Feature-based Classification of Free and Paid Android Applications on Play Store

Anand Verma - averma8@umd.edu

Abstract - **This paper builds on choosing the right classification model to predict if an Android App is free or paid based on its features. Decision Tree Classifier, Logistic Regression, K-Nearest Neighbor classifier, Naive Bayes Classifier, and Linear Discriminant analysis model were trained and tested and their accuracy was evaluated using different approaches. After testing different classification models on the unseen data, it was observed that the K-nearest neighbor model outperformed all the other classifiers with an accuracy of 82.15% and an AUC of 0.88. This paper concludes by defining how the K-NN was able to generate better accuracy than other classifiers. This study deals with a real-world scenario for which application to choose between free and Paid.**

I. INTRODUCTION

Classification is a supervised machine learning method where the model tries to predict the correct label of a given input data. In classification, the model is fully trained using the training data, and then it is evaluated on test data before being used to perform prediction on new unseen data. Classification algorithms enable machines to make sense of big datasets by building on the foundations of statistics, mathematics, and computer science. This enables activities like spam filtering, image recognition, and medical evaluation.

This paper was built around the meticulous examination of classification performance by comparing a number of Machine Learning Models (Decision Tree, Logistic Regression, K-NN, Naive Bayes, and Linear discriminant Model). Their accuracies were calculated and their performance in predicting whether an Android App is free or paid is noted. In order to compare their performance, their accuracies on test data, recall score, precision score, and Area under the curve were compared and the decision was made about which classification model with what hyperparameters to use.

Decision trees[1] are a versatile classification model that recursively splits data based on features, creating a tree-like structure that enables straightforward interpretation whereas Logistic regression[2] is a linear model adapted for binary classification, estimating the probability of an instance belonging to a particular class. KNN[3] is a non-parametric classification algorithm that classifies instances based on the majority class of their k-nearest neighbors, making it robust for various data distributions but potentially sensitive to the choice of the distance metric. Naive Bayes[4] is a probabilistic classification algorithm based on Bayes' theorem, assuming independence among features while Linear Discriminant Analysis (LDA)[5] is a linear transformation technique that finds the optimal projection to maximize the separation between classes, making it effective for high-dimensional data and situations where the assumption of normality and equal covariance matrices is reasonable.

Column	Detail
App	Application Name
Category	Application Category
Rating	App Rating on Play Store
Reviews	Number of Reviews
Size	Size of the application
Installs	Number of Installation
Type	Free of Paid
Price	Price of application
Content Rating	Which Group to show
Genres	App Genre
Last Updated	Last Update Date
Current Ver	Current Version
Android Ver	Version compatibility

TABLE I: Information about the Google Play Store Data Set

The dataset used in this paper is the Google Play Store Dataset which contains information for 10841 Play Store applications. Each instance comprises different features of the application. The ultimate goal of this study was to predict whether an app is free or paid.

The paper uses a number of Python's data analysis libraries such as pandas[6] and scikit-learn[7] which provide in-depth information on the execution of different classification models. Libraries like Seaborn and MatPlotLib were also used to visually represent the results obtained.

The Introduction section covers the basic description of the dataset and the models used in this journal. The methodology section covers what steps were taken to make the dataset ready for training and testing. After cleanup, all the classification models were trained and tested using the best case of hyperparameters,[8] and their accuracies were drawn, A confusion matrix and classification reports were generated and an ROC curve was drawn for each of them. The result section covers the different performance metrics obtained and the discussion section covers which classification Model outperformed the other models and was the best fit for the type prediction of Applications.

II. METHODOLOGY

This paper primarily focuses on achieving classification efficiency on the Google Play Store dataset. For this study, Python version 3.11.1 and for "Pandas" version 2.1.0 were used. To read the CSV file, `read_csv()` method of Pandas library was used and the name of the file to be read is passed as an argument to the method:

```
import pandas as pd
playStoreDataFrame =
pd.read_csv("googleplaystore.csv")
```

Importantly, scikit-learn library was installed, and packages such as tree and preprocessing packages were imported. With the below commands in the terminal, scikit-learn was installed and to check what version was installed, the next command was used.

```
pip install -U scikit-learn
python -m pip show scikit-learn
```

