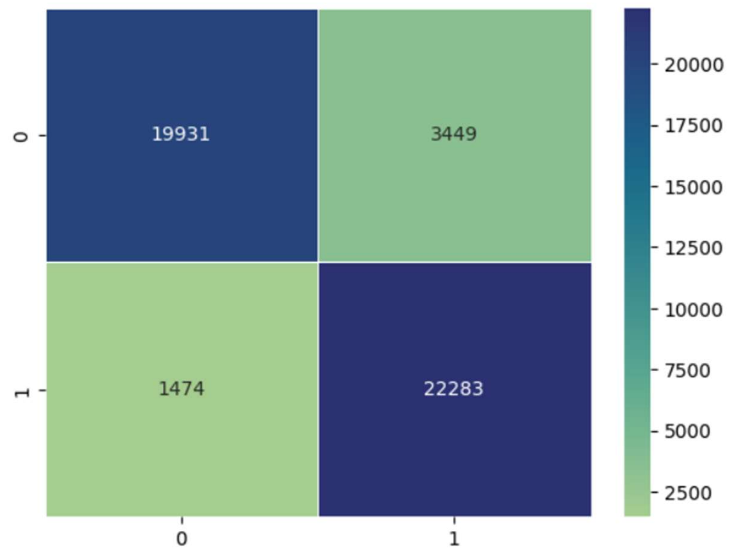


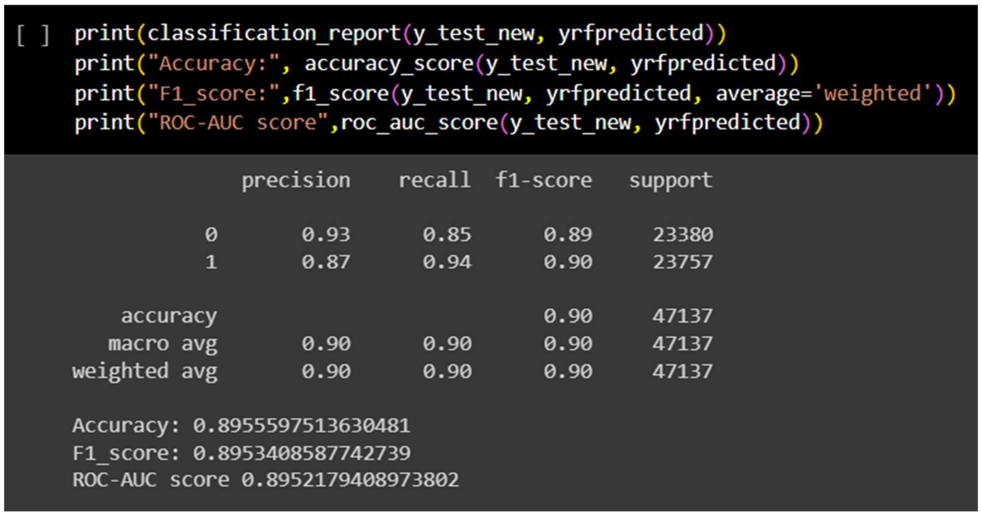
```
[ ] from sklearn.ensemble import RandomForestClassifier

    rf = RandomForestClassifier(random_state=42)
    rf.fit(X_train_new, y_train_new)

    accuracy_rf=rf.score(X_test_new, y_test_new)
    print('Accuracy:', accuracy_rf)
    print()

    yrfpredicted = rf.predict(X_test_new)
    cm=confusion_matrix(y_test_new,yrfpredicted)
    print(cm)
```

```
Accuracy: 0.8955597513630481

[[19931  3449]
 [ 1474 22283]]
```

The Random Forest Classifier, which has been enhanced with two more features, has demonstrated an overall accuracy of 89.55%, surpassing the accuracy of 88.49% of the previous optimal model. Additionally, the report that follows shows how well it performs in terms of class prediction.

