Project - Diabetes Readmission Predictor - Report

Objective:

The objective of this analysis is to evaluate the probability of hospital readmission within 30 days of discharge for patients with diabetes, based on various factors such as age, gender, medication, HbA1c levels, and medical and demographic history. The analysis aims to identify which factors are most strongly associated with hospital readmissions and to develop a predictive model that can accurately estimate the probability of readmission for individual patients.

Dataset:

Diabetes 130-US hospitals for years 1999-2008 Dataset from UCI Machine Learning Repository. (source link) The dataset consists of 101766 rows and 50 columns with various quantities that contribute towards diabetes in a person.

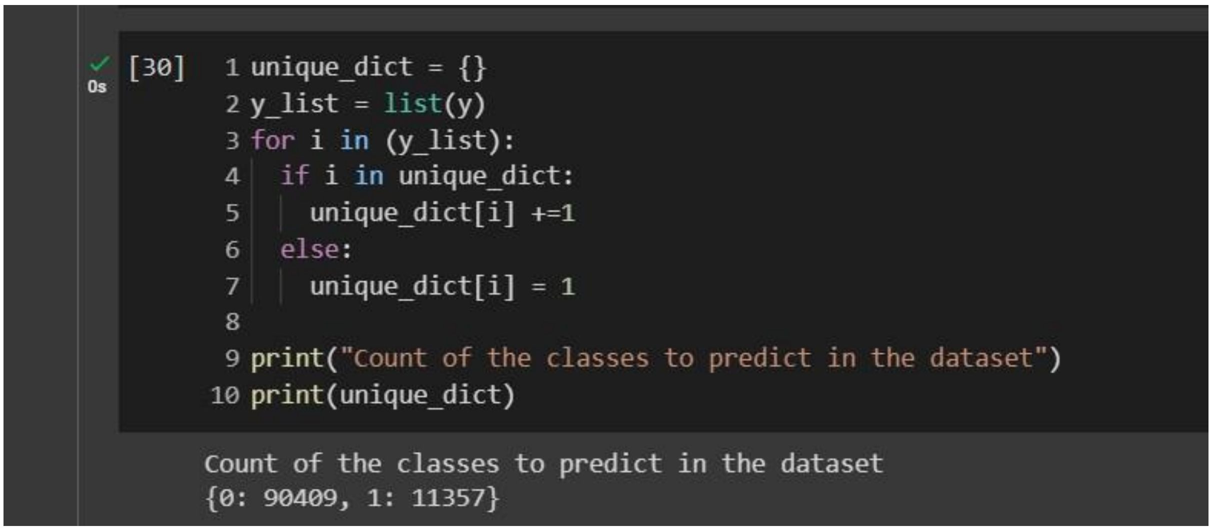
Research Paper:

Impact of HbA1c Measurement on Hospital Readmission Rates: Analysis of 70,000 Clinical Database Patient Records[1] by Beata Strack, Jonathan P. DeShazo,Chris Gennings, Juan L. Olmo, Sebastian Ventura,Krzysztof J. Cios, and John N. Clore published at BioMed Research International. (paper link)

Analysis:

The "Diabetes 130-US hospitals for years 1999-2008" dataset has been analyzed using various machine learning models to predict hospital readmissions for patients with diabetes. The model's effectiveness has been assessed using measures like recall, precision, accuracy, F1 score along with macro and weighted average and it has also been visualized using confusion matrices and ROC curves. Preprocessing and feature engineering techniques have been used to improve model performance. The analysis of this dataset has produced practical insights into predicting hospital readmissions for patients with diabetes using various machine learning models.

Issues with Data:



```
[30]
unique_dict = {}
y_list = list(y)
for i in (y_list):
  if i in unique_dict:
    unique_dict[i] +=1
  else:
    unique_dict[i] = 1

print("Count of the classes to predict in the dataset")
print(unique_dict)
```

```
Count of the classes to predict in the dataset
{0: 90409, 1: 11357}
```

From the above image, we can infer that there is a high imbalance among the two predictive classes, with class 0 containing 90409 values and class 1 containing 11357 values and the approach below is followed to handle the class imbalances.

Algorithm Used:

The algorithms used for predicting hospital readmissions for patients with diabetes include,

- Logistic Regression
- Decision Tree
- Random Forest Classifier
- Naive Bayes
- AdaBoost
- XGBoost

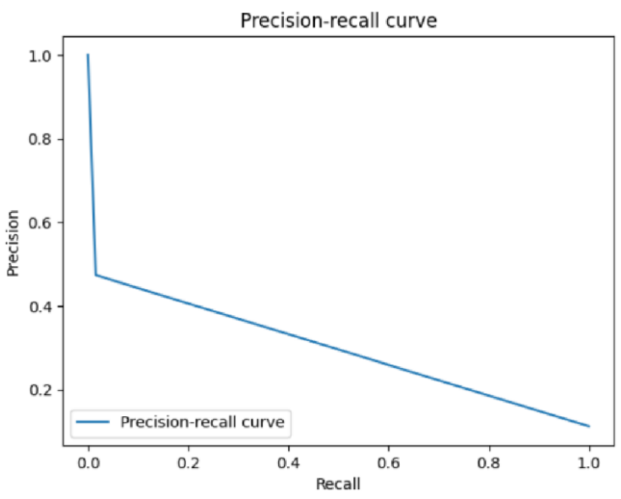
Logistic Regression:

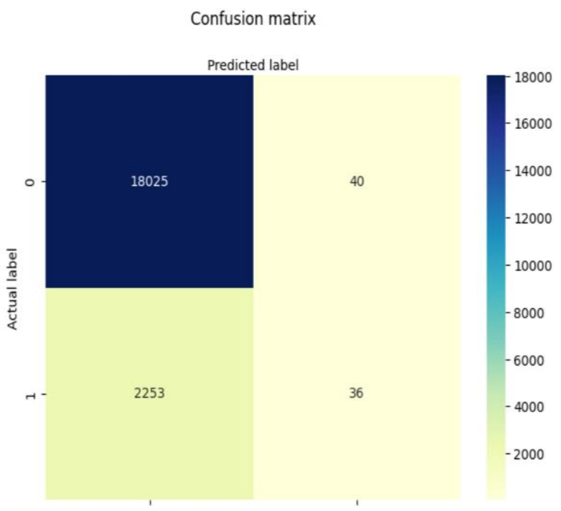
Reason:

Logistic regression was chosen for predicting hospital readmissions for patients with diabetes because it is a simple yet powerful algorithm that is effective for binary classification problems. The "readmitted" variable in the dataset is binary - 0 indicating not readmitted and 1 indicating readmitted in our case, making logistic regression a suitable choice for this problem.

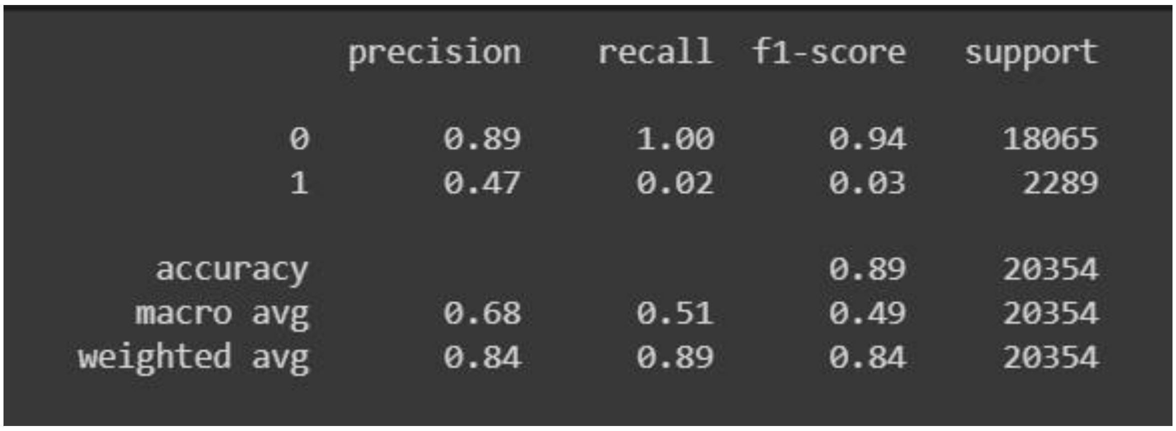
Training and Tuning:

Initially we started by training the model without any fine tuning and by just using the default parameters. From fitting and predicting the model we obtained the following results,





Precision-recall curve and Confusion Matrix



```
              precision    recall  f1-score   support

           0       0.89      1.00      0.94     18065
           1       0.47      0.02      0.03      2289

    accuracy                           0.89     20354
   macro avg       0.68      0.51      0.49     20354
weighted avg       0.84      0.89      0.84     20354
```

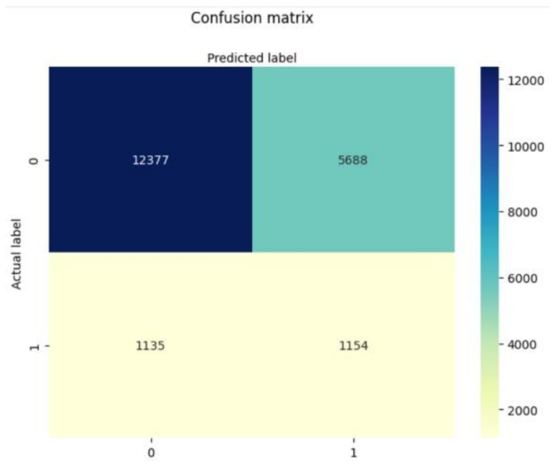
Classification Metrics for Logistic Regression with default parameters

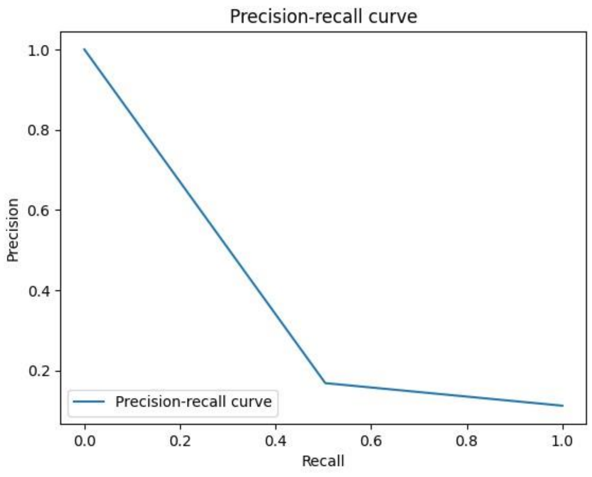
From the above metrics details we can infer that although the value of accuracy is high, the recall value is very less which is more important for a medical dataset, since we do not want anyone to miss a diagnosis i.e, identifying an individual as false negative. From the confusion matrix above, there are very few values correctly identified for class 1.

Fine Tuning:

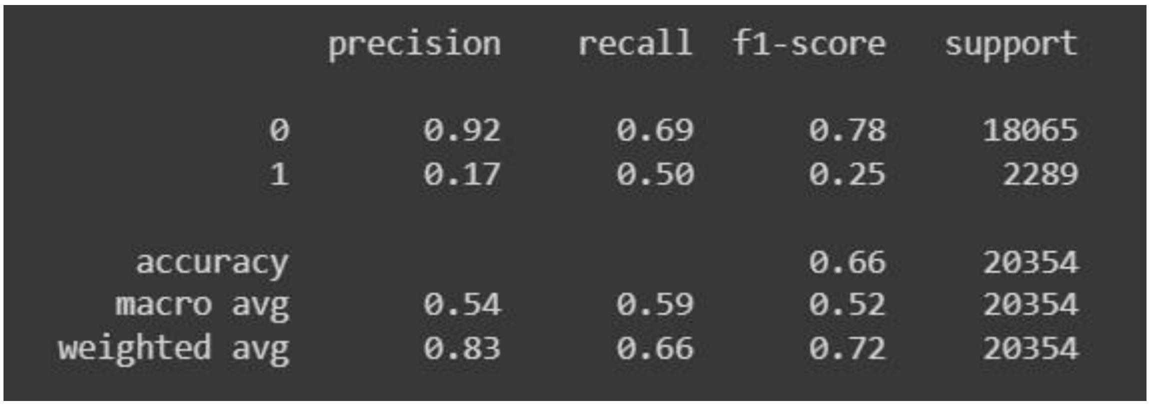
Using solver and balanced weight:

Then, we did it by using the solver 'sag' and class weight 'balanced' which handles the imbalanced data in the dataset. The "solver" parameter specifies the algorithm to use for optimization during the training of the model. The "sag" solver is particularly effective for large datasets with many features and can lead to faster convergence of the optimization. By doing this, the Logistic Regression model's performance on this substantial and intricate dataset can be enhanced. The "class_weight" parameter changes the weights of the classes in the dataset to manage imbalanced data. In this case, the "balanced" class weight is used to give the minority class 1 ("<30") more weight during model training because the "readmitted" variable has more 0 ("NO", and ">30") values than it does 1 ("<30"). This can improve the model's ability to accurately forecast diabetes patients' readmissions. Thus, by accelerating convergence and handling imbalanced data, the "sag" solver and the "balanced" class weight can enhance the performance of the Logistic Regression model on the dataset.





Confusion Matrix and Precision-recall curve



```
              precision    recall  f1-score   support

           0       0.92      0.69      0.78     18065
           1       0.17      0.50      0.25      2289

    accuracy                           0.66     20354
   macro avg       0.54      0.59      0.52     20354
weighted avg       0.83      0.66      0.72     20354
```

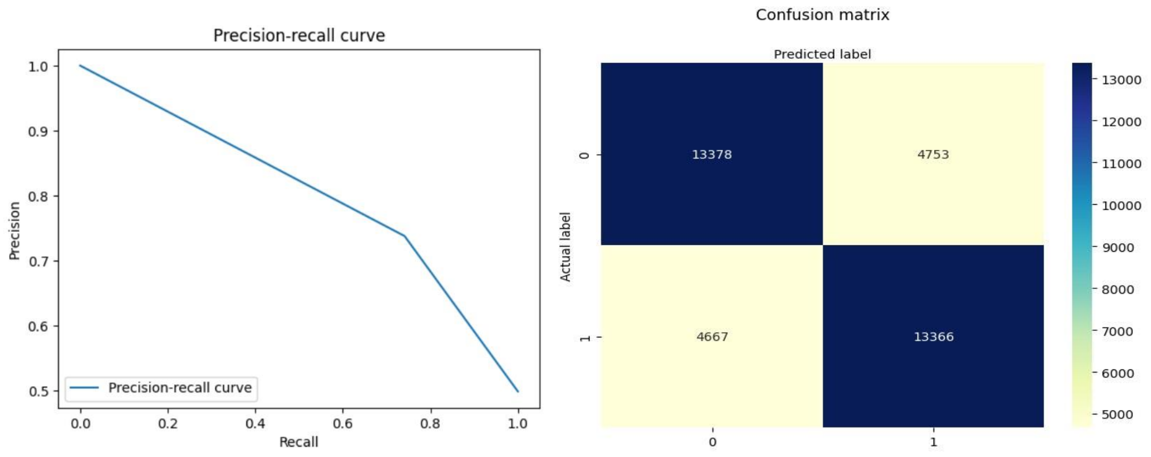
Evaluation Metrics

From the above metrics, we can infer that the recall values have been increased, but there is a decrease in accuracy. We were able to get an accuracy of 0.66 and a recall value of 0.59, although an increase from the previous one, we still tried to fine tune it and increase the values.

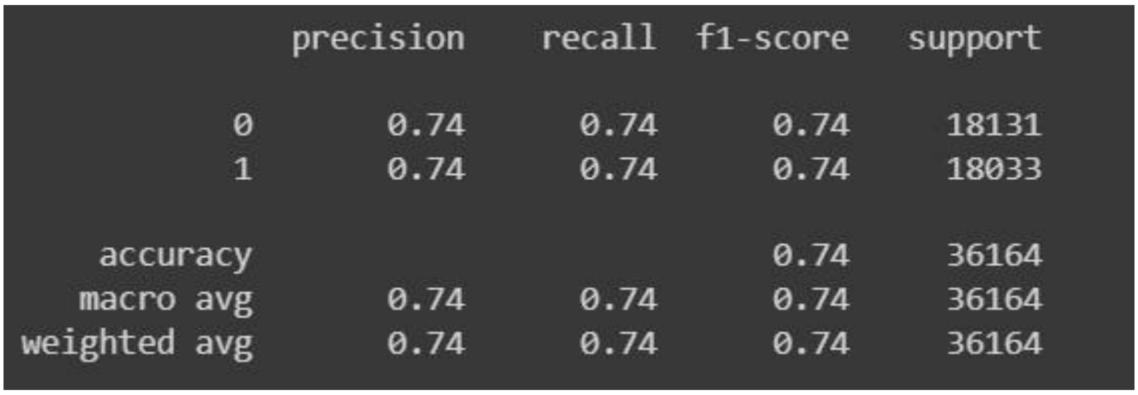
Using SMOTE + weight balancing:

We have also fine tuned the model by further using SMOT. The Logistic Regression model was then retrained using SMOTE. **SMOTE** selects a representative of a minority class and determines its k closest neighbors. Then, by arbitrarily interpolating between the chosen sample and its neighbors, it produces additional samples. Until the required level of oversampling is reached, this process is repeated in order to increase the number of minority class examples and balance the distribution of classes.

We observed that incorporating SMOTE data enhanced the logistic regression model's performance, particularly with regard to recall for the positive class. This shows that SMOTE improved the model's ability to recognize patients who would likely require readmission within 30 days.



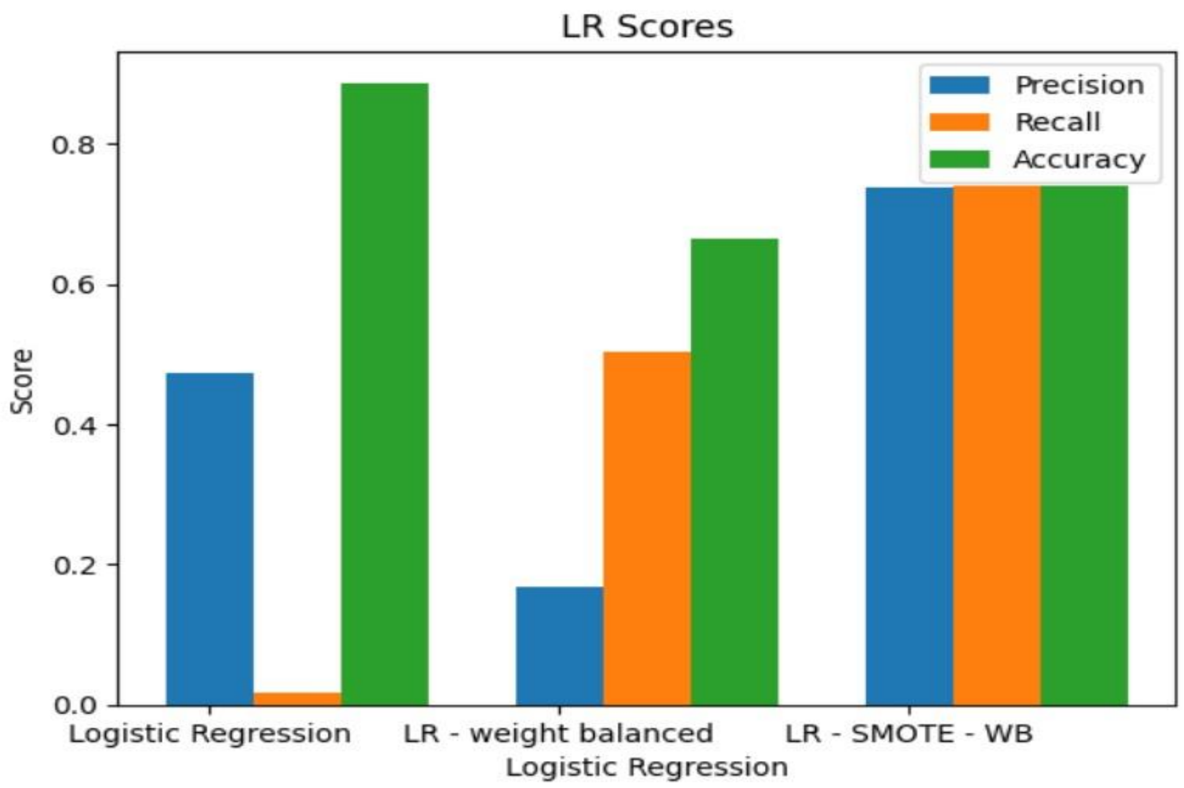
Precision-recall curve and Confusion Matrix



```
              precision    recall  f1-score   support

           0       0.74      0.74      0.74     18131
           1       0.74      0.74      0.74     18033

    accuracy                           0.74     36164
   macro avg       0.74      0.74      0.74     36164
weighted avg       0.74      0.74      0.74     36164
```

Evaluation Metrics

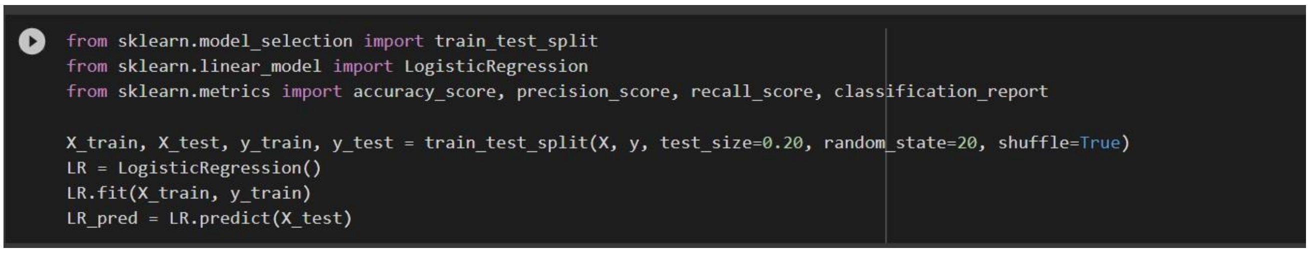
From the above metrics, we can infer that there is a significant increase in the accuracy, precision and recall all valuing 0.74 which shows our model has improved with fine tuning and is able to perform better. From the confusion matrix we can infer that the value of prediction of classes 1 has been increased, the increase in values of the confusion matrix in both the classes is due to the fact of augmentation, where SMOTE increases the data to handle data imbalance.



