# Ethereum Fraud Detection: Using Logistic Regression and K-Nearest Neighbour Models

**Lilly Sharples** lillysharples@uga.edu
*Department of Computer Science, University of Georgia*
**Kimberly Nguyen** Kimberly.Nguyen@uga.edu
*Department of Computer Science, University of Georgia*

## Abstract

The objective of this research is to create a model that can accurately classify an Ethereum transaction as fraudulent or not. Ethereum is a community-driven technology powering the cryptocurrency (ETH), as well as, thousands of decentralized applications. Our focus was that of the cryptocurrency. The imbalanced dataset used consisted of 9841 entries. Aside from the index column, there were 50 columns of information regarding the transaction, flagging fraud with a '1' and non fraud with a '0'.

To predict these fraudulent transactions, we created a Logistic Regression and K-Nearest Neighbour model. In order to get the most accurate model (LR = 0.8886, KNN_Brute = 0.9122, KNN_KD_Tree = 0.9122, KNN_Ball_Tree = 0.9073), we trained/tested each model with different variations, as well as, different ways of classifying the data. Overall, the K-Nearest Neighbour KD Tree and Brute Force classifiers gave the most accurate fraud detection models, both having an accuracy of 0.9122.

## 1. Introduction

Ethereum is a technology frequently referred to as a programmable blockchain. Along with having the cryptocurrency ETH, the community-built Ethereum platform "powers applications that everyone can use and no one can take down"[3]. This benefits many users globally as it eliminates the need for financial intermediaries. Since Ethereum is a decentralized, peer-to-peer network, there are no ways for the government to prevent/monitor fraudulent or illegal transactions. Fraudulent transactions are common through cryptocurrencies, such as Ethereum, as there is no way to confirm a user's identity/reputation through their digital address. It is possible to verify the status of a transaction by inspecting the blockchain, but the blockchain does not contain any personal information regarding the sender/receiver. On a global level, some governments are becoming interested in collecting personal data on individuals partaking in cryptocurrency transactions. There are 15 global jurisdictions, including the US, UK, France, Germany, and Canada, that are planning to develop a system to track these digital transactions. The main goal of this system is to prevent the illicit use of cryptocurrency, such as money laundering or funding terrorism.

Examples of fraudulent transactions can be fake crypto exchanges or wallets. An individual may find a crypto trading exchange with low trading fees or other benefits that appear too good to be true. It is common for these fraudulent exchanges to take your money, not allowing you to withdraw your funds after the irreversible digital deposit has been completed. Any scam possible with fiat currency is also possible with cryptocurrency, especially investment scams such as Ponzi schemes. Similar to traditional scams where a scammer takes a consumer's money without providing the service/good in return, purchases made with crypto can easily be scams. With all the possibilities of fraud in the crypto markets, we were curious to see whether there were any commonalities between these fraudulent transactions.

Our dataset contained 50 different attributes, some of those being the total number of transactions, the coin most sent to/received from a wallet, wallet address, and total wallet balance. In order to access the attributes associated with each Ethereum transaction, it was first necessary to rename each column name by replacing each space with an underscore. There were also some attributes in string form, such as the name of the most commonly sent crypto token for the account. These strings were transformed to numerical values using LabelEncoder and fit_transform from the sklearn.preprocessing package [8].

We experimented by trying different models: Logistic Regression and K-Nearest Neighbour (Brute, KD Tree, Ball Tree). After creating each of these models, we trained them using our dataset. A few attributes were not included, as they were irrelevant to our goal of detecting fraud. For example, the specific Ethereum account address linked to a user would not be beneficial, as there were 9816 unique values out of 9841 total entries for this attribute in the dataset. More information regarding the dataset can be found in the **Data-Preprocessing** section of this paper.

With 9841 total entries, we split our data into training and testing sets with the sklearn train_test_split method [8], with a 75 training/25 testing split. This dataset is imbalanced; our dataset contained 7662 entries flagged with a '0' (non-fraud) and 2179 flagged with a '1' (fraud). To address this potential issue, we used the Synthetic Minority Oversampling Technique (SMOTE) from the imbalanced-learn python package [4].

