Name: Mali Pashupathi
Roll Number: 20211A05F8

Name: Kondakalla Bhoomika Reddy
Roll Number: 20211A05D6

Name: Kondaveeti Bhargav
Roll Number:20211A05D7

Ethereum Fraud Detection using Different Models

The Machine Learning models implemented on the Ethereum dataset are:

1. Logistic Regression
2. Decision Tree
3. Random Forest
4. Gradient Boosting
5. XGBoost
6. AdaBoost
7. K - Nearest Neighbors
8. Support Vector Machines(SVM)

1. Logistic Regression

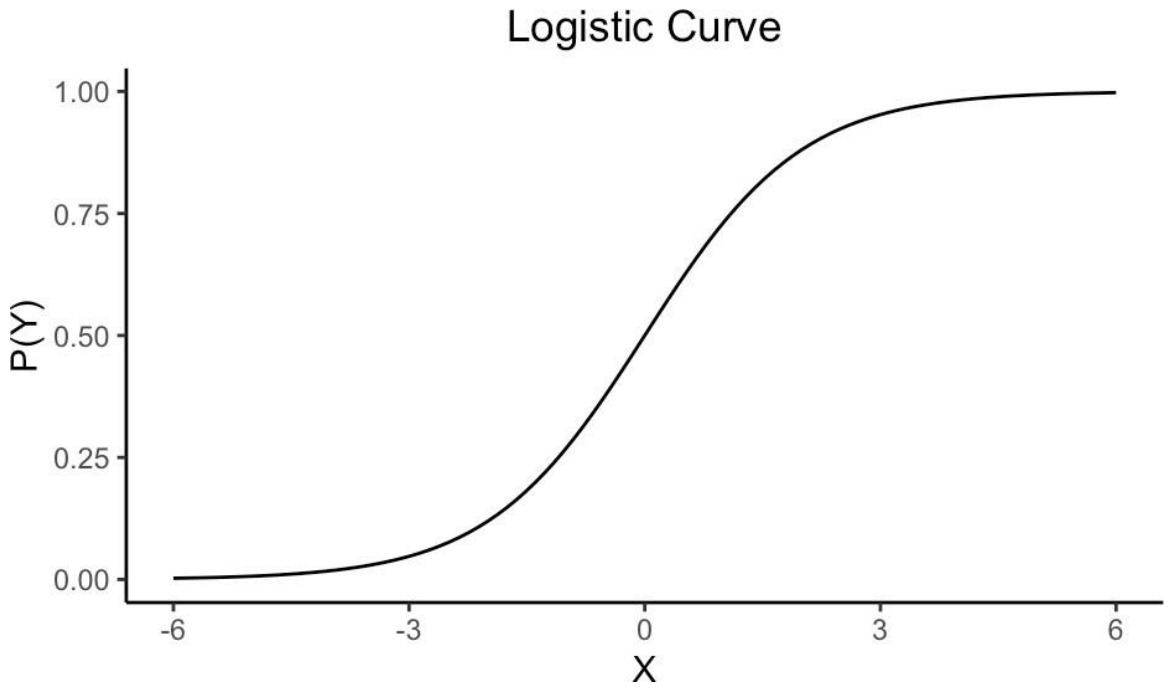
It is the appropriate regression analysis to conduct when the dependent variable is binary. The logistic regression is a predictive analysis, like all regression analyses. The continuous linear relationship between the independent variables is not necessary for logistic regression. Logistic regression requires categorical data as input, in contrast to linear regression where the dependent variable is a continuous variable.

The standard logistic function can be described before discussing logistic regression. A sigmoid function, the logistic function accepts any real input and produces an output value between 0 and 1. This is understood as input log-odds followed by output probability for the logit.

The Logistic function is of the form:

$$p(x) = \frac{1}{1 + e^{-(x-\mu)/s}}$$

Here, **μ** is mid point of the curve and **s** is a scale parameter.



Logistic Regression on Ethereum Dataset is one of the best suited mode. Because here we just need to predict wether the transaction is fraud or not. According to the definition of the Logistic Regression, if we choose a flag variable, the output of the Ethereum Dataset always falls under 0 or 1.

EDA, Preprocessing, Building model and getting the outcome are the steps followed.

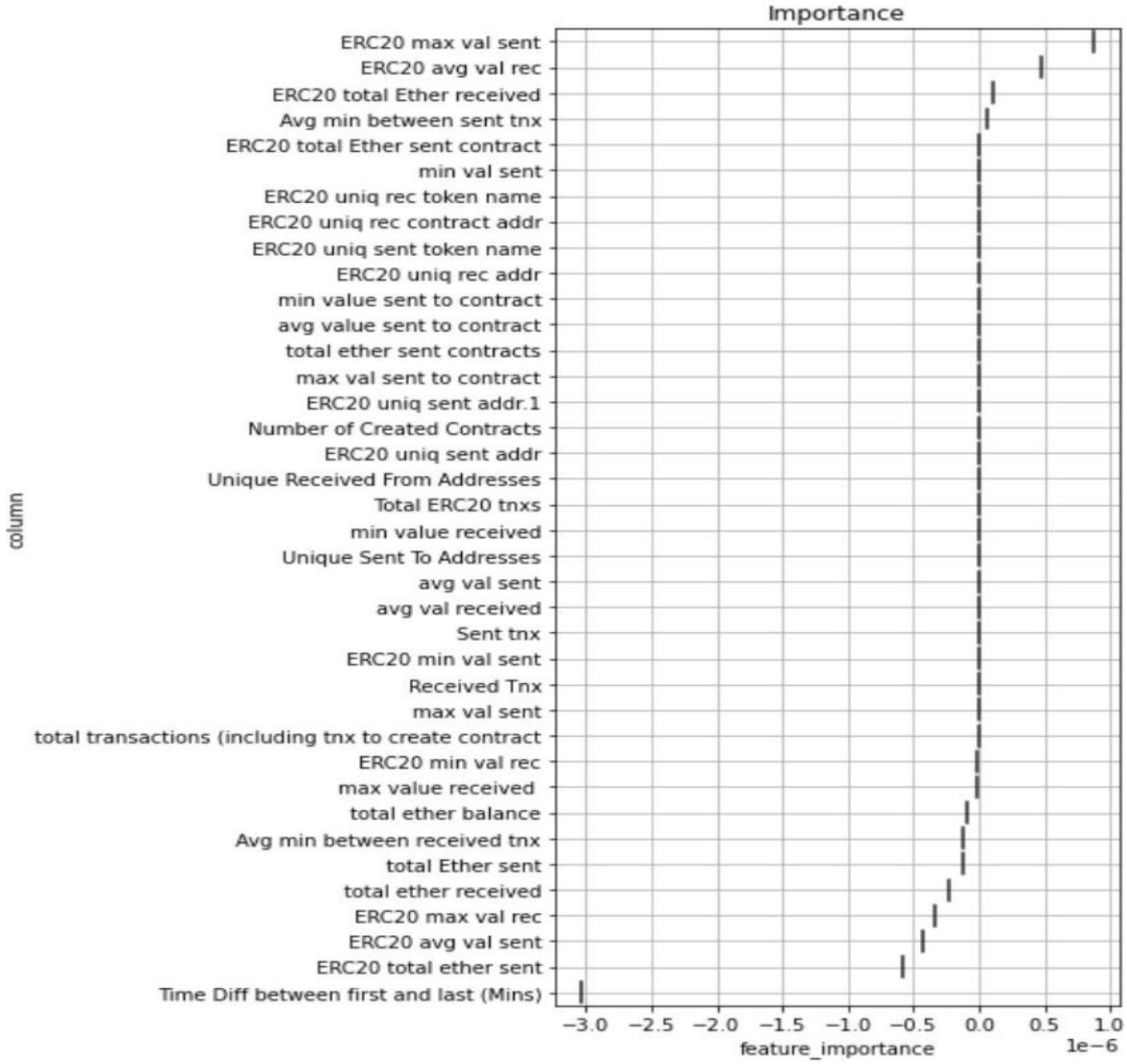
```
%%time
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression()
lr.fit(X_train, y_train)
pred = lr.predict(X_test)
print("Classification Report: ")
print(classification_report(y_test, pred))
visualize_importance(lr.coef_[0], X_train)
```

The above is the code for the implementation of Logistics Regression.

Classification Report:

	precision	recall	f1-score	support
0	0.81	1.00	0.89	1542
1	0.97	0.15	0.25	427
accuracy			0.81	1969
macro avg	0.89	0.57	0.57	1969
weighted avg	0.84	0.81	0.75	1969

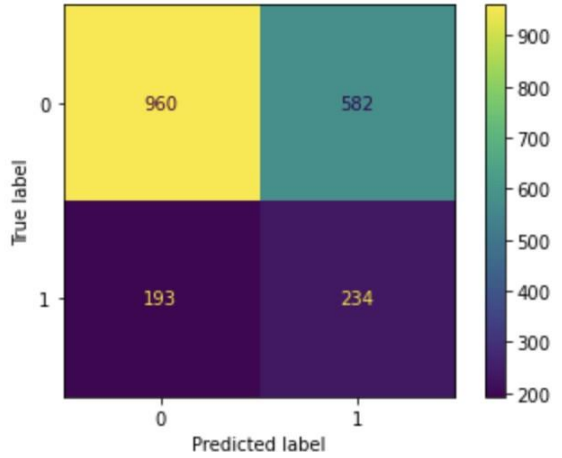
The feature importance of the columns are as follows:



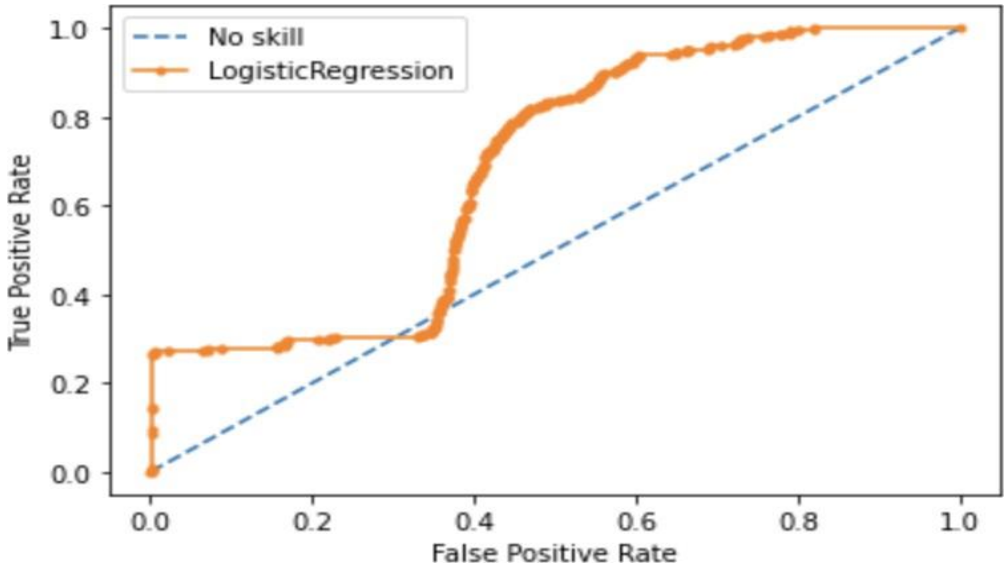
The confusion matrix of Linear regression is as follows:

```
print("Confusion matrix of Linear Regression: ")
print(confusion_matrix(y_test, pred))
norm_test_f = norm.transform(X_test)
plot_confusion_matrix(lr, norm_test_f, y_test)
```

```
Confusion matrix of Linear Regression:
[[1529   13]
 [  14  413]]

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f8fd6c69280>
```



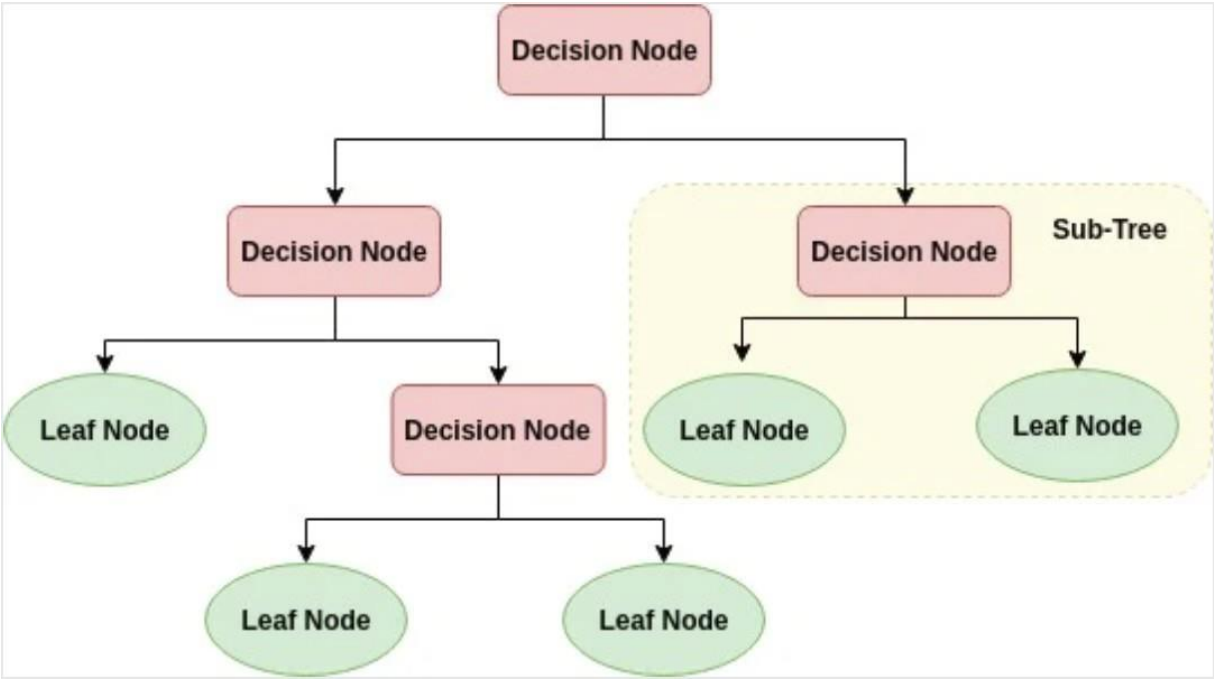
The graph between True Positive rate and False positive rate is as follows:



2. Decision Tree

Decision Tree is a Supervised Machine Learning Algorithm that uses a set of rules to make decisions, similarly to how humans make decisions. Decision trees are often referred to as CART algorithms: Classification and Regression Trees since they are capable of both classification and regression tasks. The idea behind Decision Trees is to repeatedly partition the dataset until all the data points that belong to each class are isolated by using the dataset features to produce yes/no questions.
You add a node to the tree each time you ask a question. Additionally, the root node is the first node. A question's answer divides the dataset according to the importance of a feature and adds additional nodes. The final nodes formed are known as leaf nodes if you want to terminate the process after a split. You add branches and divide the feature space into distinct regions each time you provide an answer. All of the data points on one branch of the tree indicate that the answer to the query that the rule in the preceding node implies is Yes. The remaining data points are located at a node on the other branch.

Below is the image of how Decision Tree works:



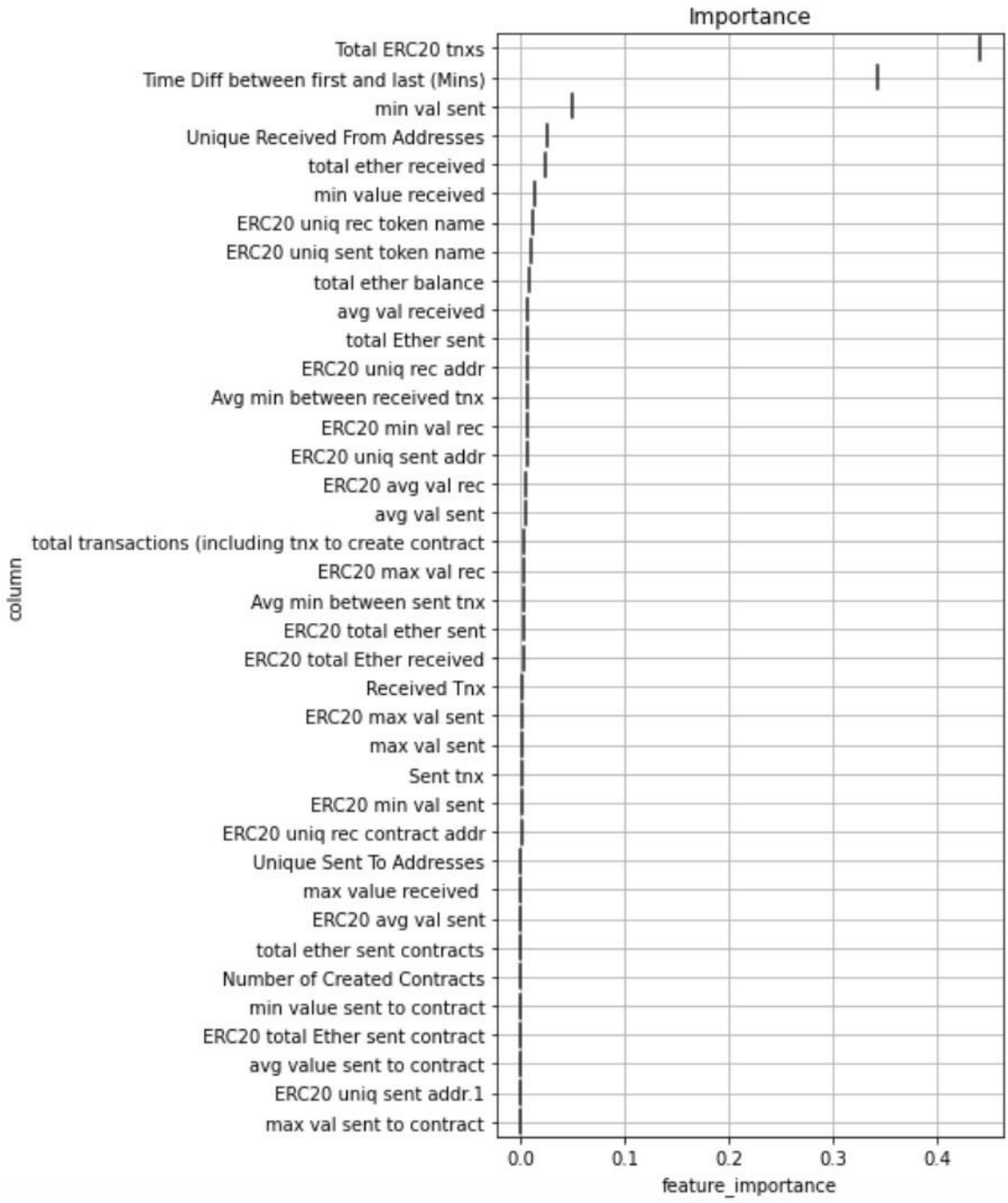
The code for implementation of Decision Tree is as follows:

```
%%time
from sklearn.tree import DecisionTreeClassifier
dtree = DecisionTreeClassifier()
dtree.fit(X_train, y_train)
pred = dtree.predict(X_test)
print("Classification Report: ")
print(classification_report(y_test, pred))
visualize_importance(dtree.feature_importances_, X_train)
```

The classification report of the Decision Tree is as follows:

```
Classification Report:
              precision    recall  f1-score   support

           0       0.98      0.96      0.97      1542
           1       0.88      0.94      0.91       427

    accuracy                           0.96      1969
   macro avg       0.93      0.95      0.94      1969
weighted avg       0.96      0.96      0.96      1969
```

