Using Machine Learning to Detect Credit Card Fraud

James Dowty, Owen Jones, Lavie Levi, Raymond Durr

June 2024

1. Introduction

Credit card fraud is a significant issue in the financial industry since anyone who obtains another person's credit card information can use it to make unauthorized transactions. However, there is a lot of information about each transaction that is captured, making machine learning a good potential way to identify unauthorized transactions. In this paper, we try four different machine learning methods (logistic regression, k-nearest neighbors, decision trees, and neural networks) and explore which ones work best and why.

1.1. Data Set

The data set we used is from Kaggle, and is published under the Database Contents License (DbCL). This data was captured over a period of two days in 2013 by European card holders. Since the features contain potentially sensitive information, they have been anonymized and labelled V1, V2, ..., V28. The three exceptions are the transaction amount, the time of the transaction, and whether the transaction was fraudulent or legitimate.

What makes this data set particularly challenging is that it is very imbalanced. Out of the 284,807 transactions, only 492 of them are fraudulent. As we soon found out, the machine learning methods that could handle this imbalance performed the best.

2. Machine Learning Methods

2.1. Logistic Regression

The first machine learning method we tried is one of the simplest: logistic regression. Logistic regression might be useful in this case since it is well suited for binary classification problems, however, the imbalanced nature of the data set may pose problems. Additionally, logistic regression assumes a linear relationship between the features, and the log odds. Since the data set has been anonymized, it is unclear if this is true for this data set.

Implementing a logistic regression model on the data is extremely simple using scikit-learn's built in class. Under the hood, scikit-learn applies the logistic sigmoid function to a linear function of the data to get:

$$p\left(C = 0 \mid \mathbf{x}\right) = \sigma\left(\mathbf{w}^\top \mathbf{x} \ + \ w_0\right) = \frac{1}{1 \ + \ e^{\mathbf{w}^\top \mathbf{x} \ + \ w_0}}$$

In the context of credit card fraud, $p\left(C = 0 \mid \mathbf{x}\right)$ represents the probability a transaction with features $\mathbf{x}$ is fraudulent. The goal is to find the weights $\mathbf{w}$ that give the best predictions for unseen data. By computing the maximum conditional likelihood and taking the negative log, it can be shown that maximizing the conditional likelihood is equivalent to minimizing

$$E(\mathbf{w}) = -\sum_{n=1}^{N}\{t_n \log(\sigma(\mathbf{w}^\top \phi(\mathbf{x}_n))) - (1 - t_n)\log(1 - \sigma(\mathbf{w}^\top \phi(\mathbf{x}_n)))\}$$

where ϕ is our vector basis functions. This is known as cross-entropy loss.

For my training data set, I took the first 240,000 data points from the set. Simply training the default Scikit-learn logistic regression model resulted in the following confusion matrix:

$$\begin{bmatrix} 21963 & 0 \\ 21 & 16 \end{bmatrix}$$

This results in an accuracy of >0.999, however, this can be misleading. Note that since the data is so imbalanced, we can achieve an accuracy of $284807/(284807 + 492) > 0.998$ by simply labelling every transaction as not fraudulent. What I found more interesting is that there were no false positives, meaning if the model suspected a transaction was fraudulent, it was correct every time, however, it only detected about 43% of the fraudulent transactions. I suspected this could be greatly improved through prepossessing, so I took a closer look at the features.

Most of the features seemed to be normally distributed with few outliers except for amount. The transaction amounts ranged from \$0 to about \$25,000, but almost all of the transactions were on the lower end. After trying different scaling methods, I found the Scikit-learns RobustScalar method worked the best. This is likely due to how well RobustScalar handles outliers. After scaling the data and training the model again, I got this confusion matrix:

$$\begin{bmatrix} 21963 & 0 \\ 16 & 21 \end{bmatrix}$$

This is a significant improvement since we still have zero false positives, but our recall increased from 0.43 to 0.57. I then used a grid search to tune the hyper-parameters. These include the regularization constant, the solving algorithm, and the regularization method and found the best results to be 0.1 for the regularization constant with l1 regularization.

When I tested my model on my test data set, I was disappointed in the results. I got the following confusion matrix:

$$\begin{bmatrix} 22787 & 2 \\ 12 & 6 \end{bmatrix}$$

Even after scaling the data and tuning the hyper-parameters, it appears the model did not generalize well to unseen data. This is likely since logistic regression is too simple of a model and our data was highly imbalanced.

2.2. K-nearest Neighbors

The K-Nearest Neighbors (K-NN) model stands out for its simplicity and effectiveness among the algorithms we used to classify the fraudulent credit card transactions data set. This section of the report examines the application of the K-NN model to the dataset, covering data preparation, feature selection, model training, and evaluation. By the end of this section, readers will gain a comprehensive understanding of how the K-NN algorithm was implemented in our project, the challenges encountered, and the outcomes achieved.

Despite its simplicity, the K-NN algorithm can achieve high performance when properly implemented and applied to the right context. For classification tasks, the algorithm assigns class labels by determining which data points in the training set are most similar to each data point in the test set. The class label is chosen from the k most similar "neighbors" through a majority vote. In the context of K-NN, similarity is measured using a distance metric, with common metrics including Euclidean, Manhattan, and Minkowski distances.