Important packages like `tree` from sklearn, `LogisticRegression` from `linear_model`, `KNeighborsClassifier` from `neighbors` package, `naive_bayes` , and `LinearDiscriminantAnalysis` from `discriminant_analysis` of sklearn were installed using the below command:

```
from sklearn import tree

from sklearn.linear_model
import LogisticRegression

from sklearn.neighbors
import KNeighborsClassifier

from sklearn import naive_bayes

from sklearn.discriminant_analysis
import LinearDiscriminantAnalysis
```

To get the confusion matrix, precision score, accuracy score, recall score and classification report, their module was imported from `sklearn.metricx` library and to generate the ROC curve with AUC `roc_curve` and `auc` were imported from `metrics` package using the below command:

```
from sklearn.metrics import
confusion_matrix, accuracy_score,
recall_score, precision_score, classification_report

from sklearn.metrics import roc_curve,auc
```

While observing the Google Play store Dataset, it was observed that the dataset was totally raw and it required a cleanup to get ready for Model training and classification. A number of steps were performed initially to prepare the data for analysis:

A. Preparing Target Variable

The primary goal of this study was to predict whether the Android app is free or paid. The column `'Type'` contained two values, `Free` and `Paid`, and was selected as a Target value. Both the `Free` and `Paid` values were encoded in 0 and 1 respectively and were put in a new column named 'Target'. This encoding simplified the representation of the target variable and made it compatible with a number of machine-learning algorithms.

```
playStoreDataFrame['Target'] =
playStoreDataFrame.apply(lambda x: 1
if x["Type"] == "Paid"
else 0, axis=1)

playStoreDataFrame.drop(columns=["Type"],
axis=1, inplace=True)
```

Once the encoding was done and the Target column was ready, the `Type` was dropped from the actual dataframe. Dropping the Target variable helps in preventing Data Leakage. If not dropped the model would essentially have access to the very information it is trying to predict, which could result in unrealistically high performance during training but poor generalization to new, unseen data.

B. Dropping Rows with Null values

After setting the `Target` variable, all the rows having empty or null values were removed. Removing cells with null values in a dataset for classification ensures data integrity, algorithm compatibility, and fair performance evaluation. To remove the empty cell rows, `dropna` method of the Pandas library was used and was provided with all the columns of the dataset as a parameter. [9]

```
playStoreDataFrame =
playStoreDataFrame.dropna(subset=
list(playStoreDataFrame.columns))
```

C. Renaming COlumns

Column names having two words in them with a space in between were then renamed, replacing space with an underscore. Removing spaces is essential for compatibility with machine learning algorithms, as models may have difficulty handling spaces in feature names. Consistent, space-free column names facilitate seamless data processing and model training.

```
playStoreDataFrame.rename(
columns = {'Content Rating' : 'Content_Rating' },
inplace = True)

playStoreDataFrame.rename(
columns={"Size":"size_Mb"}, inplace=True)
```

The `inplace` parameter is a boolean that ensures the existing dataframe is modified when set to True. It creates a new dataframe when false is set.

D. Removing Symbols from Cell Values

Columns `Installs`, `Price` and `size_MB` had symbols and letter with numeric values like `$`,`+`, `M`, `K`. These symbols and letters were then removed ensuring the cell values were converted into numeric values only. This ensured numerical consistency for classification models, preventing potential model confusion and facilitating accurate numerical processing during training.

```
playStoreDataFrame['Installs']=
playStoreDataFrame['Installs']
.apply(lambda x: x.replace('+', ""))

playStoreDataFrame['Installs']=
playStoreDataFrame['Installs']
.apply(lambda x: x.replace(',', ""))

playStoreDataFrame['Price']=
playStoreDataFrame['Price']
.apply(lambda x: x.replace('$', ''))

playStoreDataFrame['Price']=
```

```
pd.to_numeric(
playStoreDataFrame['Price'],
errors='raise')

playStoreDataFrame['size_Mb']=
playStoreDataFrame['size_Mb']
.apply(lambda x: x.replace('M', ""))
```

E. Label Encoding

Label Encoding[10] is a technique used in Machine Learning and data analysis to convert categorical variables into numerical format. The decision tree algorithm employs thresholds to determine how to divide the data into categories and transfer them to values. To map the categories of the Play store dataframe into numerical values LabelEncoder() method was used from the preprocessing module using below command:

```
labelObject = preprocessing.LabelEncoder()

playStoreDataFrame['EncodedCategory'] =
labelObject.fit_transform(
playStoreDataFrame['Category'])

playStoreDataFrame['EncodedRating'] =
labelObject.fit_transform(
playStoreDataFrame['Content_Rating'])

playStoreDataFrame['EncodedGenres'] =
labelObject.fit_transform(
playStoreDataFrame['Genres'])

playStoreDataFrame.drop(
columns=["Category","Content_Rating", "Genres"],
axis=1, inplace=True)
)
```

The above line of code created three new columns having encoded values for `Category`, `Content_Rating`, and `Genres`. These columns were then dropped from the dataframe.