CONFUSION MATRIX





```
[ ] print(classification_report(y_test_new, yrfpredicted))
    print("Accuracy:", accuracy_score(y_test_new, yrfpredicted))
    print("F1_score:",f1_score(y_test_new, yrfpredicted, average='weighted'))
    print("ROC-AUC score",roc_auc_score(y_test_new, yrfpredicted))
```

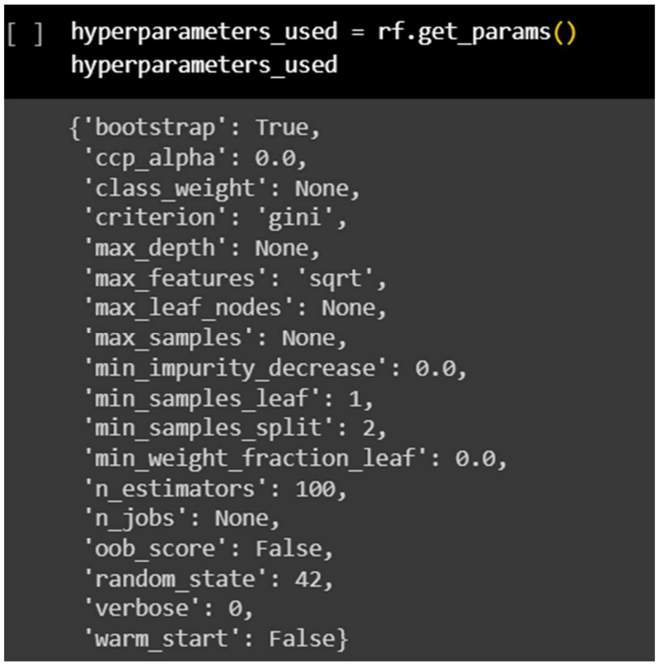
```
              precision    recall  f1-score   support

           0       0.93      0.85      0.89     23380
           1       0.87      0.94      0.90     23757

    accuracy                           0.90     47137
   macro avg       0.90      0.90      0.90     47137
weighted avg       0.90      0.90      0.90     47137

Accuracy: 0.8955597513630481
F1_score: 0.8953408587742739
ROC-AUC score 0.8952179408973802
```

1.6.2 HYPERPARAMETER TUNING

Even though the accuracy of our optimal model has increased, we can still attempt to increase it by selecting the ideal hyperparameter for our model, specifically for this set of data. The optimal hyperparameter combination is found using Grid Search, a hyperparameter tuning technique, given a set of parameters.

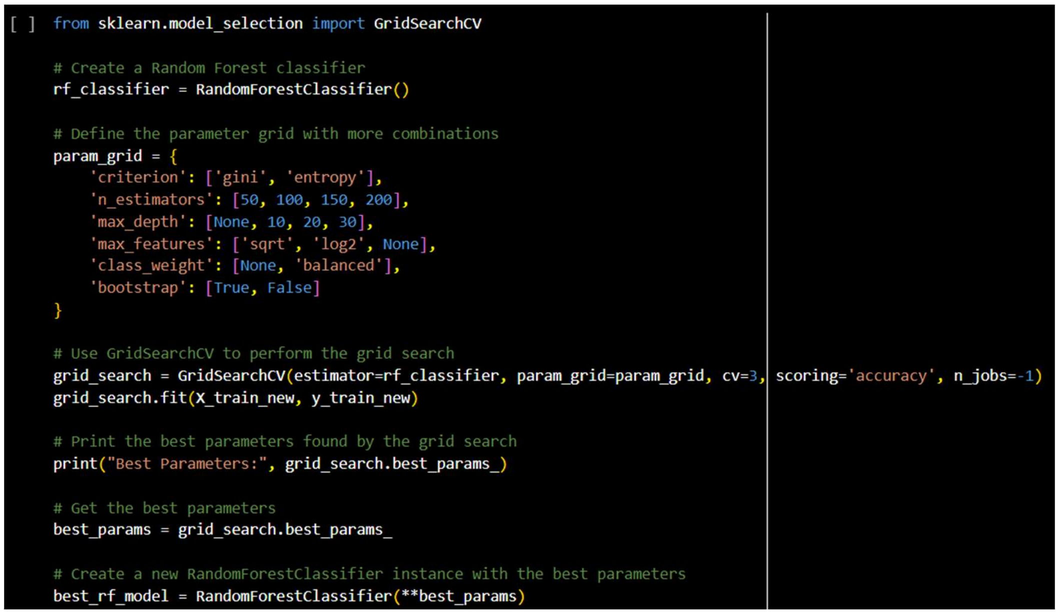
Firstly, we can examine the parameters of our previous optimal model to determine which hyperparameter to use and the range of values for our Grid Search.