Overall Comparison

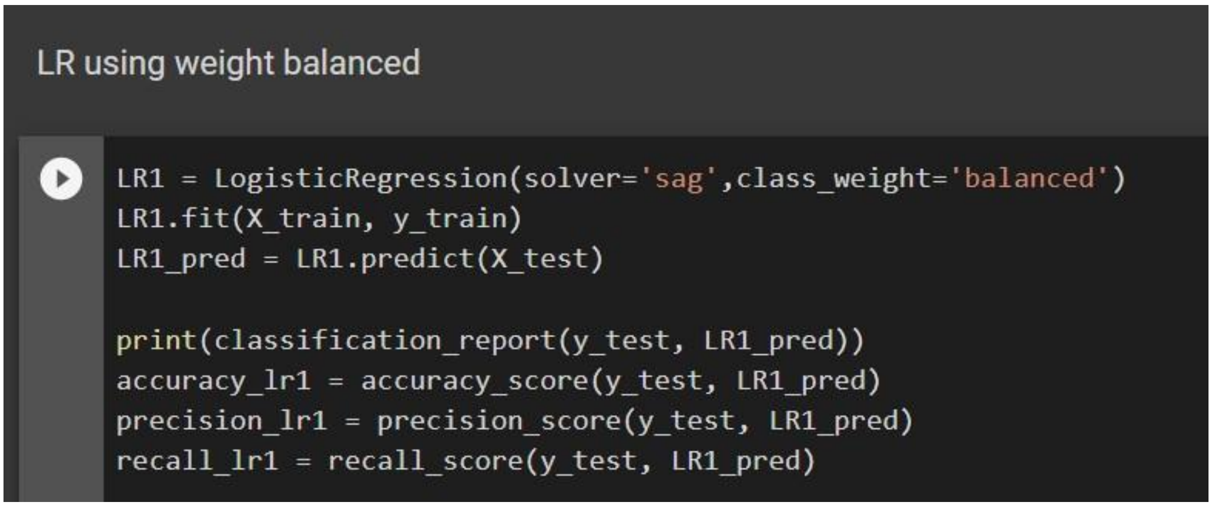
From the above, image the precision, recall and accuracies are considered for class 1, which is more important to us since we have to reduce false negative. The image above displays the overall comparison from the beginning of the model and towards fine tuning, we can see although in the initial base, the accuracy was high with a very less precision and recall values but along with fine tuning it was slowly starting to increase although with a bit trade off from the accuracy. By using SMOTE to handle imbalances, the recall and precision values have been increased to a significant amount. This is much better than having a model that is only biased to a single class and displaying very high accuracy scores. This is where we need to consider other metrics to obtain an overall better performance.

Code Snippets (Logistic Regression):



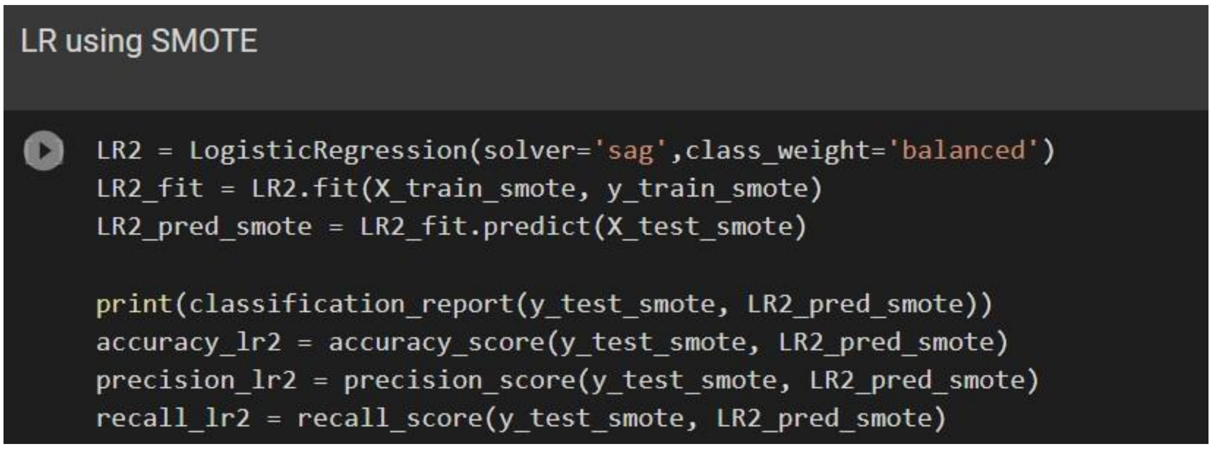
```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score, classification_report

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=20, shuffle=True)
LR = LogisticRegression()
LR.fit(X_train, y_train)
LR_pred = LR.predict(X_test)
```

Logistic Regression implementation


LR using weight balanced

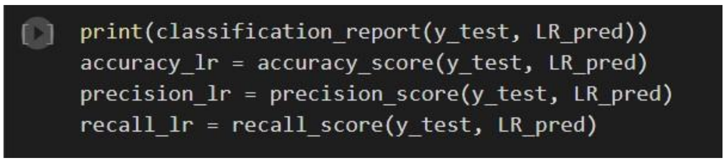
```
LR1 = LogisticRegression(solver='sag',class_weight='balanced')
LR1.fit(X_train, y_train)
LR1_pred = LR1.predict(X_test)

print(classification_report(y_test, LR1_pred))
accuracy_lr1 = accuracy_score(y_test, LR1_pred)
precision_lr1 = precision_score(y_test, LR1_pred)
recall_lr1 = recall_score(y_test, LR1_pred)
```

Logistic Regression implementation with Weight Balanced


LR using SMOTE

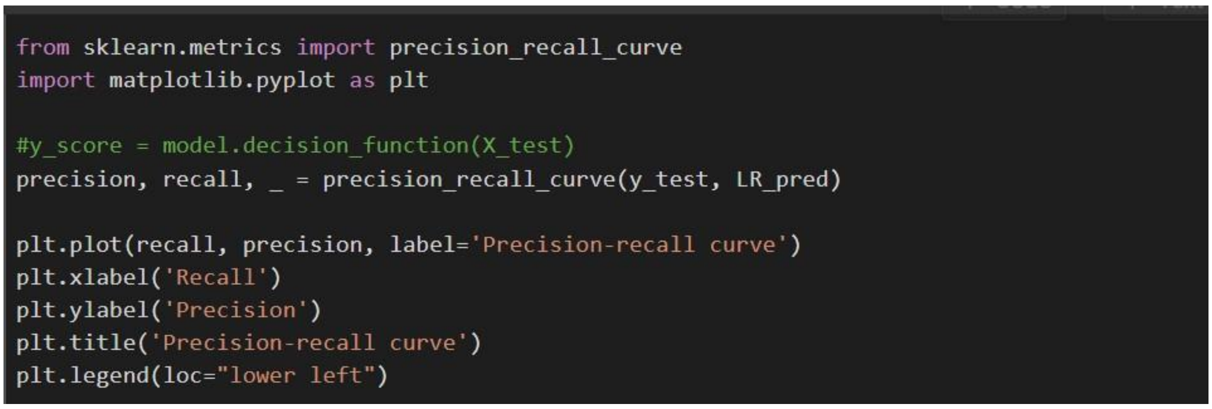
```
LR2 = LogisticRegression(solver='sag',class_weight='balanced')
LR2_fit = LR2.fit(X_train_smote, y_train_smote)
LR2_pred_smote = LR2_fit.predict(X_test_smote)

print(classification_report(y_test_smote, LR2_pred_smote))
accuracy_lr2 = accuracy_score(y_test_smote, LR2_pred_smote)
precision_lr2 = precision_score(y_test_smote, LR2_pred_smote)
recall_lr2 = recall_score(y_test_smote, LR2_pred_smote)
```

Logistic Regression implementation with SMOTE



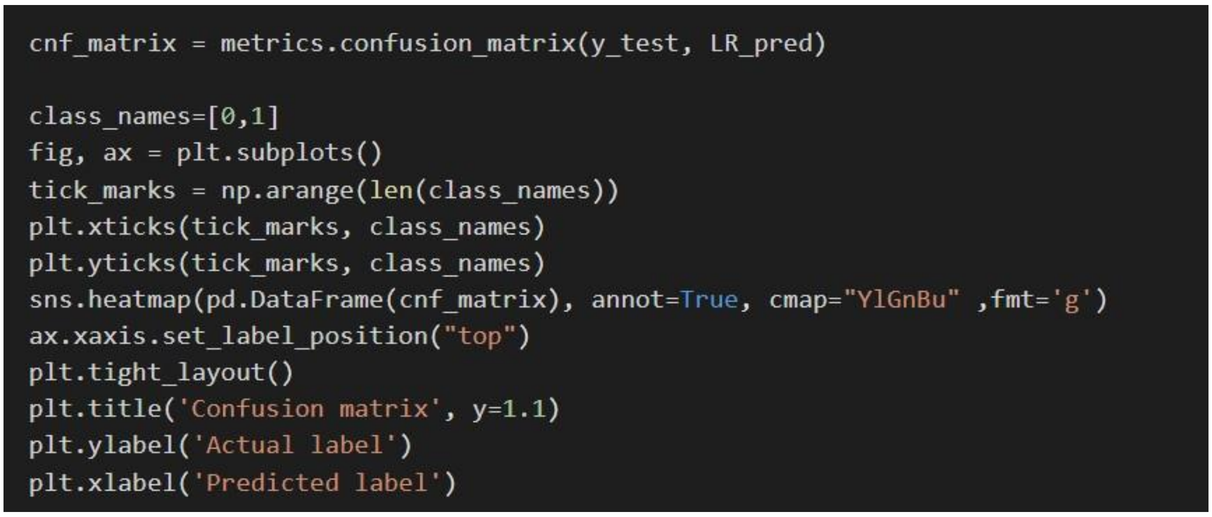
```
print(classification_report(y_test, LR_pred))
accuracy_lr = accuracy_score(y_test, LR_pred)
precision_lr = precision_score(y_test, LR_pred)
recall_lr = recall_score(y_test, LR_pred)
```

Metrics Evaluation



```
from sklearn.metrics import precision_recall_curve
import matplotlib.pyplot as plt

#y_score = model.decision_function(X_test)
precision, recall, _ = precision_recall_curve(y_test, LR_pred)

plt.plot(recall, precision, label='Precision-recall curve')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-recall curve')
plt.legend(loc="lower left")
```

Precision Recall Curve



```
cnf_matrix = metrics.confusion_matrix(y_test, LR_pred)

class_names=[0,1]
fig, ax = plt.subplots()
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names)
plt.yticks(tick_marks, class_names)
sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, cmap="YlGnBu" ,fmt='g')
ax.xaxis.set_label_position("top")
plt.tight_layout()
plt.title('Confusion matrix', y=1.1)
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
```

Confusion Matrix

Decision Tree:

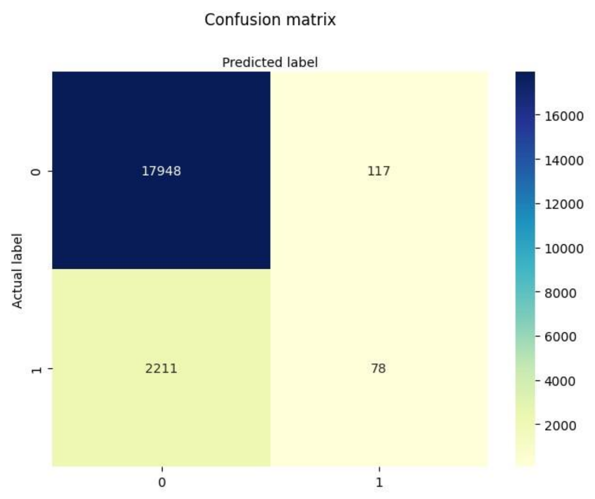
Reason:

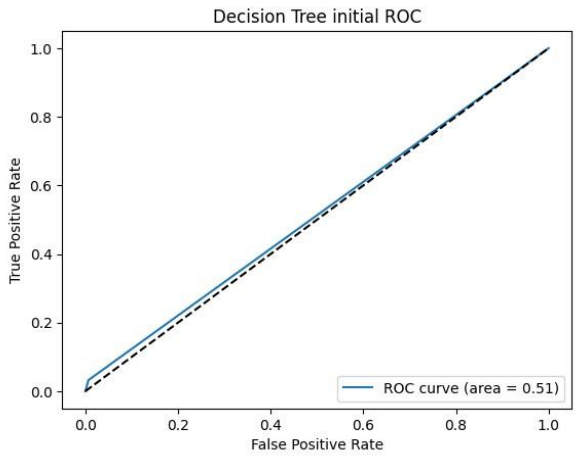
The decision tree method was used to predict hospital readmissions for diabetic patients within 30 days because it can efficiently handle binary classification issues and identify the most significant predictors for readmissions. Decision trees are very simple to analyze and visualize, which can offer important insights into the factors influencing hospital readmissions for diabetic patients. In addition to its simplicity and interpretability, Decision trees are flexible for analyzing datasets with a number of variables because they can handle both numerical and categorical data. For our dataset, which contains a variety of patient attributes and hospital information, this is especially helpful.

Training and Tuning:

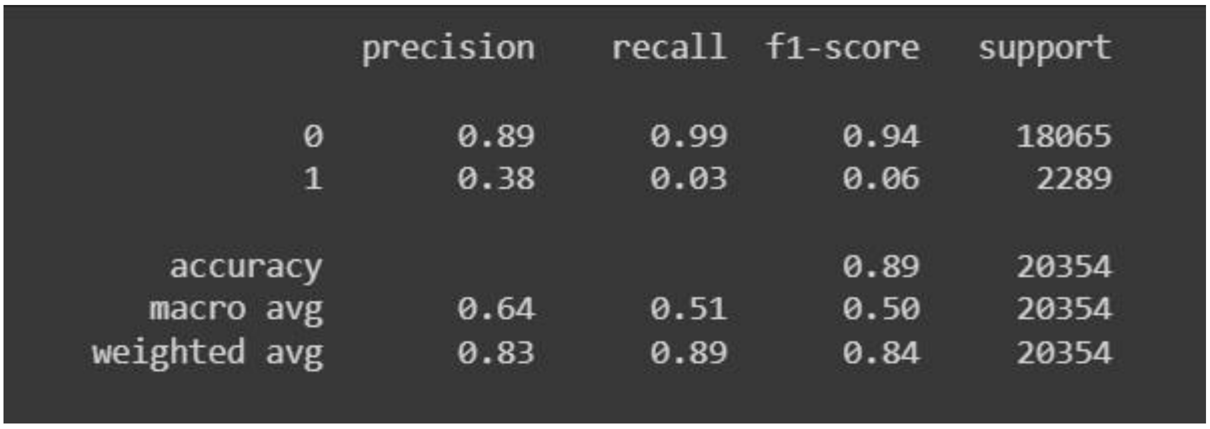
Base Model:

First we initially started with defining a decision tree model of depth 10 and entropy as criterion. After fitting the model and running the prediction the following results were observed,



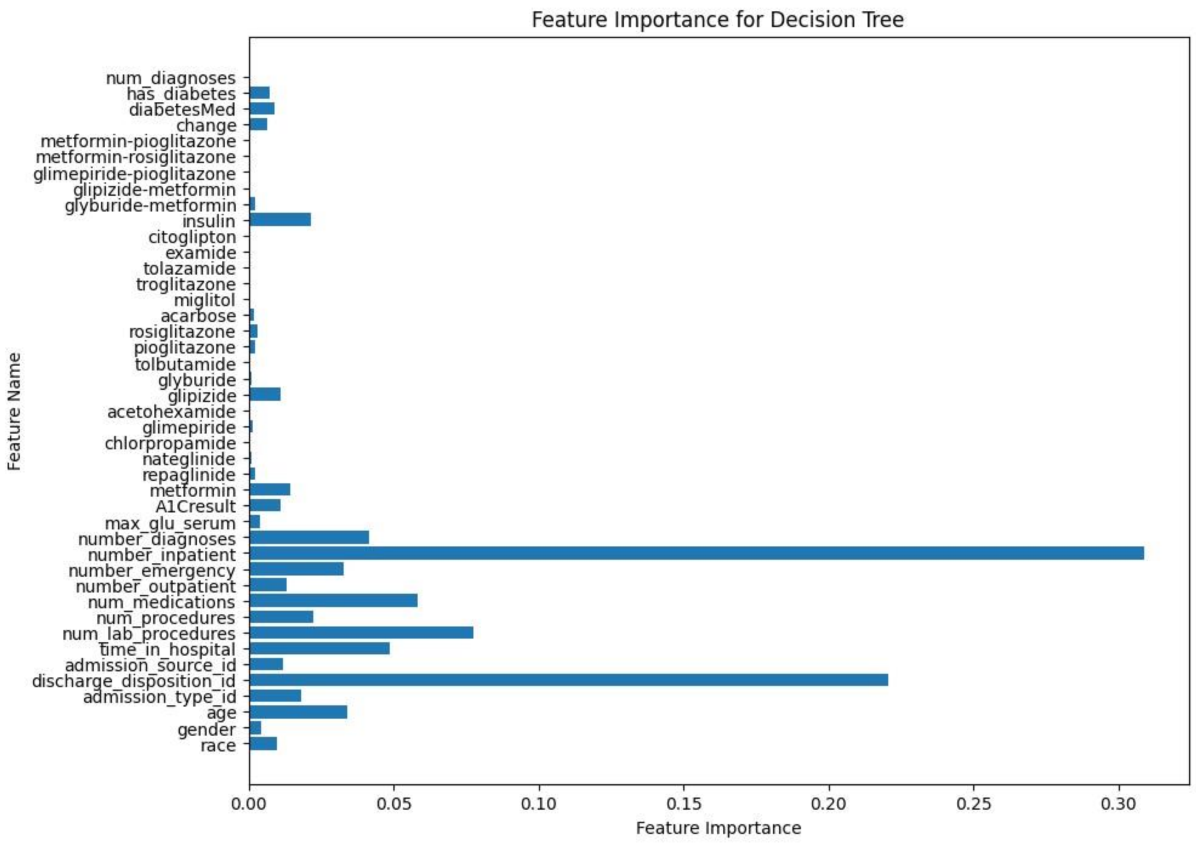


Confusion Matrix and Precision-recall curve



```
              precision    recall  f1-score   support

           0       0.89      0.99      0.94     18065
           1       0.38      0.03      0.06      2289

    accuracy                           0.89     20354
   macro avg       0.64      0.51      0.50     20354
weighted avg       0.83      0.89      0.84     20354
```

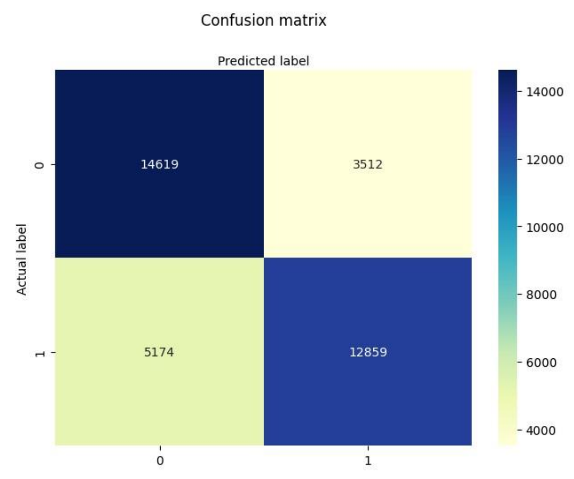
Evaluation metrics

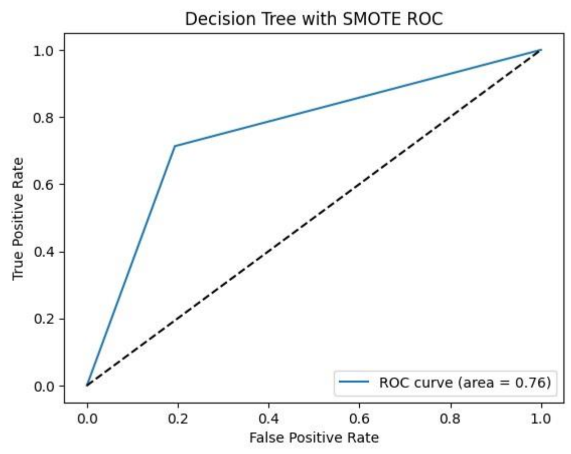


From the above screenshots we can infer that we have high accuracy of 0.89 but very less recall and precision values which is not ideal for the dataset we have chosen for the factors discussed before. From the feature importance map too, we can infer that the model is highly considering only one or two features and many other features like insulin level and diabetes are not considered which are very important. Hence we had to fine tune it to better fit the data.