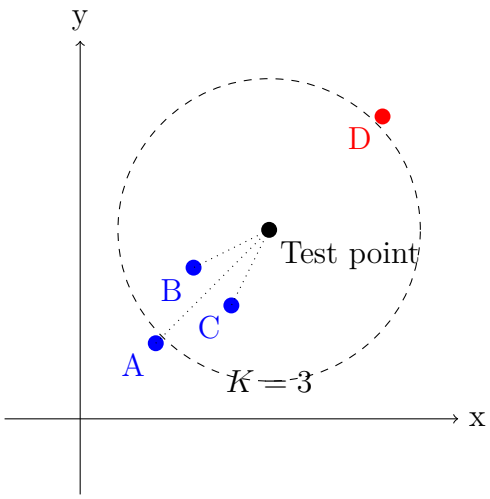

Figure 2.1: An example of a K-Nearest Neighbors diagram with K=3. Points A, B, and C represent members of one class, and D represents a different class. The Test point is classified as blue because the three closest points are all blue.

In this section, we will explore the steps involved in implementing the K-NN model for our dataset. We will start with data preparation, ensuring the data is clean and normalized. Next, we will discuss feature selection to identify the most relevant features for classification. The model training phase will cover the selection of the optimal k value, followed by an evaluation of the model's performance using cross-validation. Finally, we will reflect on the results obtained, describing key insights and any challenges we faced during the implementation process.

This report examines each phase of the K-NN model application to provide a clear and detailed account of how we used the algorithm to detect fraudulent transactions.

Data preprocessing Data preprocessing is crucial for any machine learning task, particularly for K-NN models, which are sensitive to feature scales. For this project, we used Scikit-learn's RobustScaler, which scales data using the interquartile range, minimizing the influence of outliers. We split the dataset into training (80%) and test (20%) sets, fitting the scaler to the training set and transforming both sets accordingly.

Feature selection We excluded features with a correlation close to 0 with the target variable. Specifically, any feature with an absolute correlation value less than 0.1 was removed from the dataset.

Model implementation We implemented the K-NN algorithm using a custom classifier class that includes methods for fitting the model to training data, predicting labels for test data, and evaluating performance. Euclidean distance was used as the distance metric, and we tested various values of k to identify the optimal number of neighbors. To evaluate performance, we performed 5-fold cross-validation using scikit-learn's StratifiedKFold. We averaged the accuracy, precision, and recall for each k value and recorded the results in a dictionary.

Model evaluation and results We tested k values ranging from 5 to 25 in increments of 5. Although each k value performed similarly during hyperparameter testing, k=5 yielded the best results. Ultimately, our model with k=5 achieved excellent performance on the test set, with 99.95% accuracy, 91.76% precision, and 79.59% recall.

Analysis and Discussion

Feature selection The feature selection process of removing features with an absolute correlation of less than 0.1 with the target improved the performance by ignoring features that have a weak relationship with the target variable but have a significant effect on the distance between points. While this primitive technique was useful in the context of our problem, future analysis could benefit from more sophisticated feature selection methods to achieve even better results.

Robustness and Generalization We used 5 folds in our cross-validation to ensure that the model's performance could be generalized across different subsets of the data set. Because each subset of the training set performed consistently well, we can be confident the model generalizes to unseen data.

Model performance While the model's high accuracy shows that K-NN can correctly classify transactions most of the time, it certainly does not paint a complete picture. A credit card transaction classification model must have a high precision rate to minimize the number of legitimate transactions flagged as fraudulent, but the most important metric for evaluating the model's performance in this problem is the recall, the proportion of the fraudulent transactions the algorithm classified as fraudulent. In an ideal situation, we like to design a model that classifies every fraudulent transaction as fraudulent, and while a 79.59% recall rate is decent, it shows that this model still has room to improve.

Limitations and disadvantages of K-NN

Computational efficiency: Calculating the distance between points and predicting labels are considerable bottlenecks in evaluating the test set's performance. To help mitigate this, we wrote JIT-compiled functions to increase the efficiency of calculating the distance between points and predicting class labels, but despite significant efforts, K-NN is still computationally expensive for evaluating large data sets. Future analysis could benefit from dividing the training set into regions, so data points in the test set only have to compute the distance of data points that appear in the same region.

Imbalanced data set: While we used stratification to ensure similar proportions of fraudulent transactions in different subsets of the data, fraudulent credit card transactions account for less than a fifth of a percent of all points in the data set. Because of the imbalance between classes, the legitimate transactions tend to dominate the voting process for higher values of k leading to lower recall rates.

Strengths of K-NN

Simplicity: While optimization for computational efficiency is quite difficult, K-NN models are simple to write. There aren't many hyperparameters to test, so hyperparameter tuning doesn't require testing as many combinations.

Interpretability: It's very easy to comprehend how a K-NN model classifies points in the test set.

Incremental learning: It's easy to add and remove points from the model. In the context of credit card fraud, individuals make transactions every second of the day. When the goal is constantly updating the model with recent transactions, K-NN can easily implement these changes.

2.3. Decision Trees

Our third model for classifying fraudulent credit card transactions is the Decision Tree algorithm. This algorithm is remarkably interpretable and effective. This section will examine the application of Decision Trees on our dataset, its advantages and disadvantages. We will also discuss the performance of our model, which did very well on training data.