HYPERPARAMETERS USED BY THE PREVIOUS RANDOM FOREST MODEL



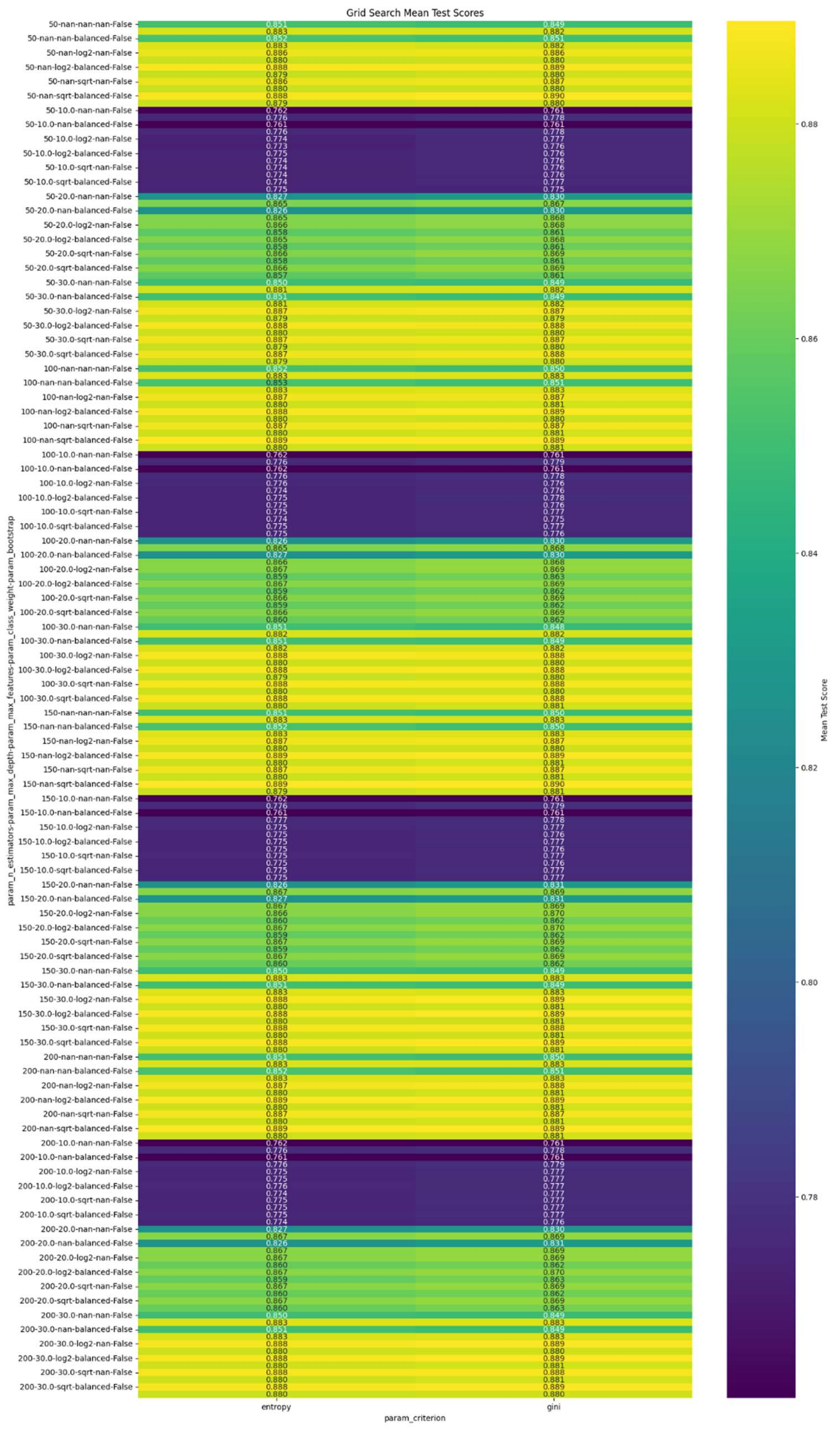
```
[ ] hyperparameters_used = rf.get_params()
    hyperparameters_used

    {'bootstrap': True,
     'ccp_alpha': 0.0,
     'class_weight': None,
     'criterion': 'gini',
     'max_depth': None,
     'max_features': 'sqrt',
     'max_leaf_nodes': None,
     'max_samples': None,
     'min_impurity_decrease': 0.0,
     'min_samples_leaf': 1,
     'min_samples_split': 2,
     'min_weight_fraction_leaf': 0.0,
     'n_estimators': 100,
     'n_jobs': None,
     'oob_score': False,
     'random_state': 42,
     'verbose': 0,
     'warm_start': False}
```