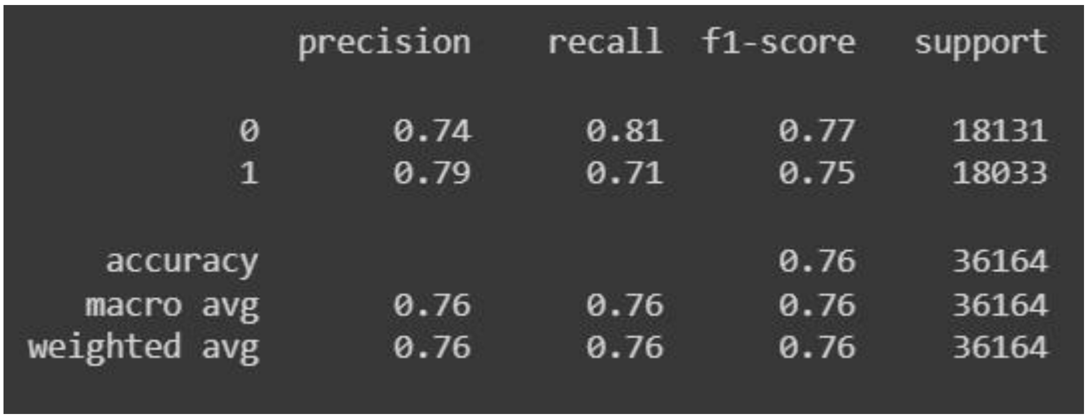
Fine tuning with SMOTE:

We used SMOTE (Synthetic Minority Over-sampling Technique) to balance the dataset's class distribution in order to fine-tune and enhance the model's performance. Since only a small percentage of patients are readmitted within 30 days of their previous discharge in our dataset, the issue of class imbalance arises. This makes it difficult for ML models to accurately predict readmissions. We then trained the Decision tree model again using SMOTE (augment the data) and the same hyperparameters as before.

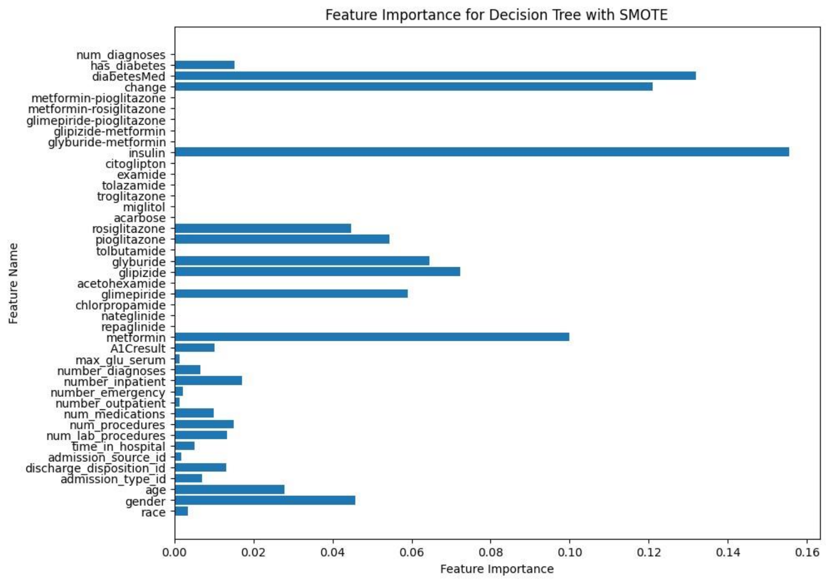




Confusion Matrix and Precision-recall curve


```
              precision    recall  f1-score   support

           0       0.74      0.81      0.77     18131
           1       0.79      0.71      0.75     18033

    accuracy                           0.76     36164
   macro avg       0.76      0.76      0.76     36164
weighted avg       0.76      0.76      0.76     36164
```

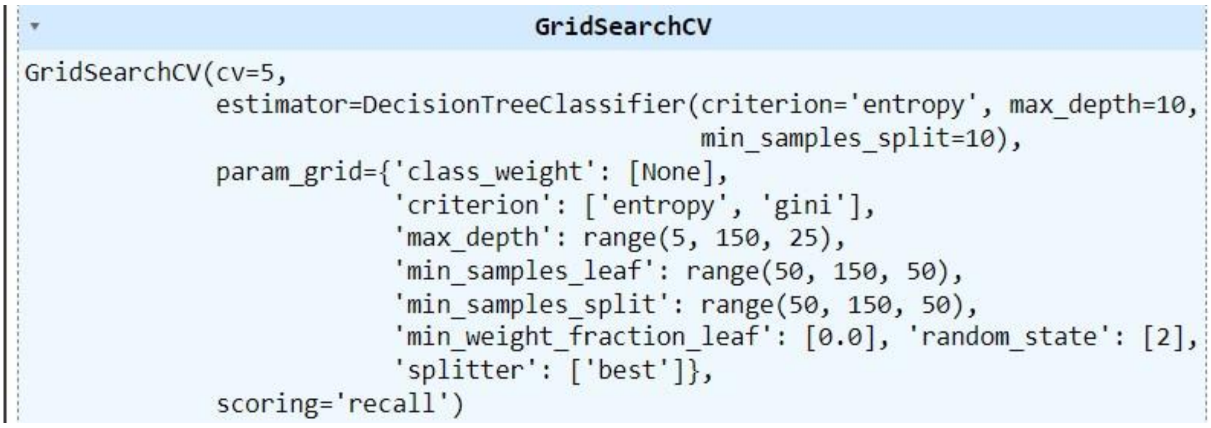
Evaluation Metrics



From the above metrics, we can see that SMOTE has increased the performance of our model and we have achieved better precision (0.74 and 0.79) and recall values (0.81 and 0.71) although there has been a little trade off in the accuracy. This is still better than having a biased model. The confusion matrix displays better classification results and the ROC curve shows a larger area, denoting better performance. The feature importance image shows the involvement of more parameters and also the ones that actually need to be considered, showing our model is performing better than before.

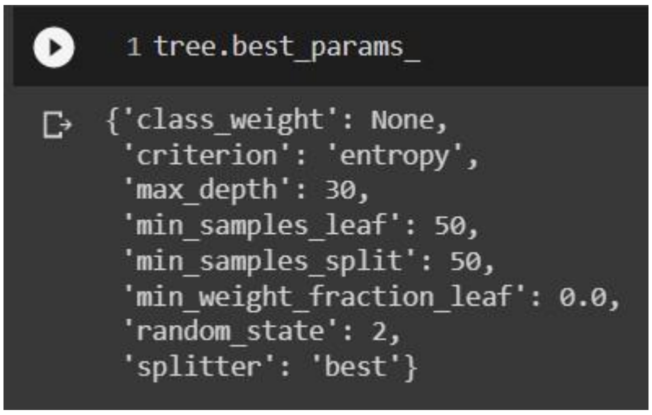
Fine Tuning using estimating best parameters:

To further train the model, we decided to make use of the GridSearchCV function to estimate the best possible parameters for the dataset for our model. These hyperparameters are max_depth, min_samples_leaf, and min_samples_split. The "gini" or "entropy" options for the criteria hyperparameter provide the quality metric that will be used to identify the best split. Reproducibility is ensured by the random_state argument, which sets the random number generator's seed. The use of these hyperparameters in tuning the decision tree model for the dataset can help to improve model performance by reducing the risk of overfitting and improving the evaluation metrics of the model.



```
GridSearchCV
GridSearchCV(cv=5,
             estimator=DecisionTreeClassifier(criterion='entropy', max_depth=10,
                                              min_samples_split=10),
             param_grid={'class_weight': [None],
                         'criterion': ['entropy', 'gini'],
                         'max_depth': range(5, 150, 25),
                         'min_samples_leaf': range(50, 150, 50),
                         'min_samples_split': range(50, 150, 50),
                         'min_weight_fraction_leaf': [0.0], 'random_state': [2],
                         'splitter': ['best']},
             scoring='recall')
```

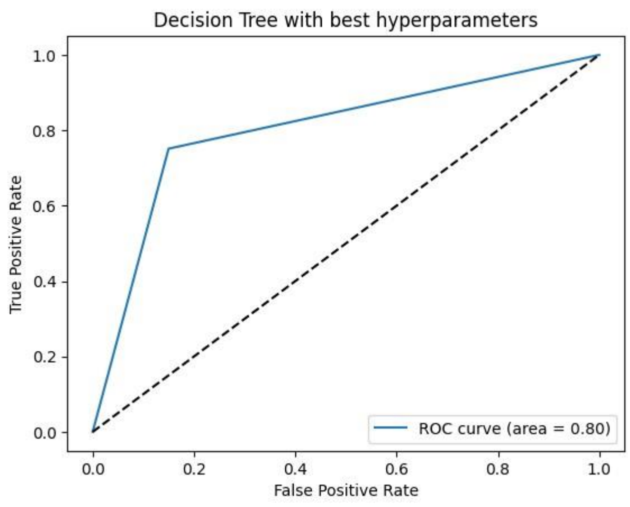
The above image shows us the parameters that we tune in order to find the best possible ones for our model. From the above GridSearch we obtain the below ones as the most optimal. The below values in the image is used as hyperparameter to obtain most efficient
result

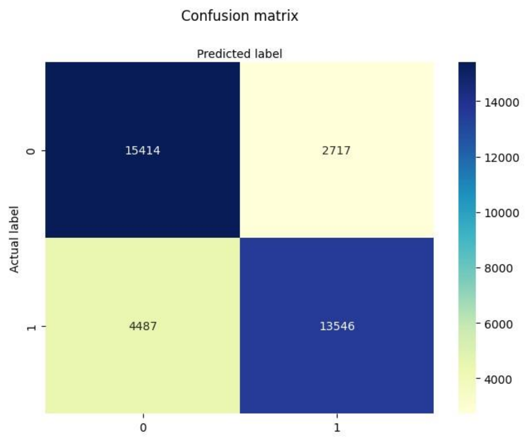


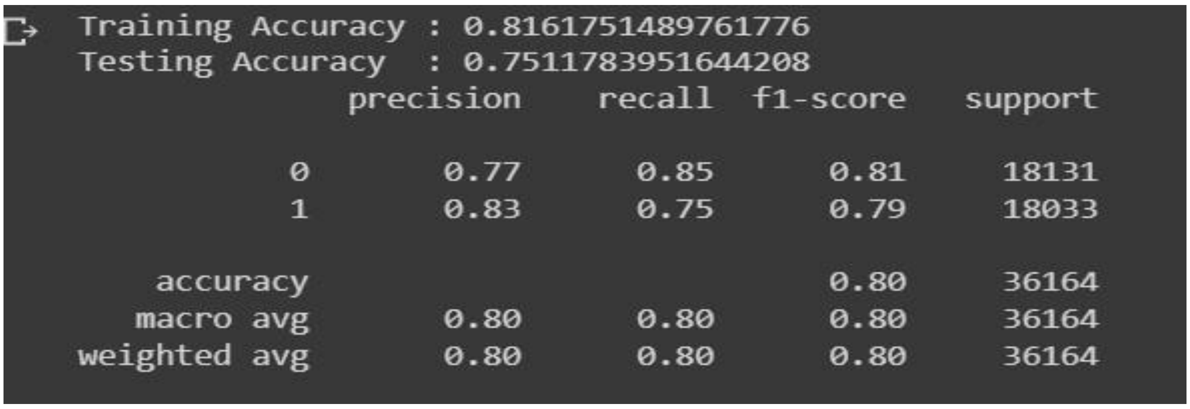
```
tree.best_params_

{'class_weight': None,
 'criterion': 'entropy',
 'max_depth': 30,
 'min_samples_leaf': 50,
 'min_samples_split': 50,
 'min_weight_fraction_leaf': 0.0,
 'random_state': 2,
 'splitter': 'best'}
```

Parameters in Decision Tree

By evaluating with the above parameters, we obtain the following results.

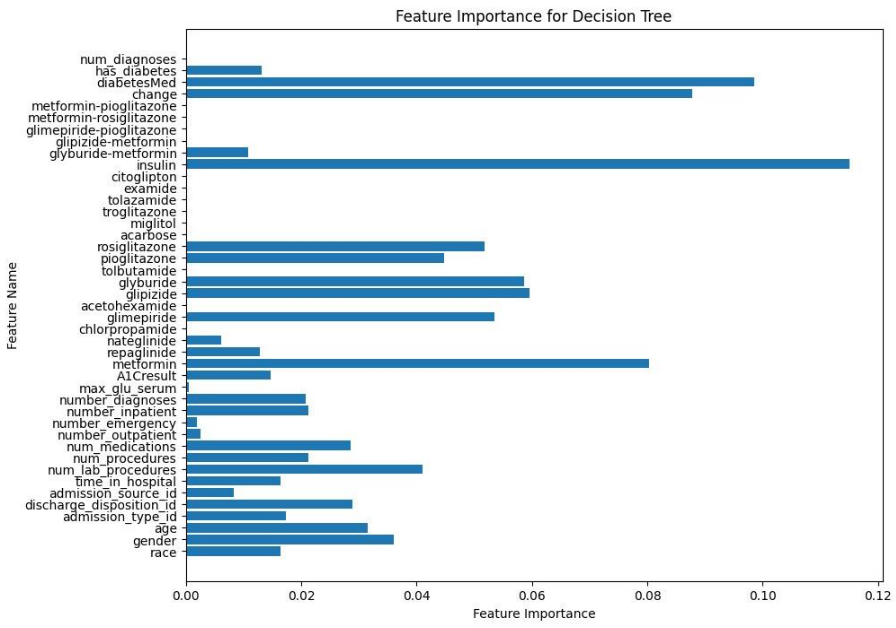






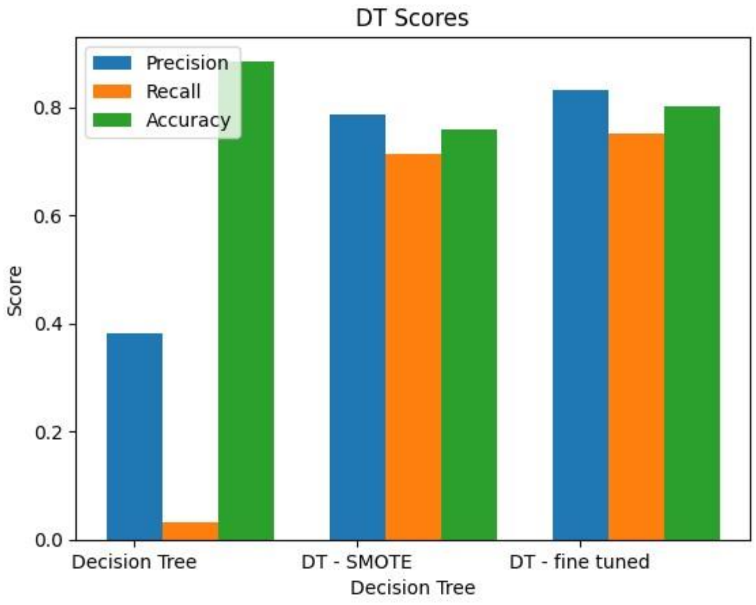
```
Training Accuracy : 0.8161751489761776
Testing Accuracy  : 0.7511783951644208
              precision    recall  f1-score   support

           0       0.77      0.85      0.81     18131
           1       0.83      0.75      0.79     18033

    accuracy                           0.80     36164
   macro avg       0.80      0.80      0.80     36164
weighted avg       0.80      0.80      0.80     36164
```

Evaluation Metrics



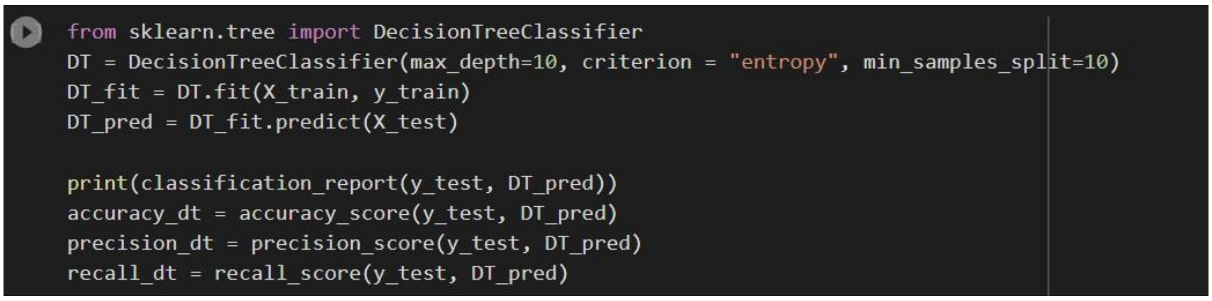
From the above images, we can infer that there is an increase in the accuracy to 0.85 and the area under the curve has also shown more area covered which involves the model performing better. The values of precision and recall have also shown an increase compared to the previous values of (0.74 and 0.79) to 0.77 and 0.83 and improved recall values of (0.81 and 0.71) to 0.85 and 0.75 indicating an overall better performance of our model. Thus GridSearch along with SMOTE provides us the most efficient results of our model. The feature importance image also considers more relevant features as important ones.

Overall Result



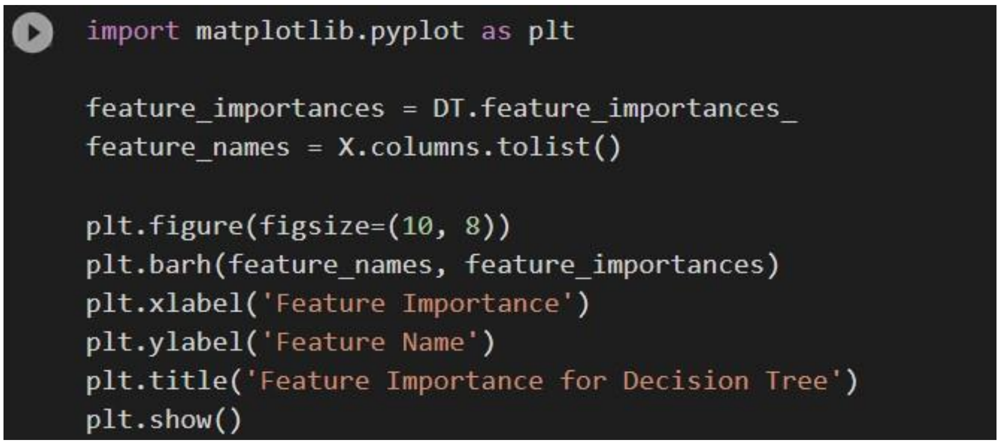
From the above, image the precision, recall and accuracies are considered for class 1, which is more important to us since we have to reduce false negative. The above image gives us a summary of the overall results obtained under each kind of fine tuning. In here DT - fine tuned denotes the Decision tree tuned with GridSearchCV and SMOTE and provides the best results overall. Our problem statement involves classifying the readmission rates and to reduce the number of false negatives and for that high recall value is necessary, which has been achieved through fine tuning.