F. Removing Unnecessary Columns

To reduce the noise in the dataset and to prevent overfitting, columns that did not add significant value to feature prediction were then removed.

```
unwanted_columns =
['Last Updated','Current Ver',
'Android Ver', 'Price','App']
playStoreDataFrame = playStoreDataFrame.drop
(columns=unwanted_columns)
```

G. Undersampling of Majority Class

Class Imbalance is a common problem in classification algorithms which can hamper the model's accuracy a big time. Most machine learning algorithms work best when the number of samples in each class is about equal. The playStoreDataFrame prepared after performing the above steps had a big difference in the number of "Free" and "Paid" classes. To fix this, undersampling on the majority class was performed here. [12]

```
minorityClassList = playStoreDataFrame
[playStoreDataFrame['Target']==1]

majorityClassList = playStoreDataFrame[
playStoreDataFrame['Target']==0]
.sample(n=len(minorityClassList),
random_state=1)

UndersampledDataframe =
pd.concat([minorityClassList,
majorityClassList])
```

With above lines of Code, minorityClassList was created having all the instances of class Paid and majorityClassList was created having all the instances of class Free having a length equal to the length of minorityClasslist. Later these two lists were combined to make a new Dataframe named `undersampledDataframe` having almost equal number of instances of both classes.

After successfully performing these five steps, the data gets ready to train and test. It was then divided into two parts, one having only the Target column that needs to be predicted in the future and another having features used to predict the Target variable.

```
y = UndersampledDataframe['Target']
X = UndersampledDataframe.drop(columns=['Target'])
```

To prevent the model from drastically overfitting the dataset, a model validation process of testing on unseen data was used to ensure the classifier can generalize effectively. 33of the dataset was used for testing the classifier while the remaining was used to train it. For the validation process, we employed the `train_test_split` method. This method randomly divides the data into two sets, one for training and the other for testing the model. The data split can be adjusted using the `test_size` attribute. While this randomized split can be done manually as well, the method contains a helpful attribute named `random_state`. To maintain consistency for comparison with other models, the `random_state` attribute was kept constant, ensuring the same seed is used for the random number generator, which is important for replicable results. In this paper, 33% of the data was used to test the performance of the model, and a seed of 1 was used to ensure consistent results and facilitate cross-comparison.

```
Xtrain, Xtest, Ytrain, Ytest =
train_test_split(X, y,
test_size=0.33, random_state=1)
```

With all the above steps performed, the dataframe was divided into train and test data and got ready for further analysis.

H. GridSearchCV

For all the classification models to run over the most efficient hyperparameters, `GridSearchCV` was used. It comes from the `model_selection` package from `sklearn` library and was installed using below command:

```
from sklearn.model_selection
import GridSearchCV
```

GridSearchCV takes in a dict of predefined values for hyperparameters and runs the model with all the possible combination of predefined values. It eventually finds the best value for hyperparameters. It takes the model to run a parameter, a dict of predefined values of hyperparameters and a number of parameters like n_jobs, refit, scoring etc. In this study, refit and verbose were used. refit = true ensures that the best estimator found during the grid search will be fit to the entire dataset.

I. Decision Tree

For Decision Tree, GridSearchCV function was passed with an instance of Decision Tree, a dict param_grid defining a grid of hyperparameters to search over, refit to use the best estimator and verbose=1 to provide some logging during the grid search. In this case, GridSearchCV explored different values for criterion (entropy or gini) and max_depth (ranging from 3 to 20). The Grid created was then fitted with training data (Xtrain features, Ytrain target variable). After fitting, best estimator found during the grid search was then used to make predictions on the test data (Xtest).

```
param_grid = {'criterion': ["entropy","gini"],
'max_depth':
[3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20]}

decisionTreeClassifier=
GridSearchCV(tree.DecisionTreeClassifier(),
param_grid, refit=True, verbose=1)

decisionTreeClassifier.fit(Xtrain, Ytrain)

testPrediction = decisionTreeClassifier.
predict(Xtest)
```

