Project Report: Credit Card Fraud Detection

MSE 491: Application of Machine Learning in Mechatronic Systems

Calvin Tse - 301330943 (50%)

Mohammad Uzair Bawany - 301296889 (50%)

Group 14

Due date: April 14th, 2022

Table of Contents

List of Figures

List of Tables

Introduction

As more of the banking industry becomes digitized, electronic fraud becomes a larger topic of interest. It is important that credit card companies can distinguish between fake and real activities. For the final project of this course, it will be on conducting various classification techniques on a set of data for detecting credit card fraud. This data is from transactions in September 2013 by various European cardholders [1]. The dataset has a total of 30 features and 284808 samples. However due to issues of confidentiality, there are 28 unnamed features. The other two features are time between the transaction in the current dataset and the first transaction in the dataset, and the transaction amount. For the classification of this data, the methods employed are Logistic Regression, K-Nearest Neighbor, Gaussian Naive Bayes, Decision Tree, and Support Vector Machine (SVM). All five of these models will be trained, tested, and then their performance will be compared and discussed against the other models.

Methods

Data Visualization

To start modelling our dataset first we need to load the libraries to manipulate our data as shown in figure 1:

```
# Load General Libraries
import numpy as np
import pandas as pd
import pickle
import matplotlib.pyplot as plt
import seaborn as sns
```

Figure 1: Loading Libraries

Then we need to import our dataset (excel file) using panda (pd.read_csv), shown in figure 2:

```
# Importing the dataset (no header)
dataset = pd.read_csv("creditcard.csv", header=0)
```

Figure 2: Importing Dataset

Header = 0, specifies that names of columns are in the first row.

Next, to visualize our data first we determine how are the classes distributed within the data. In our case, how many transactions are fraudulent and how many are real. This is done by executing the following code shown in figure 3:

```
# Determining how many transactions are fraudulent (Fraud = Class 1 and NonFraud = Class 0)
dataset.Class.value_counts()
```

```
0    284315
1       492
Name: Class, dtype: int64
```

Figure 3: How many transactions are real or fraudulent

As we can see for our dataset 491 transactions are flagged as fraudulent and 284315 transactions are flagged as real.

Since the name of the features aren't described due to confidentiality, its hard to relate the exact correlation between them. However, to provide a general idea, a heatmap is plotted using seaborn library as shown in figure 4:

```
fig, ax = plt.subplots(figsize=(25,25))
sns.heatmap(dataset.corr(), annot=True, linewidths=.5, ax=ax)
```

Figure 4: Plotting Heatmap using Seaborn Library

The result for the heatmap is shown in figure 5:

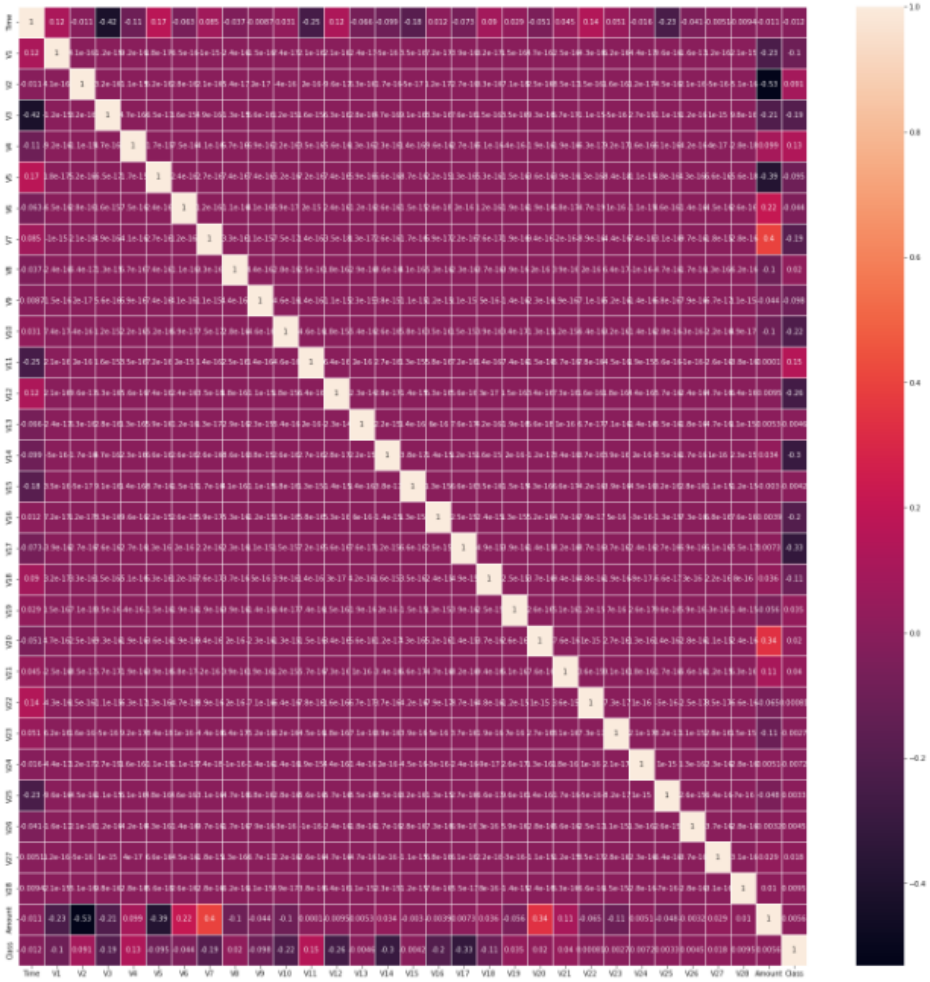

Figure 5: Heatmap

Another way to visualize the transactions is to plot a histogram of a variable from the dataset to see the skewness, shown in figure 6:

```
plt.figure(figsize=(20, 60))
for n, col in enumerate(dataset.drop('Class',axis=1).columns):
    plt.subplot(10,3,n+1)
    sns.histplot(dataset[col][dataset.Class == 1], bins=50)
    sns.histplot(dataset[col][dataset.Class == 0], bins=50)
    plt.title(col, fontsize=17)
plt.show()
```

Figure 6: Skewness check

The result prints the histogram for all 30 features against count as shown in figure 7:

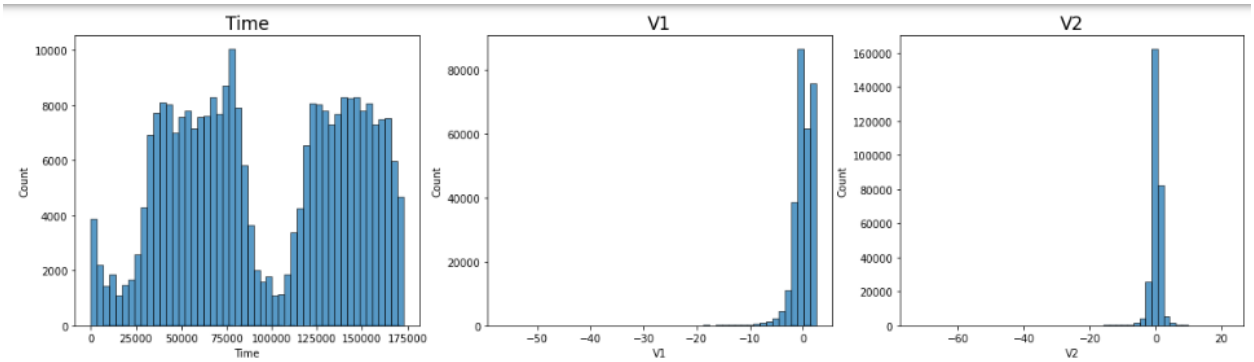

Figure 7: Histogram

Due to many histograms its difficult to fit all in here so the above figure provides an overview of how the histogram for all 30 features would look like.

Now that the data is visualized, data can now be broken down into features and target. After which to train the model on our dataset, data needs to be split into training data and test data, then the model is executed, and results are evaluated.

Logistic Regression

Logistic Regression is a supervised learning method for binary classification problems (i.e., problems with two classes). Logistic function is shown in equation 1:

$$\frac{1}{1+e^{-z}}\ , where\ z = any\ function$$

Equation 1: Logistic Function

The plot for this function is shown in figure 8 and since it's a probability function the output value is always between [0 1]:

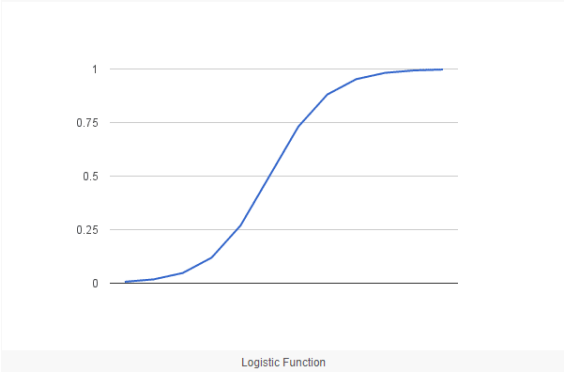

Figure 8: Logistic Function Plot

To apply the logistic regression model, first we must define the target (y) and features (x) as shown in figure 9:

```
X = dataset.iloc[:, 0:30].values
y = dataset.iloc[:, -1].values

print(X)
print(y)
```

Figure 9: Splitting dataset into Target and Features

It can be observed that we have 30 features that help us predict the 1 target value which is the class. If Class = 0, fraudulent, and if Class = 1, fraudulent.

After determining features and target, we need to split the dataset into 80% training data and 20% test data using train_test_split for our model as shown in figure 10:

```
# Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state = np.random)

print(X_train)
print(y_train)
```

Figure 10: Splitting dataset into training set and test set

After successfully splitting the dataset, its time to build the model by importing Logistic Regression from the sklearn.linear_model library and then fitting the logistic regression model with our data. Lastly, we need to test our model with the test dataset. All the steps are shown in figure 11:

```
# Import the class
from sklearn.linear_model import LogisticRegression

# Instantiating the model (using the default parameters)
logreg = LogisticRegression(max_iter=1000)

# Fitting the model with data
logreg.fit(X_train, y_train)


# Predicting the labels on test set
y_pred_test = logreg.predict(X_test)
```

Figure 11: Applying Logistic Regression to the model

To evaluate the success of our model we need to print out an evaluation matrix (confusion matrix) that highlights the accuracy of our logistic regression model as shown in figure 12:

```
# import the metrics class
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.metrics import confusion_matrix, plot_confusion_matrix
cnf_matrix = confusion_matrix(y_test, y_pred_test)
print('Confusion Matrix\n', cnf_matrix)


plt_cnf_matrix = plot_confusion_matrix(logreg, X_test, y_test,
                                       display_labels=['Real','Fraud'],
                                       cmap=plt.cm.Blues,
                                       normalize='true')
plt_cnf_matrix.ax_.set_title('Normalized Confusion Matrix for Logistic Regression')

# Print Evaluation Metrics
print("\n")
print ("CLassication metrices for test set")
print("Accuracy:",accuracy_score(y_test, y_pred_test))
print("Precision:",precision_score(y_test, y_pred_test))
print("Recall(Sensitivity):",recall_score(y_test, y_pred_test))
print("F1 Score:",f1_score(y_test, y_pred_test))
```

Figure 12: Evaluating confusion matrix for Logistic Regression

The results from this model are explained in the later section.

K-Nearest Neighbor

It is a supervised learning algorithm used for both regression and classification problem. KNN works by finding the distances between a query and all the examples in the data, selecting the specified number examples (K) closest to the query, then votes for the most frequent label (in the case of classification) or averages the labels (in the case of regression). While executing KNN the values of the distance can get very large because it uses Euclidean distance, as shown in equation 2:

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(q_i^2 - p_i^2)}$$

Equation 2: Euclidean Distance

It is a good practice to normalize the data before using KNN, the normalization for our model is done using preprocessing from sklearn library as shown in figure 13:

```
# Normalize Features
from sklearn import preprocessing
#print(X_train)
X_train_normalized = preprocessing.normalize(X_train)
#print(X_train_normalized)
X_test_normalized = preprocessing.normalize(X_test)
```

Figure 13: Normalizing Features

After normalizing the data, we need to import the KNN function from sklearn.neighbors, initialize the value of K and train the model. Lastly, we need to test our model with the test dataset. All the steps are shown in figure 14:

```
# Import the class
from sklearn.neighbors import KNeighborsClassifier

# Instantiating the model (using the default parameters)
MODEL_KNN = KNeighborsClassifier(n_neighbors=1)

# Train the Model
MODEL_KNN.fit(X_train_normalized,y_train)

# Save the Trained Model
#pickle.dump(MODEL_KNN, open('Model_KNeighborsClassifier.pkl', 'wb'))

# Predict the Trained Model on our Test data
y_pred_KNN = MODEL_KNN.predict(X_test_normalized)
```