Code Snippets(Decision Tree):



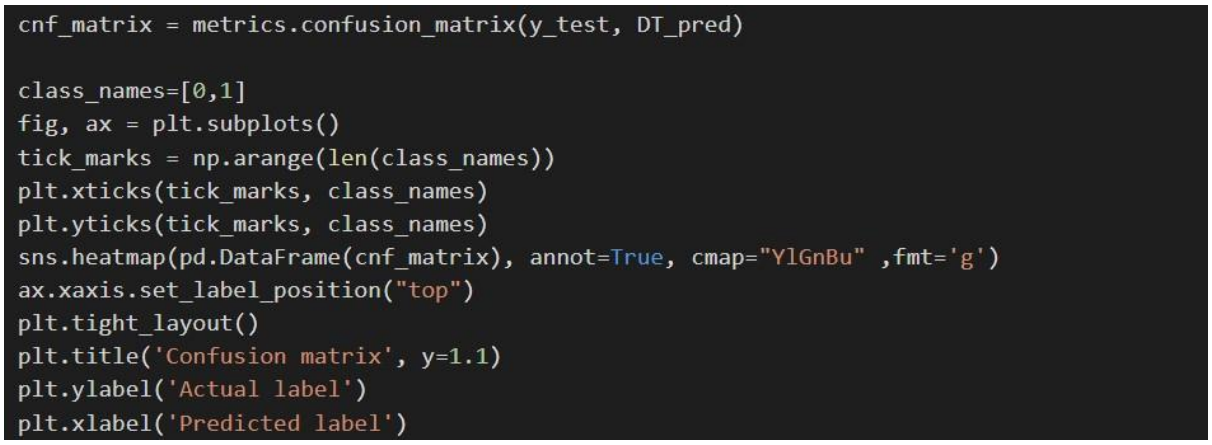
```
from sklearn.tree import DecisionTreeClassifier
DT = DecisionTreeClassifier(max_depth=10, criterion = "entropy", min_samples_split=10)
DT_fit = DT.fit(X_train, y_train)
DT_pred = DT_fit.predict(X_test)

print(classification_report(y_test, DT_pred))
accuracy_dt = accuracy_score(y_test, DT_pred)
precision_dt = precision_score(y_test, DT_pred)
recall_dt = recall_score(y_test, DT_pred)
```

Decision Tree implementation and evaluation



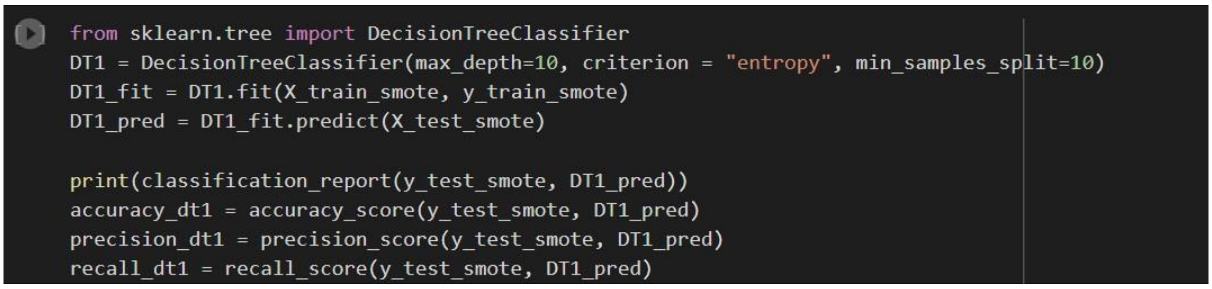
```python
import matplotlib.pyplot as plt

feature_importances = DT.feature_importances_
feature_names = X.columns.tolist()

plt.figure(figsize=(10, 8))
plt.barh(feature_names, feature_importances)
plt.xlabel('Feature Importance')
plt.ylabel('Feature Name')
plt.title('Feature Importance for Decision Tree')
plt.show()
```

Feature Importance for Decision Tree



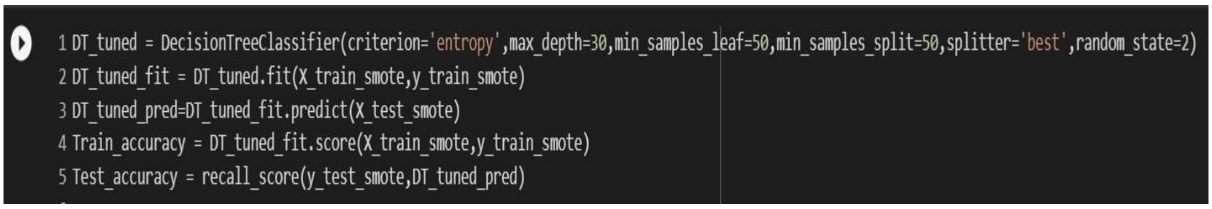
```python
cnf_matrix = metrics.confusion_matrix(y_test, DT_pred)

class_names=[0,1]
fig, ax = plt.subplots()
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names)
plt.yticks(tick_marks, class_names)
sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, cmap="YlGnBu" ,fmt='g')
ax.xaxis.set_label_position("top")
plt.tight_layout()
plt.title('Confusion matrix', y=1.1)
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
```

Confusion Matrix



```python
from sklearn.tree import DecisionTreeClassifier
DT1 = DecisionTreeClassifier(max_depth=10, criterion = "entropy", min_samples_split=10)
DT1_fit = DT1.fit(X_train_smote, y_train_smote)
DT1_pred = DT1_fit.predict(X_test_smote)

print(classification_report(y_test_smote, DT1_pred))
accuracy_dt1 = accuracy_score(y_test_smote, DT1_pred)
precision_dt1 = precision_score(y_test_smote, DT1_pred)
recall_dt1 = recall_score(y_test_smote, DT1_pred)
```

Decision Tree with SMOTE



```python
DT_tuned = DecisionTreeClassifier(criterion='entropy',max_depth=30,min_samples_leaf=50,min_samples_split=50,splitter='best',random_state=2)
DT_tuned_fit = DT_tuned.fit(X_train_smote,y_train_smote)
DT_tuned_pred=DT_tuned_fit.predict(X_test_smote)
Train_accuracy = DT_tuned_fit.score(X_train_smote,y_train_smote)
Test_accuracy = recall_score(y_test_smote,DT_tuned_pred)
```

Decision tree with best hyperparameters + SMOTE

Random Forest Classifier:

Reason:

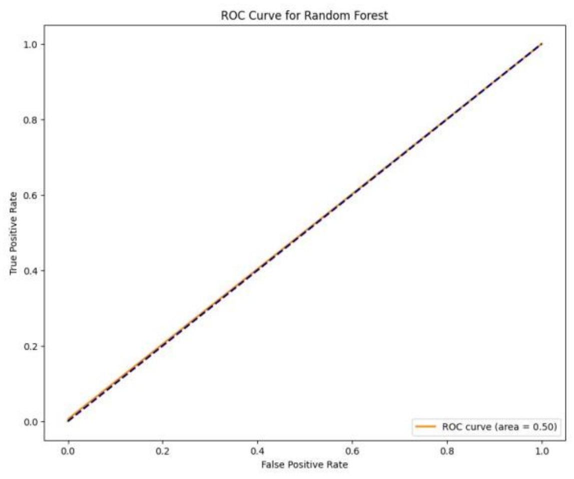
We chose the Random Forest Classifier algorithm for predicting hospital readmissions for patients with diabetes because it has been shown to perform well on classification tasks with complex datasets and high-dimensional feature spaces. The Random Forest Classifier algorithm was a good choice due to the vast amount of patient attributes and hospital characteristics that could possibly contribute to hospital readmissions. Even in the presence of unbalanced class distribution, the system was able to recognize significant predictors of readmissions and produce good predictions.

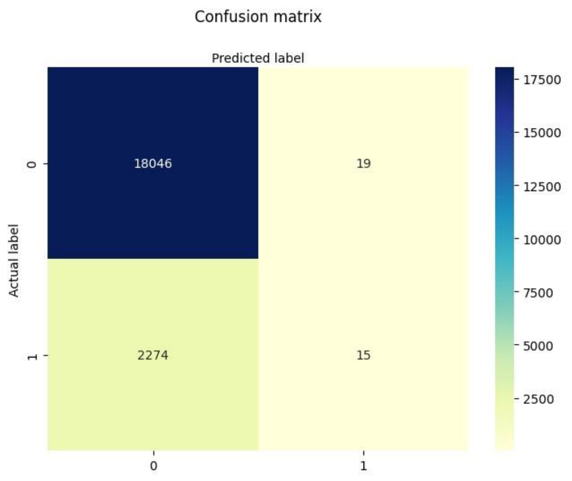
Training and Tuning:

Initial phase:

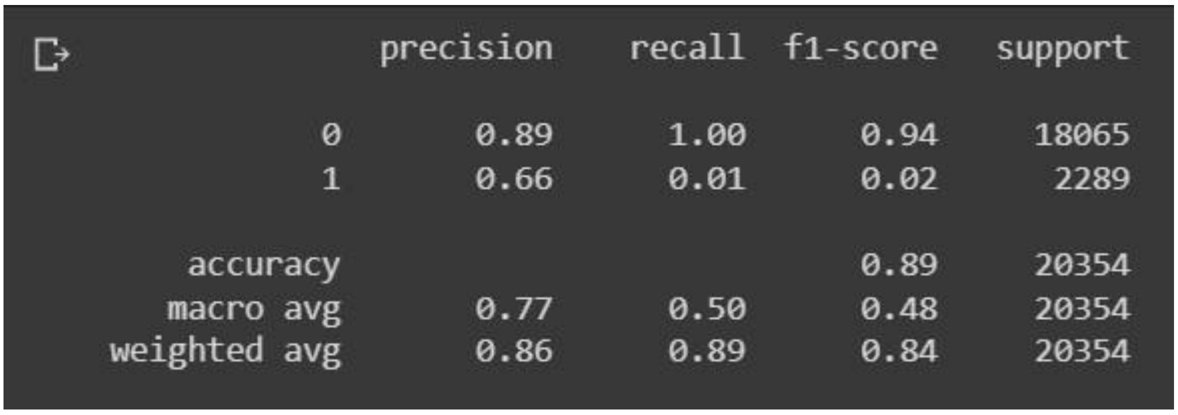
Similar to the Decision Trees above, the Random Forest was implemented in a similar way. First the model was first initialized with the following parameters: n_estimators = 10, max_ depth = 25, criteria = gini, and min_samples_split = 10. These hyperparameters were selected from experimentation on the dataset.

The results obtained from training and fitting the data is given below,





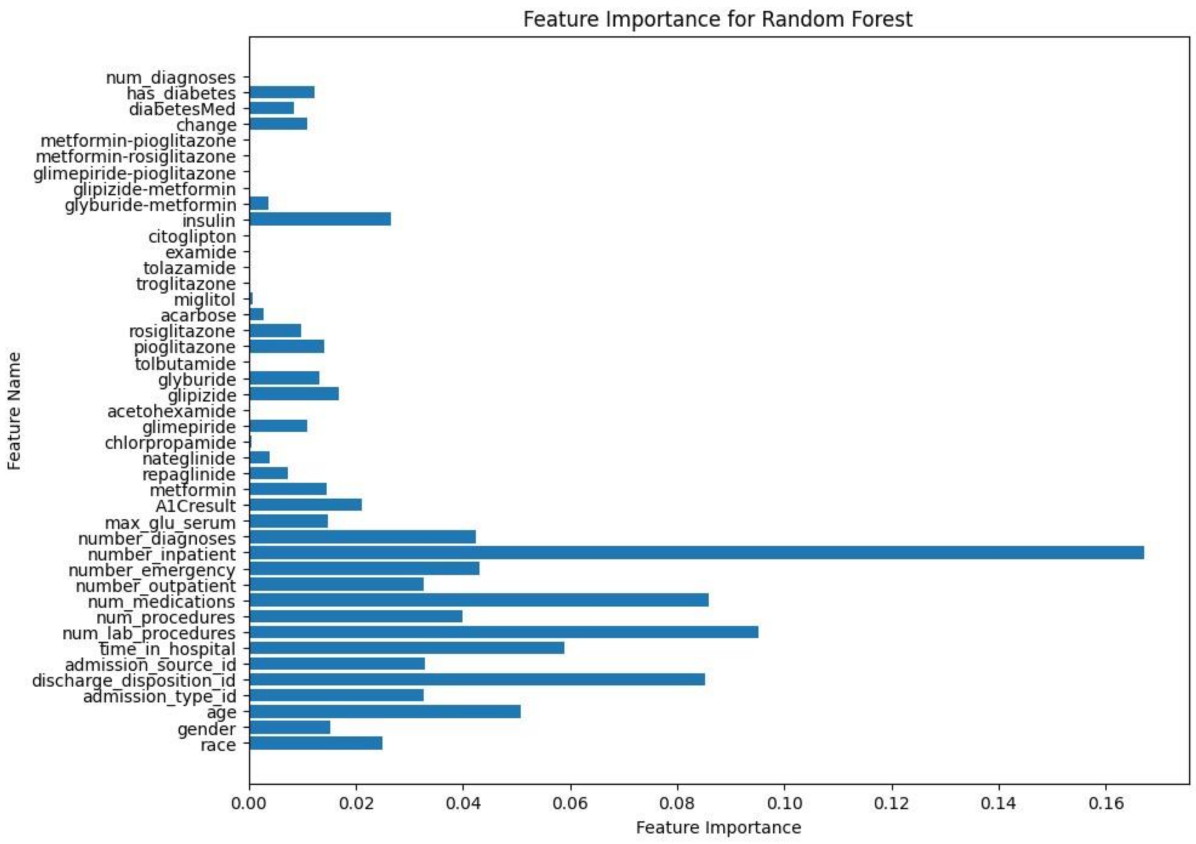
Roc Curve and Confusion Matrix



```
              precision    recall  f1-score   support

           0       0.89      1.00      0.94     18065
           1       0.66      0.01      0.02      2289

    accuracy                           0.89     20354
   macro avg       0.77      0.50      0.48     20354
weighted avg       0.86      0.89      0.84     20354
```

Evaluation Metrics

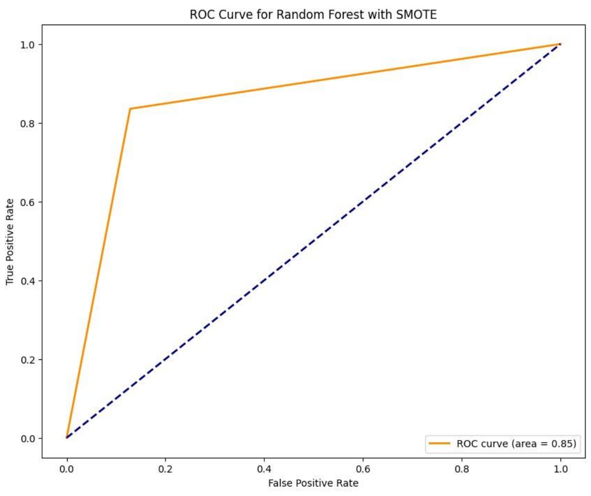
As shown in the previous approaches, the accuracy was high with 0.89 whereas the recall for class 1 was very low with only 0.01 and recall for class 0 was 1.00 due to the large imbalance in the number of classes. The precision values were class 0 - 0.89 and class1 -
0.66.

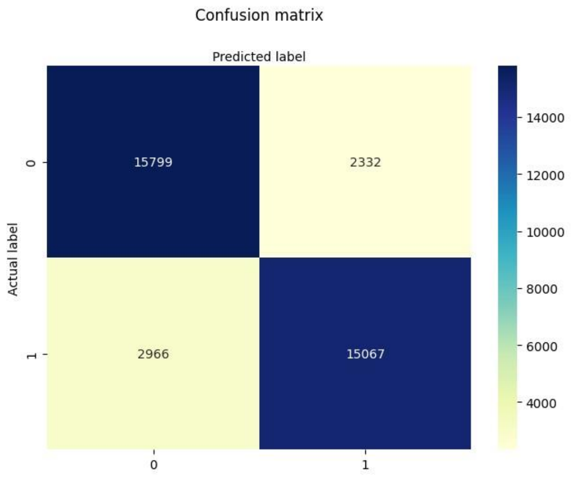


The Feature Importance values also had the issue with most of the values been given importance from very few features that too of relevantly low important than the other features in real time scenario. To handle these imbalances, fine tuning was done.

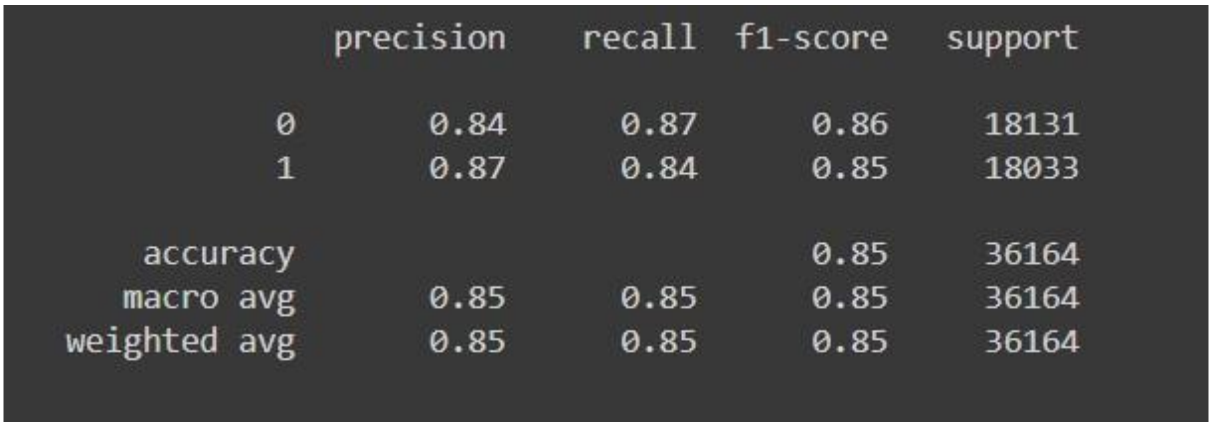
Fine Tuning using SMOTE:

Next, we tried balancing the class distribution using an SMOTE dataset. We created synthetic samples of the readmitted patients using SMOTE, which enhanced the dataset's balance and the model's performance. The results of implementing SMOTE can be observed below,



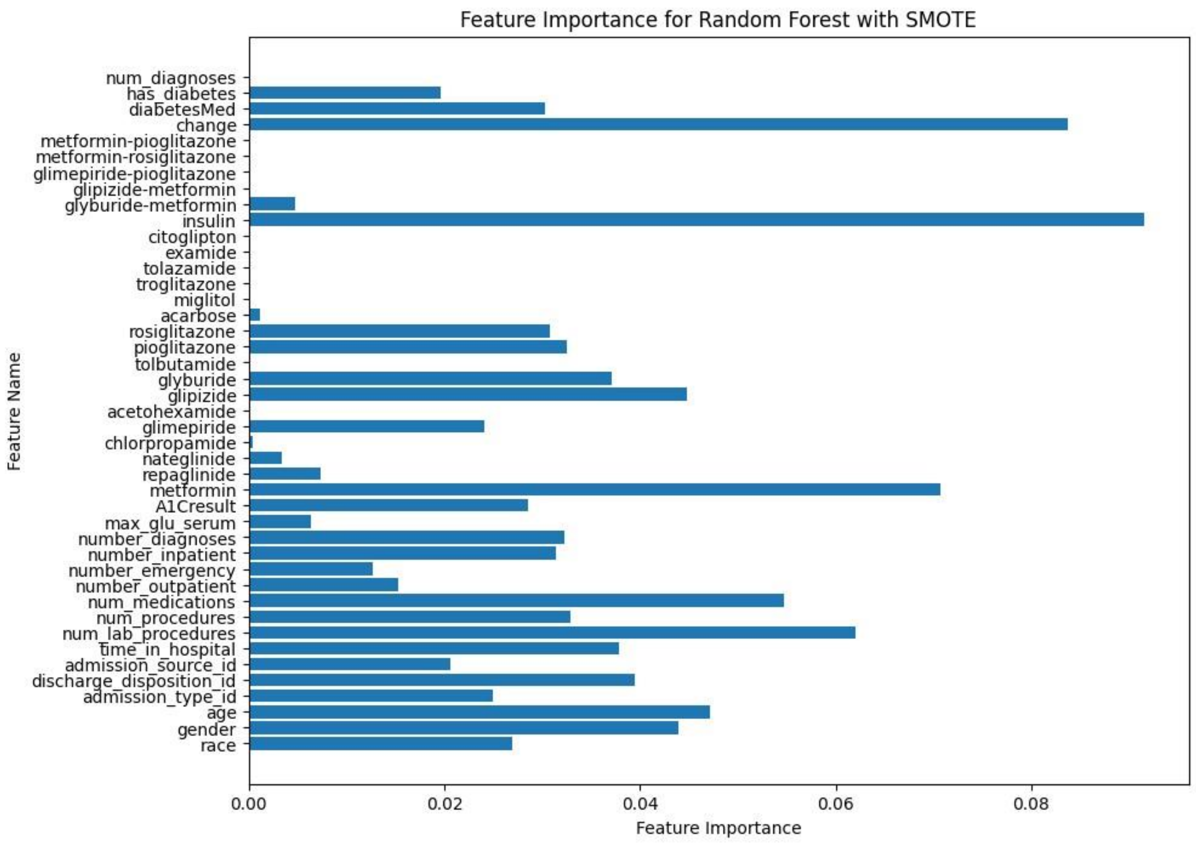


Roc Curve and Confusion Matrix



```
              precision    recall  f1-score   support

           0       0.84      0.87      0.86     18131
           1       0.87      0.84      0.85     18033

    accuracy                           0.85     36164
   macro avg       0.85      0.85      0.85     36164
weighted avg       0.85      0.85      0.85     36164
```

Evaluation Matrix



From the above metrics, we can see that SMOTE has increased the performance of our model and we have achieved better precision from (0.66 and 0.89) to (0.87 to 0.84) and recall values from (0.01 to 1.00) to 0.84 and 0.87 although there has been a little trade off in the accuracy. This is still better than having a biased model. The confusion matrix displays better classification results and the ROC curve shows a larger area, denoting better performance. The feature importance image shows the involvement of more parameters and also the ones that actually need to be considered, showing our model is performing better than before.

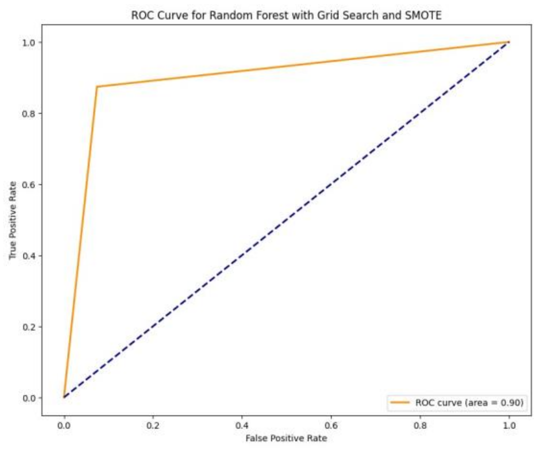
Fine Tuning with GridSearch and SMOTE:

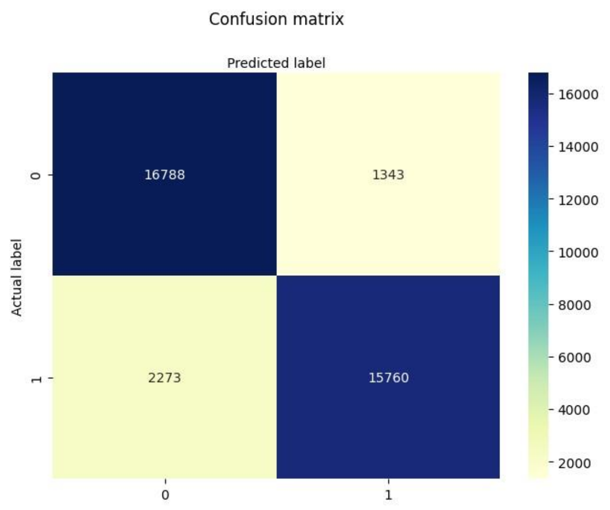
In order to determine the ideal value for n estimators, we next used a grid search. We utilized 3-fold cross-validation to assess the model's performance for each value of n_estimators in the grid search, which covered the range from 1 to 80. The best number was found to be n_estimators=66.

Lastly, we trained a Random Forest Classifier model with the SMOTE dataset and the best hyperparameters identified in the previous steps.