Decision Tree Basics Decision Trees divide the input space using many different decision boundaries to classify the data into a specific class. We can imagine these decision boundaries as nodes on a tree. At the root is an unclassified data point and, moving up the tree, we meet one node(decision boundary). This node specifies a threshold value for a specific feature within a sample. The node itself has two branches each which leads to its own sets of nodes and branches. If the feature variable for a particular sample is equal to or less than the threshold value at that node, the point is sent down the left branch. Otherwise, it is sent down the right branch.
Eventually, the sample reaches the last node(called the "leaf" node) and is classified based on the leaf's value. Because the number of decision boundaries is dependent on how "deep" we want the decision tree to be, results are sensitive to the maximum level of depth specified. The model is also somewhat sensitive to our method of determining what variable to split at.

Information Gain The best variable to split at for a given node is the variable and threshold that maximizes the information gain. Information gain is defined as the amount of information within a parent branch after subtracting the weighted amount of information from its two child branches. These weights are established by dividing the number of points in the respective branch by the total number of points from the parent branch.

$$Information\_Gain = Info_{Parent} - (W_L * Info_{LeftChild} + W_R * Info_{RightChild})$$

Information can be defined in a few different ways, but it generally is a measure of randomness. In this project, we use entropy and the Gini index.
The intuition behind this method is that by finding the variable and threshold with the highest information gain, we split the data in the direction of the lowest randomness. For example, let's say The parent has an information of 1 and there are two options for the split. Option 1 has a combined information of 0.3 and option 2 has a combined information of 0.6. At this moment we know option 2 is more random than option 1 and therefore is less desirable. The information gain under option 1 is $1 - 0.3 = 0.7$ while for option 2 it is $1 - 0.6 = 0.4$. Therefore as we can see option 1 is more desirable in terms of randomness and it also has the highest information gain.

Entropy Entropy is a measure of randomness within a sample case.

$$Entropy = \sum_{i=1}^{N} p_i log(p_i)$$

Where p_i is the probability of finding class i in the sample set. This can be calculated by dividing the number of samples of class i in the sample set by the total number of samples within the sample set.

Gini Index The Gini Index measures the likelihood of a sample being misclassified.

$$Gini = 1 - \sum_{i=1}^{N} p_i^2$$

A lower Gini Index means a lower chance of misclassification.
The Gini Index is our main method for measuring information. Because the Gini Index only involves a summation of squares, it is incredibly computationally efficient. Entropy, however, can be relatively slow due to the computation of a logarithm. It should be noted that using Entropy for information gain does lead go more accurate results.

Data Preprocessing

Feature Selection We used one preprocessing technique for the decision trees already discussed: feature selection, as described in the K-NN section.

Subsampling, Stratification, and Train-Test Splitting We also decided to subsample a dataset such that 80% of the data would be class 0(nonfraudulent transactions) and the rest would be class 1(fraudulent transactions). In total, we had a sample of 1000 points. This was done to save computational time. We also of course split the data into an 80-20 train-test split and split the training set into an 80-20 train-validation split.

We decided against using feature scaling or any more advanced techniques because the computational efficiency of decision trees isn't affected by scaling.

Implementation We coded a custom decision tree classifier class in Python (see appendix). Our model uses the Gini Index for information gain and has a maximum depth of 100 branch levels.

Results The results for our decision tree model are extremely impressive.

Class	Precision	Recall	F1 Score
0.0	0.98	1.00	0.99
1.0	1.00	0.90	0.95

Table 2.1: Decision Tree Model Results

With a recall score of 90% for fraudulent cases we are happy with the performance of our model.

Disadvantages of Decision Trees

Computational Efficiency To compute the best split for a given node the algorithm must loop through all unique values for all remaining columns within a dataset and determine the information gain for data split upon that threshold. This takes a very large amount of time and was the main issue when dealing with the algorithm.

Our solution for this problem was to split the dataset into a smaller and stratified subsample. This was done purely for computational efficiency.

Stratification The stratification was probably unnecessary as decision trees will split depending on a specific value anyway. However due to the lower presence of fraudulent transactions, and therefore the decreased randomness and increased chance of misclassification, an unstratified decision tree would probably require more depth and therefore increased runtime. Due to technological limitations, this was unapproachable, but future projects can leverage more powerful machines to work with larger and less modified datasets.

Advantages of Decision Trees

Interpretability Decision trees are extremely interpretable and make a lot of intuitive sense. Simply follow the decision tree, checking the threshold at each node and continuing to the relevant branch to figure out where and why a value was placed where it was placed.

Performance The decision tree also performed very well even given an extremely small training set. Decision Trees therefore mark an incredibly simple and intuitive model to engage in classification when datasets are relatively small.

2.4. Neural Networks

The neural network is the "heavy hammer" of machine learning models. While being the most complex, it is also extremely versatile and can be applied to a multitude of problems including regression and classification. In order to alter the functionality of the model, changes in the neural network's architecture and activations are critical. For the credit card fraud detection problem, we use a feedforward neural network with an output layer of one neuron as appropriate for binary classification.

Data Preprocessing Data preprocessing occurs in two steps:

1. Select features based on correlation with the target.
2. Undersample the majority class.

After loading the data, splitting it into the set of features (X) and the set of targets (y), and shuffling it, feature selection occurs. The two functions

```
calculate_correlations(X, y)
```

and

```
select_features_by_correlation(X, y, threshold=0.1)
```

are responsible for carrying out this task. The features whose correlation with the target set does not meet a specified threshold are rejected, as features of this nature would either be unhelpful in training the model or actively detrimental.

