UNIVERSITÀ DI PISA

Department of Information Engineering – MSc AIDE

Data Mining and Machine Learning – University of Pisa

Fraud Detection

Aida Himmiche
Michael Asante

Supervised by:
Prof. Francesco Marcelloni
Prof. Pietro Ducange

ABSTRACT

The objective of this project is to build an application which operates in the domain of banking, in order to detect fraudulent transactions from a list of credit card transactions of a customer using a classification model trained and tested on a dataset of 250 000+ transaction records.

This project was developed in the context of the Data Mining and Machine Learning course in University of Pisa, and as such it deals with the learning material covered and uses the technologies introduced; Weka for example.

This application model was implemented with Python and an integrated front-end with Flask. An analysis of various models was conducted, then the best classification model was built, trained and tested on the dataset for the best accuracy achievable with a Cross-Validation methodology.

Keywords*: Fraud Detection, Banking, Machine Learning, Cross-validation, Classification model, Weka, Python, Flask.*

TABLE OF CONTENTS

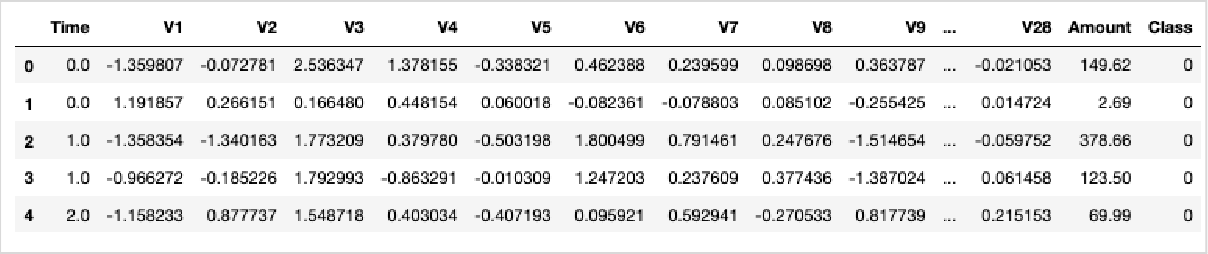
KDD Process

The Dataset

The dataset used consists of 284 807 transactions from a single customer credit card. It contains 31 columns; time, transaction amount, and transaction class, in addition to 28 more features that were PCA transformed for reasons of privacy and confidentiality, making all the columns numerical values.

Description

- Time (in s): relative to the duration between the current selected transaction and the first one recorded on that credit card.
- Amount (in €): The amount spent in each transaction.
- Class (binary): 0 for normal, 1 for fraudulent.
- V1 - V28: Features transformed with the Principal Component Analysis method.

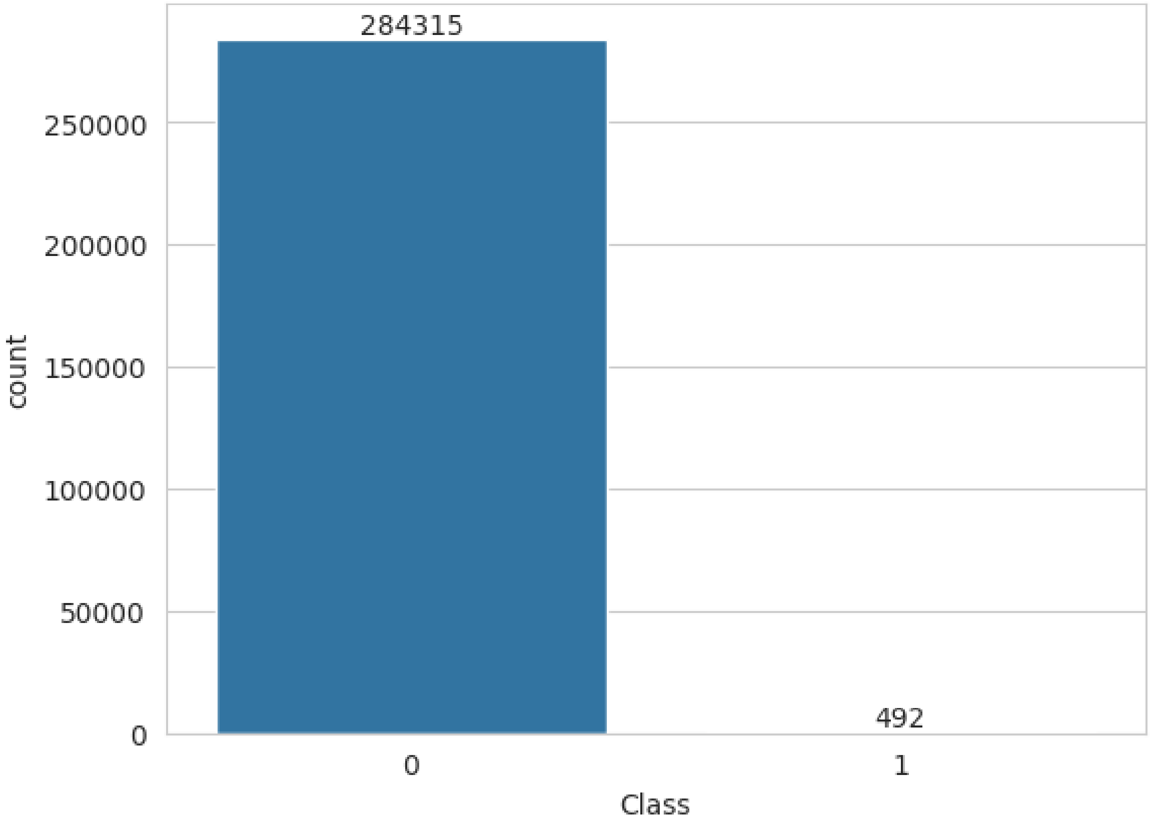


	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V28	Amount	Class
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.021053	149.62	0
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	0.014724	2.69	0
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	-0.059752	378.66	0
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	0.061458	123.50	0
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	0.215153	69.99	0

Class Distribution

The dataset is highly imbalanced, counting 284 315 instances of normal transactions and only 492 instances of fraudulent ones as shown in the following bar plot.

The bar for the positive class is so short in comparison that it can barely be seen.