For logistic regression, the data needed to be stratified and cross validated to help further ensure that the proportion of flags 0's (non-fraud) and 1's (fraud) are balanced in the testing and training datasets using StratifiedKFold() and cross_val_score() from the sklearn library [8]. While training the KNN algorithms, an important step was to determine the optimal K-value for each of the three models. By looping through and testing different K-values, we were able to find the best values to optimize the accuracy of each model. This loop consisted of KNeighborsClassifier(), RepeatedKFold(), and cross_val_score()- from the sklearn library [8]. More specifics of our experiment process can be found in the **Experiments** section.

After evaluating the performance of each model, the KNN models (Brute, KD Tree, Ball Tree) consistently had better accuracies than the logistic regression model (LR = 0.8886, KNN_Brute = 0.9122, KNN_KD_Tree = 0.9122, KNN_Ball_Tree = 0.9073). However, precision for each model was roughly the same (LR = 0.9695, KNN_Brute = 0.9623, KNN_KD_Tree = 0.9623, KNN_Ball_Tree = 0.9646). Although the results seem significant, there were some factors that may have mislead the results. For example, the models were based on limited information on fradulent transactions even though the dataset was oversampled of the flag value of '1.' Also, the different preprocessing steps between the two models may have affected the results' disparity. More information on the results' analysis can be found in the **Analysis** section.

## 2. Data – Preprocessing

The data, Ethereum Fraud Detection Dataset [1], was found on Kaggle.com. It was generated by Vagif Aliyev as an open database for public use. There are 9841 total entries of Ethereum transactions in this dataset, with 51 columns(one column being the index, one column being the FLAG value). The dataset is imbalanced with non-fraudulent transactions being much more frequent. Of the 9841 total entries, 7662 entries were flagged with a '0' (non-fraud) and 2179 were flagged with a '1' (fraud). To address this potential issue, we used the Synthetic Minority Oversampling Technique (SMOTE) from the imbalanced-learn python package [4].

```
sm = SMOTE()
X_train, Y_train = sm.fit_resample(X_train, Y_train)
```

SMOTE works to balance the distribution of a dataset by synthesizing new examples from the minority class. This technique randomly selects a member of the minority class, finds a few of the nearest neighbours, and "synthetic example is created at a randomly selected point between the two" [2]. In the case of our dataset, fraudulent transactions with a flag value of '1' are the minority class, consisting of only 22% of the total data. The accuracy of our KNN models before and after applying SMOTE are as follows:

| | **Brute** | **KD Tree** | **Ball Tree** |
|---|---|---|---|
| **Original** | .9297 | .9297 | .9281 |
| **Oversampled** | .9106 | .9106 | .9053 |

While using this dataset, there were missing values for some of the columns. Of the 51 total columns, we used 21 columns of attributes that did not contain any missing values to optimize our accuracy. Columns, such as index, FLAG, address number, and other less relevant attributes, were not included in our training dataset. We did not scale the chosen input features. The following features were extracted to use in our models to make predictions:

*'Avg_min_between_sent_tnx', 'Avg_min_between_received_tnx', 'Time_Diff_between_first_and_last_(Mins)', 'Sent_tnx', 'Received_Tnx', 'Number_of_Created_Contracts', 'Unique_Received_From_Addresses', 'Unique_Sent_To_Addresses', 'min_value_received', 'max_value_received_', 'avg_val_received', 'min_val_sent', 'max_val_sent', 'avg_val_sent', 'min_value_sent_to_contract', 'max_val_sent_to_contract', 'avg_value_sent_to_contract', 'total_transactions_(including_tnx_to_create_contract', 'total_Ether_sent', 'total_ether_received', 'total_ether_sent_contracts', 'total_ether_balance' [1]*

The first step to making this data accessible was to remove the spaces in the column titles. Using Python's replace method, any spaces in column titles were replaced with underscores.

There were two attributes in our dataset that were strings: the names of the most sent and received crypto tokens for each account. In order to incorporate these columns, it was necessary to turn these strings into numerical data. To accomplish this, we used a LabelEncoder, and then used the fit_transform method on each of these columns. Both of these were from the preprocessing package [8].