Figure 14: Applying KNN to the model

To evaluate the success of our model we need to print out an evaluation matrix (confusion matrix) and a classification report with the labels (real and fraud) that highlights the accuracy of our KNN model as shown in figure 15:

```
# Some Functions for Showing the Classifier Performance
from sklearn.metrics import classification_report
from sklearn.metrics import plot_confusion_matrix

labels = ['Real','Fraud']
def classifier_performance(model,y_pred):
    print('Classification Report: \n', classification_report(y_test,y_pred,target_names=labels))
    # Plot normalized confusion matrix
    titles_options = [("Confusion matrix, without normalization", None),
                      ("Normalized Confusion Matrix, K=1", 'true')]
    for title, normalize in titles_options:
        disp = plot_confusion_matrix(model, X_test, y_test,
                                     display_labels=labels,
                                     cmap=plt.cm.Blues,
                                     normalize='true')
        disp.ax_.set_title(title)
        print(title)
        print(disp.confusion_matrix)
    plt.show()
    return

# Print the Classification Report and Confusion Matrix
classifier_performance(MODEL_KNN,y_pred_KNN)
```

Figure 15: Function for Classification Report and Normalized Confusion Matrix

The results from this model are explained in the later section.

Gaussian Naïve Bayes

It is a supervised learning classification algorithm based on Bayes Theorem, which is used to calculate conditional probability. The algorithm is an approach to create a simple model while assuming that the data is described by a Gaussian distribution with no co-variance (independent dimensions) between dimensions.

To apply Gaussian Naïve Bayes classifier to our model, first we must import the classifier from sklearn.naive_bayes and then fit our data into the model. Lastly, we need to test our model with the test dataset. All the steps are shown in figure 16:

```
# Import the class
from sklearn.naive_bayes import GaussianNB

# Instantiating the model (using the default parameters)
MODEL_GNB = GaussianNB()

# Train the Model
MODEL_GNB.fit(X_train,y_train)

# Save the Trained Model
pickle.dump(MODEL_GNB, open('Model_GaussianNB.pkl', 'wb'))

# Predict the Trained Model on our Test data
y_pred_GNB = MODEL_GNB.predict(X_test)

# Print the Classification Report and Confusion Matrix
classifier_performance(MODEL_GNB,y_test,y_pred_GNB,X_test)
```

Figure 16: Applying GNB to the model

To evaluate the success of our model we need to print out an evaluation matrix (confusion matrix) and a classification report with the labels (real and fraud), we can use the same function for our classification report as we used for KNN (figure 15). These reports highlight the accuracy of our Gaussian Naïve Bayes model as shown in figure 17:

```
# Print the Classification Report and Confusion Matrix
classifier_performance(MODEL_GNB,y_test,y_pred_GNB,X_test)
```

Figure 17: Print Classification Report and Confusion Matrix for GNB

The results from this model are explained in the later section.

Decision Tree

Decision Tree is a supervised learning algorithm that predict value of responses by learning decision rules derived from features. It can be used for both regression and classification problem. The general form of a decision tree is shown in figure 18:

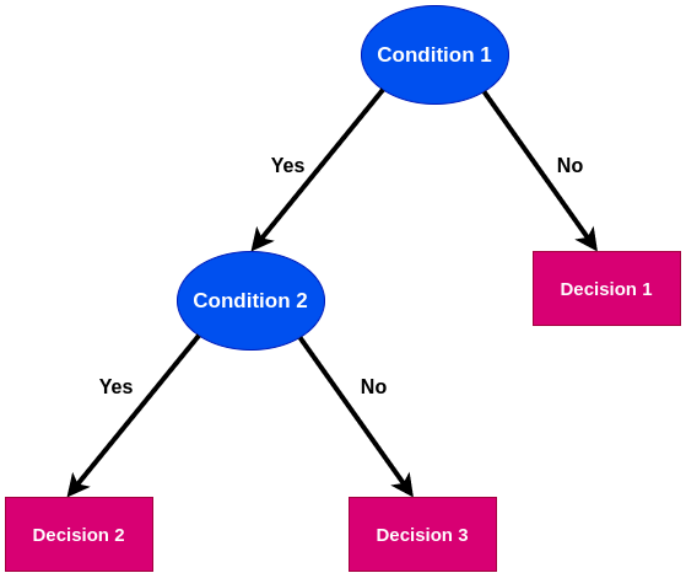

Figure 18: General Skeleton of Decision Tree

To apply Decision Tree classifier to our model, first we must import the classifier from sklearn.tree and then fit our data into the model. Lastly, we need to test our model with the test dataset. All the steps are shown in figure 19:

```
# Import the class
from sklearn.tree import DecisionTreeClassifier

# Instantiating the model (using the default parameters)
MODEL_DT = DecisionTreeClassifier()

# Train the Model
MODEL_DT.fit(X_train,y_train)

# Save the Trained Model
pickle.dump(MODEL_DT, open('DecisionTreeClassifier.pkl', 'wb'))

# Predict the Trained Model on our Test data
y_pred_DT = MODEL_DT.predict(X_test)

# Print the Classification Report and Confusion Matrix
classifier_performance(MODEL_DT,y_test,y_pred_DT,X_test)
```

Figure 19: Applying Decision Tree to the model

To evaluate the performance of the Decision Tree classifier we can use the classification report as shown in figure 20:

```
# Print the Classification Report and Confusion Matrix
classifier_performance(MODEL_DT,y_test,y_pred_DT,X_test)
```

Figure 20: Print Classification Report and Confusion Matrix for DT

The results from this model are explained in the later section.

Support Vector Machine

SVM is also a supervised learning algorithm that can be used for both classification and regression problem. In SVM, each data item is plotted as a point in n-dimensional space (where n is the number of features you have), with the value of each feature being the value of a certain coordinate. Then we classify the data by locating the hyper-plane that separates the two groups as shown in figure 21:

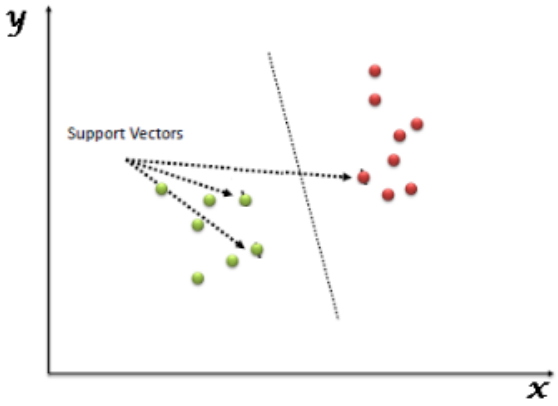

Figure 21: Skeleton for SVM

SVM separates the two classes by maximizing the distance between the hyperplane and the support vectors. Support vectors are the points closest to the hyper plane in terms of distance. Like KNN, the value for the distance can be large so it's a good practice to normalize your dataset before applying to model to your data. We will use the same approach as we used for KNN as shown in figure 22:

```
# Normalize Features
from sklearn import preprocessing
#print(X_train)
X_train_normalized = preprocessing.normalize(X_train)
#print(X_train_normalized)
X_test_normalized = preprocessing.normalize(X_test)
```

Figure 22: Normalizing Features

After normalizing we can apply SVM to our model, like every other classifier first we must import the classifier from sklearn.svm and then fit our data into the model using default parameters (rbf kernel in this case). Lastly, we need to test our model with the test dataset. All the steps are shown in figure 23:

```
# Import the class
from sklearn.svm import SVC

# Instantiating the model (using the default parameters)
MODEL_SVM = SVC()

# Train the Model
MODEL_SVM.fit(X_train_normalized,y_train)

# Save the Trained Model
pickle.dump(MODEL_SVM, open('Model_SupportVectorClasifier.pkl', 'wb'))

# Predict the Trained Model on our Test data
y_pred_SVM = MODEL_SVM.predict(X_test_normalized)

# Print the Classification Report and Confusion Matrix
classifier_performance(MODEL_SVM,y_test,y_pred_SVM,X_test_normalized)
```

Figure 23: Applying SVM to the model

To evaluate the performance of the SVM classifier we can use the classification report as shown in figure 24:

```
# Print the Classification Report and Confusion Matrix
classifier_performance(MODEL_SVM,y_test,y_pred_SVM,X_test_normalized)
```

Figure 24: Print Classification Report and Confusion Matrix for SVM

The results from this model are explained in the later section.