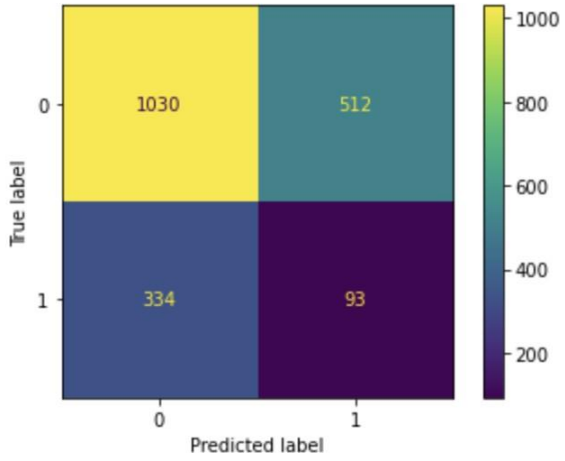
The feature importance of the columns are:



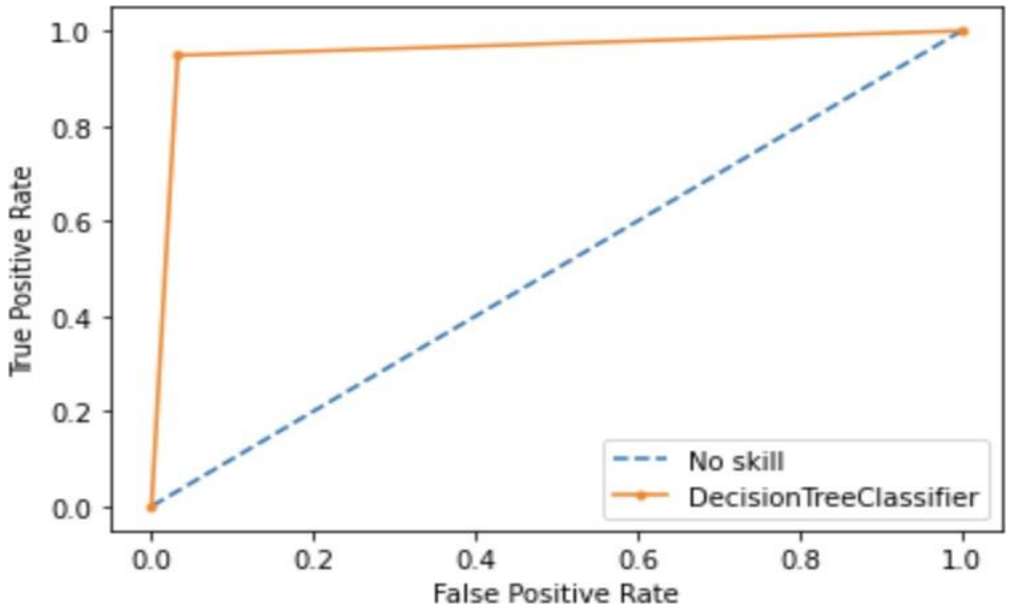
The confusion matrix of Decision Tree is as follows:

```
print("Confusion matrix of Decision Tree: ")
print(confusion_matrix(y_test, pred))
plot_confusion_matrix(dtree, norm_test_f, y_test)
```

```
Confusion matrix of Decision Tree:
[[1529   13]
 [  14  413]]

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f8fe40ca520>
```



The graph between True Positive rate and False positive rate is as follows:



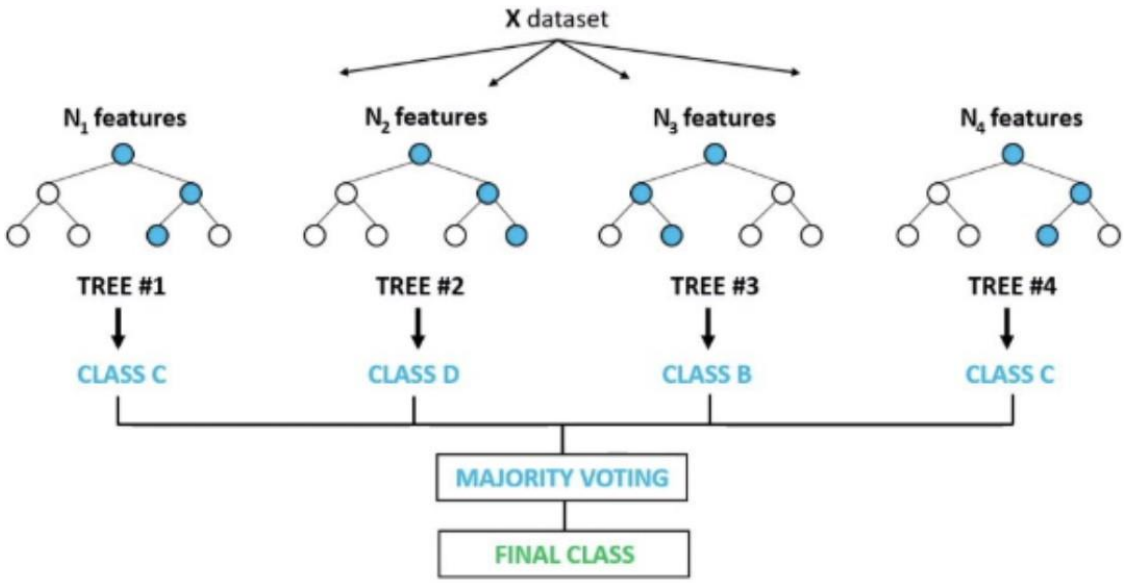
3. Random Forest

It constructs multiple decision treesat training time. The class that the majority of the trees chose is the output of the random forest for classification problems. The mean or average prediction of each individual tree is returned for regression tasks. Each and every tree in the random forest spits out a class forecast, and the classification that receives the most votes becomes the prediction made by our model. As long as they don't consistently all make the same mistake, trees defend one another from their own mistakes. Many trees will be right while some may be mistaken, allowing the trees to move together in the appropriate direction.

The pre-requisites of Random Forest to perform well:

1. In order for models created utilizing those attributes to perform better than guesswork, there must be some real signal in those features.
2. Low correlations between the predictions (and thus the mistakes) of the separate trees are required.

Below is the image of how Random Forest interacts with dataset internally:



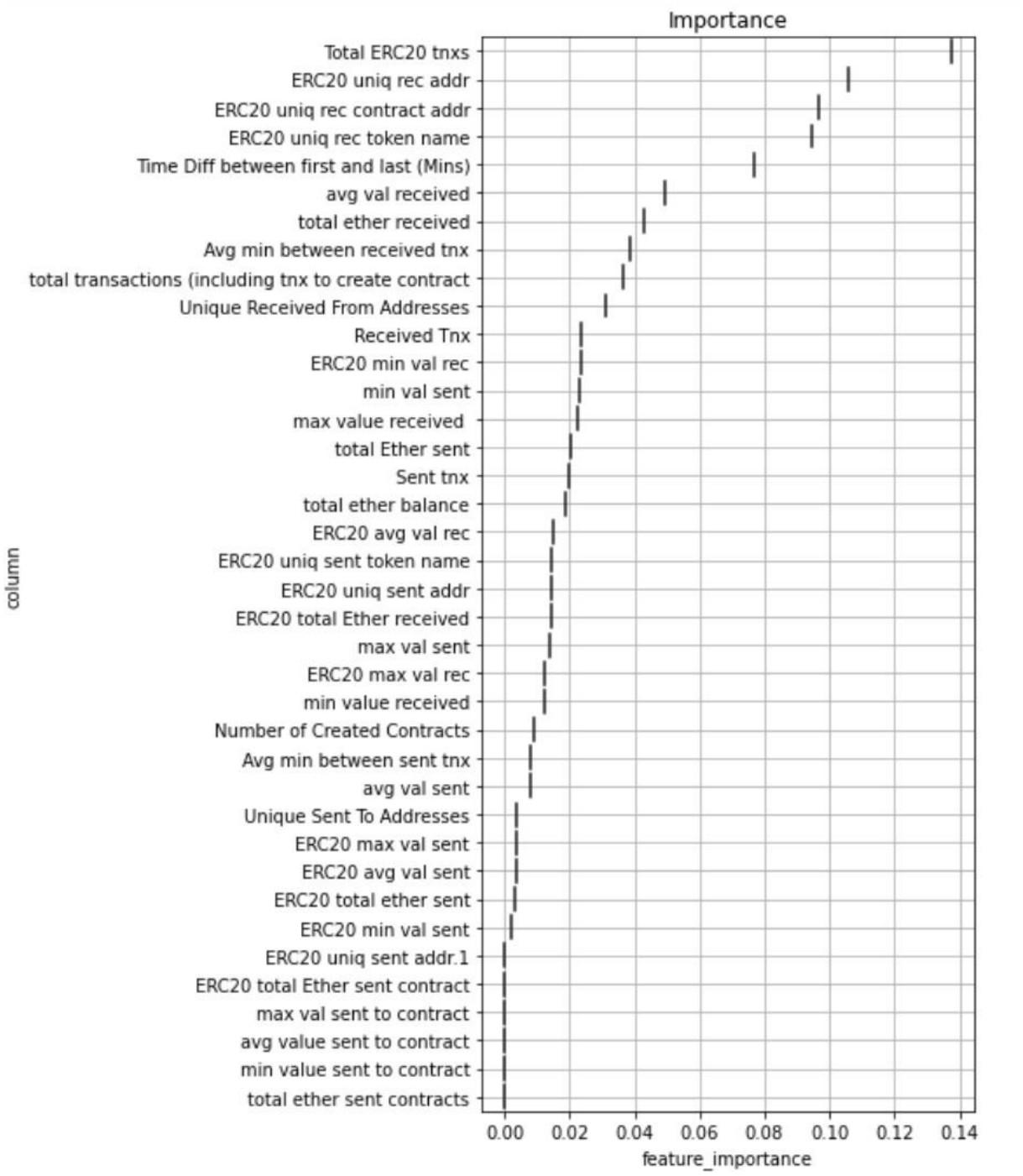
The code for implementation of Random Forest:

```
%%time
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier()
rf.fit(X_train, y_train)
pred = rf.predict(X_test)
print("Classification Report: ")
print(classification_report(y_test, pred))
visualize_importance(rf.feature_importances_, X_train)
```