Next, we undersample the majority class: 0 (non-fraudulent cases). Using Scikit-learn's utility function:

```
resample()
```

we are able to specify the majority and minority classes of the dataset, and maintain a balance between the two.

Finally, we split the preprocessed data into training and testing sets. In this example, we reserve 20% for testing.

Neural Network Design The neural network is designed as a feedforward neural network with multiple hidden layers. The hidden layers use the ReLU activation function, which helps the model to learn non-linear relationships in the data. The output layer uses a sigmoid activation function, which is suitable for binary classification tasks like fraud detection.

Key components of the neural network include:

1. **Initialization**: Weights are initialized using a method that scales with the number of input features to each neuron to help with convergence.
2. **Forward Propagation**: Computes the activations of each layer, culminating in the output prediction.
3. **Backward Propagation**: Computes the gradients of the loss with respect to each parameter to update the weights.
4. **Loss Function**: Binary Cross Entropy with L2 regularization is used to penalize large weights and prevent overfitting.

Mathematical Explanation of Forward Propagation Initialization

- `caches` is a Python dictionary used to store the activations and pre-activations of each of the layers for later use in backward propagation.
- A is initialized to the input data X.

- L is the number of layers in the network excluding the input layer.

Forward Propagation through Hidden Layers

For each layer l from 1 to $L-1$, the following operations are performed:

$$Z^l = W^l A^{l-1} + b^l$$

$$A^l = \text{ReLU}(Z^l)$$

Where:

- Z^l is the pre-activation layer.
- W^l and b^l are the weights and biases of the layer l.
- A^l is the activation of layer l.
- ReLU is the activation function of each of the hidden layers.

Forward Propagation through the Output Layer

For the output layer L, we perform the following operation:

$$Z^L = W^L A^{L-1} + b^L$$

$$A^L = \sigma(Z^L)$$

Notice that the only difference is in the choice of the activation function. For the output layer, we choose σ: logistic sigmoid, which is fitting for the binary classification problem at hand.

When forward propagation finishes, `caches` is returned.

Mathematical Explanation of Backward Propagation Initialization

- `grads` is a Python dictionary used to store the gradients.
- L is the number of layers in the network excluding the input layer.
- m is the number of training samples.
- $\hat{Y}$ is the predicted output.

Derivative of the Loss with Respect to the Output

The derivative of the loss (E) with respect to the output layer activation A^L i.e. $\hat{Y}$ is computed as:

$$\frac{\partial E}{\partial A^L} = A^L - Y$$

since the loss function used is binary cross-entropy:

$$E = -\frac{1}{m}\sum_{i=1}^{m}[Y_i \log(A_i^L) + (1 - Y_i)\log(1 - A_i^L)]$$

Gradients of the Output Layer

$$\frac{\partial E}{\partial W^L} = \frac{1}{m}(A^{L-1})^T dZ^L + \frac{\lambda}{m}W^L$$
$$\frac{\partial E}{\partial b^L} = \frac{1}{m}\sum_{i=1}^{m} dZ^L$$

Here, dZ^L is the gradient of the loss with respect to the pre-activation of the output layer, Z^L. The gradients for the weights and biases are updated with L2 regularization.

Propagation of the Gradient Backwards For each layer l from $L-1$ to 1, we compute the gradients. The multidimensional chain rule is used to propagate the gradient backwards:

$$dA^{l-1} = dZ^l dW^l$$
$$dZ^l = dA^l \cdot \text{ReLU}'(Z^l)$$

Where ReLU' is the derivative of the activation function: ReLU.

$$\frac{\partial E}{\partial W^l} = \frac{1}{m}(A^{l-1})^T dZ^l + \frac{\lambda}{m}$$

The gradients of the hidden layers' weights are calculated in a similar way to that of the output layer. The same applies to the biases:

$$\frac{\partial E}{\partial b^l} = \frac{1}{m}\sum_{i=1}^{m} dZ^l$$

Finally `grads` is returned.

Updating the Parameters For the update formula, we use Stochastic Gradient Descent with a specified learning rate. Using `grads`, calculated in the backprapagation step, the parameters are updated as follows:

$$W^l \leftarrow W^l - \eta \cdot \frac{\partial E}{\partial W^l}$$
$$b^l \leftarrow b^l - \eta \cdot \frac{\partial E}{\partial b^l}$$

Where η is the learning rate.

Hyperparameter Tuning Hyperparameter tuning involves searching for the best set of hyperparameters that yields the highest performance of the neural network.

- **Layer Sizes**: The architecture of the neural network can significantly impact its ability to learn from data. Different combinations of the number of layers and the number of neurons in each layer are tested.
- **Learning Rate**: This controls the size of the steps the optimizer takes during the weight update. A small learning rate ensures the model converges smoothly, while a large learning rate speeds up the training process but may overshoot the optimal solution.
- **L2 Regularization (l2_lambda)**: This term helps prevent overfitting by penalizing large weights in the model. Different values of the regularization term are tested to find the optimal balance between fitting the training data and generalizing to unseen data.

The grid search process evaluates all possible combinations of these hyperparameters and selects the best set based on the F1 score, which balances precision and recall, making it suitable for the imbalanced nature of the fraud detection problem.