For the logistic regression model, the train_test_split method from the model_selection package [8] was used to split our original dataset into training and testing sets. Our parameters to train_test_split were X: containing our dataframe of chosen attributes, Y: containing the 0/1 fraud FLAG for each entry, the default train_size of .25, and the default test_size value of .75. Cross validation was performed on the data using the StratifiedKFold method from the model_selection package [8] with the parameter n_splits of 10. 10 stratified subsets were iterated through to compute the mean accuracy of the model using the cross_val_score method from the model_selection package [8]. This preserves class proportions in datasets while creating 10 random subsets to evaluate more reliable accuracies (stratified 10-fold cross-validation).

x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.25) **
k_fold = StratifiedKFold(n_splits=10)

For the K-Nearest-Neighbour models, the train_test_split method from the model_selection package [8] was used to split our original dataset into training and testing sets. Our parameters to train_test_split were X: containing our dataframe of chosen attributes, Y: containing the 0/1 fraud FLAG for each entry, a random_state value of 3, the default train_size of .25, and the default test_size value of .75. The random_state value was chosen by manually testing different values and comparing the accuracy.

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, random_state=3) **

** X and Y refer to data (refer to features listed above) and target (FLAG=0,1), respectively.

## 3. Experiments

Since we are attempting to classify fraud in Ethereum transactions, logistic regression was a viable option since it separates instances into the appropriate classes (fraud or non-fraud). Logistic regression can be calculated using the following equation, in which a binary variable is mapped:

$$h_\theta(x) = \frac{1}{1+e^{-\Sigma_{j=0}^{k} \theta_j x_j}}.$$

[6]

After preprocessing the data, a logistic regression model was created that allowed the max number of iterations to be 100000000.

logistic = LogisticRegression(solver='lbfgs',max_iter=100000000)

The model was then fitted given the training data.

```
logistic.fit(x_train, y_train)
```

Scores were evaluated within each iteration using cross-validation score method.

```
scores = cross_val_score(logistic,X=x_test,y=y_test,cv=k_fold)
```

The K-Nearest-Neighbour framework is proven to be better suited for fraud detection in this dataset than the Logistic Regression framework. When trained, the KNN algorithm "computes the distance between the new data point with every training example...picks K entries in the database which are closest to the new data point" [5] to be the neighbours, and assigns the class based on the most common class among those neighbours.

Within the KNN framework, we tested three variations of the algorithm. The KNN Brute Force compares each entry in the test set by calculating the euclidean distance to all the entries in the training set. Brute Force is best used for smaller samples as every data point is visited, increasing the complexity for larger data sets. The KNN K-D Tree variation focuses on reducing the number of distance comparisons/calculations by splitting data into a binary tree around the median instance. Best used in our dataset, the KNN Ball Tree "assumes the data in multidimensional dimensional space and creates the nested hyper spheres" [7], which is said to improve performance compared to the other algorithm variations.

```
knn1 = KNeighborsClassifier(n_neighbors=18, algorithm='brute')
knn1.fit(X_train, Y_train)
knn2 = KNeighborsClassifier(n_neighbors=18, algorithm='kd_tree')
knn2.fit(X_train, Y_train)
knn3 = KNeighborsClassifier(n_neighbors=17, algorithm='ball_tree')
knn3.fit(X_train, Y_train)
predictedKnn1 = knn1.predict(X_test)
predictedKnn2 = knn2.predict(X_test)
predictedKnn3 = knn3.predict(X_test)
```

When using each of these models on our dataset, we did not notice any particular model requiring significantly more time to run than the others. The actual complexities are given below, where D is the number of dimensions/attributes and N is the number of samples [7].