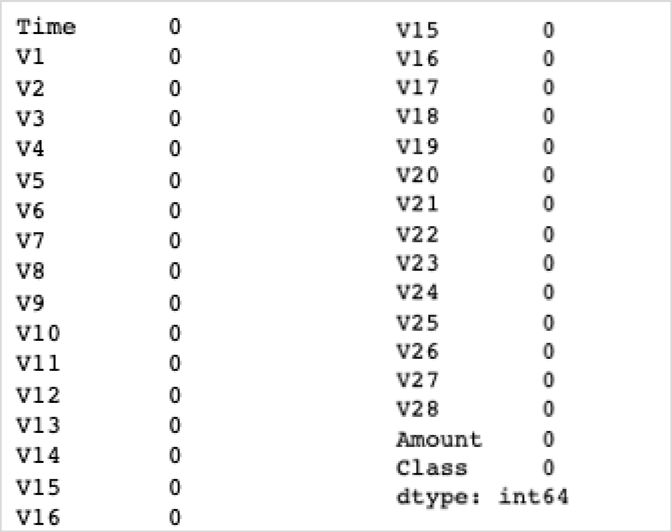


Preprocessing

To maximize the pattern recognition of the model for this dataset, a few preprocessing steps were carried out. We first treated the dataset on Weka, and since we are using Python for the implementation, those steps were replicated using Pandas's data structures and other python modules.

Missing values

Using the *isnull()* count method in Pandas, we see that this dataset did not contain any null or missing values so there was no need to perform any alterations either by replacement or elimination.



```
Time      0                V15       0
V1        0                V16       0
V2        0                V17       0
V3        0                V18       0
V4        0                V19       0
V5        0                V20       0
V6        0                V21       0
V7        0                V22       0
V8        0                V23       0
V9        0                V24       0
V10       0                V25       0
V11       0                V26       0
V12       0                V27       0
V13       0                V28       0
V14       0                Amount    0
V15       0                Class     0
V16       0                dtype: int64
```

Duplicate Removal

After running a duplicate search on the dataset, there were 1081 identical records. After dropping those duplicated instances the new class distribution is as follows. Therefore we notice a different of {0: -1062, 1: -19}

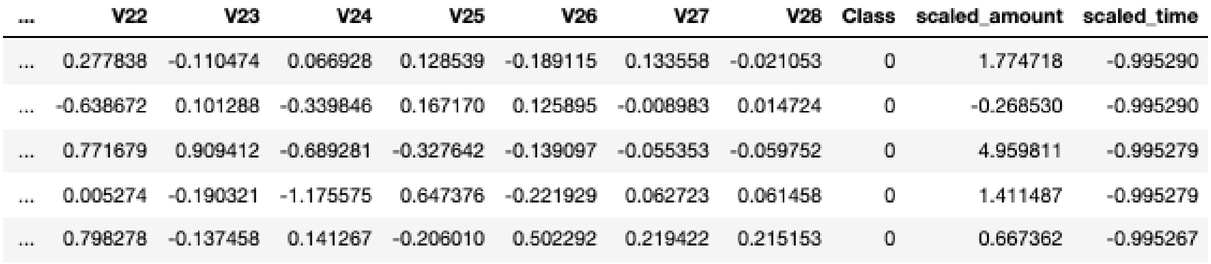
Class	Count
0	283 253
1	473

Scaling and Normalization

Seeing that the values of the attributes Time and Amount did not fall in the same range as the PCA transformed features V1 through V28, we decided to normalize them and transform them into values in the scale -1 to 1.

To achieve that we used the RobustScaler() object from the *“sklearn.preprocessing”* package by calling the method transform() and providing said scale.

The Time and Amount columns are displayed as “scaled_time” and “scaled_amount” in the following figure.



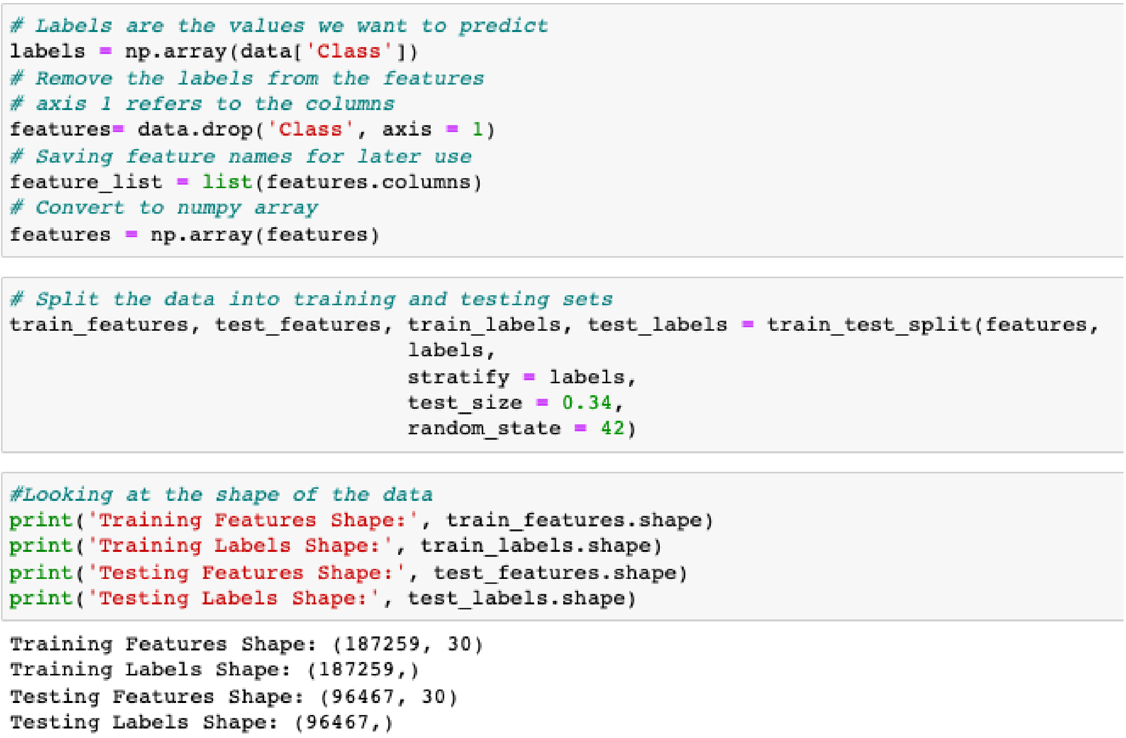
...	V22	V23	V24	V25	V26	V27	V28	Class	scaled_amount	scaled_time
...	0.277838	-0.110474	0.066928	0.128539	-0.189115	0.133558	-0.021053	0	1.774718	-0.995290
...	-0.638672	0.101288	-0.339846	0.167170	0.125895	-0.008983	0.014724	0	-0.268530	-0.995290
...	0.771679	0.909412	-0.689281	-0.327642	-0.139097	-0.055353	-0.059752	0	4.959811	-0.995279
...	0.005274	-0.190321	-1.175575	0.647376	-0.221929	0.062723	0.061458	0	1.411487	-0.995279
...	0.798278	-0.137458	0.141267	-0.206010	0.502292	0.219422	0.215153	0	0.667362	-0.995267