The classification report of the Random Forest:

```
Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      1542
           1       0.96      0.96      0.96       427

    accuracy                           0.98      1969
   macro avg       0.97      0.97      0.97      1969
weighted avg       0.98      0.98      0.98      1969
```

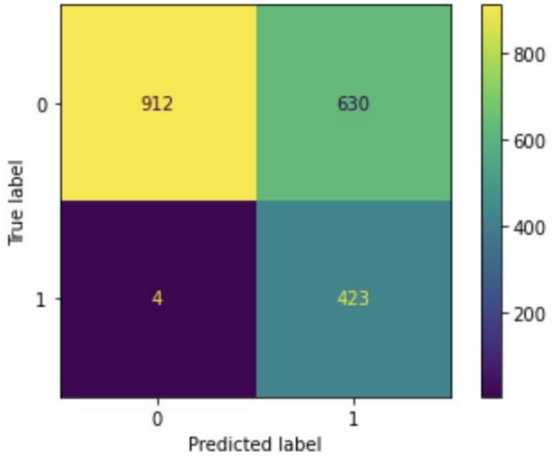
The feature importance of the columns are:



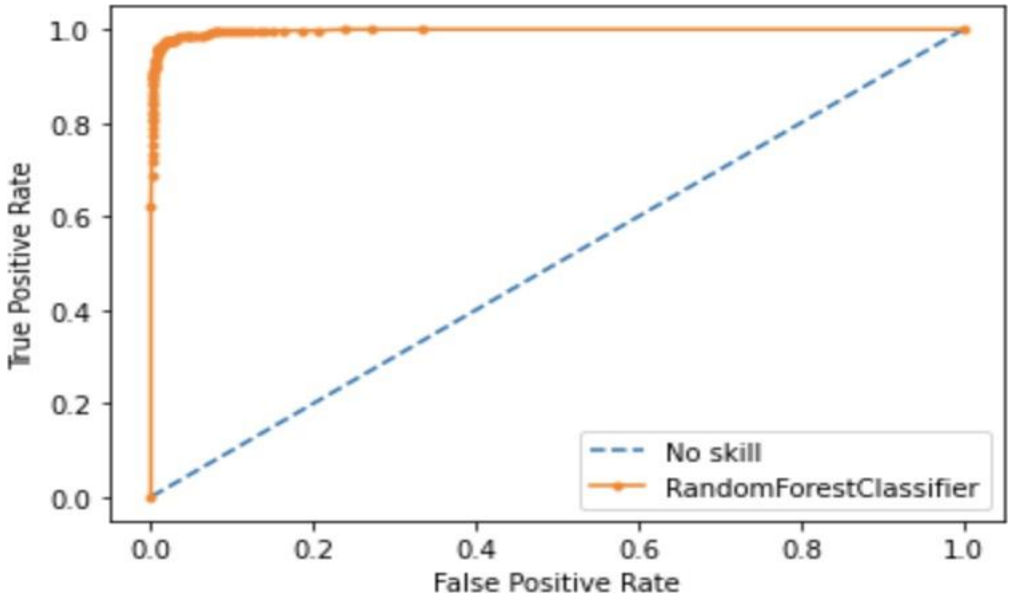
The confusion matrix of the Random Forest:

```
print("Confusion matrix of Random Forest: ")
print(confusion_matrix(y_test, pred))
plot_confusion_matrix(rf, norm_test_f, y_test)
```

```
Confusion matrix of Random Forest:
[[1529   13]
 [  14  413]]

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f8fe4093c40>
```

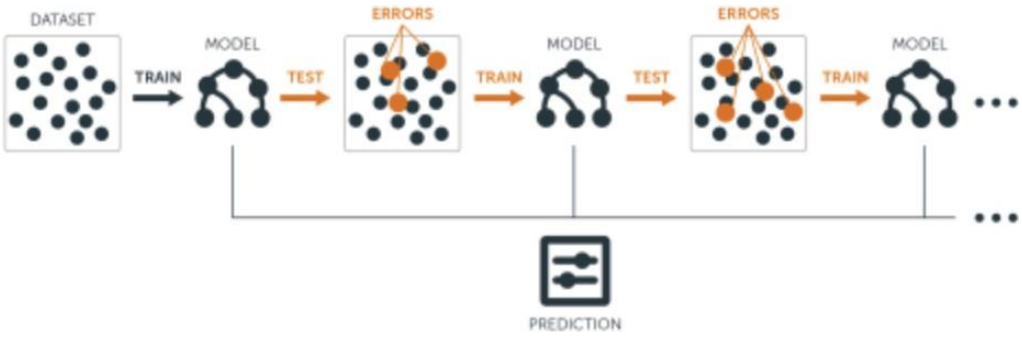


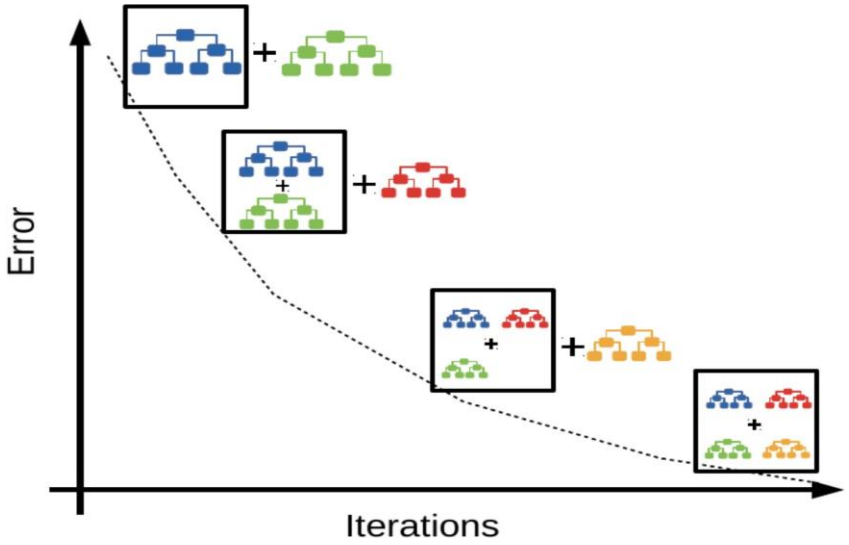
The graph between True Positive rate and False positive rate is as follows:



4. Gradient Boosting

Gradient Boosting actually make predictions on the predecessors residuals. The first thing Gradient Boosting does is that is starts of with a Dummy Estimator. Next, an estimator is trained to forecast the residuals of the first predictor rather than a new estimator to predict the target using the data. This predictor typically takes the form of a Decision Tree with restrictions, like the maximum permitted number of leaf nodes. The leaf node's value is determined by averaging the residuals from all occurrences that have residuals in the same leaf node. It creates a new prediction for each instance by adding the value of the base estimator to the instance's expected residual value as predicted by the decision tree. The residuals between the predicted and actual values are then calculated once more. This procedure is repeated until the residual difference is very minimal or a certain threshold is met. It gives the instance to every decision tree it generates, adds the predictions it generates, and adds the value of the base estimator to create a forecast for an unseen instance.





The code for implementation of Gradient Boosting:

```
%%time
from sklearn.ensemble import GradientBoostingClassifier
gb = GradientBoostingClassifier()
gb.fit(X_train, y_train)
pred = gb.predict(X_test)
print("Classification Report: ")
print(classification_report(y_test, pred))
visualize_importance(gb.feature_importances_, X_train)
```

The classification report of Gradient Boosting:

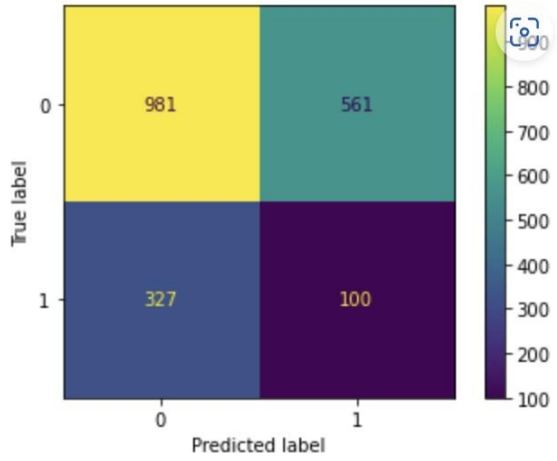
```
Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.98      0.98      1542
           1       0.92      0.97      0.95       427

    accuracy                           0.98      1969
   macro avg       0.96      0.97      0.96      1969
weighted avg       0.98      0.98      0.98      1969
```

The feature importance of the columns when using Gradient Boosting are:

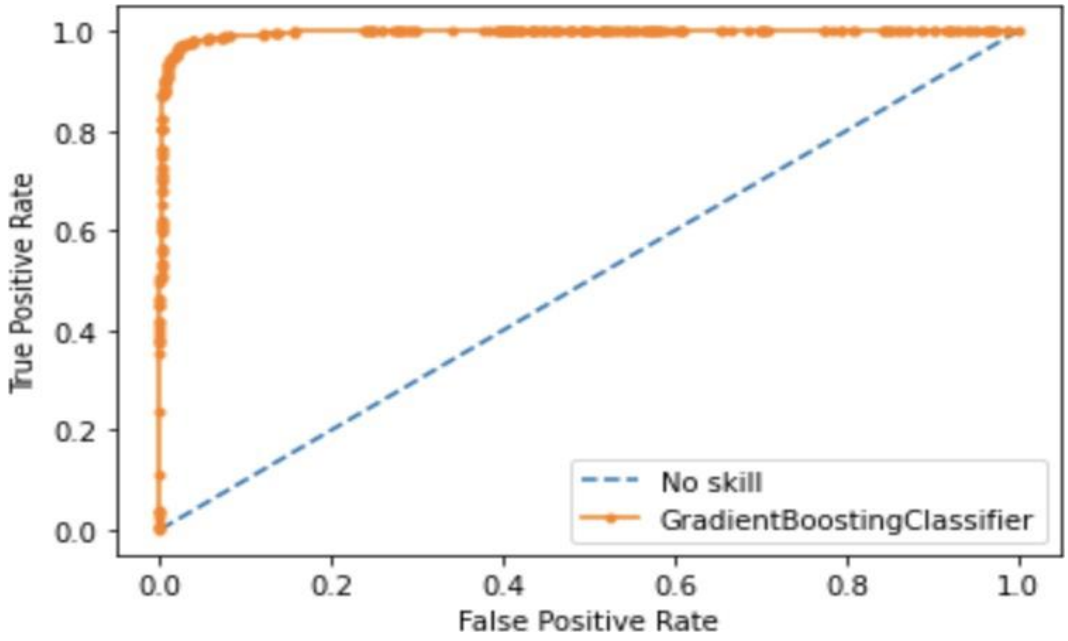
The confusion matrix of Gradient Boosting:

```
print("Confusion matrix of Gradient Boosting: ")
print(confusion_matrix(y_test, pred))
plot_confusion_matrix(gb, norm_test_f, y_test)
```

```
Confusion matrix of Gradient Boosting:
[[1529   13]
 [  14  413]]

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f7f98709a60>
```



The graph between True Positive rate and False positive rate is as follows:



5. XGBoost classifier

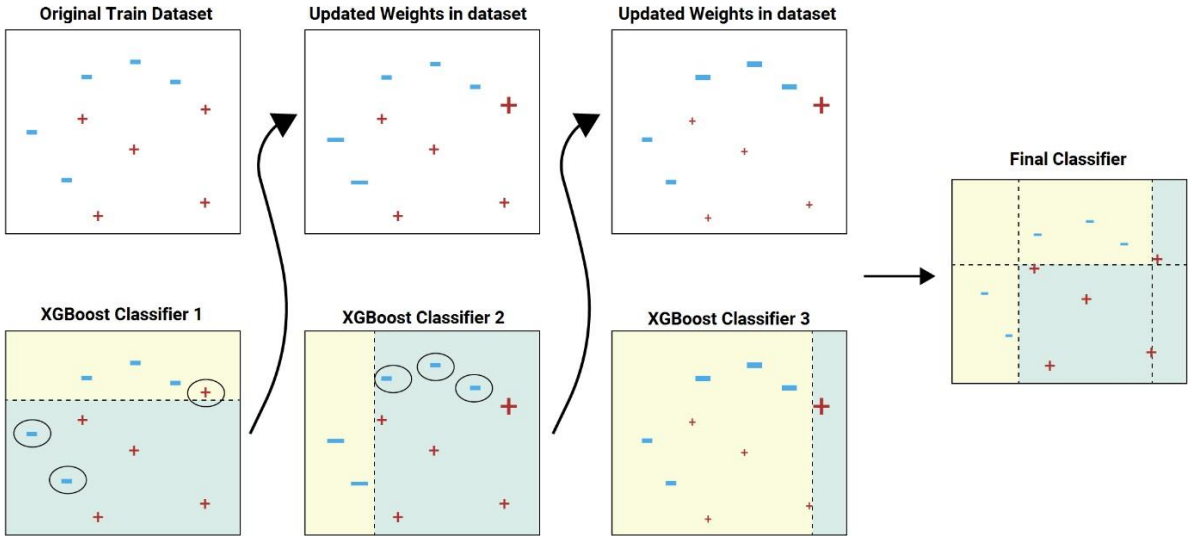
XGBoost classifier's speed and performance are unparalleled and it consistently outperforms any other algorithms aimed at supervised learning tasks.

The Basic and main requirements for XGBoost to perform well are:

1. Numeric features should be scaled.
2. Categorical features should be encoded.

As an ensemble learning algorithm, XGBoost combines the output from numerous base learners to provide a prediction.

The image of how XGBoost works is as follows:



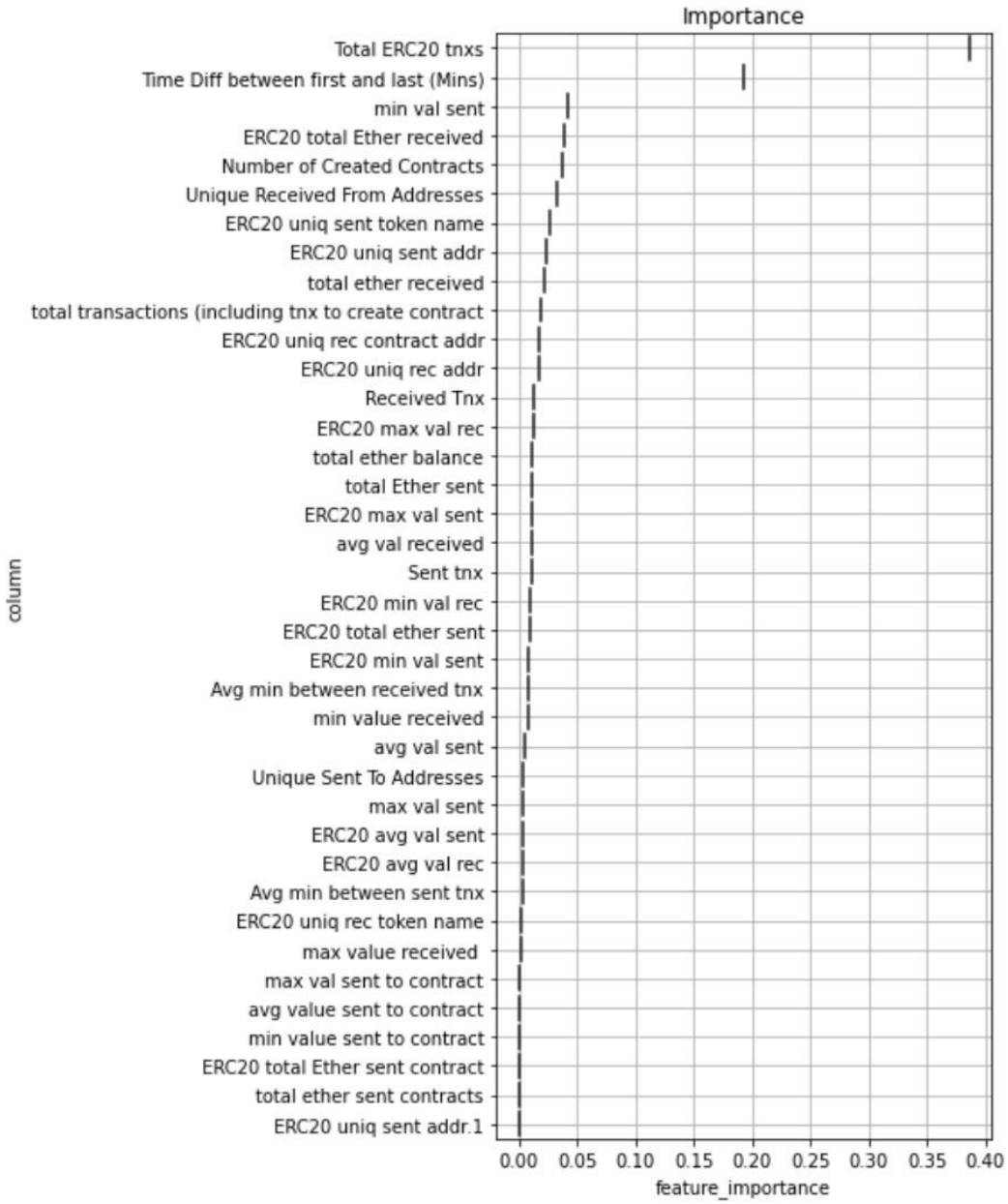
The code for implementation of XGBoost:

```
from xgboost import XGBClassifier
xgb = XGBClassifier()
xgb.fit(X_train, y_train)
pred = xgb.predict(X_test)
print("Classification Report: ")
print(classification_report(y_test, pred))
visualize_importance(xgb.feature_importances_, X_train)
```

The classification report of XGBoost:

```
Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.99      0.99      1542
           1       0.97      0.97      0.97       427

    accuracy                           0.99      1969
   macro avg       0.98      0.98      0.98      1969
weighted avg       0.99      0.99      0.99      1969
```

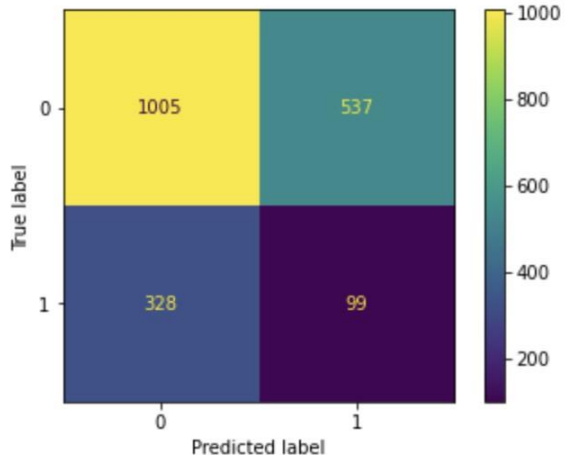
The feature importance of the columns when using XGBoost are:



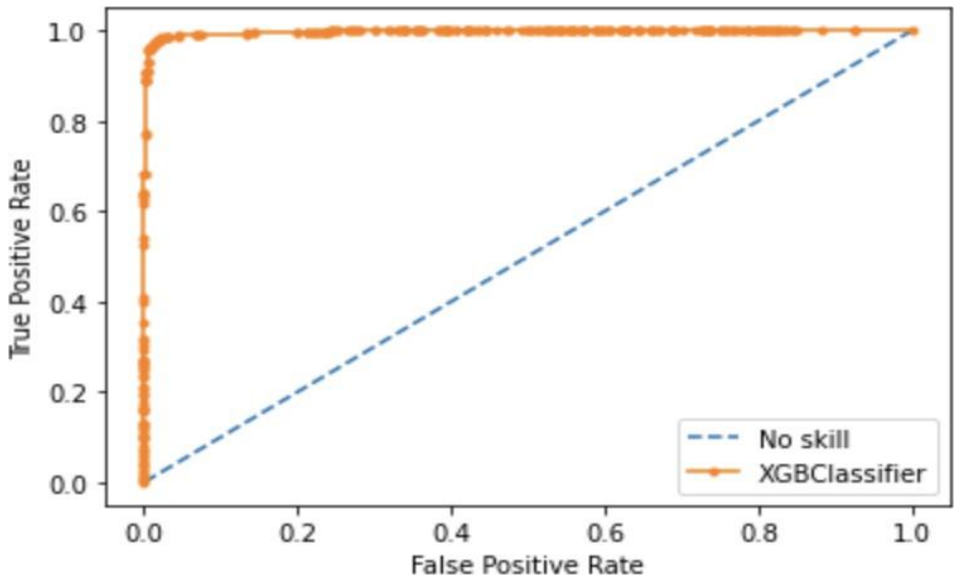
The confusion matrix of XGBoost:

```
print("Confusion matrix of XGBoost classifier: ")
print(confusion_matrix(y_test, pred))
plot_confusion_matrix(xgb, norm_test_f, y_test)
```

```
Confusion matrix of XGBoost classifier:
[[1529   13]
 [  14  413]]

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f7f8afca340>
```



The graph between True Positive rate and False positive rate is as follows:



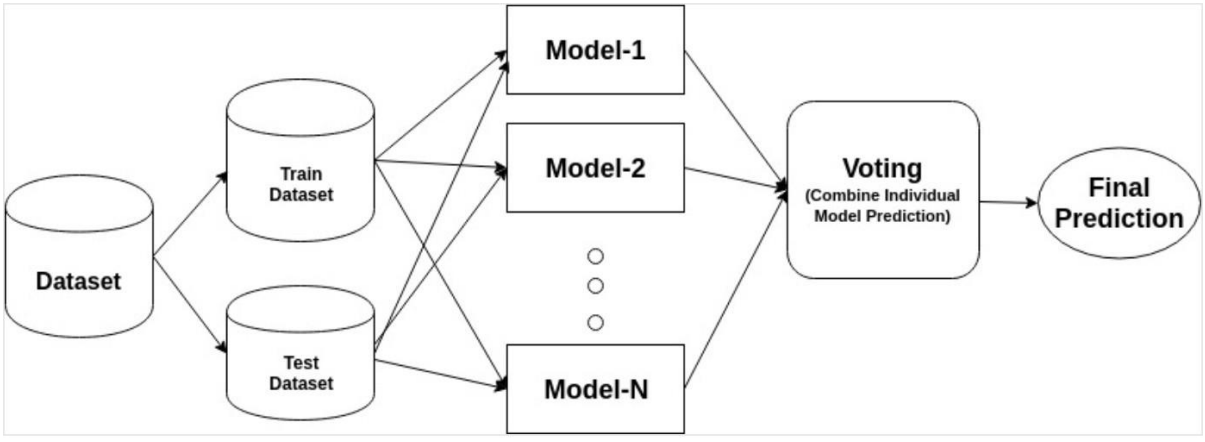
6. AdaBoost Classifier

One of the first boosting algorithms to be applied in solving techniques is AdaBoost. Adaboost enables you to create a single "strong classifier" out of several "weak classifiers". The fundamental idea underlying Adaboost is to train the data sample and set the classifier weights in each iteration in a way that provides accurate predictions of uncommon observations. Imagine training a Decision Tree method to generate predictions on our training data in order to construct an AdaBoost classifier. Then the weight of the misclassified training instances is increased. The second classifier is trained, accepts the modified weights, and again executes the process. We end up increasing the weights of the cases that were incorrectly classified at the conclusion of each model prediction so that the following model performs better on them, and so on. AdaBoost continually improves the ensemble by adding predictions to it. This algorithm's main drawback is that it prevents parallelization of the model because each predictor can only be learned after the preceding one has been trained and evaluated.

Like many other models AdaBoost also should meet two conditions to perform well:

1. On a variety of weighed training instances, the classifier should be trained interactively.
2. It strives to minimize training error in order to offer the best match possible for these instances in each iteration.

Below is the representation of how actually AdaBoost works:



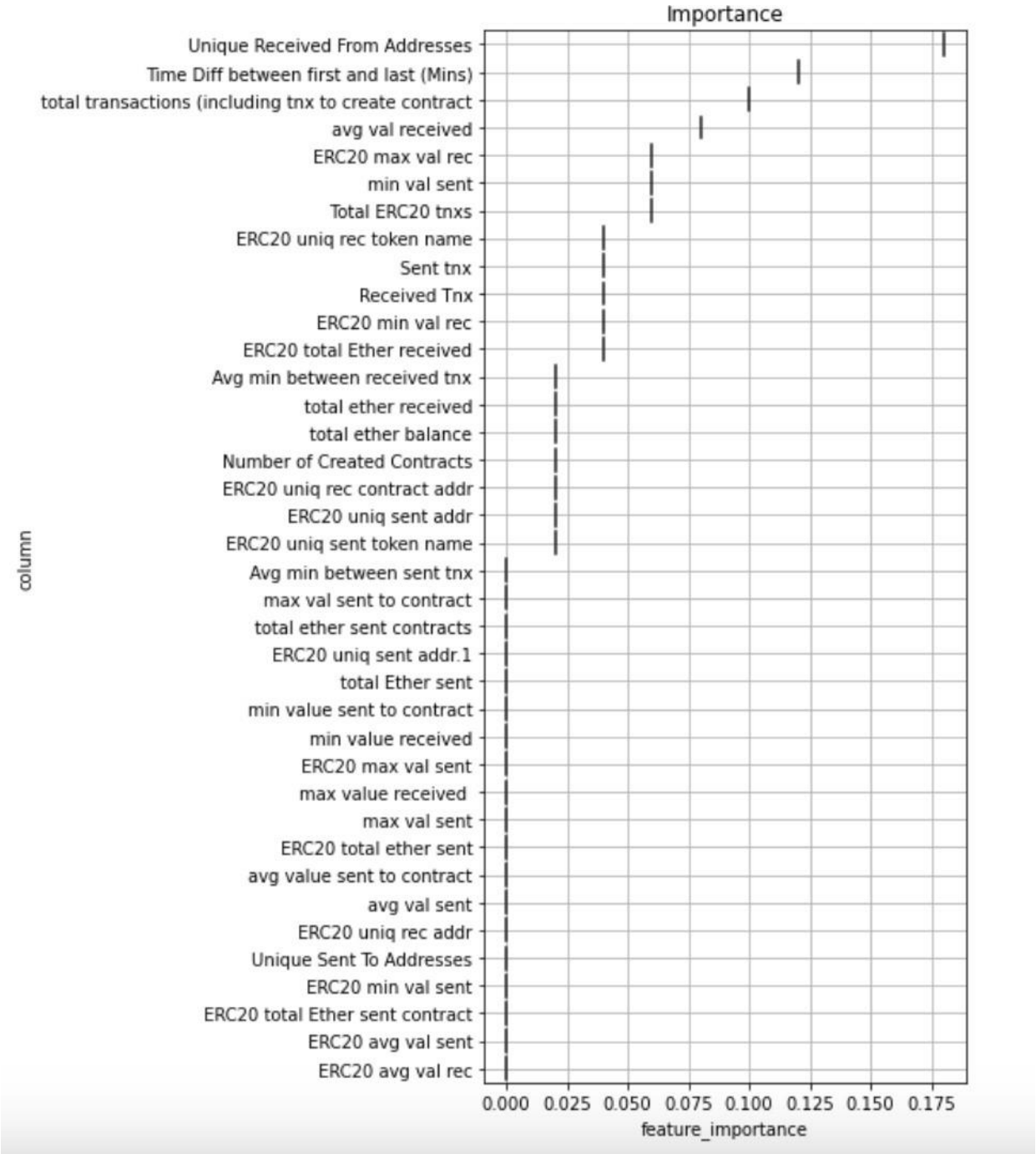
The code for implementation of AdaBoost:

```
from sklearn.ensemble import AdaBoostClassifier
num_trees = 50
seed=7
ada = AdaBoostClassifier(n_estimators=num_trees, random_state=seed)
ada.fit(X_train, y_train)
pred = ada.predict(X_test)
print("Classification Report: ")
print(classification_report(y_test, pred))
```

The classification Report of AdaBoost:

```
Classification Report:
              precision    recall  f1-score   support

           0       0.99      0.96      0.98      1542
           1       0.88      0.96      0.92       427

    accuracy                           0.96      1969
   macro avg       0.93      0.96      0.95      1969
weighted avg       0.97      0.96      0.96      1969
```

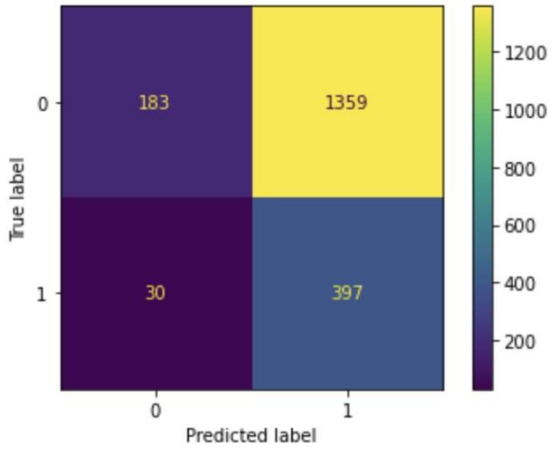
The feature importance of the columns when using AdaBoost are:



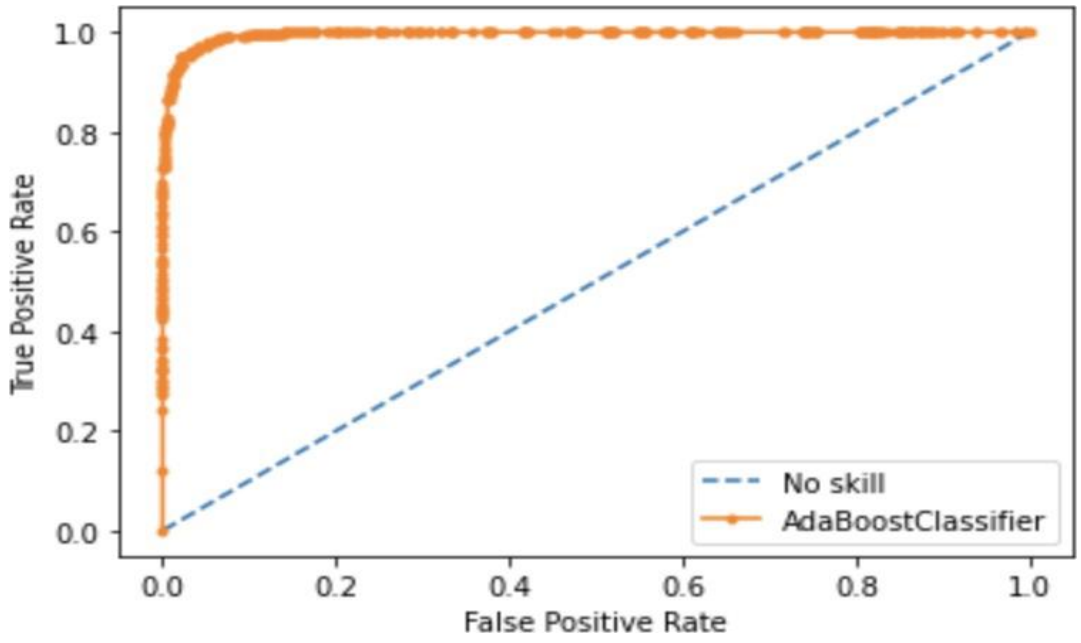
The confusion matrix of AdaBoost:

```
print("Confusion matrix for ADABoost classifier: ")
print(confusion_matrix(y_test, pred))
plot_confusion_matrix(ada, norm_test_f, y_test)
```

```
Confusion matrix for ADABoost classifier:
[[1486   56]
 [  16  411]]

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f7f7864f100>
```



The graph between True Positive rate and False positive rate is as follows:

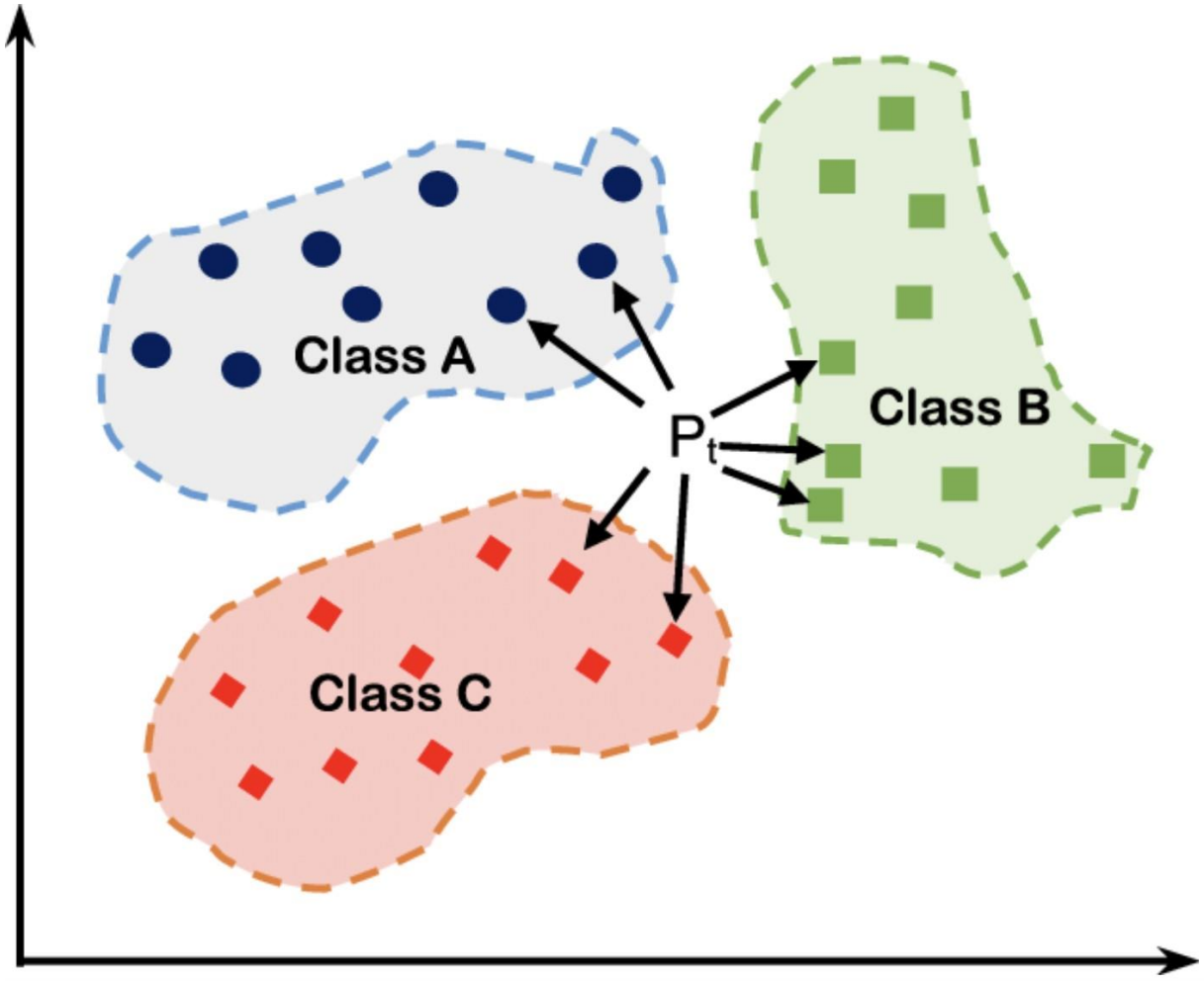


7. K - Nearest neighbors

A straightforward, user-friendly supervised machine learning approach known as the k-nearest neighbors (KNN) algorithm can be utilized to resolve classification and regression issues. KNN uses the arithmetic we may have learned as children—calculating the distance between points on a graph—to encapsulate the idea of similarity (also called distance, proximity, or closeness).

Choosing the correct **K** value:

1. When the K value decreases(=1), the predictions become less stable.
2. Increasing K value means greater the stability. After certain limit with the accuracy becomes constant we need to know that we increased K value to a greater extent.
3. K value usually be an odd number!!!!



The code for implementation of KNN:

```
knn = KNeighborsClassifier(n_neighbors=15, metric='minkowski', p= 2)
knn.fit(X_train, y_train)
pred = knn.predict(X_test)
print("Classification Report: ")
print(classification_report(y_test, pred))
```

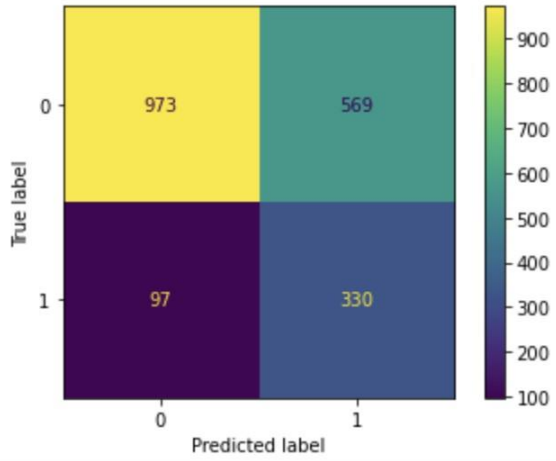
Classification Report of KNN:

```
Classification Report:
              precision    recall  f1-score   support

           0       0.96      0.85      0.90      1542
           1       0.62      0.88      0.73       427

    accuracy                           0.86      1969
   macro avg       0.79      0.87      0.82      1969
weighted avg       0.89      0.86      0.86      1969
```

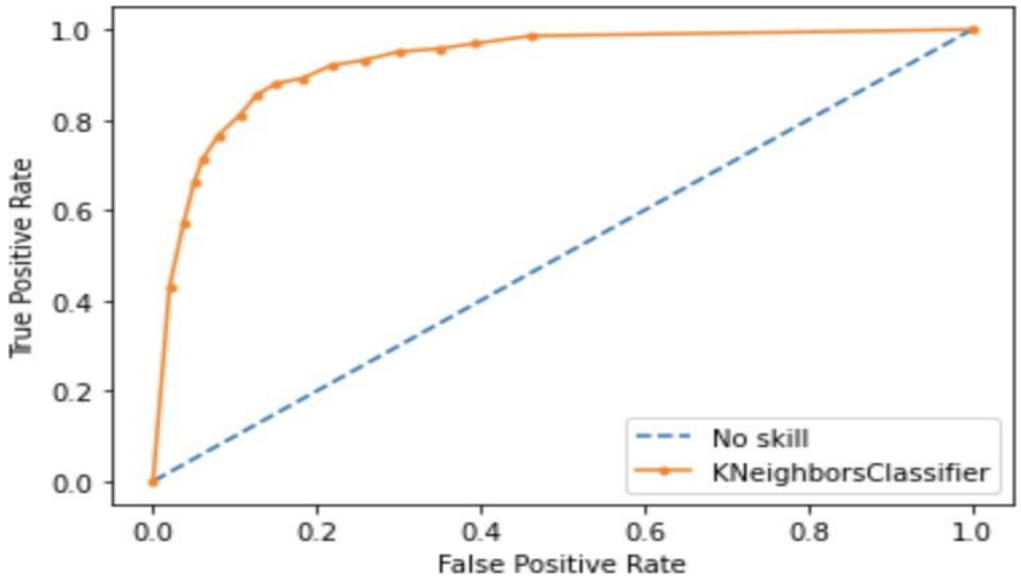
Confusion matrix of KNN:

```
print("Confusion matrix of K-Nearest neighbors: ")
print(confusion_matrix(y_test, pred))
plot_confusion_matrix(knn, norm_test_f, y_test)
```

```
Confusion matrix of K-Nearest neighbors:
[[1311  231]
 [  51  376]]

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f7f899202b0>
```