HYPERPARAMETERS USED FOR THE GRID SEARCH

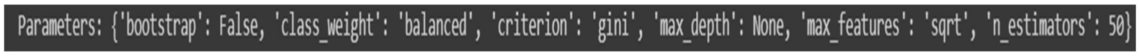


```
[ ] from sklearn.model_selection import GridSearchCV

    # Create a Random Forest classifier
    rf_classifier = RandomForestClassifier()

    # Define the parameter grid with more combinations
    param_grid = {
        'criterion': ['gini', 'entropy'],
        'n_estimators': [50, 100, 150, 200],
        'max_depth': [None, 10, 20, 30],
        'max_features': ['sqrt', 'log2', None],
        'class_weight': [None, 'balanced'],
        'bootstrap': [True, False]
    }

    # Use GridSearchCV to perform the grid search
    grid_search = GridSearchCV(estimator=rf_classifier, param_grid=param_grid, cv=3, scoring='accuracy', n_jobs=-1)
    grid_search.fit(X_train_new, y_train_new)

    # Print the best parameters found by the grid search
    print("Best Parameters:", grid_search.best_params_)

    # Get the best parameters
    best_params = grid_search.best_params_

    # Create a new RandomForestClassifier instance with the best parameters
    best_rf_model = RandomForestClassifier(**best_params)
```