Data Mining

This section is dedicated to the work done on predicting the **Class** of the transactions in the dataset. For this step of the KDD process we performed an evaluation of different models using Weka, then mainly used Python's Scikit-learn library for the implementation of the chosen model.

Dataset Splitting

Before beginning to use the dataset for the classification task at hand, it was first split into Training and Testing sets, using the percentages 64% and 34% respectively, the split was stratified. The testing set was put aside and the training set was used for the Cross-Validation analysis of the different algorithms in Weka.



```
# Labels are the values we want to predict
labels = np.array(data['Class'])
# Remove the labels from the features
# axis 1 refers to the columns
features= data.drop('Class', axis = 1)
# Saving feature names for later use
feature_list = list(features.columns)
# Convert to numpy array
features = np.array(features)
```

```
# Split the data into training and testing sets
train_features, test_features, train_labels, test_labels = train_test_split(features,
                                                            labels,
                                                            stratify = labels,
                                                            test_size = 0.34,
                                                            random_state = 42)
```

```
#Looking at the shape of the data
print('Training Features Shape:', train_features.shape)
print('Training Labels Shape:', train_labels.shape)
print('Testing Features Shape:', test_features.shape)
print('Testing Labels Shape:', test_labels.shape)
```

```
Training Features Shape: (187259, 30)
Training Labels Shape: (187259,)
Testing Features Shape: (96467, 30)
Testing Labels Shape: (96467,)
```

Cross-Validation

This step was carried out using the tool Weka, which provides many preprocessing filters and classifiers ready to use.

Since our dataset is highly imbalanced, it was likely to only learn the majority class only. Therefore, we used a combination of algorithms to rebalance it as described in the following section.

Dataset Rebalancing

This step was achieved using two filters from Weka's instance-based supervised algorithms for oversampling and undersampling.

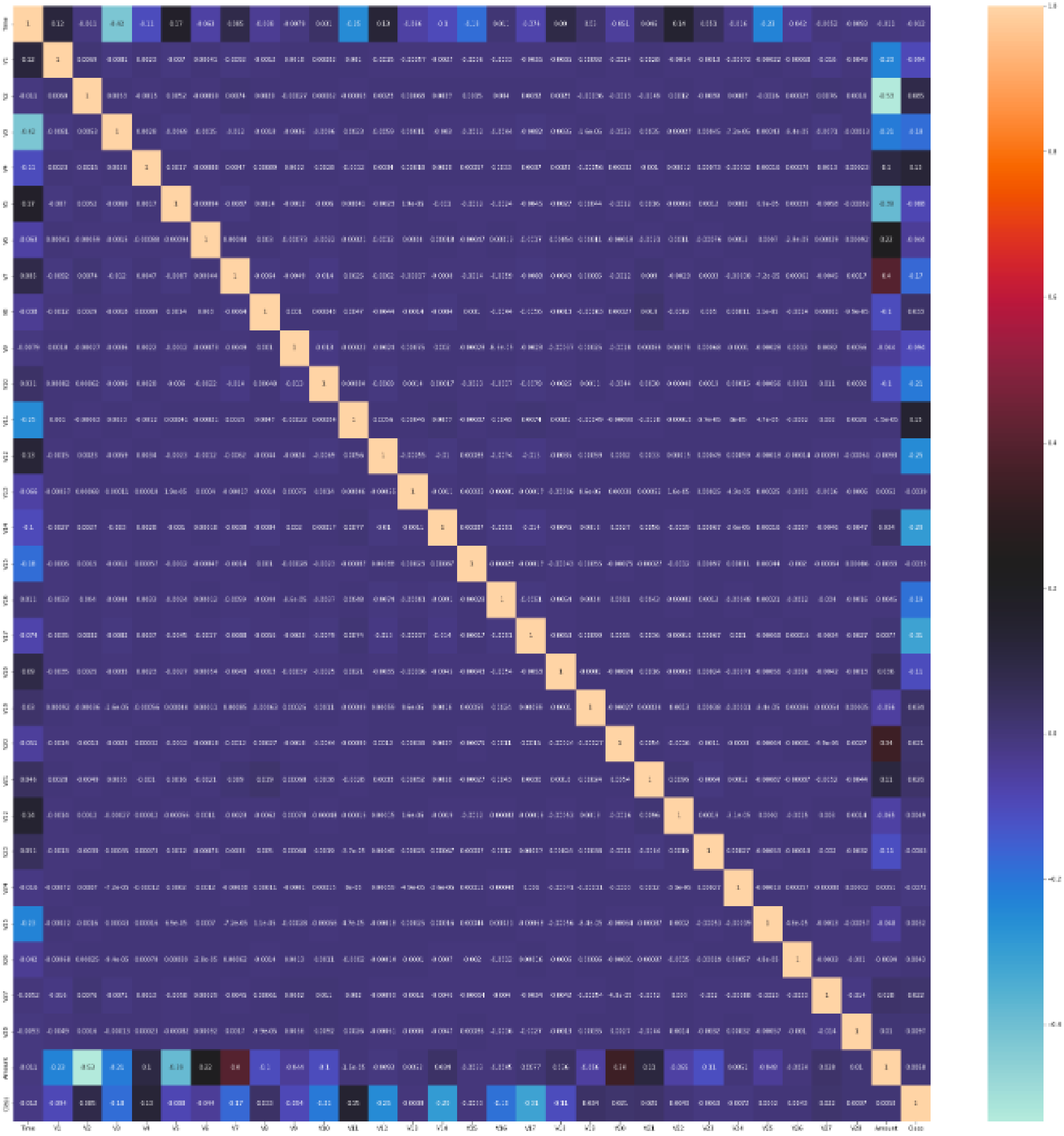
The supervised algorithm “Resample” was used to reduce the majority class, however it also increases the minority class by duplicating its samples. Therefore, after resampling, we use the unsupervised “Remove Duplicates” filter and take the minority class back to its original size.