5-Fold Cross-validation was then performed on the dataset to estimate the performance of specific folds of data and results were observed. Cross-validation was performed to use a limited sample in order to estimate how the model is expected to perform in general when used to make predictions on data not used during the training of the model. The k-fold cross-validation procedure divides a limited dataset into 5 discrete folds. Each of the 5 folds is given an opportunity to be used as a held-back test set, while all other folds collectively are used as a training dataset. A total of 5 models are fit and evaluated on the 5 hold-out test sets and the mean performance and standard deviation are reported. The main advantage of K-fold cross-validation is that it allows us to obtain a more accurate estimate of a model's performance, as it ensures that each data point in the dataset is used for both training and testing. The results of these classifiers were compared with classifiers trained on the testing data.

```
accuracies = cross_val_score
(decisionTreeClassifier, Xtrain, Ytrain, cv=5)

std = np.std(accuracies)
mean = np.mean(accuracies)
```

The cross_val_score returns a list containing the results of the scoring measure used for each fold tested using the decision tree classifier. The mean and standard deviation will be calculated using mean() and std() functions from the NumPy library.

Accuracy Score, Recall Score and Precision Score were then calculated on the testing data. Accuracy score is a measure of the overall correctness of a classification model on the testing data. Precision score is a measure of the accuracy of the positive predictions made by the model. It is the ratio of correctly predicted positive observations to the total predicted positives. Recall Score measures the ability of the model to capture all the relevant instances of the positive class. It is the ratio of correctly predicted positive observations to the total actual positives.

```
accuracy = accuracy_score
(Ytest, testPrediction)*100

recallScore = recall_score
(Ytest, testPrediction, average='weighted') * 100

precisionScore = precision_score
(Ytest, testPrediction, average='weighted')* 100
```

Confusion Matrix and Classification Report were then drawn. Confusion matrices serve as a visual tool to gauge how accurately predicted labels align with the true labels. Scikitlearn provides a built-in utility confusion_matrix for generating these matrices, making it more convenient than the challenging task of creating them manually. To enhance the visualization, the generated matrix is plotted using the Seaborn visualization library and Matplotlib. The result analysis is supplemented through Scikit-learn's classification_report [19] utility that provides an indepth analysis of the performance of the classifier.

To generate the ROC curve, roc_curve method from the metrics module was called and was passed with Ytest and decisionTreeProba. Area under the curve was then calculated using auc() method from sklearn.metrics and was passed false positive rate and true positive rate.

```
decisionTreeProba =
decisionTreeClassifier.predict_proba(Xtest)[:,1]

falsePositiveRate_DT, truePositiveRate_DT,
threshold_DT = roc_curve(Ytest, decisionTreeProba)

DecisionTreeAUC =
auc(falsePositiveRate_DT, truePositiveRate_DT)
```

The ROC curve used in this report is a graphical representation of the trade-off between true positive rate (sensitivity) and false positive rate (1 - specificity) for different threshold values.It plots the True Positive Rate (TPR) on the y-axis against the False Positive Rate (FPR) on the x-axis. The diagonal line in the ROC plot represents random chance (an uninformative classifier), and a good classifier should curve toward the top-left corner, maximizing the true positive rate while minimizing the false positive rate. AUC provides a scalar value that represents the area under the ROC curve. A perfect classifier would have an AUC of 1.0, while a purely random classifier would have an AUC of 0.5.

J. Logisctic Regression

Similar to the Decision Tree, for Logistic Regression, GridSearchCV function was passed with an instance of

Logistic Regression, a dict `param_grid` defining a grid of hyperparameters to search over, `refit` to use the best estimator and `verbose`=1 to provide some logging during the grid search.`param_grid` was a dict having different values for `max_iter` ranging from 100 to 1000. `refit` was used to use the best estimator and `verbose`=1 to provide some logging during the grid search.

```
param_grid = {'max_iter':
[100,200, 300, 400, 500, 600, 700, 800, 900, 1000]}

logisticRegressionclassifier=
GridSearchCV(LogisticRegression(), param_grid,
refit=True, verbose=1)

logisticRegressionclassifier.fit(Xtrain, Ytrain)
testPrediction =
logisticRegressionclassifier.predict(Xtest)
```