Results and Discussions

The criteria for the results of using different classifiers are evaluated and compared based on the confusion matrix, F1 score. F1 score is the weighted average of precision and recall, it is evaluated using the following formula shown in equation 3:

$$F1 = 2 * (\frac{Recall * Precision}{Recall + Precision})$$

Equation 3: F1 Score

Recall also known as sensitivity is the ratio of correctly predicted positive observations vs all the observations in the class. The following equation 4 shows how recall is evaluated:

$$Recall = \frac{True\ Positive}{True\ Positive + False\ Negative}$$

Equation 4: Recall (Sensitivity)

Precision is the ratio of correctly predictive positive observations vs total predicted positive observations. The following equation 5 shows how precision is evaluated:

$$Precision = \frac{True\ Positive}{True\ Positive + False\ Positive}$$

Equation 5: Precision

Logistic Regression

For the logistic regression model, the model evaluated had an F1 Score of 0.74. Overall, it had a very fast training and evaluation time which was approximately only a few seconds. In Figure 25, there is a Normalized Confusion Matrix of the trained model. From Figure 25, it is visualized that the data is heavily skewed due to the many samples of real transactions relative to the number of samples for fraudulent ones.

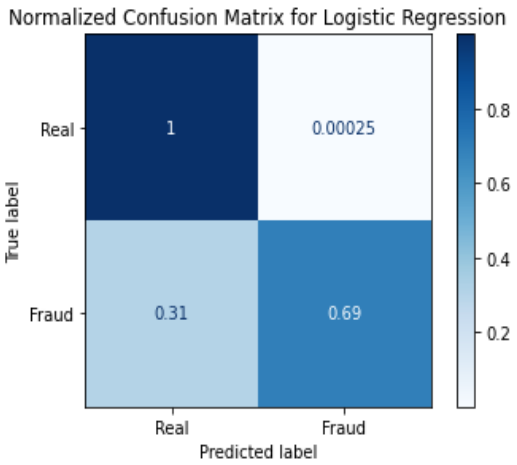

Figure 25: Normalized Confusion Matrix for Logistic Regression

K-Nearest Neighbor

For the K-Nearest Neighbor, the model was evaluated multiple times with varying values of K. From there, Table 1 was generated to find the most effective K value. Furthermore, Figure 26 illustrates the data and when K=1, the F1 score yields the best results of 0.79. To find the optimum K value, the first value was with K=400 which is slightly lower than the square root of the number of the samples and from there, K was reduced to find the K value with the best F1 Score. An important note for this model is the time it took to train and test. With each iteration, it would take 2-5 minutes to run the code. In comparison, Logistic Regression took only a few seconds to train and test on the same dataset.

Table 1: F1 Score at Different K Values

K Value	F1 Score	K Value	F1 Score
1	0.79	25	0.58
2	0.76	30	0.57
3	0.76	35	0.58
4	0.73	40	0.57
5	0.74	45	0.54
6	0.70	50	0.47
7	0.69	75	0.42
8	0.69	100	0.41
9	0.68	125	0.38
10	0.67	150	0.33
15	0.65	200	0.21
20	0.61	400	0.04

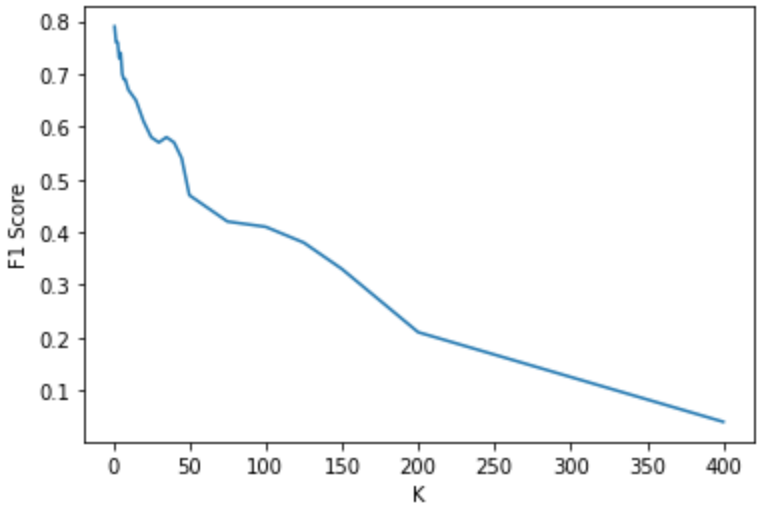

Figure 26: F1 Score Versus K Value for K Nearest Neighbor

Gaussian Naive Bayes

For GNB, the evaluation time for the model was quick with an F1 score of 0.2496, as shown in the figure 27:

```
GNB Accuracy: 0.9922931076858257
GNB Precision: 0.15176715176715178
GNB Recall: 0.7019230769230769
GNB F1 Score: 0.2495726495726496
```

Figure 27: Evaluation Metrics for GNB

Figure 28 shows the normalized confusion matrix for this classifier.