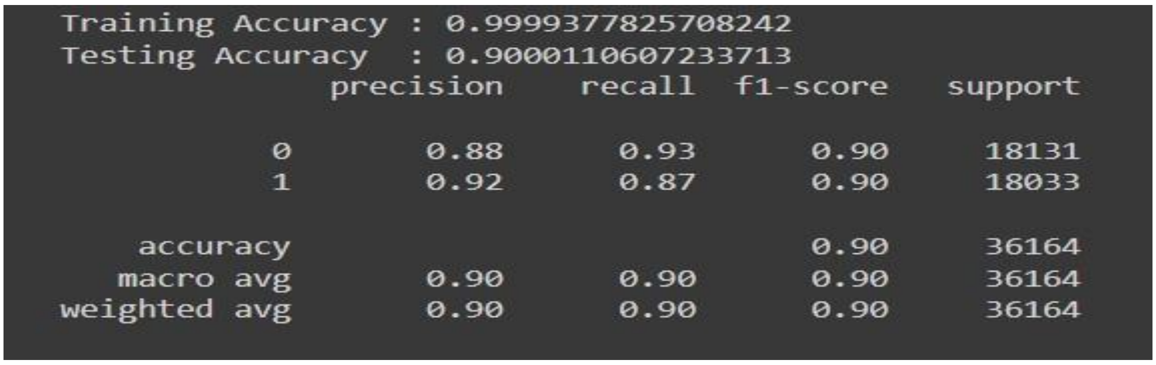
We were able to learn more about hospital readmissions for diabetic patients through this process of fine-tuning and training the Random Forest Classifier model. Even though the Random Forest Classifier model produced some encouraging predictions for hospital readmissions, we can still improve model performance by focusing exclusively on this model.

The results are shown below,



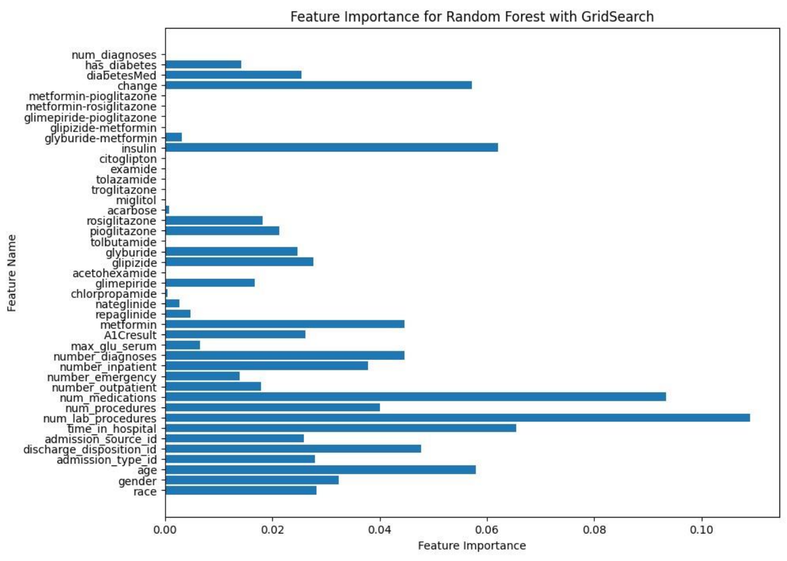


Roc curve and Confusion Matrix



```
Training Accuracy : 0.9999377825708242
Testing Accuracy  : 0.9000110607233713
              precision    recall  f1-score   support

           0       0.88      0.93      0.90     18131
           1       0.92      0.87      0.90     18033

    accuracy                           0.90     36164
   macro avg       0.90      0.90      0.90     36164
weighted avg       0.90      0.90      0.90     36164
```

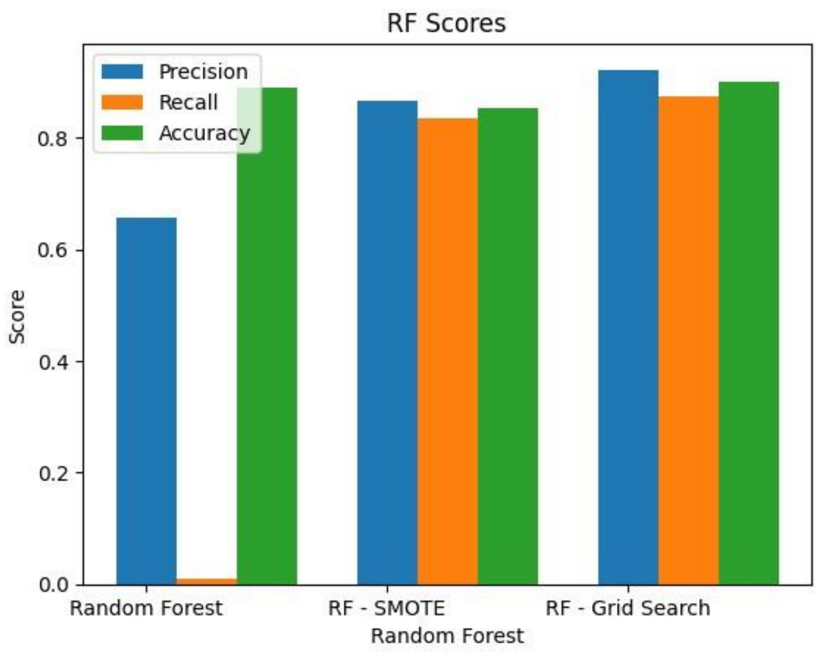
Evaluation Metrics



The ROC curve value shows an increase in value to 0.90 from the previous value of 0.85 and the confusion matrix better values with the increase in number of classes predicted correctly, with 15760 class 0 identified correctly and 16788 class 1 identified correctly. This is an increase from the previous value of 15067 and 15799. From the classification report, the values of accuracy, precision and recall have shown an increase in values, with recall increasing from 0.84 class 1 and 0.87 class 0 to 0.87 and

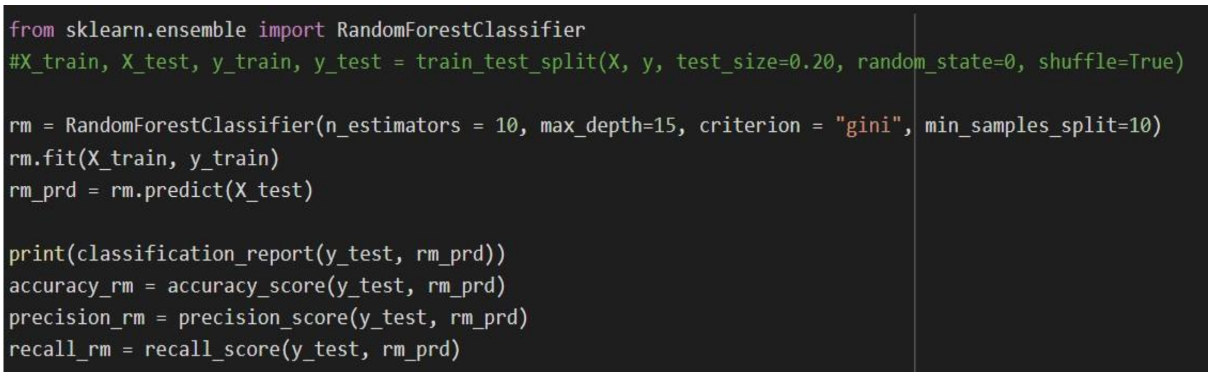
0.93. Similarly precision showed an increase from 0.87 and 0.84 for both the classes to 0.92 and 0.88.

Overall Result:



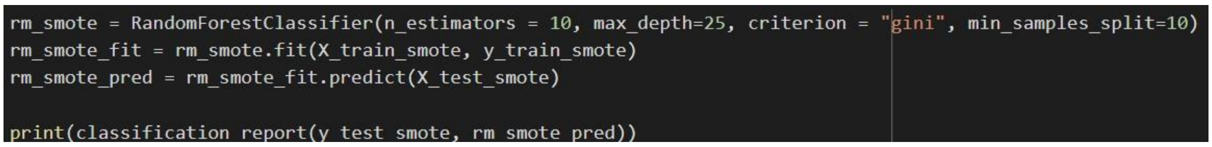
From the above, image the precision, recall and accuracies are considered for class 1, which is more important to us since we have to reduce false negative. The above image gives us a summary of the overall results obtained under each kind of fine tuning. In here RF - Grid Search tuned denotes the Random Forest tuned with GridSearchCV and SMOTE and provides the best results overall. Our problem statement involves classifying the readmission rates and to reduce the number of false negatives and for that high recall value is necessary, which has been achieved through fine tuning.

Code Snippet:



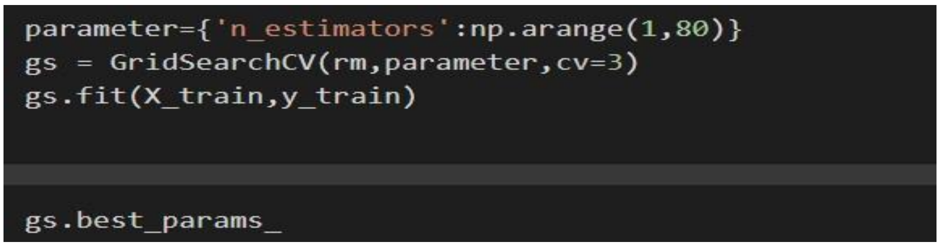
```
from sklearn.ensemble import RandomForestClassifier
#X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, random_state=0, shuffle=True)

rm = RandomForestClassifier(n_estimators = 10, max_depth=15, criterion = "gini", min_samples_split=10)
rm.fit(X_train, y_train)
rm_prd = rm.predict(X_test)

print(classification_report(y_test, rm_prd))
accuracy_rm = accuracy_score(y_test, rm_prd)
precision_rm = precision_score(y_test, rm_prd)
recall_rm = recall_score(y_test, rm_prd)
```

Random Forest Implementation and Evaluation



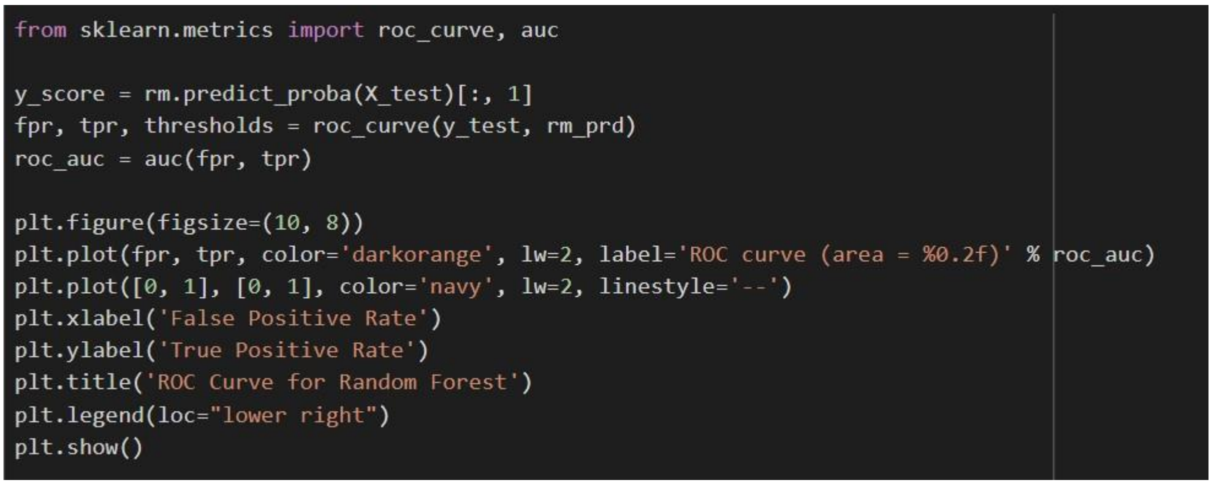
```
rm_smote = RandomForestClassifier(n_estimators = 10, max_depth=25, criterion = "gini", min_samples_split=10)
rm_smote_fit = rm_smote.fit(X_train_smote, y_train_smote)
rm_smote_pred = rm_smote_fit.predict(X_test_smote)

print(classification_report(y_test_smote, rm_smote_pred))
```

Random Forest With SMOTE



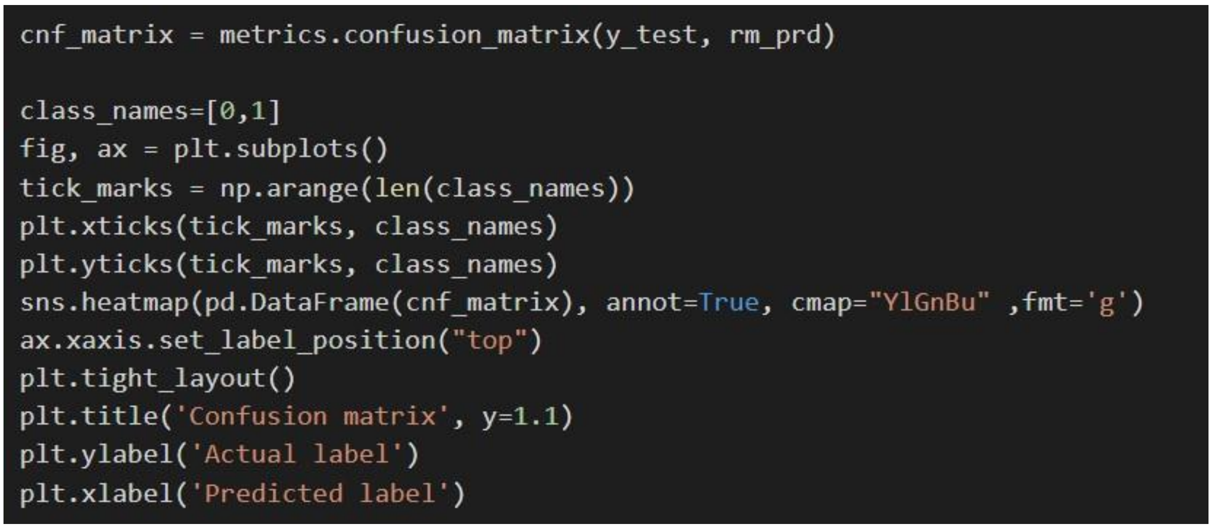
```
parameter={'n_estimators':np.arange(1,80)}
gs = GridSearchCV(rm,parameter,cv=3)
gs.fit(X_train,y_train)

gs.best_params_
```

Random Forest With Best Parameters



```
from sklearn.metrics import roc_curve, auc

y_score = rm.predict_proba(X_test)[:, 1]
fpr, tpr, thresholds = roc_curve(y_test, rm_prd)
roc_auc = auc(fpr, tpr)

plt.figure(figsize=(10, 8))
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' % roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve for Random Forest')
plt.legend(loc="lower right")
plt.show()
```

Roc Curve Implementation



```
cnf_matrix = metrics.confusion_matrix(y_test, rm_prd)

class_names=[0,1]
fig, ax = plt.subplots()
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names)
plt.yticks(tick_marks, class_names)
sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, cmap="YlGnBu" ,fmt='g')
ax.xaxis.set_label_position("top")
plt.tight_layout()
plt.title('Confusion matrix', y=1.1)
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
```

Confusion Matrix

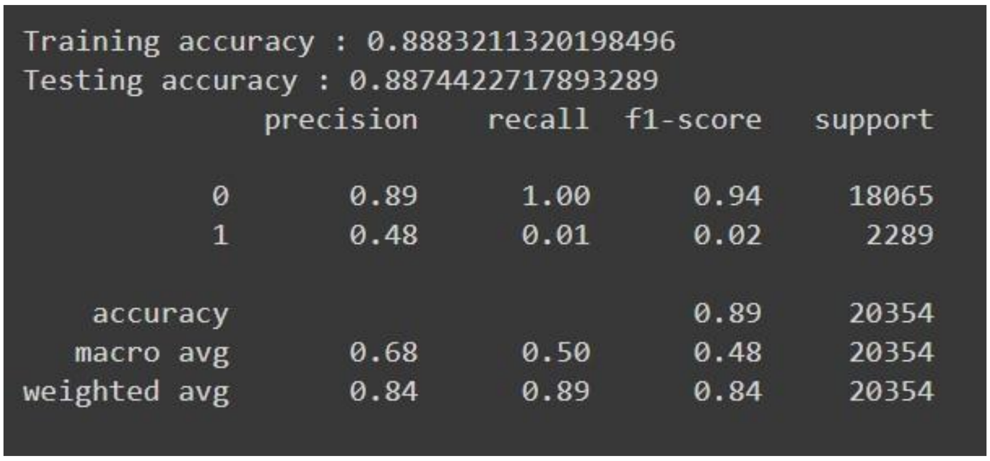
<u>AdaBoost:</u>

Reason:

We decided to utilize the AdaBoost algorithm because it can help weak learners perform better by increasing their accuracy through iterative learning. This is especially helpful for datasets like medical data that contain complicated relationships and interactions between variables. AdaBoost is a versatile algorithm that may be used for a variety of issues, including binary classification issues like predicting hospital readmissions. Many applications, including medical diagnostics and drug reaction prediction, have demonstrated it to work effectively.

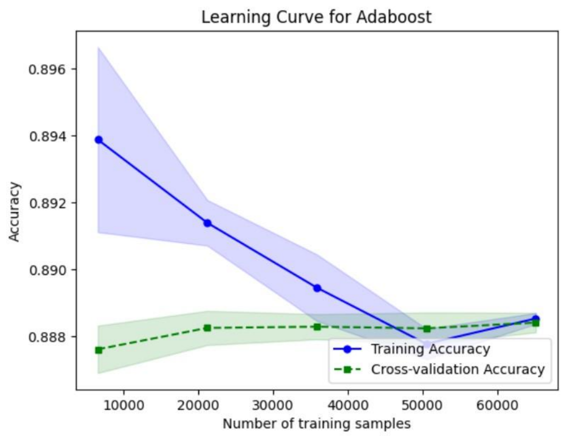
Training and Tuning:

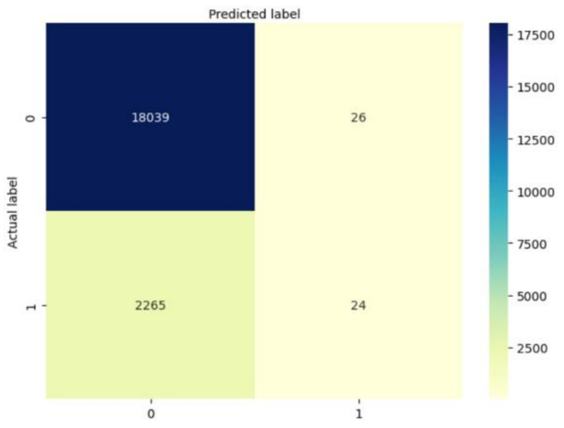
Using the preprocessed dataset, we trained a typical Adaboost model by using the default hyperparameters. The model's performance on the testing set was then assessed by computing measures like accuracy, precision, recall, and F1 score. We trained a classification model on our preprocessed dataset using the AdaBoostClassifier algorithm and a random state value of 20. We divided the dataset into training and testing sets, and then we used the fit() function to fit the AdaBoostClassifier to the training set of data. After making predictions on the testing set using the predict() function, we utilized the score() method to determine the model's accuracy.



```
Training accuracy : 0.8883211320198496
Testing accuracy : 0.8874422717893289
              precision    recall  f1-score   support

           0       0.89      1.00      0.94     18065
           1       0.48      0.01      0.02      2289

    accuracy                           0.89     20354
   macro avg       0.68      0.50      0.48     20354
weighted avg       0.84      0.89      0.84     20354
```

Evaluation Metrics for AdaBoost





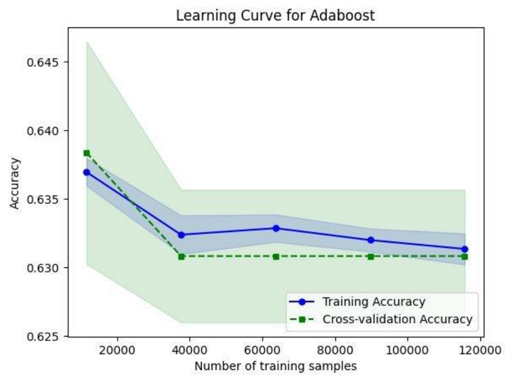
Evaluation Metrics for AdaBoost (Without Hyper-parameter Tuning)

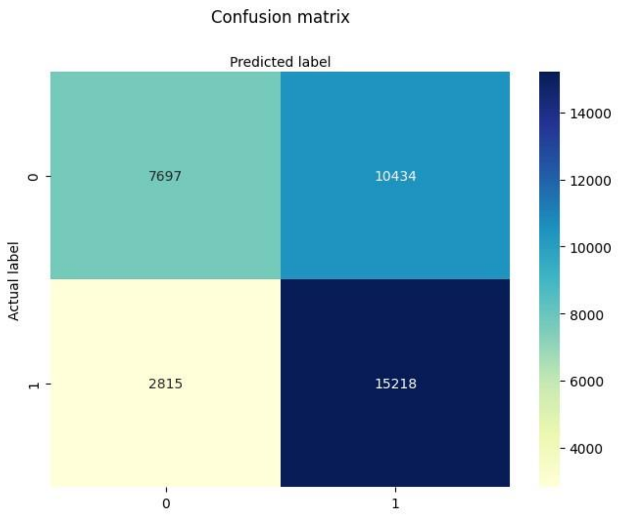
From the metrics above, we can infer that due to class imbalances the precision and recall values are much lesser for class 1 being 0.48 and 0.01. Even though there is a high accuracy, this trade off renders our model highly biased which is not suitable for a medical dataset. The confusion matrix also shows that for class 1 only 24 are predicted correctly and 2265 is predicted wrong, which displays that our model is highly biased.