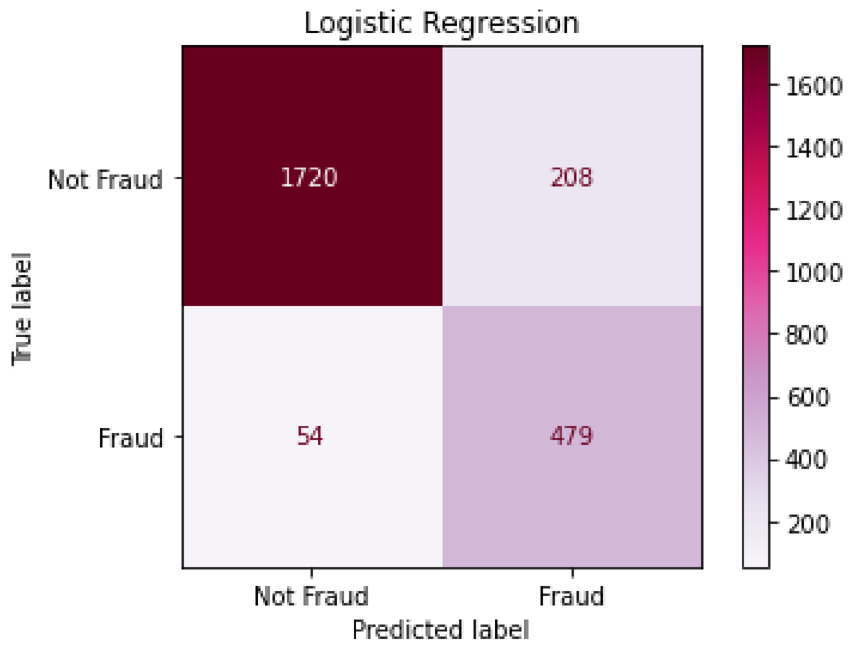
| | |
|---|---|
| Logistic Regression | O[D] |
| KNN Brute Force | O[DN$^2$] |
| KNN K-D Tree | O[D N *log(N)] |
| KNN Ball Tree | O[Dlog(N)] |

To perform the logistic regression experiment, a 2015 MacBook Pro with 2.9 GHz Dual-Core Intel Core i5 processor in version 11.2.2 of macOS Big Sur was used. To perform the KNN experiments, a 2018 MacBook Air with 1.6 GHz Dual-Core Intel Core i5 processor in version 10.15.4 of macOS Catalina was used. They were coded in the The Scientific Python Development Environment, more commonly referred to as Spyder. Git version control software was also used to track changes remotely.

## 4. Analysis

The results are significant, because the data was oversampled to include more fraudulent account information than non-fraudulent account information (oversampling of minority class) to better detect fraud. Overall, all of the KNN models (Brute, KD Tree, Ball Tree) have consistently higher accuracies compared to the logistic regression model, which indicates that the two ML framework results are significantly different. It is possible that the partitioned data may have led to misleading results. Even though the data was oversampled of the minority class, the original dataset is highly imbalanced since approximately only 22% of the data indicated fraud (FLAG=1), so the models were based on limited information on fraudulent accounts. Also, the data preprocessing slightly differs between the two models, which may have affected the accuracies. Stratification and cross validation were performed in the logistic regression model, while they were not performed in the KNN models.

### 4.1 Logistic Regression

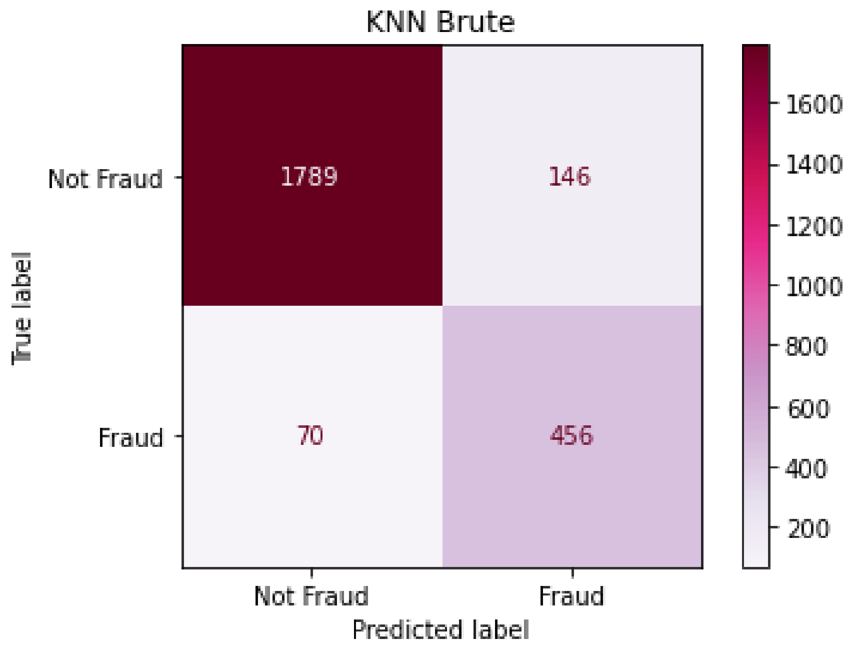


Success Rate (Accuracy) = 0.8886606760804451 (variation between 0.8500 - 0.9500)
True Positive Rate = 0.8921
False Positive Rate = 0.1013
Error Rate = 0.1113
Precision = 0.9695
Recall = 0.8921