Results

```
Best Hyperparameters:
{'layer_sizes': [11, 64, 32, 16, 8, 1], 'learning_rate': 0.1, 'l2_lambda': 0.0001}
Best F1 Score: 0.9359605911330049
```

The above hyperparameters were selected as the best choices for the architecture of the network, the learning rate for SGD, and the L2 regularization term for the loss function after executing the grid search process.

Experimentation

Results

```
Epoch 1/100, Loss: 1.1003273306324894
Epoch 2/100, Loss: 0.5723685925046424
Epoch 3/100, Loss: 0.4819781463530347
...
Epoch 98/100, Loss: 0.15716932042262496
Epoch 99/100, Loss: 0.1569818336369603
Epoch 100/100, Loss: 0.15679738361540987

Accuracy: 0.934010152284264
```

```
Precision: 0.9693877551020408
Recall: 0.9047619047619048
F1 Score: 0.9359605911330049
Confusion Matrix:
[[89  3]
 [10 95]]
```

Analysis The confusion matrix reflects the following:

- True Positives: 95
- True Negatives: 89
- False Negatives: 10
- False Positives: 3

These scores, combined with the high accuracy of 93.40% suggest that the model correctly predicts a high proportion of both fraudulent and non-fraudulent transactions. The precision of 96.94% indicates that among all transactions predicted as fraudulent, a significantly high percentage of them were indeed so. This is particularly important in credit card fraud detection due to the nature of the dataset, being highly imbalanced, causing precision to be a difficult score to optimize.

All in all, the neural network exhibits strong performance in terms of all of the evaluation metrics. Further work could focus on improving recall, as there are still some fraudulent transactions that were missed, perhaps with methods such as oversampling the minority class or further tuning the hyperparameters.

3. Final Thoughts

We found that the most important metric for the performance of our models was recall since it is the number of true positives, and the top-performing models in this sense were the neural network and decision tree. Both of these models required under-sampling and re-stratification of the data to help reduce computational complexity. One interesting thing to note is that the logistic regression had very high precision, meaning if it identified a transaction as fraudulent then it is extremely likely to be fraudulent in reality. Because of this, we think a possible strategy would be to use multiple machine learning models in conjunction and compare the results.

4. Appendix

Logistic Regression Colab Notebook
Decision Tree Colab Notebook

4.1. Logistic Regression Code

```
train, test, val = preprocessed_df[:240000], preprocessed_df
   [240000:262000], preprocessed_df[262000:]
X_train, y_train = train.drop('Class', axis=1), train['Class']
X_test, y_test = test.drop('Class', axis=1), test['Class']
X_val, y_val = val.drop('Class', axis=1), val['Class']

model = LogisticRegression(max_iter=1000)
model.fit(X_train, y_train)

y_pred = model.predict(X_test)

print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
print("\nAccuracy Score:")
print(accuracy_score(y_test, y_pred))


from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV

pipeline = Pipeline([
    ('scaler', RobustScaler()),
    ('classifier', LogisticRegression())
])

param_grid = {
    'classifier__C': [0.01, 0.1, 1, 10, 100],
    'classifier__solver': ['liblinear', 'lbfgs', 'saga'],
    'classifier__penalty': ['l2', 'l1']
}

```

```
grid_search = GridSearchCV(estimator=pipeline, param_grid=
    param_grid, cv=5, scoring='recall', verbose=2, n_jobs=-1)

grid_search.fit(X_train, y_train)

print("Best Parameters:", grid_search.best_params_)
print("Best Score:", grid_search.best_score_)


train, test, val = preprocessed_df[:240000], preprocessed_df
    [240000:262000], preprocessed_df[262000:]
X_train, y_train = train.drop('Class', axis=1), train['Class']
X_test, y_test = test.drop('Class', axis=1), test['Class']
X_val, y_val = val.drop('Class', axis=1), val['Class']

model = LogisticRegression(max_iter=1000, C=0.1, penalty='l1',
    solver='liblinear')
model.fit(X_train, y_train)

y_pred = model.predict(X_test)

print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))
print("\nClassification Report:")
print(classification_report(y_test, y_pred))
print("\nAccuracy Score:")
print(accuracy_score(y_test, y_pred))


y_pred = model.predict(X_val)

print("Confusion Matrix:")
print(confusion_matrix(y_val, y_pred))
print("\nClassification Report:")
print(classification_report(y_val, y_pred))
print("\nAccuracy Score:")
print(accuracy_score(y_val, y_pred))
```