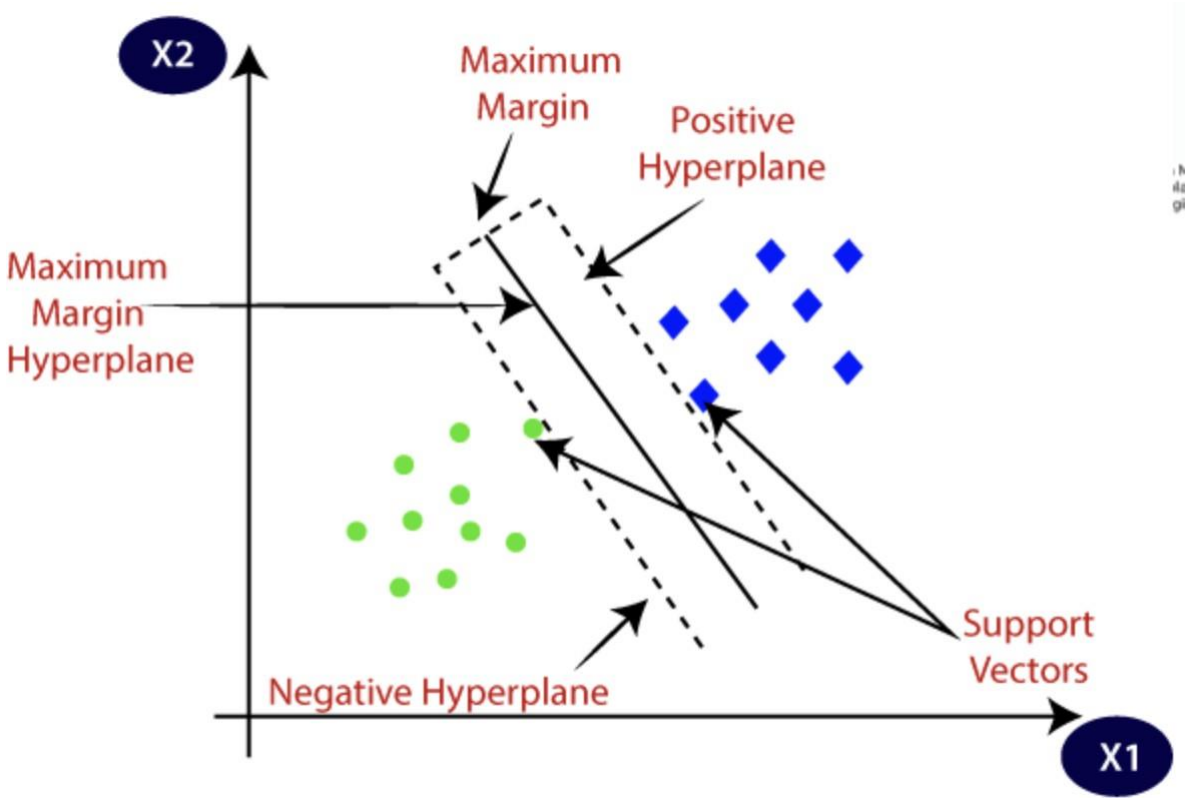
The graph between True Positive rate and False positive rate is as follows:



8. Support Vector Machines(SVM)

Finding a hyperplane in an N-dimensional space (N is the number of features) that categorizes the data points clearly is the goal of the support vector machine algorithm. There are a variety of different hyperplanes that might be used to split the two classes of data points. Finding a plane with the greatest margin—that is, the greatest separation between data points from both classes—is our goal. Maximizing the margin distance adds some support, increasing the confidence with which future data points can be categorised. Decision boundaries known as hyperplanes assist in categorizing the data points. Different classes can be given to the data points that fall on each side of the hyperplane. Additionally, the amount of features affects how big the hyperplane is.

Support vectors are data points that are closer to the hyperplane and have an impact on the hyperplane's position and orientation. By utilizing these support vectors, we increase the classifier's margin. The hyperplane's location will vary if the support vectors are deleted.



The code for implementation of SVM:

```
svm = LinearSVC()
svm.fit(X_train, y_train)
pred = svm.predict(X_test)
print("Classification Report: ")
print(classification_report(y_test, pred))
```

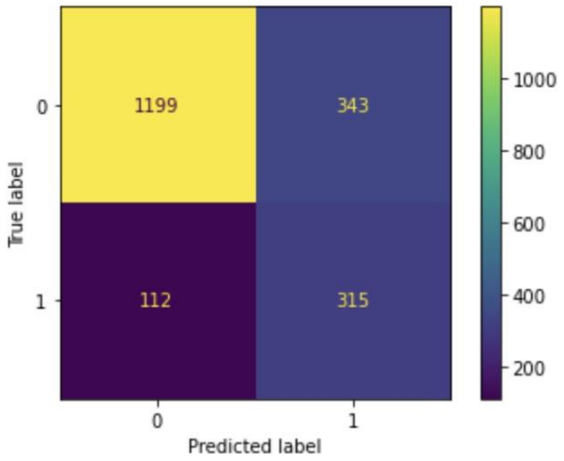
Classification report of SVM:

```
Classification Report:
              precision    recall  f1-score   support

           0       0.88      0.71      0.79      1542
           1       0.39      0.66      0.49       427

    accuracy                           0.70      1969
   macro avg       0.64      0.69      0.64      1969
weighted avg       0.78      0.70      0.72      1969
```

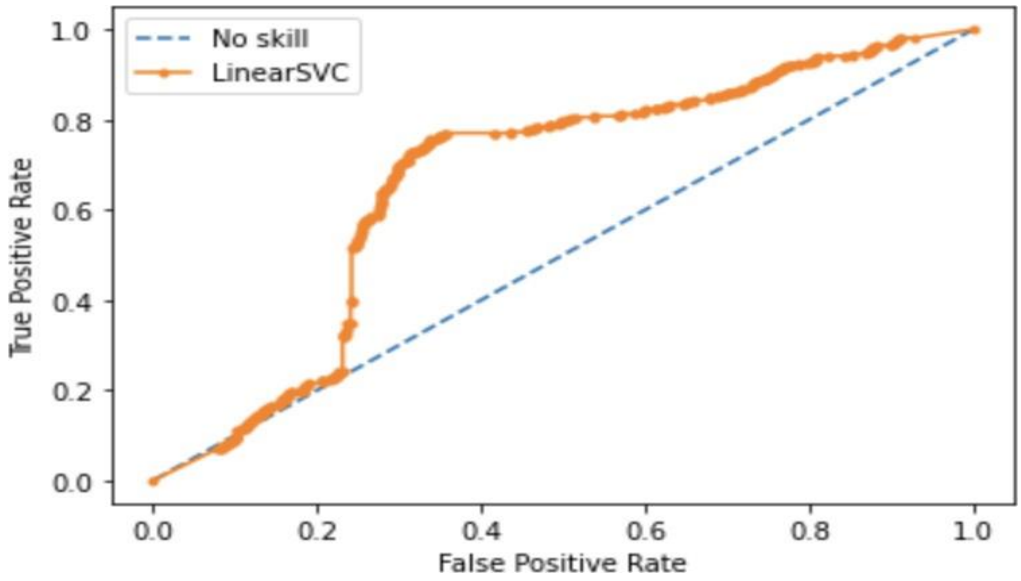
Confusion matrix of SVM:

```
print("Confusion matrix of SVM: ")
print(confusion_matrix(y_test, pred))
plot_confusion_matrix(svm, norm_test_f, y_test)
```

```
Confusion matrix of SVM:
[[1486   56]
 [  16  411]]

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f7f60226be0>
```



The graph between True Positive rate and False positive rate is as follows:



Model	Accuracy
Logistic Regression	81%
Decision Tree	96%
Random Forest	98%
Gradient Boosting	98%
XGBoost	99%
AdaBoost	96%
KNN	86%
SVM	70%

Table 1

Model	AUC value
Logistic Regression	0.572
Decision Tree	0.957
Random Forest	0.975
Gradient Boosting	0.973
XGBoost	0.979
AdaBoost	0.963
KNN	0.865
SVM	0.686

Table 2

Why I have only chosen Decision Tree, Random Forest, Gradient Boosting, XGBoost, AdaBoost as the models for my dataset?

1. Accuracy

By seeing the results of Table-1, we can easily come to a conclusion that Logistic Regression, KNN, SVM performs poor compared to the other algorithms. When we are working with the real world one always thinks of getting the accurate results for a problem. So, the above mentioned five models (Decision Tree, Random Forest, Gradient Boosting, XGBoost, AdaBoost) satisfies this constraint.

2. AUC values

AUC means area under the ROC curve.
ROC curve - It is a graph showing the performance of a classification model at all classification thresholds.

By observing the values of Table 2, only 50% and 60% predicted values of Linear Regression and SVM falls under the curve. The means the False Positive rate while prediction is very high.
Compared to other five models (RF, DT, GB, XGB, ADA), the value of KNN is also about 10% lower.So, it is safe to stick to the other five models while drawing conclusions.

Which is the perfect model for this dataset?

On comparing all the results obtained (Accuracy, Macro Average, Weighted Average, F1 score, Precision, Recall, Support, AUC values), it is safe to say that **XGBoost** performs well while predicting. The other four models also performs equally with XGBoost, But just falling behindwith 1 or 2%.