88.86% of the predictions made by the model were accurate (success rate), which means 11.13% of the predictions were incorrect (error rate). 89.21% were true positives (true positive rate, recall). 10.13% were false positives (false positive rate). 96.95% were not false positives (precision).

## 4.2 KNN

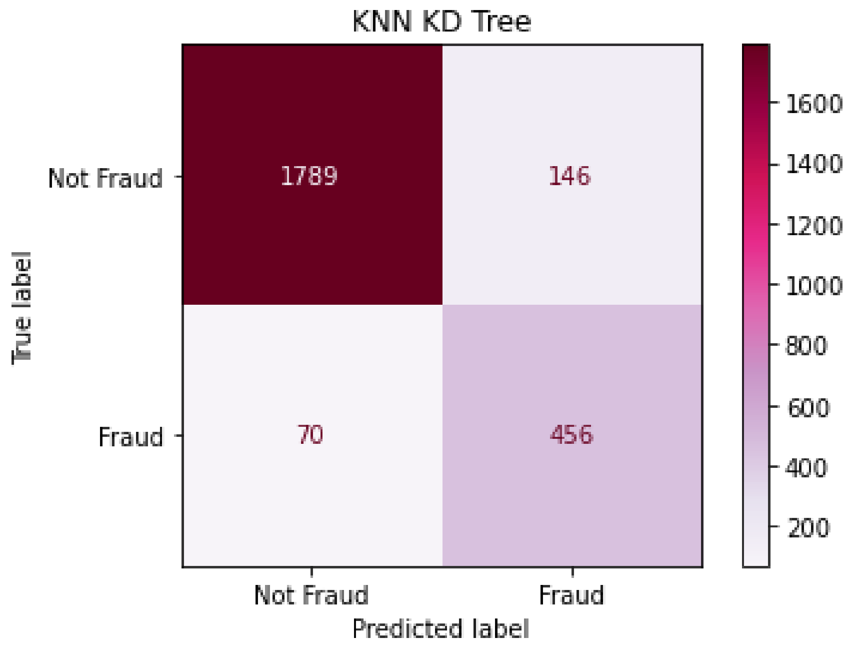
### 4.2.1 KNN Brute



Success Rate (Accuracy) = 0.9122308004876066 (variation between 0.9000 - 0.9200)
True Positive Rate = 0.9245
False Positive Rate = 0.1330
Error Rate = 0.0877
Precision = 0.9623
Recall = 0.9245

91.22% of the predictions made by the model were accurate (success rate), which means 8.77% of the predictions were incorrect (error rate). 92.45% were true positives (true positive rate, recall). 13.31% were false positives (false positive rate). 96.23% were not false positives (precision).