AdaBoost with Hyper-parameter tuning (Fine Tuning):

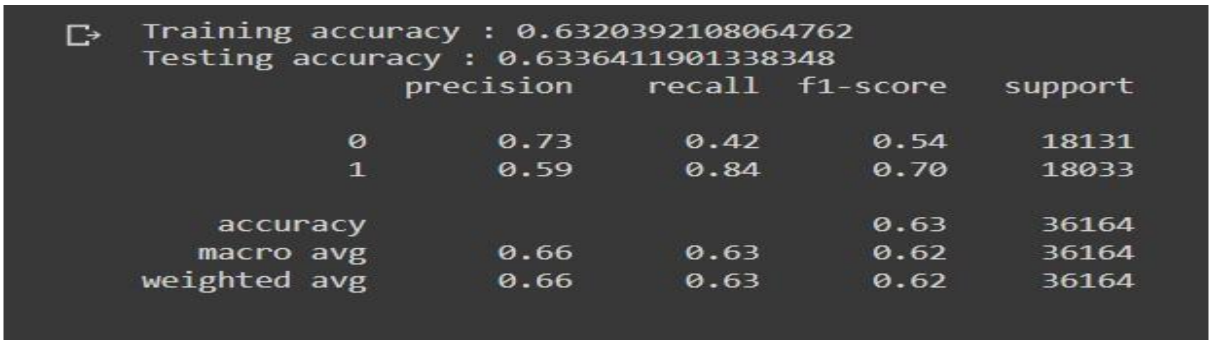
We used the hyperparameter tuning to train a new Adaboost model. On the testing set, we assessed the model's performance and contrasted it with that of the standard Adaboost model.

The performance of the model was then further enhanced by hyperparameter tuning. That is, we kept the random state at 20 for consistency. The optimum hyperparameters for this dataset, according to our research, were n_estimators=73 and learning_rate=0.01. The model was then fitted to the preprocessed training data using SMOTE oversampling, and a new instance of the AdaBoostClassifier class was produced with the tuned hyperparameters. Using the.score() method, we assessed the performance of this tuned model on both the training and testing sets. The adb tuned pred variable included the predictions generated by the tuned model.



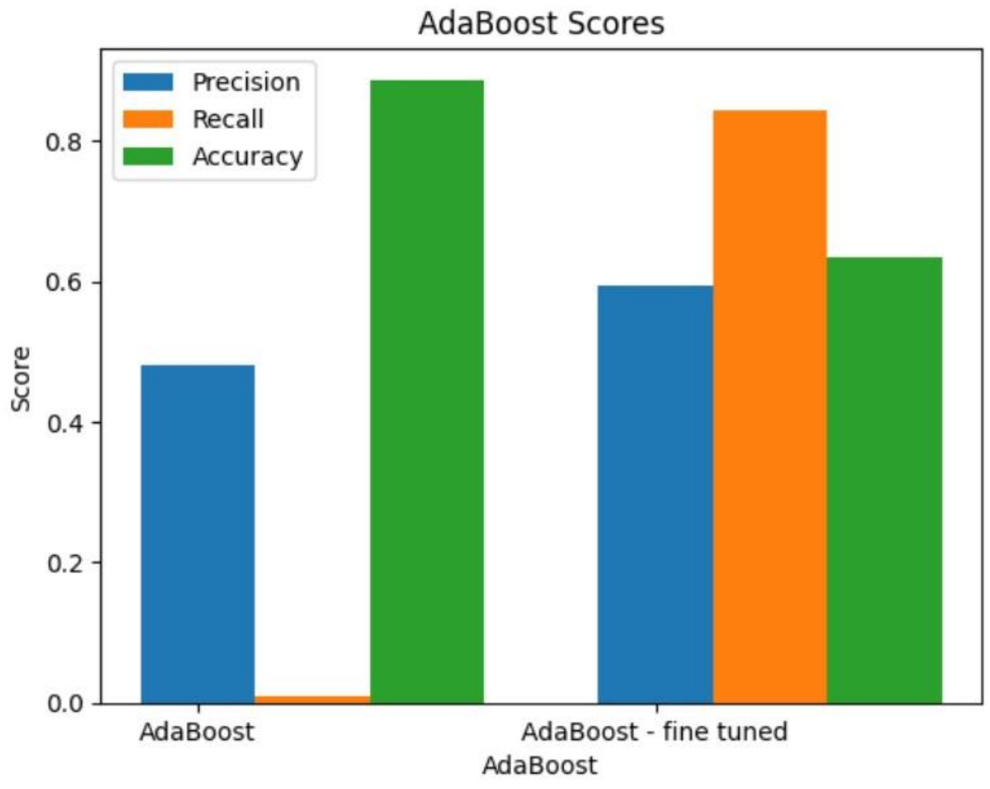


Evaluation Metrics for AdaBoost
(With Hyper-parameter Tuning)


```
Training accuracy : 0.6320392108064762
Testing accuracy : 0.6336411901338348
              precision    recall  f1-score   support

           0       0.73      0.42      0.54     18131
           1       0.59      0.84      0.70     18033

    accuracy                           0.63     36164
   macro avg       0.66      0.63      0.62     36164
weighted avg       0.66      0.63      0.62     36164
```

From the above metrics, we can see an increase in the precision and recall values for class 1 from 0.48 and 0.01 to 0.59 and 0.84 with a trade off in accuracy form 0.89 to 0.63. Although there is an increase, compared with Decision Trees and Random Forest the AdaBoost does not fit the model really well, but it took very less time to run and fit the model.

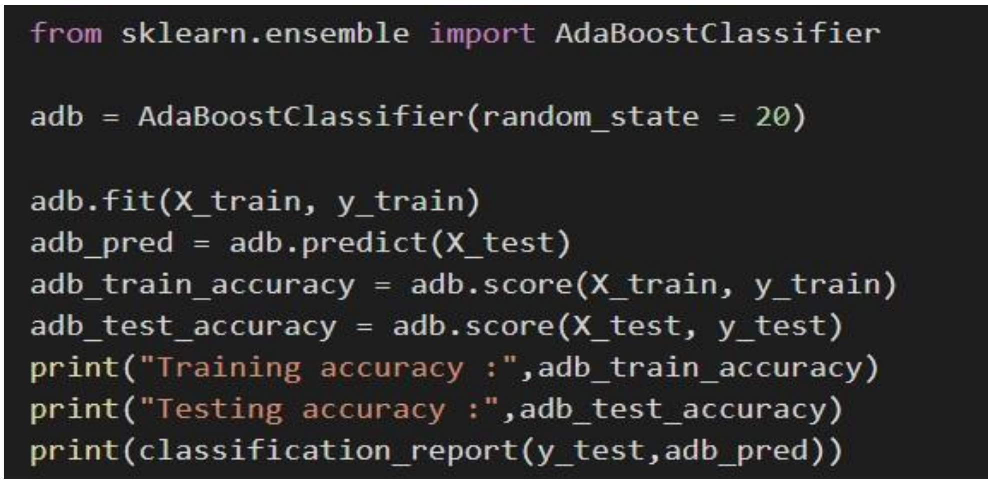
Overall Results:



Adaboost - With and Without Hyper-parameter tuning comparison

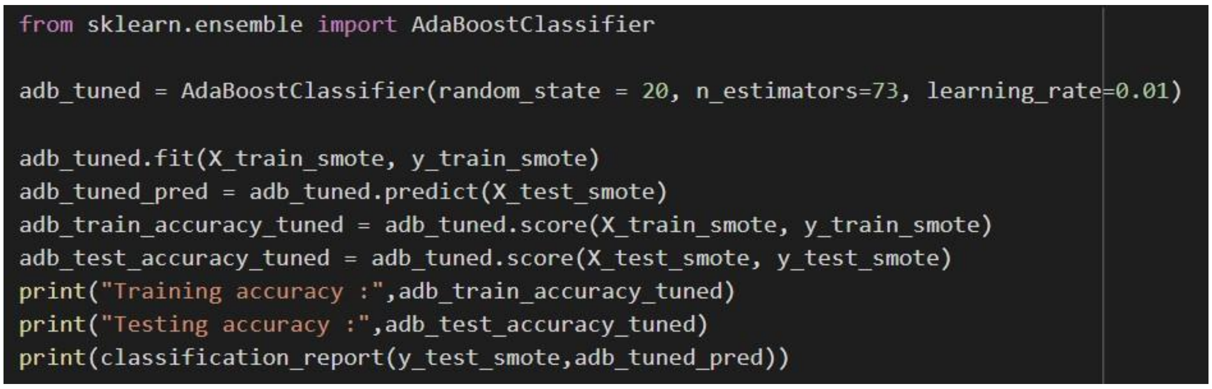
The above image gives us a summary of the overall results obtained under each kind of fine tuning. In here AdaBoost - fine tuned denotes the dataset with SMOTE and provides the best results overall. From the above, image the precision, recall and accuracies are considered for class 1, which is more important to us since we have to reduce false negative. Our problem statement involves classifying the readmission rates and to reduce the number of false negatives and for that high recall value is necessary, which has been achieved through fine tuning. Although the accuracies are lesser than the previous two approached, we were able to fine tune the model to fit our dataset in a better way.

Code Snippets:



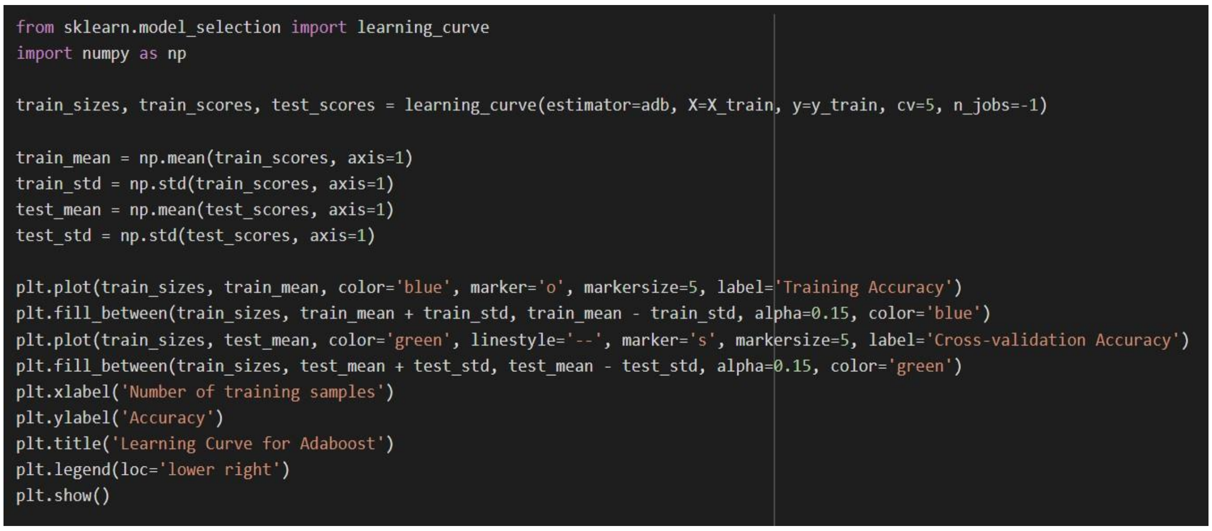
```
from sklearn.ensemble import AdaBoostClassifier

adb = AdaBoostClassifier(random_state = 20)

adb.fit(X_train, y_train)
adb_pred = adb.predict(X_test)
adb_train_accuracy = adb.score(X_train, y_train)
adb_test_accuracy = adb.score(X_test, y_test)
print("Training accuracy :",adb_train_accuracy)
print("Testing accuracy :",adb_test_accuracy)
print(classification_report(y_test,adb_pred))
```

AdaBoost Implementation and Evaluation



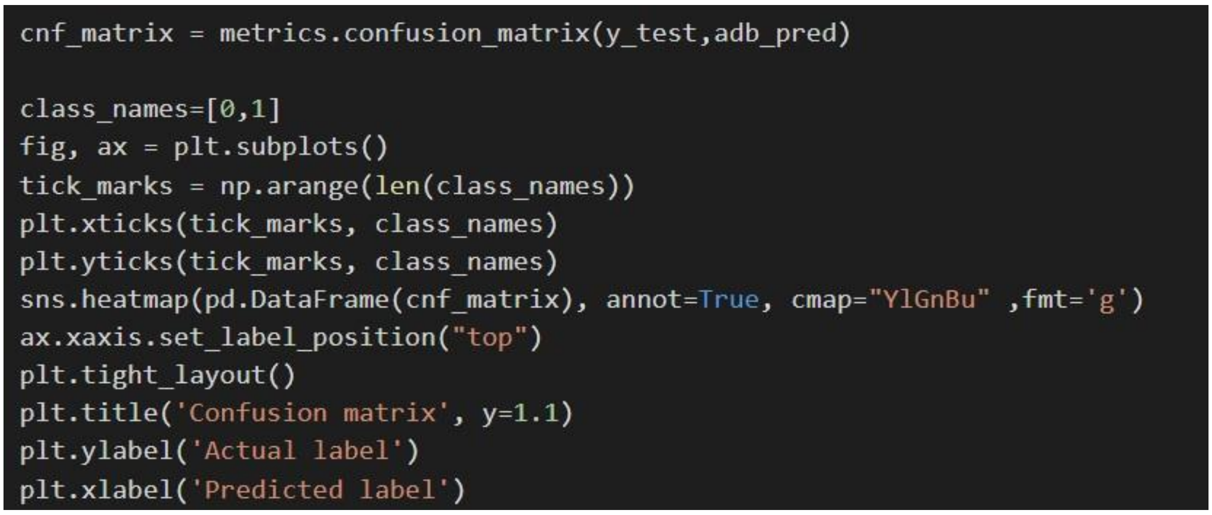
```
from sklearn.ensemble import AdaBoostClassifier

adb_tuned = AdaBoostClassifier(random_state = 20, n_estimators=73, learning_rate=0.01)

adb_tuned.fit(X_train_smote, y_train_smote)
adb_tuned_pred = adb_tuned.predict(X_test_smote)
adb_train_accuracy_tuned = adb_tuned.score(X_train_smote, y_train_smote)
adb_test_accuracy_tuned = adb_tuned.score(X_test_smote, y_test_smote)
print("Training accuracy :",adb_train_accuracy_tuned)
print("Testing accuracy :",adb_test_accuracy_tuned)
print(classification_report(y_test_smote,adb_tuned_pred))
```

AdaBoost with Hyper-parameter tuning - Implementation and Evaluation



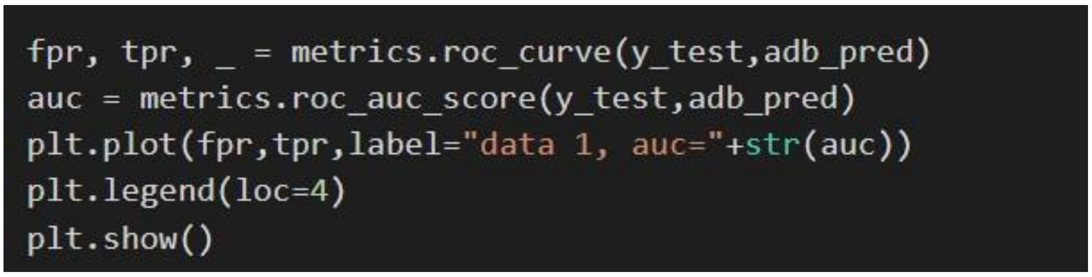
```python
from sklearn.model_selection import learning_curve
import numpy as np

train_sizes, train_scores, test_scores = learning_curve(estimator=adb, X=X_train, y=y_train, cv=5, n_jobs=-1)

train_mean = np.mean(train_scores, axis=1)
train_std = np.std(train_scores, axis=1)
test_mean = np.mean(test_scores, axis=1)
test_std = np.std(test_scores, axis=1)

plt.plot(train_sizes, train_mean, color='blue', marker='o', markersize=5, label='Training Accuracy')
plt.fill_between(train_sizes, train_mean + train_std, train_mean - train_std, alpha=0.15, color='blue')
plt.plot(train_sizes, test_mean, color='green', linestyle='--', marker='s', markersize=5, label='Cross-validation Accuracy')
plt.fill_between(train_sizes, test_mean + test_std, test_mean - test_std, alpha=0.15, color='green')
plt.xlabel('Number of training samples')
plt.ylabel('Accuracy')
plt.title('Learning Curve for Adaboost')
plt.legend(loc='lower right')
plt.show()
```

Learning Curve for AdaBoost



```python
cnf_matrix = metrics.confusion_matrix(y_test,adb_pred)

class_names=[0,1]
fig, ax = plt.subplots()
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names)
plt.yticks(tick_marks, class_names)
sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, cmap="YlGnBu" ,fmt='g')
ax.xaxis.set_label_position("top")
plt.tight_layout()
plt.title('Confusion matrix', y=1.1)
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
```

Confusion Matrix



```python
fpr, tpr, _ = metrics.roc_curve(y_test,adb_pred)
auc = metrics.roc_auc_score(y_test,adb_pred)
plt.plot(fpr,tpr,label="data 1, auc="+str(auc))
plt.legend(loc=4)
plt.show()
```

ROC Curve

Bernoulli's Naive Bayes:

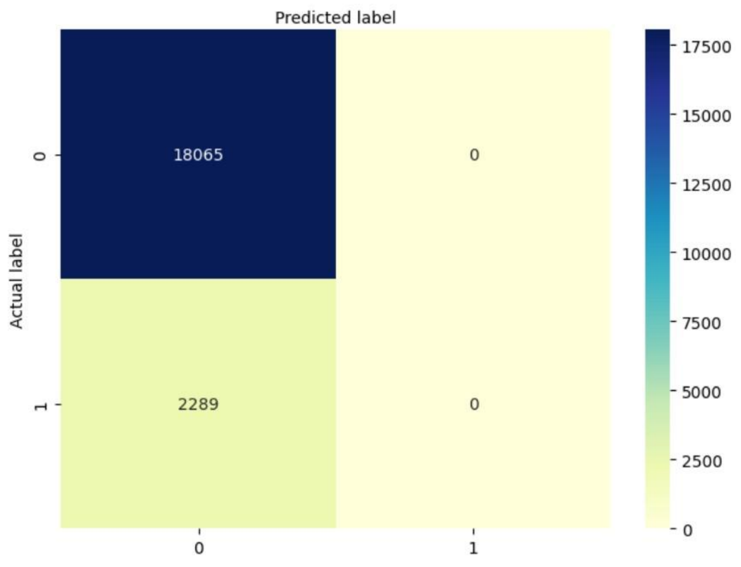
Reason:

The Naive Bayes algorithm is a probabilistic classification algorithm which uses the Bayes theorem to determine the likelihood that a sample belongs to a particular class. Naive Bayes has been used in the analysis of medical data to predict a variety of outcomes, including readmissions to hospitals. This is due to the fact that Naive Bayes is effective at handling high-dimensional features seen in medical data, such as laboratory test results and medical history. For binary classification issues, Bernoulli Naive Bayes is a variation of the Naive Bayes algorithm that is frequently employed. It presumes the characteristics are binary, which means they can only take on one of two possible values, which are typically denoted by the numbers 0 or 1.

Training and Tuning:

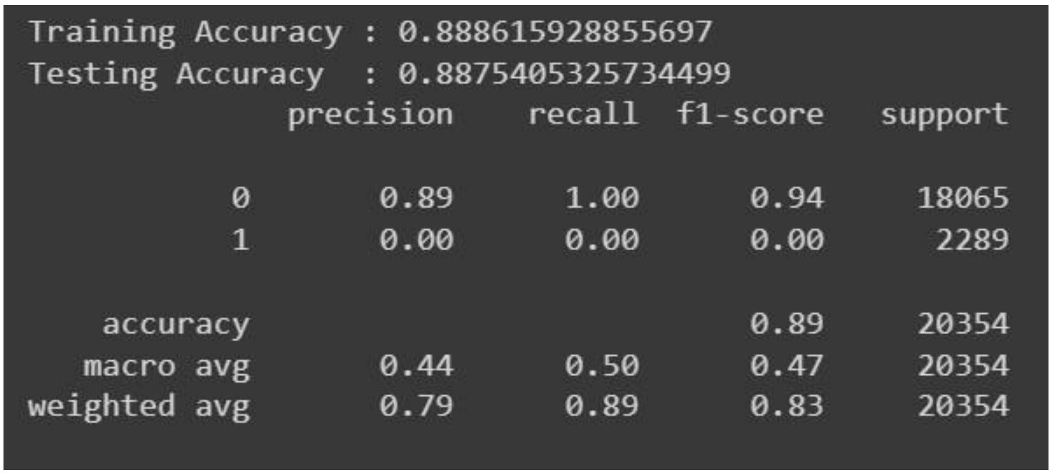
Base Model:

We first divided our data into training and validation sets in order to train and fine-tune the models. After that, we fitted the models for Bernoulli Naive Bayes using the training set.



Confusion Matrix for Naive Bayes

We evaluated the effectiveness of the algorithm by measuring various metrics such as accuracy, precision, recall and F1 score along with macro and weighted average. We also visualized the results using confusion matrices, where all the values in class 0 are identified correctly, whereas no class 1 is identified correctly. This results in a high false negative rate.