Then, we follow with “SMOTE” (Synthetic Minority Oversampling Technique), a more useful way of oversampling which actually introduces new synthetic data to the dataset and can serve the model to make more accurate patterns.

The algorithm chooses a random example from the minority class, then the k nearest neighbors of that example, a synthetic example is then created by choosing a point between the random neighbor and the example.

Feature Selection

One of the fastest ways to identify features that will be relevant in predicting the class is to find the attributes that are most correlated with the class column, either positively or negatively. To do so, we can visualize those correlation factors into a color coded matrix as follows.



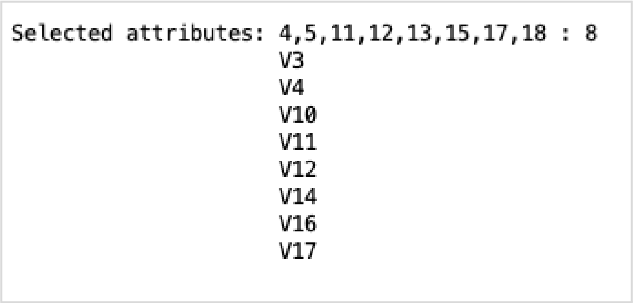
The class attribute is the bottom row and right-most column. We can see how 15 features have the most noticeable correlations with it.

However there are other methods to select relevant attributes, a few of which were tried against the dataset to observe the different results and identify the least recurring or least "important" features. This was first done using Weka.

The first combination was CfsSubsetEval with a Best First search method. This algorithm finds the features that have the highest correlation with the class but not much correlation with other features. The selected features will therefore promise a higher individual predictability with regards to the class because of a clear pattern, unconfused with other values in the dataset.

The best first search moves forward or backward, or even from any point to search bidirectionally, expanding the most promising nodes that fulfill a specific rule, in this case the high correlation with the class.

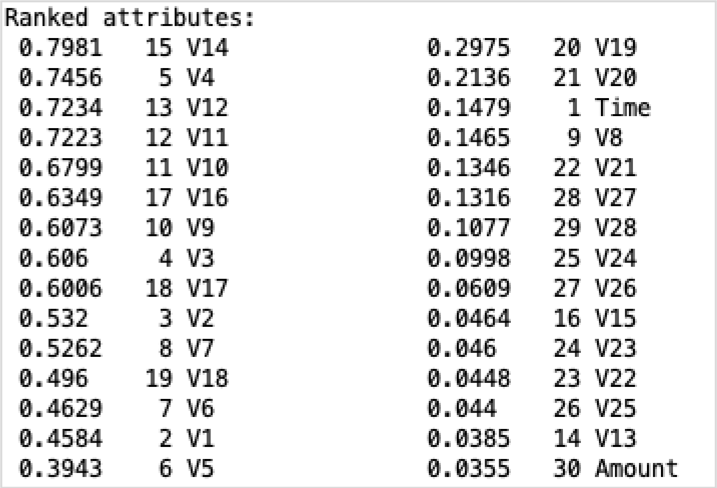
The results for this combination were as follows.



```
Selected attributes: 4,5,11,12,13,15,17,18 : 8
                     V3
                     V4
                     V10
                     V11
                     V12
                     V14
                     V16
                     V17
```

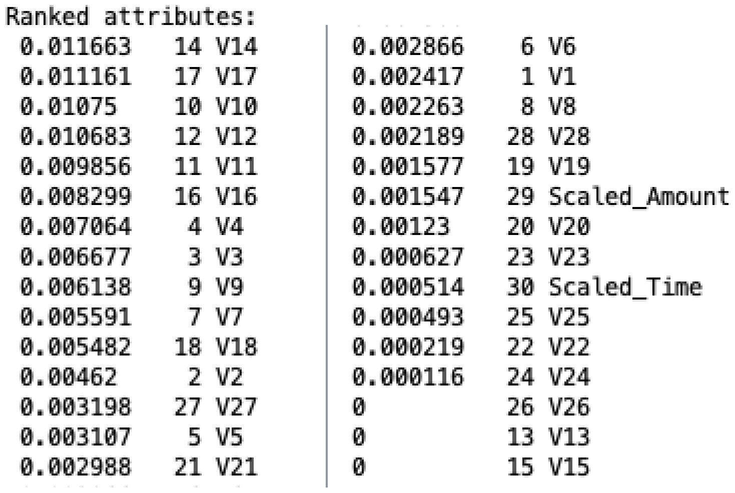
The second combination tried was a CorrelationAttributeEval with a Ranker search method. This one calculates the Pearson correlation (worth) of features with the class and ranks them from most important to least.