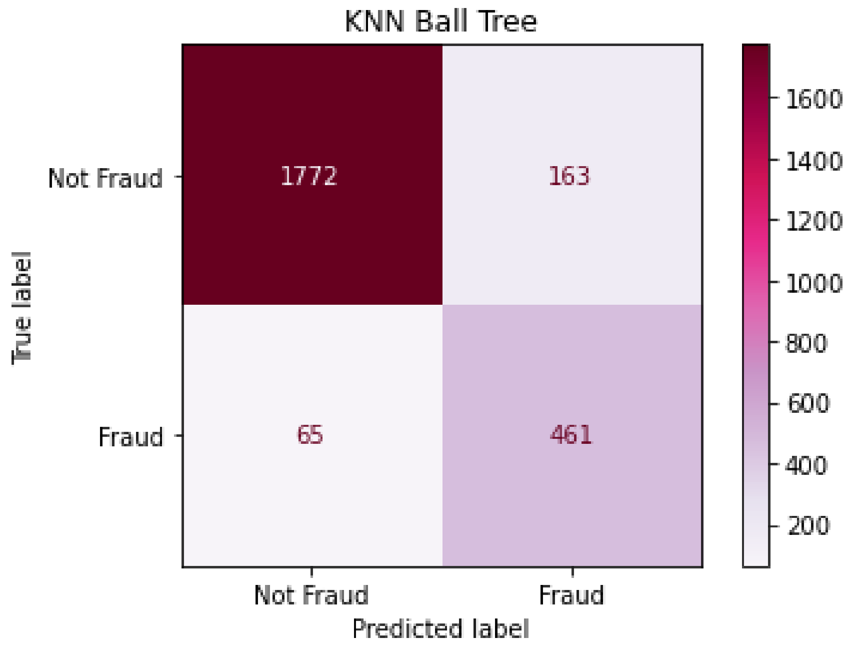
#### 4.2.2 KNN KD Tree



Success Rate (Accuracy) = 0.9122308004876066 (variation between 0.9000 - 0.9200)
True Positive Rate = 0.9245
False Positive Rate = 0.1330
Error Rate = 0.0877
Precision = 0.9623
Recall = 0.9245

91.22% of the predictions made by the model were accurate (success rate), which means 8.77% of the predictions were incorrect (error rate). 92.45% were true positives (true positive rate, recall). 13.30% were false positives (false positive rate). 96.23% were not false positives (precision).

### 4.2.3 KNN Ball Tree



Success Rate (Accuracy) = 0.9073547338480292 (variation between 0.9000 - 0.9200)
True Positive Rate = 0.9157
False Positive Rate = 0.1235
Error Rate = 0.0926
Precision = 0.9646
Recall = 0.9157

90.73% of the predictions made by the model were accurate (success rate), which means 9.26% of the predictions were incorrect (error rate). 91.57% were true positives (true positive rate, recall). 12.35% were false positives (false positive rate). 96.46% were not false positives (precision).

## 5. Conclusion

Two different models were used to classify fraudulent Ethereum transactions: logistic regression and KNN. The dataset had an issue of being imbalanced, which limited the amount of fraudulent transaction information that the model could use to make predictions. However, the data was oversampled using Synthetic Minority Oversampling Technique (SMOTE) to help proportion the amount of 0's (non-fraud) and 1's (fraud) in the testing and training datasets. Overall, the KNN models performed better than the logistic regression model since they consistently yielded higher accuracies (LR = 0.8886, KNN_Brute = 0.9122, KNN_KD_Tree = 0.9122, KNN_Ball_Tree = 0.9073). In addition to accurately classifying an Ethereum transaction as fraudulent or not, the current work can be further extended to possibly detecting the locations of where fraudulent transactions are made. Also, the models used 21 features to make predictions so another possibility for future work would be to calculate each feature's weight in terms of relevance/importance when detecting fraud.

## References

[1] Vagif Aliyev. *Ethereum Fraud Detection Dataset.* Jan. 2021. URL: `https://www.kaggle.com/vagifa/ethereum-frauddetection-dataset`.

[2] Jason Brownlee. *SMOTE for Imbalanced Classification with Python.* Jan. 2021. URL: `https://machinelearningmastery.com/smote-oversampling-for-imbalanced-classification/`.

[3] Ethereum.org. *What is Ethereum?* URL: `https://ethereum.org/en/what-is-ethereum/`.

[4] Guillaume Lemaître, Fernando Nogueira, and Christos K. Aridas. "Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning". In: *Journal of Machine Learning Research* 18.17 (2017), pp. 1–5. URL: `http://jmlr.org/papers/v18/16-365`.

[5] Sanjay M. *KNN using scikit-learn.* Nov. 2018. URL: `https://towardsdatascience.com/knn-using-scikit-learn-c6bed765be75`.

[6] Frederick Maier. *Logistic Regression.*

[7] Savan Patel. *Chapter 4: K Nearest Neighbors Classifier.* May 2017. URL: `https://medium.com/machine-learning-101/k-nearest-neighbors-classifier-1c1ff404d265`.

[8] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.