Similar to the Decision Tree, 5-fold cross validation score was generated along with its mean and standard deviation, accuracy score, recall score, precision score was calculated, and classification report and confusion matrix were drawn.

```
accuracies =
cross_val_score(logisticRegressionclassifier,
Xtrain, Ytrain, cv=5)

accuracy = accuracy_score(Ytest, testPrediction)*100

recallScore = recall_score(Ytest,
testPrediction, average='weighted')

precisionScore = precision_score(Ytest,
testPrediction, average='weighted')

cm_df = pd.DataFrame(confusion_matrix(
Ytest, testPrediction))
```

To generate the ROC curve, `roc_curve` method from the metrics module was called and was passed with Ytest and logisticRegressionProba. Area under the curve was then calculated using auc() method from sklearn.metrics and was passed false positive rate and true positive rate.

```
logisticRegressionProba =
logisticRegressionclassifier.
predict_proba(Xtest)[::,1]

falsePositiveRate_LR, truePositiveRate_LR,
threshold_lr = roc_curve(Ytest,
logisticRegressionProba)

LogisticRegressionAUC =
auc(falsePositiveRate_LR, truePositiveRate_LR)
```

K. K-NN

To generate the K-NN with the optimal value of `n_neighbors`, K-NN was modeled with a number of `n_neighbors` odd values ranging from 3 to 15. After running the Model on all the values it was observed that with `n_beighbor` value=7, the K-NN classifier had the maximum accuracy. After this analysis, K-NN was modeled with 7 neighbors and 'euclidean' metric.

```
knnclassifier =
KNeighborsClassifier(n_neighbors=7,
metric="euclidean")

knnclassifier.fit(Xtrain, Ytrain)

testPrediction = knnclassifier.predict(Xtest)
```

Similar to the Decision Tree, 5-fold cross validation score was generated along with its mean and standard deviation, accuracy score, recall score, precision score was calculated, and classification report and confusion matrix were drawn.

```
accuracies =
cross_val_score(knnclassifier,
Xtrain, Ytrain, cv=5)
```

To generate the ROC curve, `roc_curve` method from the metrics module was called and was passed with Ytest and knnclassifierProba. Area under the curve was then calculated using auc() method from sklearn.metrics and was passed false positive rate and true positive rate.

```
knnclassifierProba =
knnclassifier.predict_proba(Xtest)[::,1]

falsePositiveRate_KNN, truePositiveRate_KNN,
threshold_knn = roc_curve(Ytest, knnclassifierProba)

kNNAUC =
auc(falsePositiveRate_KNN, truePositiveRate_KNN)
```

L. Naive Bayes

In this section, similar to Decision Tree, for Naive Bayes, `GridSearchCV` function was passed with an instance of Naive Bayes, a dict `param_grid` defining a grid of hyperparameters to search over, `refit` to use the best estimator and `verbose`=1 to provide detailed logging.`param_grid` was a dict having different values for `var_smooting`. `var_smoothing` is the portion of the largest variance of all features that are added to variances for calculation stability. `refit` was used to use the best estimator and `verbose`=1 to provide some logging during the grid search.

```
param_grid = {'var_smoothing': [1e-2, 1e-3, 1e-4,
1e-5, 1e-6, 1e-7, 1e-8, 1e-9, 1e-10, 1e-11,
1e-12, 1e-13, 1e-14, 1e-15, 1e-16, 1e-17,
1e-18, 1e-19, 1e-20]}

naiveBayesClassifier=
GridSearchCV(naive_bayes.GaussianNB(),
param_grid, refit=True, verbose=2)


# naiveBayesClassifier = naive_bayes.GaussianNB()
naiveBayesClassifier.fit(Xtrain, Ytrain)

testPrediction =
naiveBayesClassifier.predict(Xtest)
```

Similar to the Decision Tree, 5-fold cross validation score was generated along with its mean and standard deviation, accuracy score, recall score, precision score was calculated, and classification report and confusion matrix were drawn.

```
accuracies =
cross_val_score(naiveBayesClassifier,
Xtrain, Ytrain, cv=5)
```

To generate the ROC curve, `roc_curve` method from the metrics module was called and was passed with Ytest

and naiveBayesClassifierProba. Area under the curve was then calculated using auc() method from sklearn.metrics and was passed false positive rate and true positive rate.

```
naiveBayesClassifierProba =
naiveBayesClassifier.predict_proba(Xtest)[::,1]

falsePositiveRate_NB, truePositiveRate_NB,
threshold_NB = roc_curve(Ytest,
naiveBayesClassifierProba)

naiveBayesAUC =
auc(falsePositiveRate_NB, truePositiveRate_NB)
```