4.2. K-NN Code

K-NN Code

```
import pandas as pd
import numpy as np
from numba import jit, njit, prange
from sklearn.model_selection import train_test_split,
   StratifiedKFold
from sklearn.preprocessing import RobustScaler

df = pd.read_csv('creditcard.csv')
df_clean = df[['V11','V4','V1','V18','V7','V3','V16','V10','V12
   ','V14','V17','Class']]

target_correlations = df.corr()['Class'].sort_values(ascending=
   False)

X = df_clean.drop('Class', axis=1).values
y = df_clean['Class'].values

X_train, X_test, y_train, y_test = train_test_split(X, y,
   stratify=y, test_size=0.2, random_state=42)

scaler = RobustScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s = scaler.transform(X_test)

@njit(parallel=True, fastmath=True)
def euclidean_distance(X_train, X_test):
    num_train, num_features = X_train.shape
    num_test = X_test.shape[0]
    distances = np.empty((num_test, num_train), dtype=np.
       float64)

    for i in prange(num_test):
        for j in range(num_train):
            diff = X_train[j] - X_test[i]
            distances[i, j] = np.sqrt(np.sum(diff ** 2))

    return distances

@njit(parallel=True, fastmath=True)
def predict_labels(distances, y_train, k):
    num_test = distances.shape[0]
    predictions = np.empty(num_test, dtype=np.int32)
```

```
38
39      for i in prange(num_test):
40          neighbors_indices = np.argsort(distances[i])[:k]
41          neighbor_labels = y_train[neighbors_indices]
42          count_1 = np.sum(neighbor_labels == 1)
43          count_0 = np.sum(neighbor_labels == 0)
44          predictions[i] = 1 if count_1 > count_0 else 0
45
46      return predictions
47
48  class KNNClassifier:
49
50      def __init__(self, k=5):
51          self.k = k
52
53      def fit(self, X_train, y_train):
54          self.X_train = X_train
55          self.y_train = y_train
56
57      def predict(self, X_test, batch_size=100):
58          num_samples = X_test.shape[0]
59          predictions = np.empty(num_samples, dtype=np.int32)
60
61          for i in range(0, num_samples, batch_size):
62              end_index = min(i + batch_size, num_samples)
63              batch_X_test = X_test[i:end_index]
64              distances = euclidean_distance(self.X_train,
                    batch_X_test)
65              batch_predictions = predict_labels(distances, self.
                    y_train, self.k)
66              predictions[i:end_index] = batch_predictions
67
68          return predictions
69
70      def evaluate(self, X_test, y_test):
71          self._predictions = self.predict(X_test)
72
73          self._accuracy = np.sum(self._predictions == y_test) /
                len(y_test)
74
75          # Compute the number of true positives, false positives
                , and false negatives
```

```
76          true_positives = np.sum((self._predictions == 1) & (
               y_test == 1))
77          false_positives = np.sum((self._predictions == 1) & (
               y_test == 0))
78          false_negatives = np.sum((self._predictions == 0) & (
               y_test == 1))
79
80          # Compute precision and recall
81          self._precision = true_positives / (true_positives +
               false_positives)
82          self._recall = true_positives / (true_positives +
               false_negatives)
83
84      def predictions(self):
85          return self._predictions  # Return the predictions
86
87      def metrics(self):
88          return np.array([self._accuracy, self._precision, self.
               _recall])  # Return metrics
89
90  def cross_validate_knn(X, y, k_values, n_splits=5):
91      kf = StratifiedKFold(n_splits=n_splits, shuffle=True,
           random_state=42)
92      results = []
93
94      for k in k_values:
95          knn = KNNClassifier(k=k)
96          accuracies = np.array([])
97          precisions = np.array([])
98          recalls = np.array([])
99
100         for train_index, val_index in kf.split(X, y):
101             X_train, X_val = X[train_index], X[val_index]
102             y_train, y_val = y[train_index], y[val_index]
103
104             knn.fit(X_train, y_train)
105             knn.evaluate(X_val, y_val)
106             accuracy, precision, recall = knn.metrics()
107
108             accuracies = np.append(accuracies, accuracy)
109             precisions = np.append(precisions, precision)
110             recalls = np.append(recalls, recall)
```

```
111
112         results.append({
113             'k': k,
114             'accuracy': np.mean(accuracies),
115             'precision': np.mean(precisions),
116             'recall': np.mean(recalls)
117         })
118         print({
119             'k': k,
120             'accuracy': np.mean(accuracies),
121             'precision': np.mean(precisions),
122             'recall': np.mean(recalls)
123         })
124
125     return results
126
127 results = cross_validate_knn(X_train_s, y_train, np.arange(5,
    30, 5))
128
129 model = KNNClassifier(k=5)
130 model.fit(X_train_s, y_train)
131 model.evaluate(X_test_s, y_test)
132 model.metrics()
```

4.3. Decision Tree Code

```
1  class Node():
2      def __init__(self, feature = None, threshold = None, left_branch = None, right_branch =
       None, info_gain = None, leaf_value = None):
3          # for branches
4          self.feature = feature
5          self.threshold = threshold
6          self.left_branch = left_branch
7          self.right_branch = right_branch
8          self.info_gain = info_gain
9          # for leafs
10         self.value = leaf_value
```