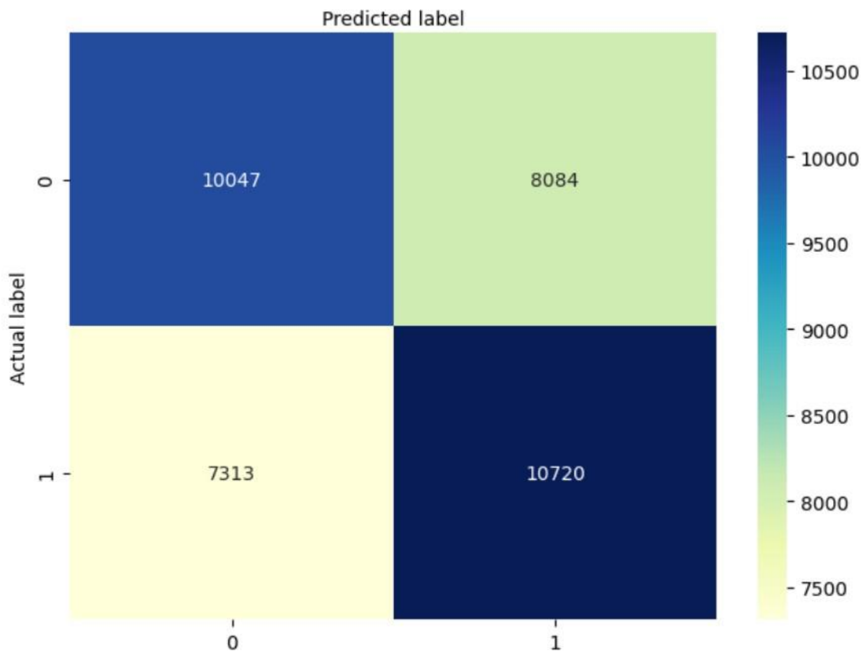


```
Training Accuracy : 0.888615928855697
Testing Accuracy  : 0.8875405325734499
              precision    recall  f1-score   support

           0       0.89      1.00      0.94     18065
           1       0.00      0.00      0.00      2289

    accuracy                           0.89     20354
   macro avg       0.44      0.50      0.47     20354
weighted avg       0.79      0.89      0.83     20354
```

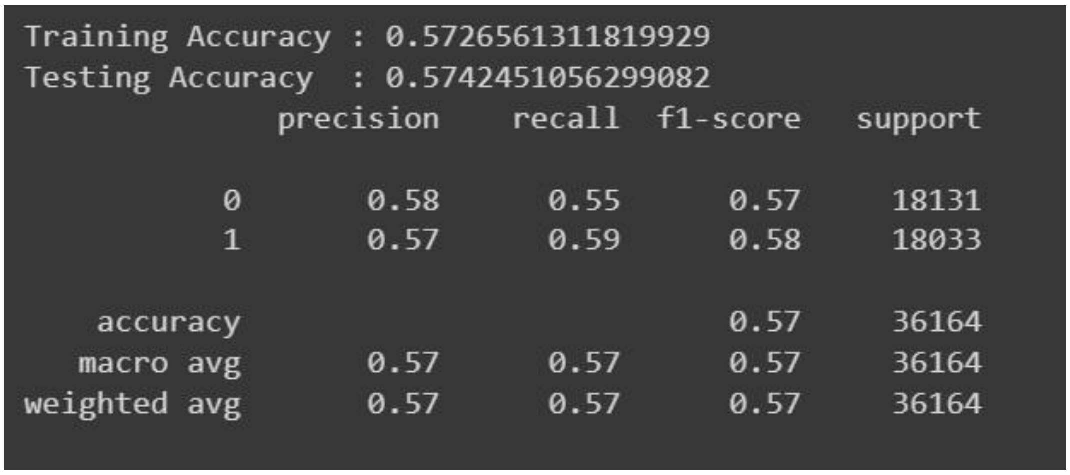
We observed that the recall measure was substantially lower than other metrics, such as precision and recall (0 for class 1) and accuracy, after evaluating the performance of our Naive Bayes classifier on our dataset. Furthermore, Naive Bayes assumes that the features are conditionally independent given the class label, which may not hold true for our dataset. This can result in a loss of information and may contribute to the low recall, which is more important to our model.

Fine Tuned Model (with SMOTE):

In Bernoulli Naive Bayes with SMOTE, each feature is modeled using a Bernoulli distribution, which describes the probability of observing a particular value (0 or 1) for that feature. Bernoulli Naive Bayes can be effective for binary classification problems with many features, especially when the features are highly sparse and many of them are irrelevant for classification.



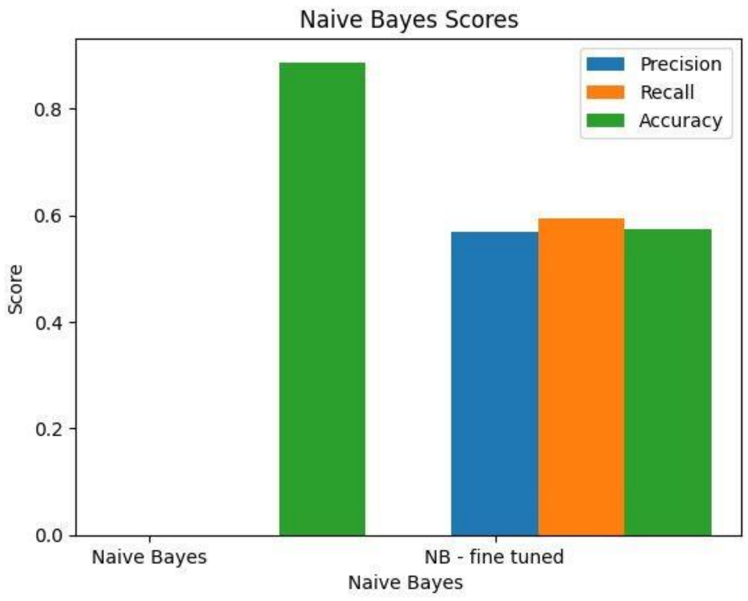
Confusion Matrix for Bernoulli Naive Bayes



```
Training Accuracy : 0.5726561311819929
Testing Accuracy  : 0.5742451056299082
              precision    recall  f1-score   support

           0       0.58      0.55      0.57     18131
           1       0.57      0.59      0.58     18033

    accuracy                           0.57     36164
   macro avg       0.57      0.57      0.57     36164
weighted avg       0.57      0.57      0.57     36164
```

Using criteria like accuracy, precision, recall, and F1 score on the validation set, we assessed each model's performance. We can infer that now the model acts unbiased by detecting the class 1 and with the improved precision and recall values.

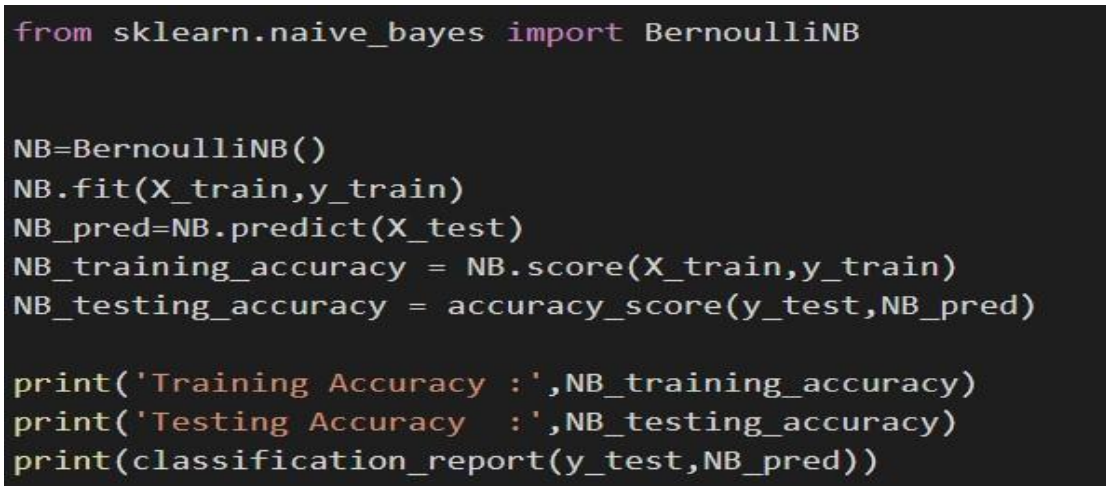
Overall Result:



Naive Bayes with and without SMOTE NB (Fine Tuned) comparison

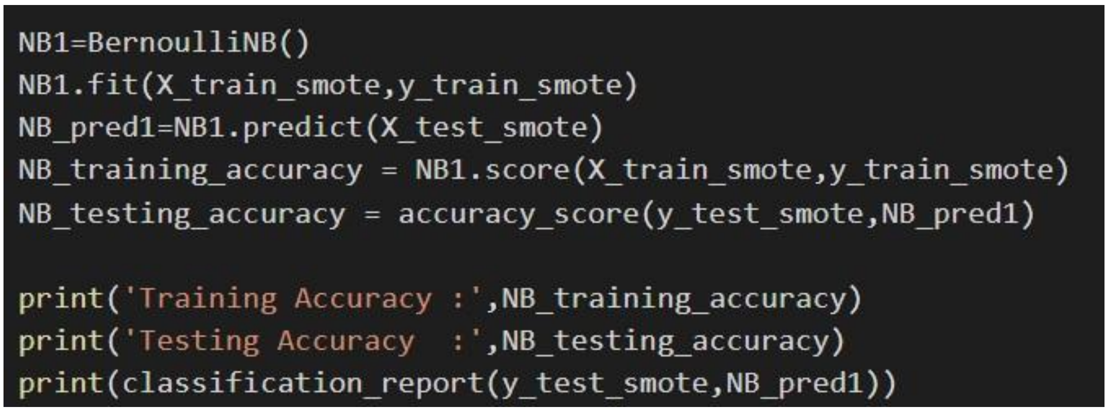
From the above image the precision, recall and accuracies are considered for class 1, which is more important to us since we have to reduce false negatives. We can see that during the initial approach, only the accuracy is higher which is not suitable for our data and after fine tuning we are able to achieve an equilibrium of accuracy, precision and recall. This model displays a lesser accuracy and other metrics compared to above approaches.

Code Snippets:



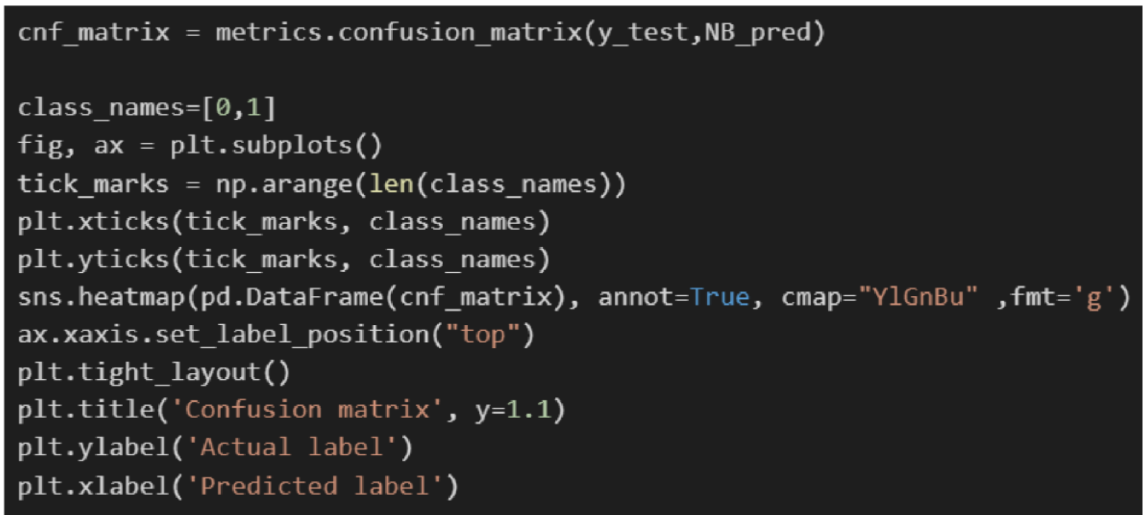
```python
from sklearn.naive_bayes import BernoulliNB


NB=BernoulliNB()
NB.fit(X_train,y_train)
NB_pred=NB.predict(X_test)
NB_training_accuracy = NB.score(X_train,y_train)
NB_testing_accuracy = accuracy_score(y_test,NB_pred)

print('Training Accuracy :',NB_training_accuracy)
print('Testing Accuracy  :',NB_testing_accuracy)
print(classification_report(y_test,NB_pred))
```

Naive Bayes implementation and Evaluation



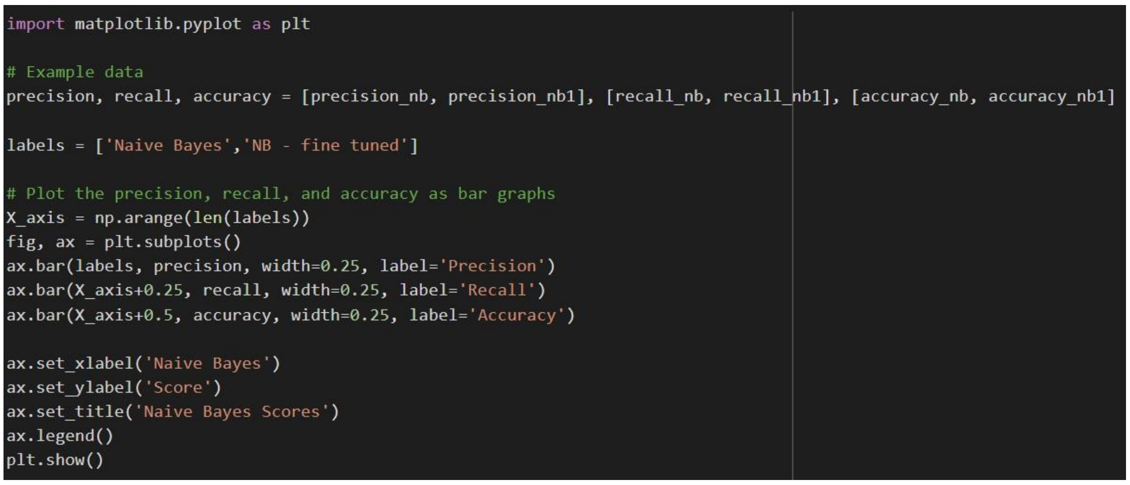
```python
NB1=BernoulliNB()
NB1.fit(X_train_smote,y_train_smote)
NB_pred1=NB1.predict(X_test_smote)
NB_training_accuracy = NB1.score(X_train_smote,y_train_smote)
NB_testing_accuracy = accuracy_score(y_test_smote,NB_pred1)

print('Training Accuracy :',NB_training_accuracy)
print('Testing Accuracy  :',NB_testing_accuracy)
print(classification_report(y_test_smote,NB_pred1))
```

Naive Bayes with SMOTE implementation and Evaluation



```python
cnf_matrix = metrics.confusion_matrix(y_test,NB_pred)

class_names=[0,1]
fig, ax = plt.subplots()
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names)
plt.yticks(tick_marks, class_names)
sns.heatmap(pd.DataFrame(cnf_matrix), annot=True, cmap="YlGnBu" ,fmt='g')
ax.xaxis.set_label_position("top")
plt.tight_layout()
plt.title('Confusion matrix', y=1.1)
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
```

Confusion Matrix



```
import matplotlib.pyplot as plt

# Example data
precision, recall, accuracy = [precision_nb, precision_nb1], [recall_nb, recall_nb1], [accuracy_nb, accuracy_nb1]

labels = ['Naive Bayes','NB - fine tuned']

# Plot the precision, recall, and accuracy as bar graphs
X_axis = np.arange(len(labels))
fig, ax = plt.subplots()
ax.bar(labels, precision, width=0.25, label='Precision')
ax.bar(X_axis+0.25, recall, width=0.25, label='Recall')
ax.bar(X_axis+0.5, accuracy, width=0.25, label='Accuracy')

ax.set_xlabel('Naive Bayes')
ax.set_ylabel('Score')
ax.set_title('Naive Bayes Scores')
ax.legend()
plt.show()
```

Comparison chart

XGBoost:

Reason:

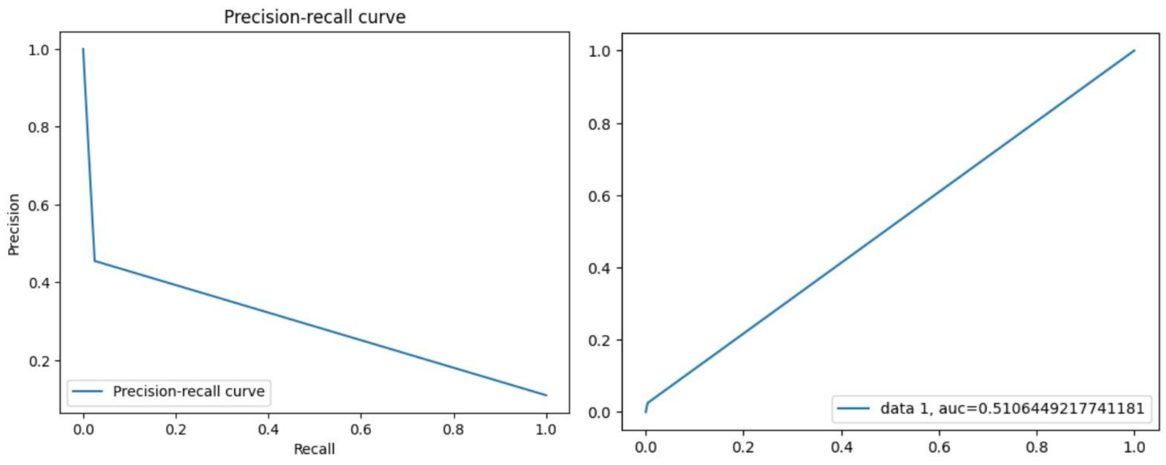
XGBoost is a popular gradient boosting algorithm that is frequently applied to structured data-based machine learning problems like regression and classification. It is robust because of its accuracy, scalability, and speed, making it suitable for huge datasets with a variety of attributes. Moreover, it offers parallel processing, which speeds up model training compared to many other techniques.

The effectiveness with which XGBoost can deal with missing data and outliers is one of its main advantages. L1 and L2 regularization approaches, which help avoid overfitting and enhance generalization, are used to achieve this. The number of trees in the model, the learning rate, and the depth of the trees are a few more hyperparameters in XGBoost that can be adjusted to improve performance.

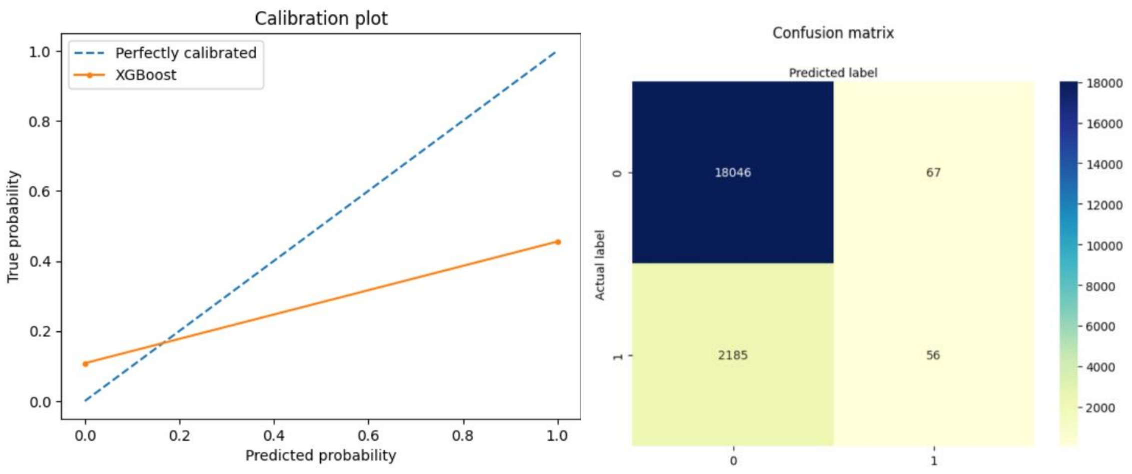
Training and Tuning:

Base Model:

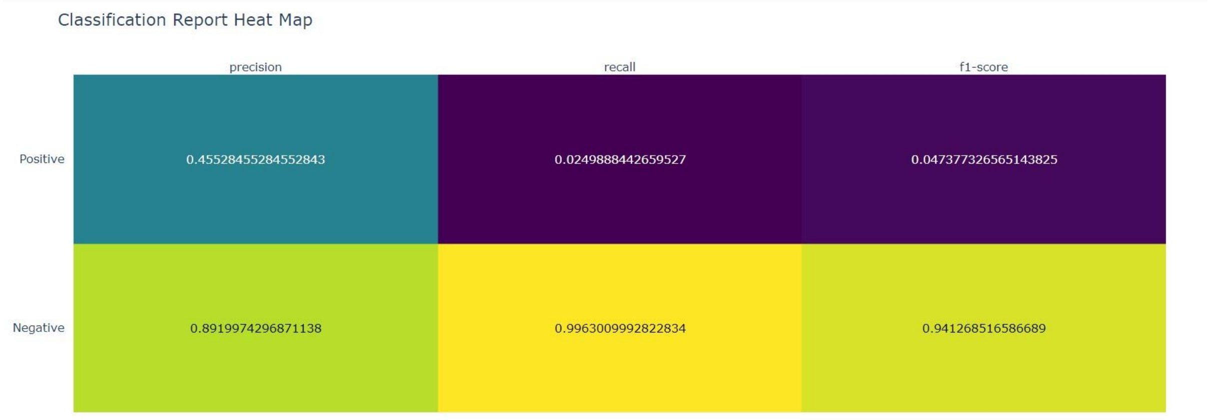
For tuning and training the XGBoost model, two methods were followed: using normal data and using SMOTE data. In the first approach, neither oversampling or undersampling strategies were used, the model was trained on the original dataset. The model was trained on the entire dataset and assessed on a different test set after the optimum hyperparameters had been identified.