The outcome was somewhat similar to the first in the way that it placed the same selected attributes in the top 8 except V2, instead replacing it with V9 and calculating the Pearson correlation to be higher.



```
Ranked attributes:
 0.7981   15 V14             0.2975   20 V19
 0.7456    5 V4              0.2136   21 V20
 0.7234   13 V12             0.1479    1 Time
 0.7223   12 V11             0.1465    9 V8
 0.6799   11 V10             0.1346   22 V21
 0.6349   17 V16             0.1316   28 V27
 0.6073   10 V9              0.1077   29 V28
 0.606     4 V3              0.0998   25 V24
 0.6006   18 V17             0.0609   27 V26
 0.532     3 V2              0.0464   16 V15
 0.5262    8 V7              0.046    24 V23
 0.496    19 V18             0.0448   23 V22
 0.4629    7 V6              0.044    26 V25
 0.4584    2 V1              0.0385   14 V13
 0.3943    6 V5              0.0355   30 Amount
```

The third was infoGainAttributeEval which evaluates the worth of an attribute by measuring the information gain with respect to the class. This was used along with Ranker search method and gave the following results:

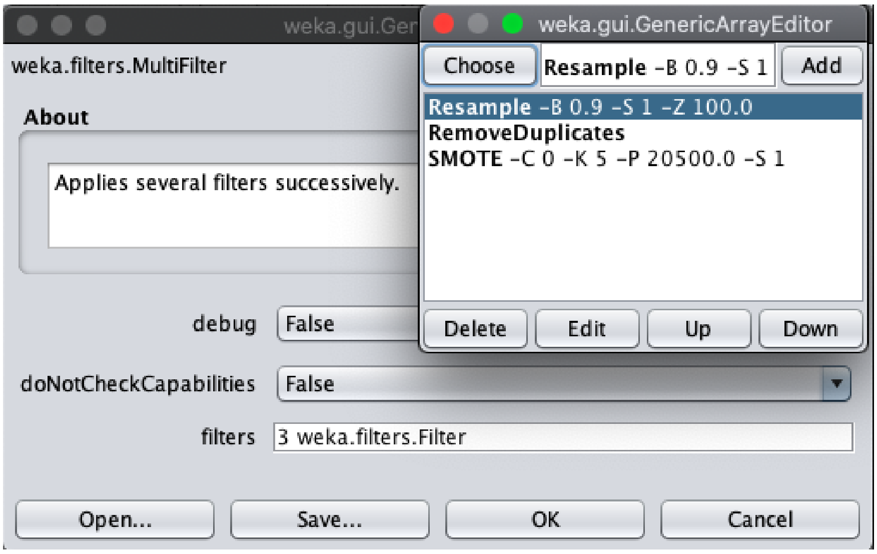


```
Ranked attributes:
 0.011663    14 V14       0.002866     6 V6
 0.011161    17 V17       0.002417     1 V1
 0.01075     10 V10       0.002263     8 V8
 0.010683    12 V12       0.002189    28 V28
 0.009856    11 V11       0.001577    19 V19
 0.008299    16 V16       0.001547    29 Scaled_Amount
 0.007064     4 V4        0.00123     20 V20
 0.006677     3 V3        0.000627    23 V23
 0.006138     9 V9        0.000514    30 Scaled_Time
 0.005591     7 V7        0.000493    25 V25
 0.005482    18 V18       0.000219    22 V22
 0.00462      2 V2        0.000116    24 V24
 0.003198    27 V27       0           26 V26
 0.003107     5 V5        0           13 V13
 0.002988    21 V21       0           15 V15
```

WEKA Nested Filters

These two steps were applied during the Cross Validation step and not before, by using the “MultiFilter” to stack them in order, inside the “FilteredClassifier” classifier where we could choose the classification algorithm. For the attribute selection, we used the “AttributeSelectedClassfier” for in the classifier options.

The Rebalancing is done at each of the cross-validation iterations, only on the partition considered for training while the validation fold is kept as is. Here is the stack of filters applied to achieve the rebalancing of majority/minority classes.



weka.filters.MultiFilter

About

Applies several filters successively.

debug False

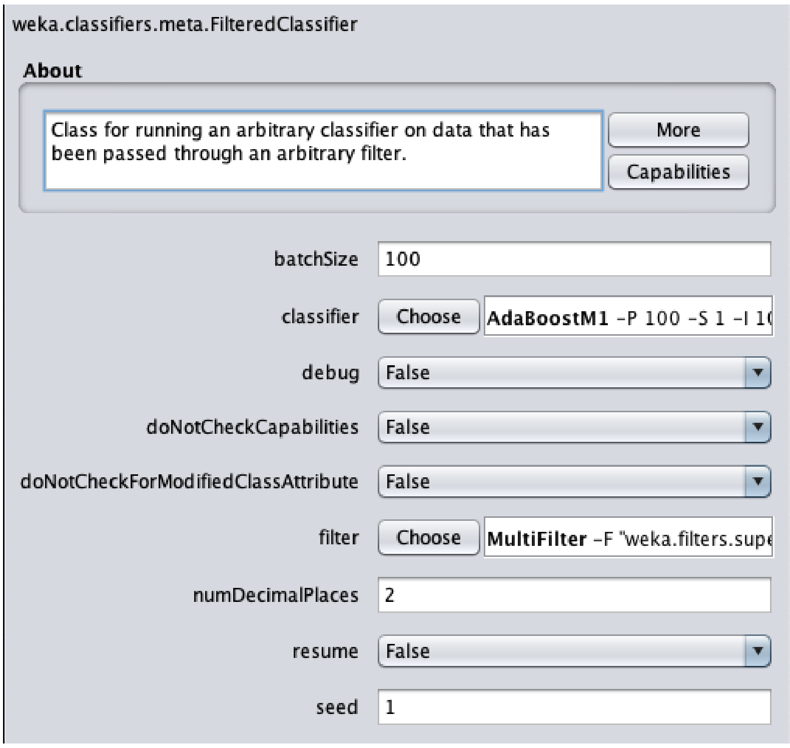
doNotCheckCapabilities False

filters 3 weka.filters.Filter

Resample -B 0.9 -S 1

```
Resample -B 0.9 -S 1 -Z 100.0
RemoveDuplicates
SMOTE -C 0 -K 5 -P 20500.0 -S 1
```

Normal Classifier:



weka.classifiers.meta.FilteredClassifier

About

Class for running an arbitrary classifier on data that has been passed through an arbitrary filter.