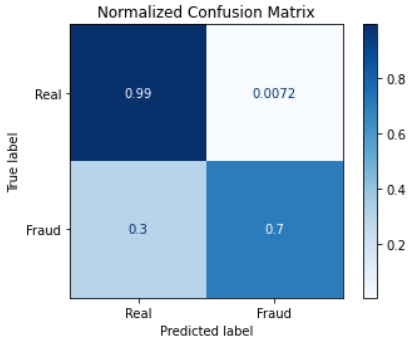

Figure 28: Confusion Matrix for GNB

The above figure shows that using GNB classifier it predicted 70% of the real fraudulent transactions which is not that great. To increase the accuracy for detecting the fraudulent transactions, we can train more data or have a data that is more balanced than what we have for this problem.

Decision Tree

For Decision Tree classifier our test model gave the F1 score of 0.8195, which is better than all the above classifiers. However, it wasn't as quick as GNB and Logistic Regression classifiers but faster than KNN classifier. Figure 29 highlights the evaluation metrics for Decision Tree:

```
DT Accuracy: 0.9993504441557529
DT Precision: 0.8316831683168316
DT Recall: 0.8076923076923077
DT F1 Score: 0.8195121951219512
```

Figure 29: Evaluation Metrics for DT

Normalized confusion matrix for Decision Tree is shown in figure 30:

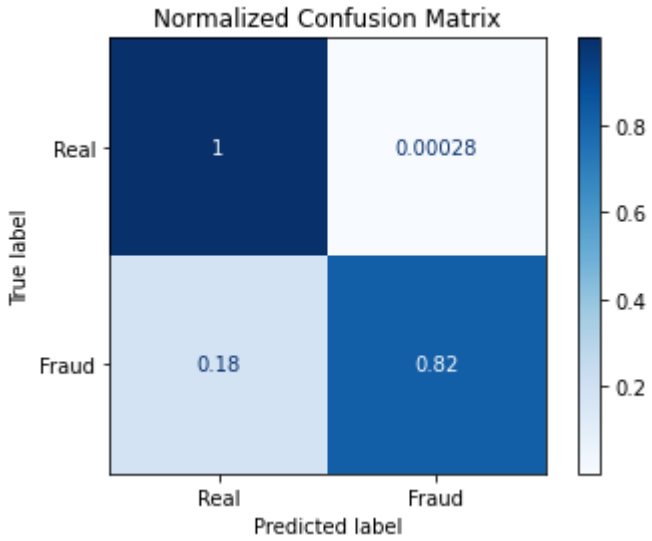

Figure 30: Confusion Matrix for DT

The figure above shows that on the test data, Decision Tree classifier successfully predicted 82% of the fraudulent transactions which is the highest amongst the other classifiers. The reason for this could be due to the nature of Decision Tree as it provides predictive modeling with higher accuracy, better stability and ease of interpretation.

Support Vector Machine

For SVM classifier there was no concrete results shown as it returned an F1 score of 0 as shown in figure 31:

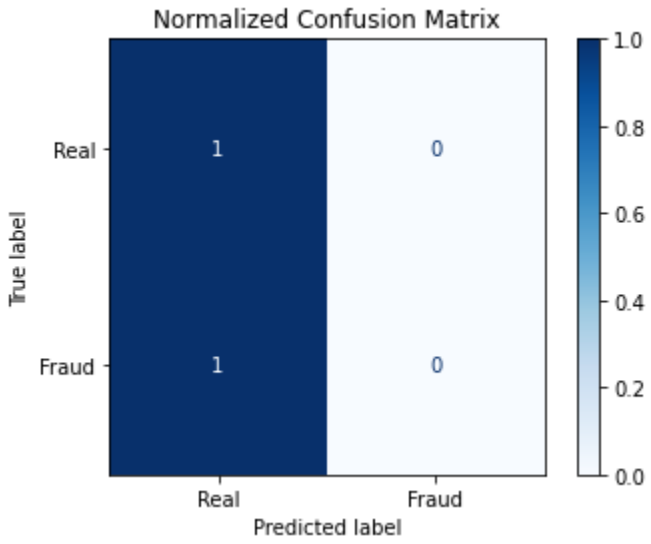

Figure 31: Confusion Matrix for SVM

The figure shows that the classifier that didn't perform well. The reason SVM algorithm didn't perform well is because it isn't suitable for large data sets. SVM does not perform very well when the data set has more noise i.e., target classes are overlapping. In cases where the number of features for each data point exceeds the number of training data samples, the SVM will underperform. However, to identify the best classifier, we will only be comparing the results from Logistic Regression, KNN, Gaussian Naïve Bayes and Decision Tree.