M. Linear Discriminant Analysis

Following the same pattern, for Linear Discriminant Analysis, `GridSearchCV` function was passed with an instance of Linear Discriminant Analysis, a dict `param_grid` defining a grid of hyperparameters to search over, `refit` to use the best estimator and `verbose`=1 to provide detailed logging.`param_grid` was a dict having different values for `solver`. The `solver` parameter specifies the method used to estimate the coefficients of the linear discriminants. `refit` was used to use the best estimator and `verbose`=1 to provide some logging during the grid search.

```
param_grid = {'solver': ['svd','lsqr','eigen']}

LinearDiscriminantAnalysisModel=
GridSearchCV(LinearDiscriminantAnalysis(),
param_grid, refit=True, verbose=1)

LinearDiscriminantAnalysisModel
.fit(Xtrain, Ytrain)

testPrediction =
LinearDiscriminantAnalysisModel.predict(Xtest)
```

Similar to the Decision Tree, 5-fold cross-validation score was generated along with its mean and standard deviation, accuracy score, recall score, precision score was calculated, and classification report and confusion matrix were drawn.

```
accuracies = cross_val_score(
LinearDiscriminantAnalysisModel,
Xtrain, Ytrain, cv=5)
```

To generate the ROC curve, `roc_curve` [13] method from the metrics module was called and was passed with Ytest and LinearDiscriminantAnalysisModelProba. Area under the curve was then calculated using auc() method from sklearn.metrics and was passed false positive rate and true positive rate.

```
LinearDiscriminantAnalysisModelProba =
LinearDiscriminantAnalysisModel.
predict_proba(Xtest)[::,1]

falsePositiveRate_LDA, truePositiveRate_LDA,
threshold_LDA = roc_curve(Ytest,
LinearDiscriminantAnalysisModelProba)

ldaAUC = auc(falsePositiveRate_LDA,
truePositiveRate_LDA)
```

The results for also analyzed by plotting a confusion matrix for which matplotlib and seaborn library were used. This confusion matrix was drawn for every model.

```
fig, ax = plt.subplots()
ax.xaxis.tick_top()
ax.xaxis.set_label_position("top")
sn.heatmap(cm_df, annot=True, cmap="Purples", fmt="d
"
)


plt.title("Decision Tree model")
plt.xlabel("Predicted")
plt.ylabel("True")
plt.show()
```

To visualize the ROC Curve, matplotlib library was used and was passed with False positive rate, True Postive rate and Area under the curve for every model.

```
plt.figure(figsize=(10, 6))
plt.plot(falsePositiveRate_DT, truePositiveRate_DT, label=f'Decision Tree (AUC = {DecisionTreeAUC:.2f})', color='green')
plt.plot(falsePositiveRate_LR, truePositiveRate_LR, label=f'Logistic Regression (AUC = {LogisticRegressionAUC:.2f})', color='blue')
plt.plot(falsePositiveRate_KNN, truePositiveRate_KNN, label=f'K-NN (AUC = {KNNAUC:.2f})', color='pink')
plt.plot(falsePositiveRate_NB, truePositiveRate_NB, label=f'Naive Bayes (AUC = {naiveBayesAUC:.2f})', color='red')
plt.plot(falsePositiveRate_LDA, truePositiveRate_LDA, label=f'Linear Discriminant Model (AUC = {ldaAUC:.2f})', color='brown')
plt.plot([0, 1], [0, 1], linestyle='--', color='yellow', label='Random Guess (AUC = 0.5)')

# Customize the plot
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()
plt.show()
```

Fig. 1: Enter Caption

III. RESULTS

This section focuses on the classification results obtained when all the models were tested on the unseen data.The Raw dataset taken had `10841` instances in the beginning which were reduced to `1120` instances after undersampling of Majority Class. After the data was split into training and testing, 381 instances were used for testing purpose.

A. Decision Tree

Decision Tree when tested on the unseen testing data generated an accuracy of 79.27% whereas its mean accuracy after cross-validation was 76.83% with a standard deviation of 0.03. It generated a Recall Score of 79 and a precision Score of 80.7.