Listing 1: Node Class

```
1  class DecisionTrees():
2      def __init__(self, max_depth = 100, min_samp = 2):
3          ''' max_depth is the number of rows of nodes allowed on the tree
4              min_samp is the minimum number of samples within a node allowing for a split
5              such that if min_samp is 2 the dataset can be split into another two branches if
6              there are at least two data points
7          '''
```

```
        self.max_depth = max_depth
        self.min_samp = min_samp
        self.tree = None
    def fit(self, sample, target):
        self.tree = self.grow_tree(sample, target)
    def grow_tree(self, sample, target, depth = 0):
        num_samp, num_feat = sample.shape
        # if there are enough samples in the dataset and the tree is not too deep
        if num_samp >= self.min_samp and depth <= self.max_depth:
            # find best split
            best_split = self.best_split(sample, target, num_samp, num_feat)
            if best_split['info_gain'] > 0:
                left_branch = self.grow_tree(best_split['data_left'], best_split['
                    target_left'], depth = depth + 1)
                right_branch = self.grow_tree(best_split['data_right'], best_split['
                    target_right'], depth = depth + 1)
                # return decision node
                return Node(best_split['feature'], best_split['threshold'], left_branch,
                    right_branch, best_split['info_gain'])
        # get leaf node
        val = self.get_leaf_val(target)

        return Node(leaf_value = val)
    def best_split(self, X, y, num_samp, num_feat):
        best_gain = -np.Inf
        best_split = {}

        #feature loop
        for feature in range(num_feat):
            unique_val = np.unique(X[:, feature]) #unique values are potential thresholds
            #now loop through potential thresholds
            for threshold in unique_val:
                #get left and right split
                left = np.array([sample for sample in np.c_[X,y]if sample[feature] <=
                    threshold])
                right = np.array([sample for sample in np.c_[X,y] if sample[feature] >
                    threshold])
                if len(left) > 0 and len(right) > 0: #if they're not null
                    gain = information_gain(left, right, type = 'gini')
                    if gain > best_gain:
                        best_gain = gain
                        #data
                        best_split['data_left'] = left[:, :-1]
                        best_split['target_left'] = left[:, -1]
                        best_split['data_right'] = right[:, :-1]
                        best_split['target_right'] = right[:, -1]
                        #node parameters
                        best_split['info_gain'] = gain
                        best_split['feature'] = feature
                        best_split['threshold'] = threshold
        return best_split
    def get_leaf_val(self, target):
        target = list(target)
        return max(target,key = target.count) # returns the unique value which shows up the
            most amount of times in the dataset
    def predict(self,X):
        predictions = [self.move_to_leaf(samp, self.tree) for samp in X]
        return predictions
    def move_to_leaf(self, samp, node):

        #for leaf nodes
        if node.value != None:
            return node.value
        if samp[node.feature] <= node.threshold:
            return self.move_to_leaf(samp, node.left_branch)
```

```
        else:
            return self.move_to_leaf(samp, node.right_branch)
```

Listing 2: DecisionTree Class

4.4. Neural Network Code

Link to Google Colab notebook: NeuralNetwork.ipynb

```
class NeuralNetwork:
    def __init__(self, layer_sizes, learning_rate=0.01, l2_lambda=0.001):
        self.layer_sizes = layer_sizes
        self.learning_rate = learning_rate
        self.l2_lambda = l2_lambda
        self.parameters = self.initialize_parameters(layer_sizes)

    def initialize_parameters(self, layer_sizes):
        np.random.seed(42)
        parameters = {}
        for i in range(1, len(layer_sizes)):
            parameters[f'W{i}'] = np.random.randn(layer_sizes[i-1], layer_sizes[i]) * np.
                sqrt(
                2. / layer_sizes[i-1])
            parameters[f'b{i}'] = np.zeros((1, layer_sizes[i]))
        return parameters

    def relu(self, Z):
        return np.maximum(0, Z)

    def relu_derivative(self, Z):
        return (Z > 0).astype(float)

    def sigmoid(self, Z):
        return 1 / (1 + np.exp(-Z))

    def binary_cross_entropy_loss(self, Y, Y_hat):
        m = Y.shape[0]
        loss = -(1/m) * np.sum(Y * np.log(Y_hat) + (1 - Y) * np.log(1 - Y_hat))
        # Add L2 regularization
        L2_reg = (self.l2_lambda / (2 * m)) * sum(
            np.sum(np.square(self.parameters[f'W{i}'])) for i in range(1, len(self.
                layer_sizes)))
        return loss + L2_reg

    def forward_propagation(self, X):
        caches = {'A0': X}
        A = X
        L = len(self.layer_sizes) - 1
        for i in range(1, L):
            Z = np.dot(A, self.parameters[f'W{i}']) + self.parameters[f'b{i}']
            A = self.relu(Z)
            caches[f'Z{i}'] = Z
            caches[f'A{i}'] = A
        ZL = np.dot(A, self.parameters[f'W{L}']) + self.parameters[f'b{L}']
        AL = self.sigmoid(ZL)
        caches[f'Z{L}'] = ZL
        caches[f'A{L}'] = AL
        return caches

    def backward_propagation(self, X, Y, caches):
        grads = {}
        L = len(self.layer_sizes) - 1
```

```
        m = X.shape[0]
        Y_hat = caches[f'A{L}']

        dZL = Y_hat - Y.reshape(Y_hat.shape)
        grads[f'dW{L}'] = (np.dot(caches[f'A{L-1}'].T, dZL) / m) + (
            self.l2_lambda / m) * self.parameters[f'W{L}']
        grads[f'db{L}'] = np.sum(dZL, axis=0, keepdims=True) / m

        dA_prev = np.dot(dZL, self.parameters[f'W{L}'].T)
        for i in reversed(range(1, L)):
            dZ = dA_prev * self.relu_derivative(caches[f'Z{i}'])
            grads[f'dW{i}'] = (np.dot(caches[f'A{i-1}'].T, dZ) / m) + (
                self.l2_lambda / m) * self.parameters[f'W{i}']
            grads[f'db{i}'] = np.sum(dZ, axis=0, keepdims=True) / m
            dA_prev = np.dot(dZ, self.parameters[f'W{i}'].T)
        return grads

    def update_parameters(self, grads):
        L = len(self.layer_sizes) - 1
        for i in range(1, L+1):
            self.parameters[f'W{i}'] -= self.learning_rate * grads[f'dW{i}']
            self.parameters[f'b{i}'] -= self.learning_rate * grads[f'db{i}']

    def train(self, X, Y, epochs, print_loss=False):
        for epoch in range(epochs):
            caches = self.forward_propagation(X)
            Y_hat = caches[f'A{len(self.layer_sizes) - 1}']
            loss = self.binary_cross_entropy_loss(Y.reshape(-1, 1), Y_hat)
            if print_loss:
                print(f'Epoch {epoch+1}/{epochs}, Loss: {loss}')
            grads = self.backward_propagation(X, Y.reshape(-1, 1), caches)
            self.update_parameters(grads)

    def predict(self, X, threshold=0.5):
        caches = self.forward_propagation(X)
        predictions = (caches[f'A{len(self.layer_sizes) - 1}'] > threshold).astype(int)
        return predictions
```