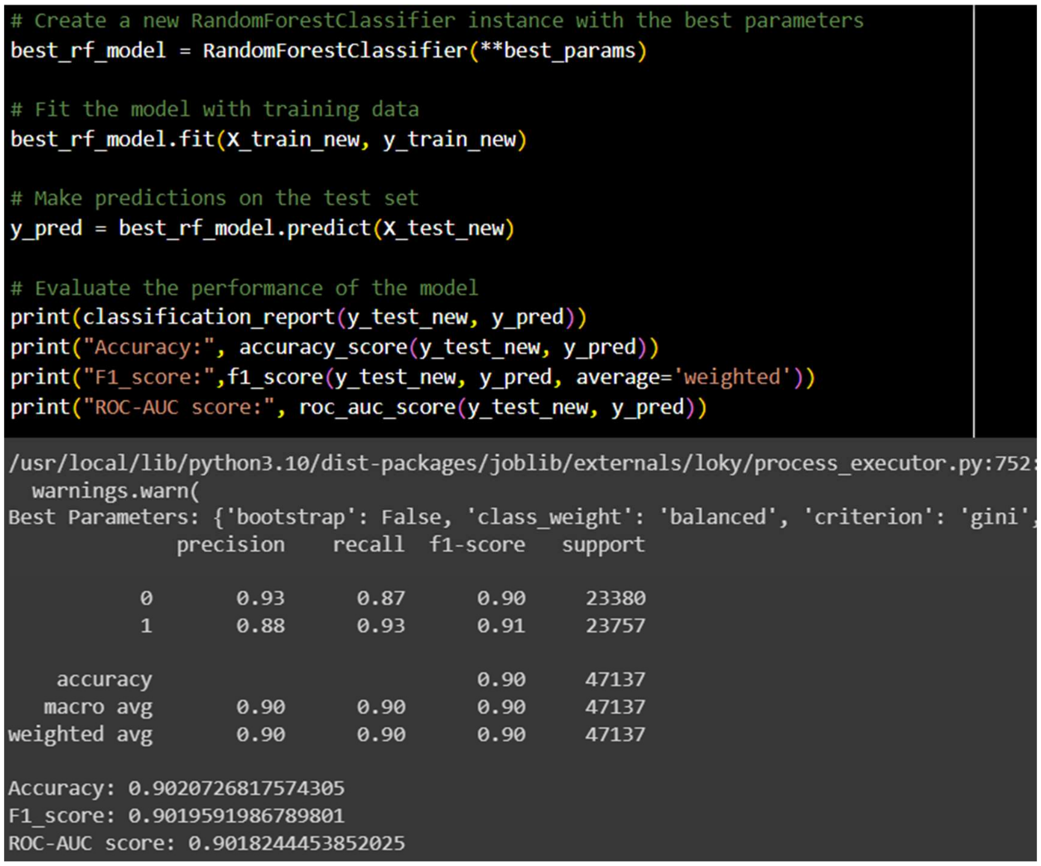
To find the optimal hyperparameter combinations, the grid search now fits our data to the Random Forest Classifier using the hyper parameter combinations of the param_grid displayed in the above figure. The outcome is shown in the section below.



The ideal hyperparameters for our model are as follows.