Precision-Recall Curve and ROC curve for Without SMOTE



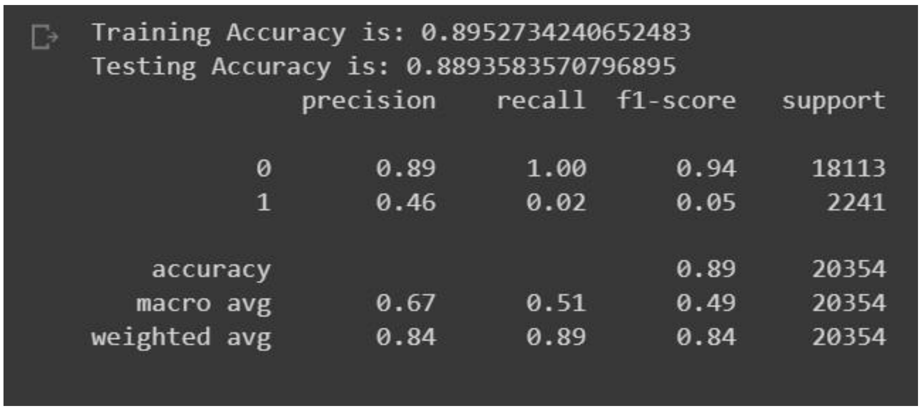
Calibration Plot and Confusion Matrix for Without SMOTE


Classification Report Heat Map

	precision	recall	f1-score
Positive	0.45528455284552843	0.0249888442659527	0.047377326565143825
Negative	0.8919974296871138	0.9963009992822834	0.941268516586689

Classification Report Heatmap for Without SMOTE

We evaluated the effectiveness of the algorithm by measuring various metrics such as accuracy, precision, recall and F1 score along with macro and weighted average. We also visualized the results using confusion matrices and ROC curves.

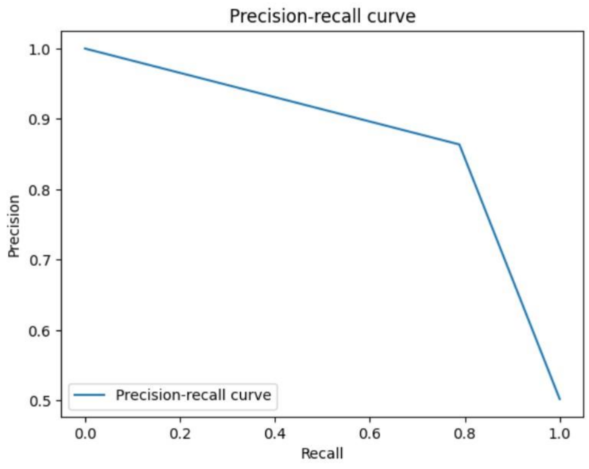


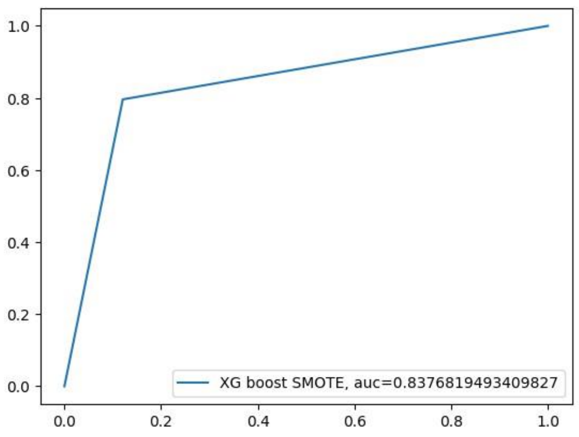
```
Training Accuracy is: 0.8952734240652483
Testing Accuracy is: 0.8893583570796895
              precision    recall  f1-score   support

           0       0.89      1.00      0.94     18113
           1       0.46      0.02      0.05      2241

    accuracy                           0.89     20354
   macro avg       0.67      0.51      0.49     20354
weighted avg       0.84      0.89      0.84     20354
```

It can be analyzed from the above image that the recall values are low due to data imbalance. From the above metrics details we can infer that although the value of accuracy is high, the recall value is very less which is more important for a medical dataset, since we do not want anyone to miss a diagnosis i.e, identifying an individual as false negative. From the confusion matrix above, there are very few values correctly identified for class 1. The model achieved an overall accuracy of 0.89, with a precision of 0.89 for predicting the negative class (0) and a precision of 0.46 for predicting the positive class (1). The recall of the positive class was very low at 0.02, resulting in an F1-score of 0.05. The macro-average F1-score was 0.49, indicating that the model performed poorly in predicting the positive class. Despite this, the weighted-average F1-score was 0.84, suggesting that the model's overall performance was reasonable. Further optimization of the model was required to improve its ability to predict the positive class.

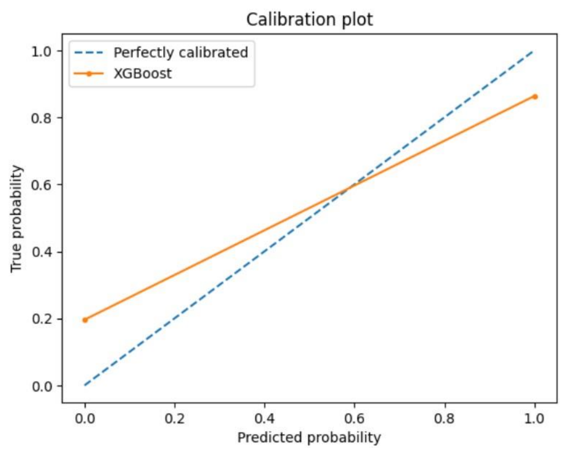
Fine Tuning with SMOTE:

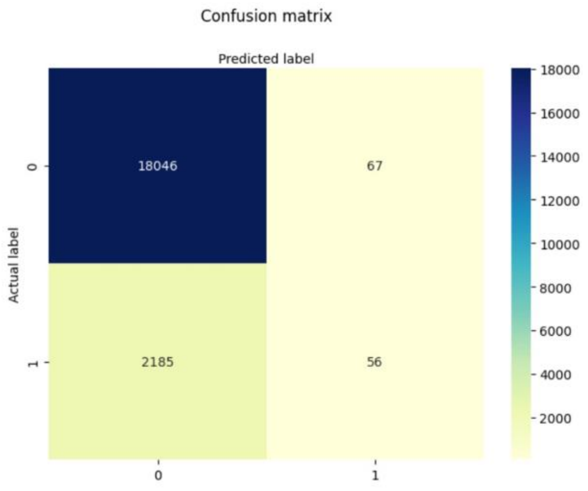
In the second approach, the model was trained using the SMOTE dataset, a method for addressing class imbalance by oversampling the minority class. Before training the model, SMOTE was applied to the dataset to tweak the model. The model was trained on the entire smote dataset and assessed on a different test set after the optimum hyperparameters had been identified.



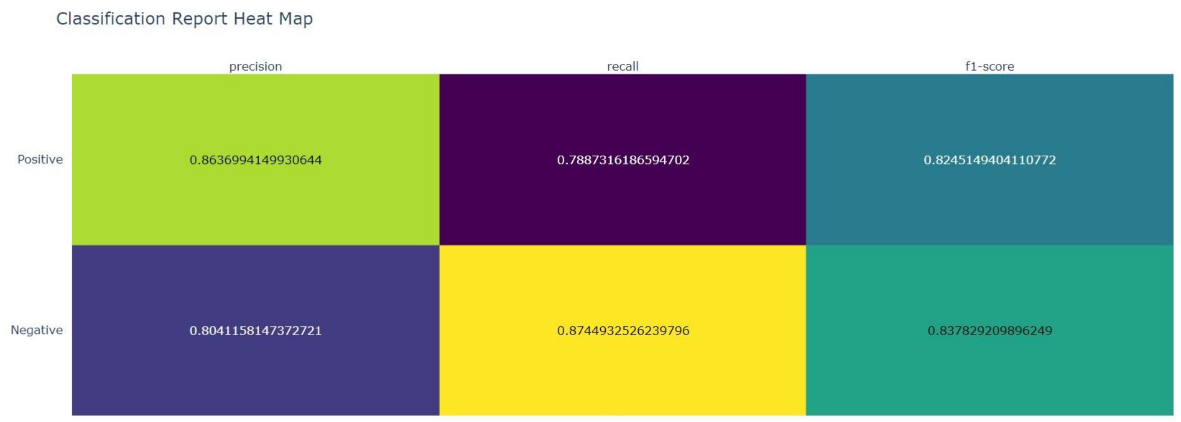


ROC Curve and Precision-Recall Curve for With SMOTE





Calibration Plot and Confusion Matrix for With SMOTE

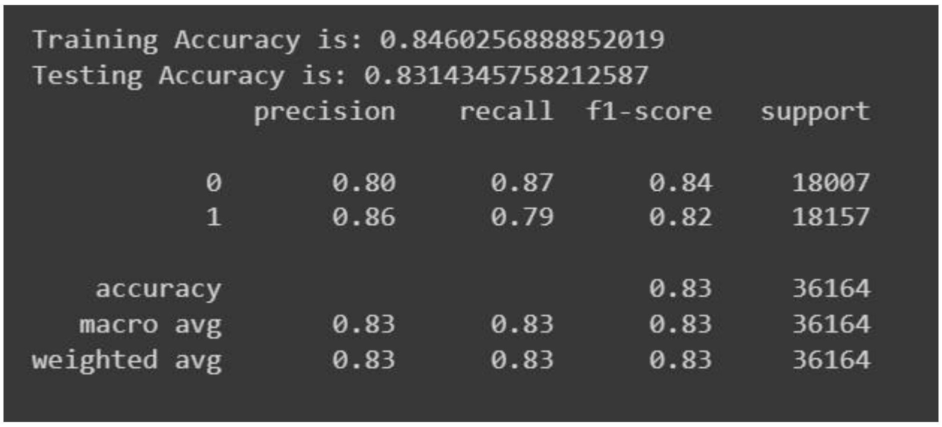

Classification Report Heat Map

	precision	recall	f1-score
Positive	0.8636994149930644	0.7887316186594702	0.8245149404110772
Negative	0.8041158147372721	0.8744932526239796	0.837829209896249

Classification Report Heatmap for With SMOTE

The effectiveness of the XGBoost models was evaluated using several metrics. These metrics provide a comprehensive view of the model's performance and can be used to compare the performance of different models.

The feature importances that XGBoost models offer can be utilized to analyze the model and learn more about the underlying data. Understanding which characteristics are most crucial for accurately anticipating the desired result can be gained from this.

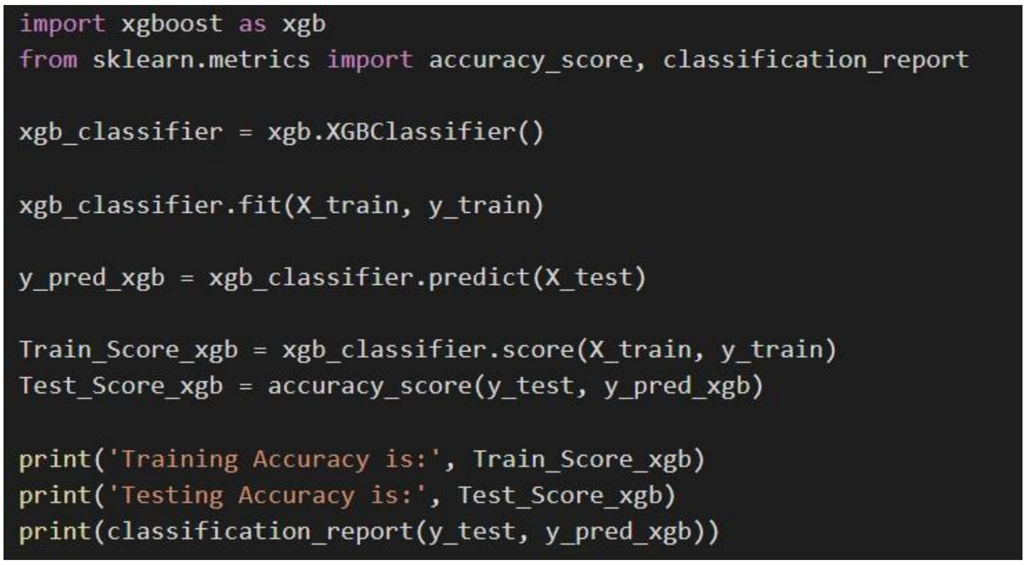


```
Training Accuracy is: 0.8460256888852019
Testing Accuracy is: 0.8314345758212587
              precision    recall  f1-score   support

           0       0.80      0.87      0.84     18007
           1       0.86      0.79      0.82     18157

    accuracy                           0.83     36164
   macro avg       0.83      0.83      0.83     36164
weighted avg       0.83      0.83      0.83     36164
```

Evaluation Metrics

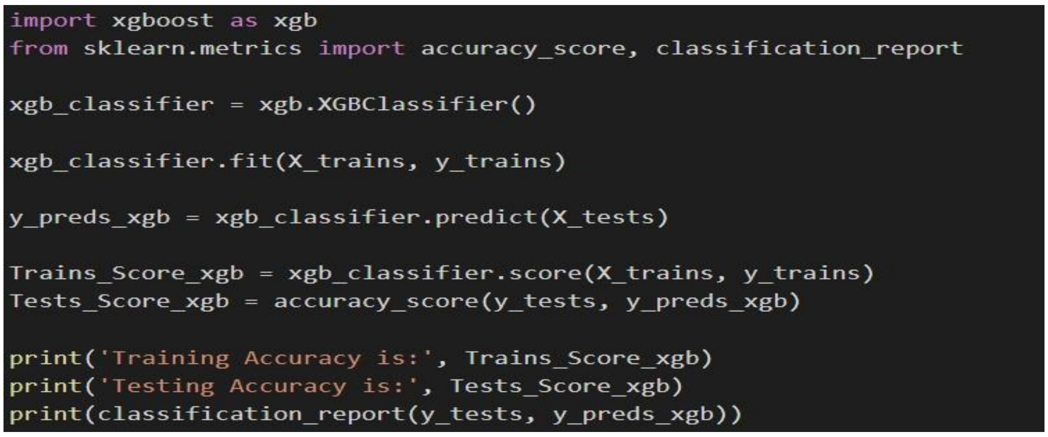
Here, it can be analyzed from the above image that the recall values have improved after performing SMOTE on the data to overcome data imbalance. The recall values have been increased, however there is a decrease in accuracy.
The model achieved an overall accuracy of 84%, with a precision of 81% and recall of 88% for class 0, and a precision of 87% and recall of 80% for class 1. The f1-score was 0.84 for both classes. The macro-average f1-score was 0.84, indicating a balanced performance across both classes. The weighted-average f1-score was also 0.84, suggesting that the model's performance was consistent across all classes. Overall, XGBoost performed well in our SMOTE dataset, demonstrating good accuracy and balanced performance across both classes.

Code Snippets:



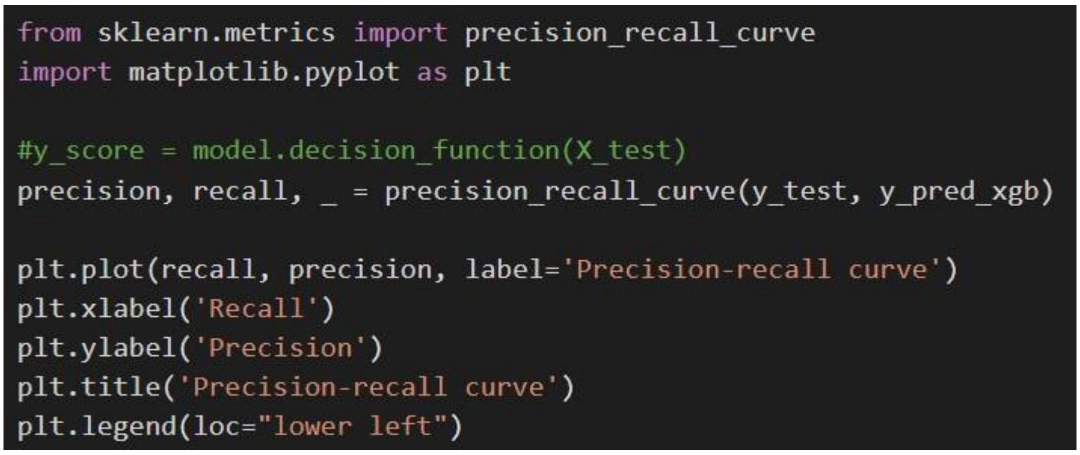
```python
import xgboost as xgb
from sklearn.metrics import accuracy_score, classification_report

xgb_classifier = xgb.XGBClassifier()

xgb_classifier.fit(X_train, y_train)

y_pred_xgb = xgb_classifier.predict(X_test)

Train_Score_xgb = xgb_classifier.score(X_train, y_train)
Test_Score_xgb = accuracy_score(y_test, y_pred_xgb)

print('Training Accuracy is:', Train_Score_xgb)
print('Testing Accuracy is:', Test_Score_xgb)
print(classification_report(y_test, y_pred_xgb))
```

XGBoost Implementation without SMOTE



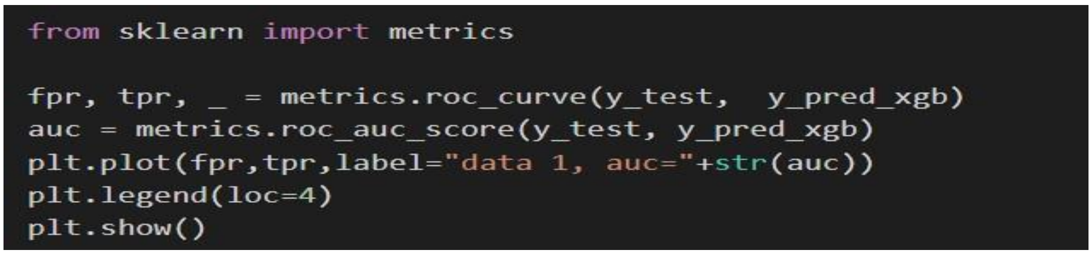
```
import xgboost as xgb
from sklearn.metrics import accuracy_score, classification_report

xgb_classifier = xgb.XGBClassifier()

xgb_classifier.fit(X_trains, y_trains)

y_preds_xgb = xgb_classifier.predict(X_tests)

Trains_Score_xgb = xgb_classifier.score(X_trains, y_trains)
Tests_Score_xgb = accuracy_score(y_tests, y_preds_xgb)

print('Training Accuracy is:', Trains_Score_xgb)
print('Testing Accuracy is:', Tests_Score_xgb)
print(classification_report(y_tests, y_preds_xgb))
```

XGBoost Implementation with SMOTE



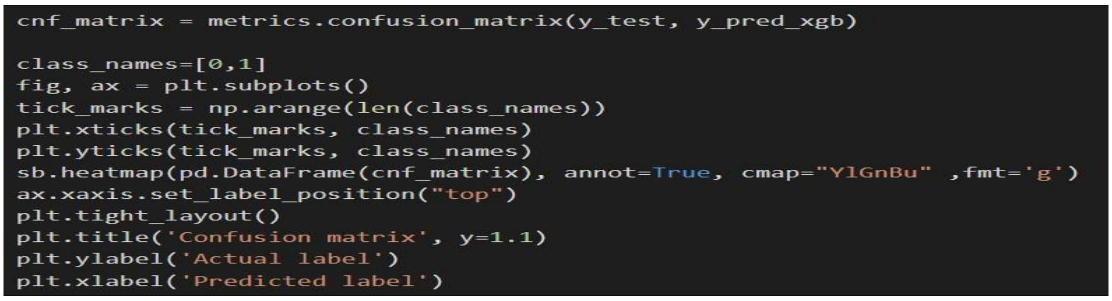
```
from sklearn.metrics import precision_recall_curve
import matplotlib.pyplot as plt

#y_score = model.decision_function(X_test)
precision, recall, _ = precision_recall_curve(y_test, y_pred_xgb)

plt.plot(recall, precision, label='Precision-recall curve')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-recall curve')
plt.legend(loc="lower left")
```

Precision-Recall Curve



```
from sklearn import metrics

fpr, tpr, _ = metrics.roc_curve(y_test,  y_pred_xgb)
auc = metrics.roc_auc_score(y_test, y_pred_xgb)
plt.plot(fpr,tpr,label="data 1, auc="+str(auc))
plt.legend(loc=4)
plt.show()
```

ROC Curve



```
cnf_matrix = metrics.confusion_matrix(y_test, y_pred_xgb)

class_names=[0,1]
fig, ax = plt.subplots()
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks, class_names)
plt.yticks(tick_marks, class_names)
sb.heatmap(pd.DataFrame(cnf_matrix), annot=True, cmap="YlGnBu" ,fmt='g')
ax.xaxis.set_label_position("top")
plt.tight_layout()
plt.title('Confusion matrix', y=1.1)
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
```

Confusion Matrix

References:

[1]. Diabetes 130-US hospitals for years 1999-2008 Dataset from UCI Machine Learning Repository.

[2]. Impact of HbA1c Measurement on Hospital Readmission Rates: Analysis of 70,000 Clinical Database Patient Records.

[3] https://plotly.com/python/discrete-color/#discrete-color-annotations-on-heatmaps

[4]https://pandas.pydata.org/docs/

[5]https://scikit-learn.org/stable/user_guide.html

[6]https://imbalanced-learn.org/stable/references/generated/imblearn.over_sampling.SMOTE.html

[7]https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html

[8]https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

[9]https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[10]https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html

[11]https://xgboost.readthedocs.io/en/stable/get_started.html

[12]https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html

[13]https://plotly.com/python-api-reference/generated/plotly.figure_factory.html

[14]https://scikit-learn.org/stable/modules/generated/sklearn.calibration.calibration_curve.html

[15]https://scikit-learn.org/stable/modules/naive_bayes.html

[16]https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.BernoulliNB.html

[17]https://pythonbasics.org/matplotlib-bar-chart/