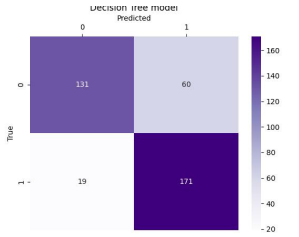

Fig. 2: Confusion Matrix for Decision Tree

It was observed from the Confusion Matrix that the classifier incorrectly predicted 60 instances of 'Free' class instances while performing relatively better on predicting 'Paid' class instances with 19 wrongs.

B. Logistic Regression

Logistic Regression when tested on the unseen testing data generated an accuracy of 73.23% whereas its mean accuracy after cross-validation was 63.66% with a standard deviation of

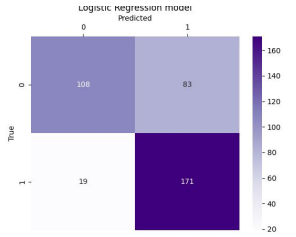

Fig. 3: Confusion Matrix for Logistic Regression

0.1. It generated a Recall Score of 73.2 and a precision Score of 76.20.

Logistic Regression performed poorly in predicting 'Free' class instances and incorrectly predicted 83 instances. It performed a little better in predicting 'Paid' class instances and only predicted 19 instances wrong.

C. K-NN

Decision Tree when tested on the unseen testing data generated an accuracy of 82.15% whereas its mean accuracy after cross-validation was 79.29% with a standard deviation of 0.02. It generated a Recall Score of 82.1 and a precision Score of 82.24.

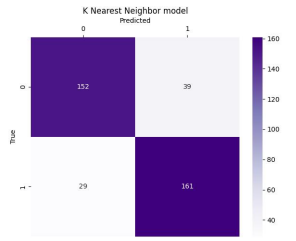

Fig. 4: Enter Caption

K-NN Model performed better in predicting 'Free' class instances and incorrectly predicted 39 instances only While it predicted 29 'Paid' instances incorrectly.

D. Naive Bayes

Naive Bayes when tested on the unseen testing data generated an accuracy of 61.94% whereas its mean accuracy after cross-validation was 64.94% with a standard deviation of 0.06. It generated a Recall Score of 61.9 and a precision Score of 77.39.

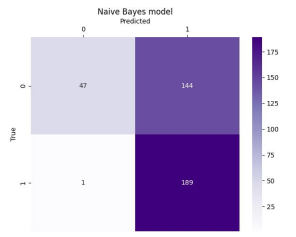

Fig. 5: Confusion Matrix for Naive Bayes

It could be observed that Naive Bayes performed exceptionally well in predicting 'Paid' class instances but at the same time performed very poorly in predicting 'Free' class instances with 144 wrong predictions.

E. Linear Discriminant Model

Linear Discriminant Model when tested on the unseen testing data generated a poor accuracy of 58.27% whereas its mean accuracy after cross-validation was 52.90% with a standard deviation of 0.03. It generated a Recall Score of 58.27 and a precision Score of 59.47.

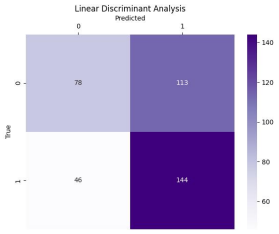

Fig. 6: Confusion Matrix for LDA

LDA performed relatively poorly in predicting instances of both the class and predicted 113 and 46 wrong instances of 'Free' and 'Paid' relatively.

F. ROC Curve

It could be observed from the ROC Curve that K-NN has the highest Area under the curve followed by Decision tree and Logistic Regression.

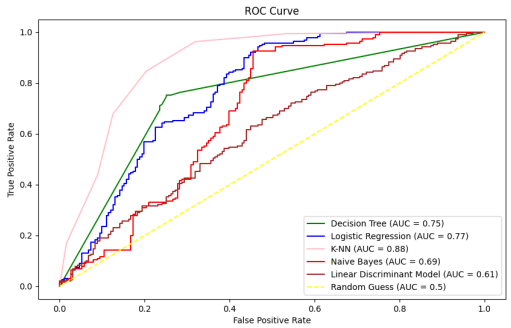

Fig. 7: ROC Curve

Considering the results obtained after testing the model, K-NN was considered an optimal choice for the classification of 'Free' and 'Paid' instances of the Play Store data set. With an Area under the curve of 0.88 and an overall testing accuracy of 82.15% it has overperformed every other model.

One of the reasons why K-NN performed relatively better than other classifiers is because the relationship between the features and the target in the dataset is non-linear in the dataset. K-NN captures non-linear data relatively better than LR and DT.