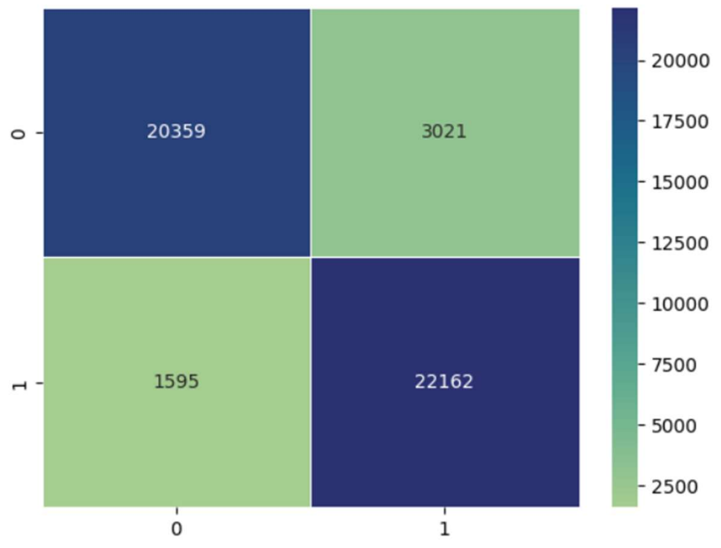


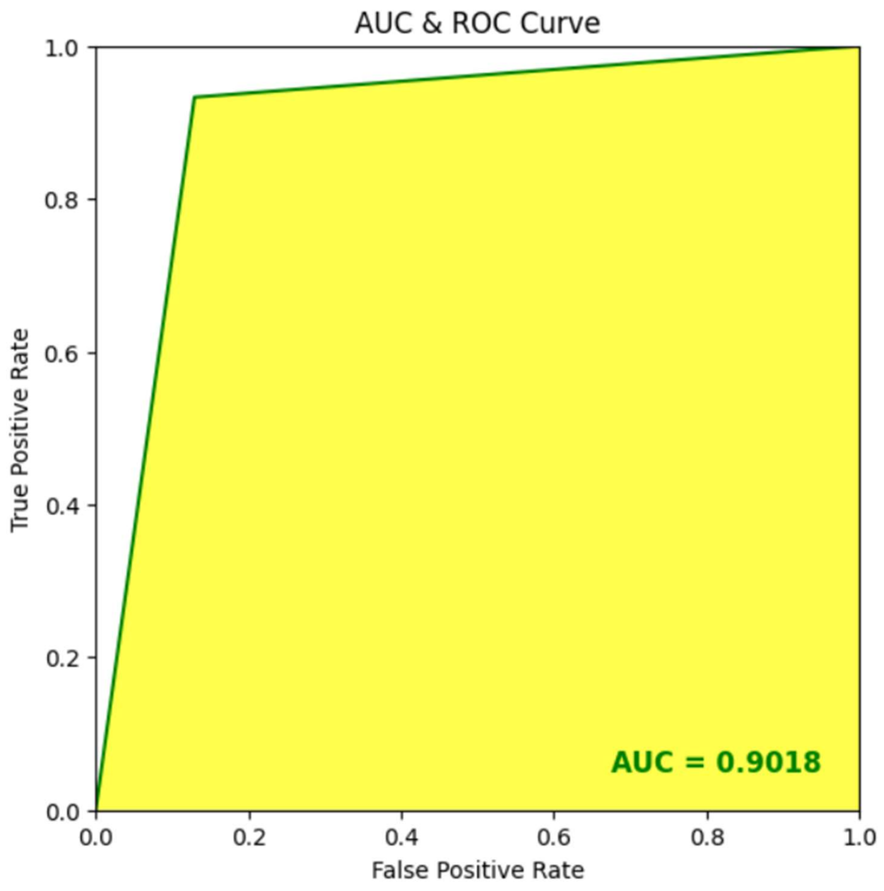
```
Parameters: {'bootstrap': False, 'class_weight': 'balanced', 'criterion': 'gini', 'max_depth': None, 'max_features': 'sqrt', 'n_estimators': 50}
```



```
# Create a new RandomForestClassifier instance with the best parameters
best_rf_model = RandomForestClassifier(**best_params)

# Fit the model with training data
best_rf_model.fit(X_train_new, y_train_new)

# Make predictions on the test set
y_pred = best_rf_model.predict(X_test_new)

# Evaluate the performance of the model
print(classification_report(y_test_new, y_pred))
print("Accuracy:", accuracy_score(y_test_new, y_pred))
print("F1_score:",f1_score(y_test_new, y_pred, average='weighted'))
print("ROC-AUC score:", roc_auc_score(y_test_new, y_pred))
```

```
/usr/local/lib/python3.10/dist-packages/joblib/externals/loky/process_executor.py:752:
  warnings.warn(
Best Parameters: {'bootstrap': False, 'class_weight': 'balanced', 'criterion': 'gini',
              precision    recall  f1-score   support

           0       0.93      0.87      0.90     23380
           1       0.88      0.93      0.91     23757

    accuracy                           0.90     47137
   macro avg       0.90      0.90      0.90     47137
weighted avg       0.90      0.90      0.90     47137

Accuracy: 0.9020726817574305
F1_score: 0.9019591986789801
ROC-AUC score: 0.9018244453852025
```





In conclusion, our Random Forest Classifier has successfully predicted 93% of the positive class samples and 87% of the negative class samples from the actual positive and negative class, yielding an overall accuracy of 90.20%. It also has an AUC-ROC Score of 90.18% and an F1-Score of 90.19%. Consequently, we have developed our final, ideal model that can predict heart attack cases 90% of the time.