Based on the F1 scores for the four classifiers as shown in figure 32:

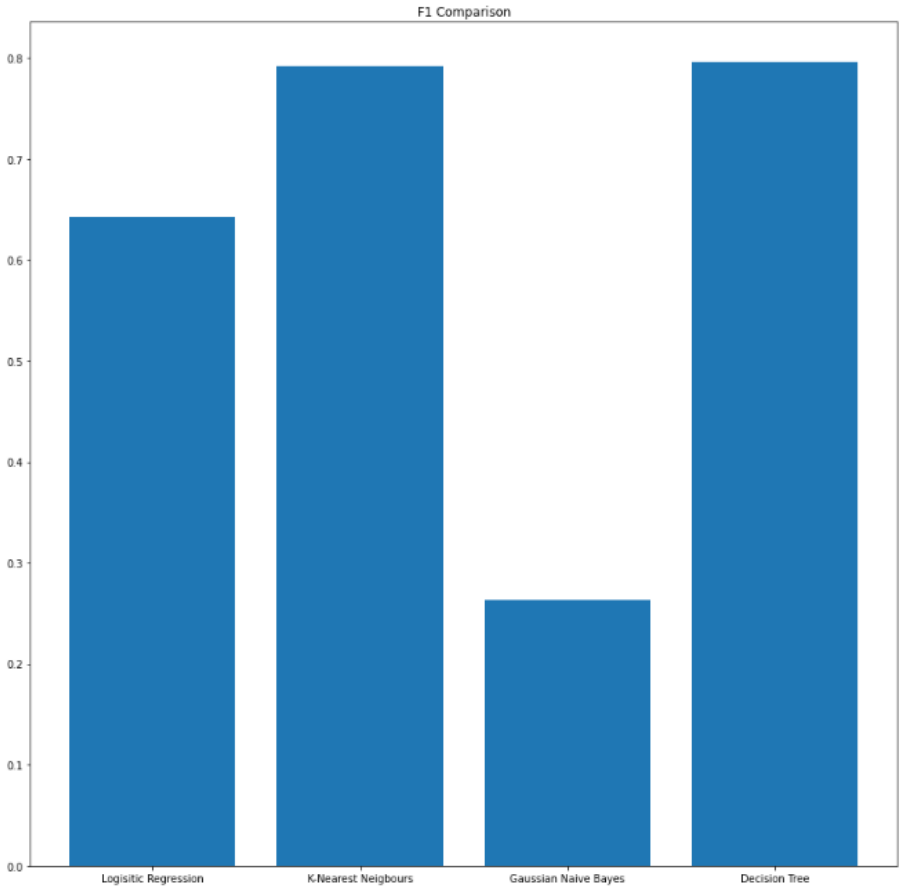

Figure 32: Bar chart comparing F1 score for different classifiers

It can be concluded that for this dataset Decision Tree and KNN provided relatively good results than Logistic Regression classifier while Gaussian Naïve Bayes classifier gave the least satisfactory result. The difference between KNN and Decision Tree was the execution time as KNN took a lot of time to train the model. Its better to use Decision Tree when we just need to classify examples.

Conclusion

In this project, machine learning techniques like Logistic Regression, KNN, Gaussian Naïve Bayes, Decision Tree and Support Vector Machines were used to detect the fraud in credit card transactions. F1 scores were used to evaluate the performance for the dataset. The F1 score obtained for different classifiers are shown in table 2:

Table 2: F1 scores for different classifiers

Classifier	**F1 Score**
Logistic Regression	0.6429
K Nearest Neighbor	0.7917
Gaussian Naïve Bayes	0.2629
Decision Tree	0.7960

From the table we can conclude that Decision Tree performed best for our dataset.

References

[1] "Credit Card Fraud Detection", *Kaggle.com*, 2022. [Online]. Available: https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud. [Accessed: 14- Apr- 2022].