batchSize 100

classifier AdaBoostM1 -P 100 -S 1 -I 10

debug False

doNotCheckCapabilities False

doNotCheckForModifiedClassAttribute False

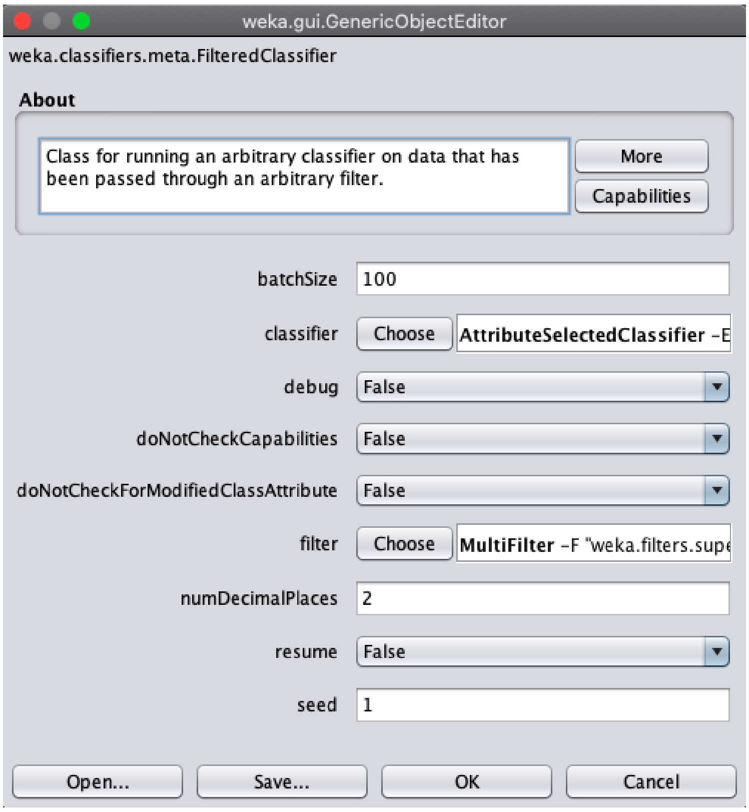
filter MultiFilter -F "weka.filters.supe

numDecimalPlaces 2

resume False

seed 1

Classifier with Attribute Selection:



weka.classifiers.meta.FilteredClassifier

About

Class for running an arbitrary classifier on data that has been passed through an arbitrary filter.

batchSize 100

classifier Choose AttributeSelectedClassifier -E

debug False

doNotCheckCapabilities False

doNotCheckForModifiedClassAttribute False

filter Choose MultiFilter -F "weka.filters.supe

numDecimalPlaces 2

resume False

seed 1

10-fold CV Method

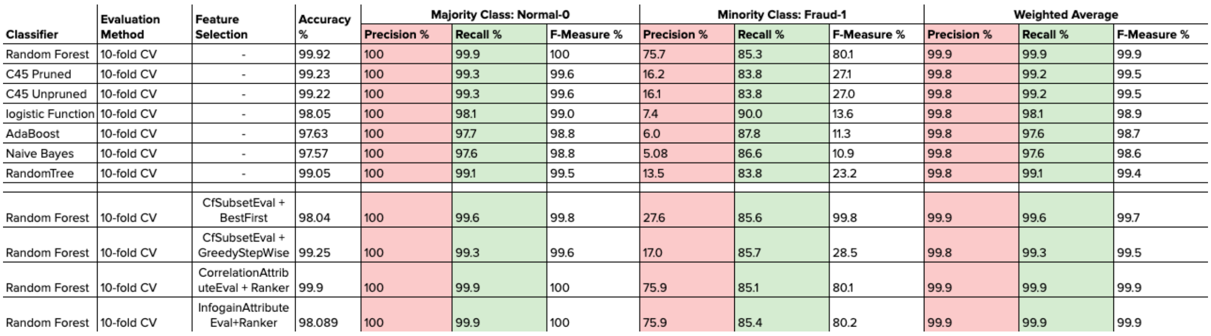
To make sure the model doesn’t overfit our training data and rather learn the essential patterns from it, it is necessary to use a cross-validation method. This will allow us to know if the model will perform well on unseen data or not by keeping a portion(s) of it out for the testing phase.

In the case of this project, the cross-validation method used is the **k-fold** method, which defines that the dataset is split into training and validation sets at each iteration, the validation set changing k times along the data set.

We chose the 10-fold cross validation setting, which means we feed our model 9/10 of the dataset during the training phase, then validate its learning on the 1/10 remaining samples (the process is repeated 10 times).

Classification

Different models were tried against our preprocessed dataset (all of which were kept at default parameter values), to find the one that classifies the transactions most accurately. The results are summarized in the following table:



Classifier	Evaluation Method	Feature Selection	Accuracy %	Majority Class: Normal-0			Minority Class: Fraud-1			Weighted Average		
				Precision %	Recall %	F-Measure %	Precision %	Recall %	F-Measure %	Precision %	Recall %	F-Measure %
Random Forest	10-fold CV	-	99.92	100	99.9	100	75.7	85.3	80.1	99.9	99.9	99.9
C45 Pruned	10-fold CV	-	99.23	100	99.3	99.6	16.2	83.8	27.1	99.8	99.2	99.5
C45 Unpruned	10-fold CV	-	99.22	100	99.3	99.6	16.1	83.8	27.0	99.8	99.2	99.5
logistic Function	10-fold CV	-	98.05	100	98.1	99.0	7.4	90.0	13.6	99.8	98.1	98.9
AdaBoost	10-fold CV	-	97.63	100	97.7	98.8	6.0	87.8	11.3	99.8	97.6	98.7
Naive Bayes	10-fold CV	-	97.57	100	97.6	98.8	5.08	86.6	10.9	99.8	97.6	98.6
RandomTree	10-fold CV	-	99.05	100	99.1	99.5	13.5	83.8	23.2	99.8	99.1	99.4
Random Forest	10-fold CV	CfSubsetEval + BestFirst	98.04	100	99.6	99.8	27.6	85.6	99.8	99.9	99.6	99.7
Random Forest	10-fold CV	CfSubsetEval + GreedyStepWise	99.25	100	99.3	99.6	17.0	85.7	28.5	99.8	99.3	99.5
Random Forest	10-fold CV	CorrelationAttributeEval + Ranker	99.9	100	99.9	100	75.9	85.1	80.1	99.9	99.9	99.9
Random Forest	10-fold CV	InfogainAttributeEval+Ranker	98.089	100	99.9	100	75.9	85.4	80.2	99.9	99.9	99.9

The measure considered for evaluation are :

- **Accuracy:** The percentage of correctly classified instances compared to the total number of instances considered.
- **Precision**: How many instances classified in a class actually belong to the class.
- **Recall:** How many instances were correctly classified compared to the total instances of that class.
- **F-Measure:** Calculated from the precision and recall of the "test", the harmonic mean of the two. The highest possible value of an F-score is 1.0, indicating perfect precision and recall.