Listing 3: NeuralNetwork Class

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.utils import shuffle, resample

def calculate_correlations(X, y):
    correlations = X.corrwith(y)
    return correlations.abs().sort_values(ascending=False)

def select_features_by_correlation(X, y, threshold=0.1):
    correlations = calculate_correlations(X, y)
    selected_features = correlations[correlations > threshold].index.tolist()
    return selected_features

# Load the data
df = pd.read_csv('creditcard.csv')
df = shuffle(df)

# Split features and target
X = df.drop(columns=['Class'])
y = df['Class']

# Select relevant features based on correlation with the target variable
selected_features = select_features_by_correlation(X, y, threshold=0.1)
```

```
X_selected = X[selected_features]

# Combine X and y into one DataFrame
df_selected = pd.concat([X_selected, y], axis=1)

# Separate the majority and minority classes
df_majority = df_selected[df_selected.Class == 0]
df_minority = df_selected[df_selected.Class == 1]

# Undersample the majority class
df_majority_undersampled = resample(df_majority,
                                    replace=False,
                                    n_samples=len(df_minority)
                                    )

# Combine minority class with undersampled majority class
df_undersampled = pd.concat([df_majority_undersampled, df_minority])

# Shuffle the combined data
df_undersampled = shuffle(df_undersampled)

# Split features and target
X_undersampled = df_undersampled.drop(columns=['Class'])
y_undersampled = df_undersampled['Class']

# Convert to NumPy arrays
X_selected_values = X_undersampled.values
y_values = y_undersampled.values

# Split the data
X_train, X_test, y_train, y_test = train_test_split(
    X_selected_values, y_values, test_size=0.2)

# Standardize the data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

Listing 4: Data Preprocessing

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,
    confusion_matrix
import itertools
from joblib import Parallel, delayed

# Define the function to train and evaluate the neural network
def train_and_evaluate(layer_sizes, learning_rate,
            l2_lambda, X_train, y_train, X_test, y_test, epochs=100):
    print(f'Testing layer_sizes={layer_sizes}, learning_rate={learning_rate}, l2_lambda={
        l2_lambda}')

    # Initialize the neural network with the current hyperparameters
    nn = NeuralNetwork(
        layer_sizes, learning_rate=learning_rate, l2_lambda=l2_lambda)

    # Train the neural network
    nn.train(X_train, y_train, epochs=epochs)

    # Make predictions
    y_pred = nn.predict(X_test)

    # Evaluate the model
    f1 = f1_score(y_test, y_pred)
    print(f'F1 Score: {f1}')
```

```
23
24      return (layer_sizes, learning_rate, l2_lambda, f1)
25
26  # Define the grid search function
27  def grid_search_parallel(X_train, y_train, X_test, y_test,
28      layer_sizes_list, learning_rates, l2_lambdas, epochs=100, n_jobs=-1):
29      # Generate all possible combinations of hyperparameters
30      param_combinations = list(
31          itertools.product(layer_sizes_list, learning_rates, l2_lambdas))
32
33      # Use joblib to parallelize the training and evaluation
34      results = Parallel(n_jobs=n_jobs)(
35          delayed(train_and_evaluate)(
36              layer_sizes, learning_rate, l2_lambda, X_train, y_train, X_test, y_test, epochs)
37          for layer_sizes, learning_rate, l2_lambda in param_combinations
38      )
39
40      # Find the best hyperparameters based on F1 score
41      best_params = None
42      best_f1 = 0
43      for layer_sizes, learning_rate, l2_lambda, f1 in results:
44          if f1 > best_f1:
45              best_f1 = f1
46              best_params = {
47                  'layer_sizes': layer_sizes,
48                  'learning_rate': learning_rate,
49                  'l2_lambda': l2_lambda
50              }
51
52      return best_params, best_f1
53
54  # Define the hyperparameter ranges
55  layer_sizes_list = [
56      [X_train_scaled.shape[1], 16, 8, 1],
57      [X_train_scaled.shape[1], 32, 16, 8, 1],
58      [X_train_scaled.shape[1], 64, 32, 16, 8, 1]
59  ]
60  learning_rates = [0.001, 0.01, 0.1]
61  l2_lambdas = [0.0001, 0.001, 0.01]
62
63  # Perform grid search with parallel processing
64  best_params, best_f1 = grid_search_parallel(
65      X_train_scaled, y_train, X_test_scaled, y_test,
66          layer_sizes_list, learning_rates, l2_lambdas, epochs=100, n_jobs=-1)
67
68  # Print the best hyperparameters and F1 score
69  print('Best Hyperparameters:')
70  print(best_params)
71  print(f'Best F1 Score: {best_f1}')
```

Listing 5: Hyperparameter Tuning