With consistent overall accuracy in each fold obtained during 5-fold cross-validation, and a standard deviation of (0.03),

TABLE II: Performance Comparision of Different Models

Metric	DT	LR	K-NN	NB	LDA
Fold 1 Accuracy	80.6%	50.3%	81.8%	63.8%	59.3%
Fold 2 Accuracy	72.9%	73.5%	78.7%	65.1%	50.3%
Fold 3 Accuracy	81.9%	52.2%	80.6%	63.2%	51.6%
Fold 4 Accuracy	70.7%	68.8%	74.0%	65.5%	51.9%
Fold 5 Accuracy	77.9%	73.3%	81.1%	65.5%	51.2%

K-NN performed really well across different sets of unseen data. Therefore, it can be extrapolated that the performance on subsequent unseen data may follow the same trend as well.

IV. DISCUSSION

Selecting a Classification model in Data Analytics is a very important step. Selecting machine learning algorithms for various types of problems requires an impartial approach to evaluate the performance of different classifiers. This paper focuses on selecting an appropriate model for handling complex Airline Delay Dataset. As model selection does not have a right or wrong answer, different aspects and parameters should be considered while making a decision.

In this report, the performance accuracy of the K-NN over the other classification models was greatly expected because of the non-linear relationship between the features and the target. K-NN [11] being a non-parametric algorithm does not assume a specific functional form for the decision boundary. It can capture complex relationships in the data, especially when the true decision boundary is non-linear which is also a reason why K-NN has good precision score. It was observed that Naive Bayes had the minimum standard deviation among all the classifiers yet it had poor accuracy and Area under the curve. One of the reasons for this could be that the classes were not easily separable by a linear boundary. Same reason goes for the poor performance of Logistic Regression as it assumes a linear decision boundary between classes. If the true decision boundary is non-linear, Logistic Regression struggles to capture the complex relationships in the data. Also, LDA assumes that the classes have Gaussian distributions with the same covariance matrix and aims to find a linear combination of features that best separates the classes. As a result, on performs relatively poor on non-linear datasets.

In non-linear scenarios, where the decision boundary is not a straight line, KNN's flexibility allows it to better approximate intricate patterns and capture the non-linear relationships between features and the target variable resulting in higher AUC than others. Model selection is an essential step in classification problems as it helps to avoid overfitting, enhance accuracy, minimize computational expenses, improve interpretability, and tackle domain-specific challenges. By carefully selecting the most appropriate model from a range of candidates, machine learning practitioners can optimize their classification models and achieve better performance. Classifying an Android application based on its features if free or not is a real-world problem. This study explores the selection process of a classification model considering all the hyperparameters and choosing the best for the Dataset and elaborates on the different metrics that should be used.

A new of things could be done in the future to get even better results over this dataset. An important thing would be to more thoroughly examine the features of the dataset to decide which other features can be added or removed to get better performance. Another approach would be to choose a more complex Model suitable for a non-linear dataset.In this study, undersampling of Majority classes was performed, in the future, oversampling could be tried.

REFERENCES

[1] Decision Trees https://scikit-learn.org/stable/modules/tree.html#:~:text=Decision%20Trees%20(DTs)%20are%20a,as%20a%20piecewise%20constant%20approximation.

[2] Logistic regression in data analysis https://www.researchgate.net/publication/227441142_Logistic_regression_in_data_analysis_An_overview

[3] K-NN https://www.datacamp.com/tutorial/k-nearest-neighbor-classification-scikit-learn

[4] Naive Bayes https://scikit-learn.org/stable/modules/naive_bayes.html

[5] Linear discriminant analysis https://scikit-learn.org/stable/modules/generated/sklearn.discriminant_analysis.LinearDiscriminantAnalysis.html

[6] Pandas https://pandas.pydata.org/

[7] sk-learn https://scikit-learn.org/

[8] Hyperparameters https://towardsdatascience.com/parameters-and-hyperparameters-aa609601a9ac

[9] Dropping rows https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.dropna.html

[10] Label Encoding https://www.geeksforgeeks.org/ml-label-encoding-of-datasets-in-python/

[11] K-NN on non-linear dataset https://arxiv.org/abs/2305.17695

[12] Undersampling https://www.sciencedirect.com/science/article/pii/S1877050919313456

[13] ROC curve https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5