The following section is an analysis of the results in a more detailed description:

Random Forest

This first model is a supervised classification model which builds decision trees by fitting them on samples of provided data, creating different training subsets, and uses averaging to improve the predictive accuracy and control over-fitting.

This model was the best performing in terms of Precision and Recall for the minority class with 75.7% and 85.3% respectively.

We later try Random Forest with two combinations of attribute selection methods to see if we can improve the performance by having the limited "important" features considered only; the model demonstrates almost the same as the first results (InfoGainAttributeEval), but not better.

C4.5

We tried the C4.5 (labeled J48 in Weka), and used it in two modes:

- **Pruned tree:** The default setting for this algorithm, performed at second best with a Precision of 16.2% and Recall of 83.8%, which means it is misclassifying the Normal class.

- **Unpruned tree:** Gave Precision of 16.2% and Recall of 83.8%, not far from the pruned version and suffering from the same issue.

Random Tree

Our next option was another decision tree algorithm. This classifier creates a tree that considers K randomly chosen attributes at each node, and performs no pruning.

This model achieved Precision of 13.5% and Recall of 83.8%, as much as the “Fraud” class detection is performing well, samples from the “Normal ”class are being mistaken as fraudulent.

Logistic Regression

This classification technique uses a logistic function to model the dependent variable. The dependent variable is dichotomous (binary) in nature, i.e. there could only be two possible classes.

Always with an arguably good performance, this model shows a Precision of 7.4% and Recall of 90%, which means it is the best at detecting the actual samples from “Fraud” class so far, but it’s misclassifying many “Normal” samples.

AdaBoost

This classifier fits a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted, such that subsequent classifiers focus more on difficult cases.

The model should theoretically be performing better than others because it involves a correction mechanism and should predict classes better after learning from its errors, but the issue here becomes over-fitting, which is likely the reason why it only achieved 6% in Precision and 87% in Recall.

Naive Bayes

Naive Bayes is a probabilistic classifier based on the Bayes algorithm where we can find out whether A happened based on B's occurrence.

This algorithm makes the assumption that features are independent, uncorrelated, which is most likely why its performance was a lowest 5.08% in Precision and 86.6% in Recall; we can observe from the correlation matrix that there is a level of correlation between certain features, and that may have misled the classifier to an extent.

Model Selection

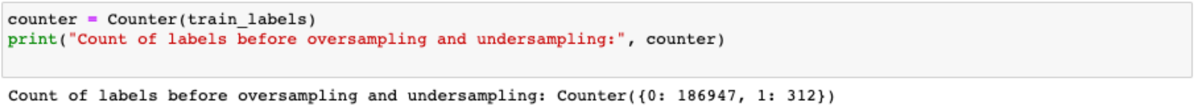
After comparing all these results and evaluation metrics we can see that all the model have around similar values of Recall ranging from 80-90%, meaning they are detecting the actual "Fraud" samples rather well, but most have very low Precision since they misclassify them as normal transactions as fraudulent as well.

Therefore, looking at the PRF measures, we can clearly see that **Random Forest** is the best classifier for this task and dataset.

Implementation

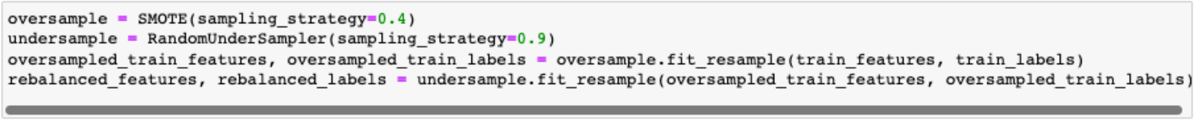
Training Set Rebalancing

As mentioned, the same steps done on Weka during cross validation were replicated in Python on the training set.

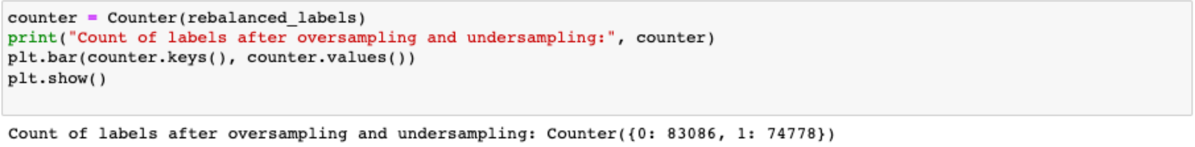


```
counter = Counter(train_labels)
print("Count of labels before oversampling and undersampling:", counter)
```

```
Count of labels before oversampling and undersampling: Counter({0: 186947, 1: 312})
```

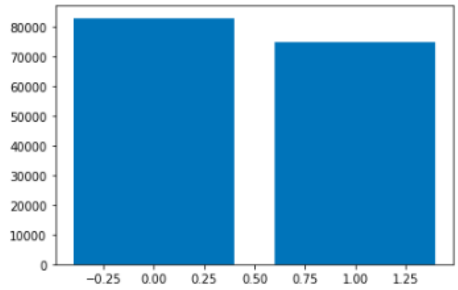


```
oversample = SMOTE(sampling_strategy=0.4)
undersample = RandomUnderSampler(sampling_strategy=0.9)
oversampled_train_features, oversampled_train_labels = oversample.fit_resample(train_features, train_labels)
rebalanced_features, rebalanced_labels = undersample.fit_resample(oversampled_train_features, oversampled_train_labels)
```



```
counter = Counter(rebalanced_labels)
print("Count of labels after oversampling and undersampling:", counter)
plt.bar(counter.keys(), counter.values())
plt.show()
```

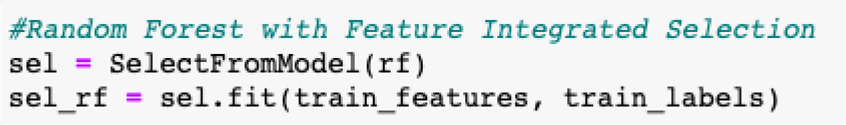
```
Count of labels after oversampling and undersampling: Counter({0: 83086, 1: 74778})
```



For dataset **rebalancing** In the python notebook this was done using an oversample/undersample combination or SMOTE and RandomUnderSampler, with sampling strategies of 0.4 and 0.9 respectively. Ending up with a class distribution of 83086 normal and 74778 fraud.

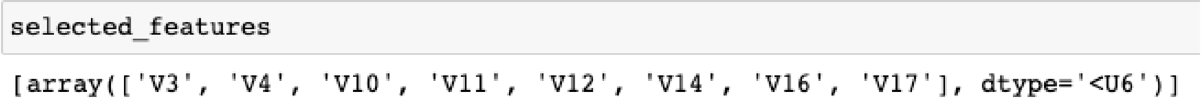
Feature Selection

Since as seen through Weka, the feature selection methods do not improve the performance of the model, if not make it worse, so we decided not to use any of them. However, there was another option using Python and an integrated feature selection with the RandomForest that we decided was worth a try. This is done through the "SelectFromModel" Random Forest model in "sklean.feature_selection"



```
#Random Forest with Feature Integrated Selection
sel = SelectFromModel(rf)
sel_rf = sel.fit(train_features, train_labels)
```

We create it and fit it to the training features and labels, then retrieve the selection attributes.



```
selected_features
```

```
[array(['V3', 'V4', 'V10', 'V11', 'V12', 'V14', 'V16', 'V17'], dtype='<U6')]
```

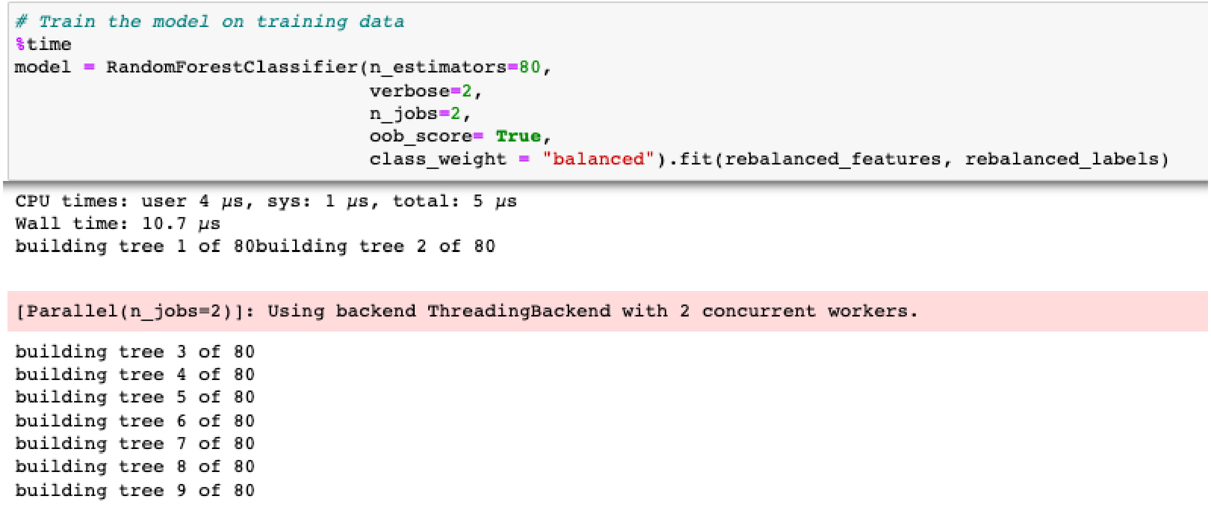
The result was the exact same list of attributes selected in the CfsSubsetEval method, so we did not go through with it.

Random Forest Evaluation

To build our chosen model, we used the “sklearn.ensemble” module as it provides a “RandomForestClassifier” implementation.

We start by creating our model with 80 decision trees (decided after different changes and trial and error), and a 2-job parallelization. We then call the *fit()* method and pass our training features and appropriate labels. The model takes 1.1 minutes to build and learn the 66% training data.

The class weight is set to “balanced” to give the majority/minority classes weight inversely proportional to their distribution, seeing the severe imbalancing of the dataset.



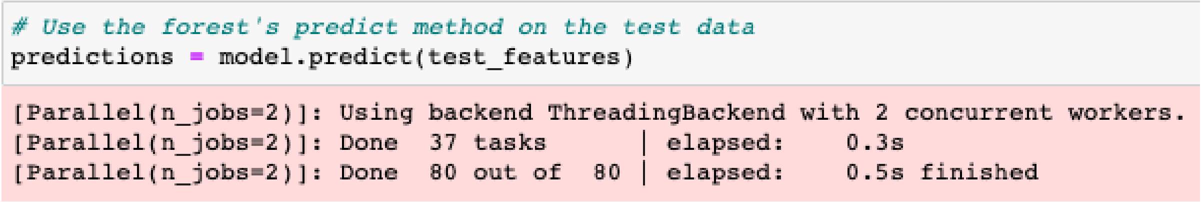
```
# Train the model on training data
%time
model = RandomForestClassifier(n_estimators=80,
                               verbose=2,
                               n_jobs=2,
                               oob_score= True,
                               class_weight = "balanced").fit(rebalanced_features, rebalanced_labels)
```

```
CPU times: user 4 µs, sys: 1 µs, total: 5 µs
Wall time: 10.7 µs
building tree 1 of 80building tree 2 of 80

[Parallel(n_jobs=2)]: Using backend ThreadingBackend with 2 concurrent workers.

building tree 3 of 80
building tree 4 of 80
building tree 5 of 80
building tree 6 of 80
building tree 7 of 80
building tree 8 of 80
building tree 9 of 80
```

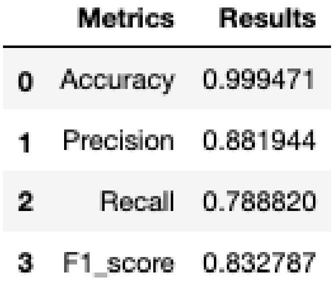
Now we test the model on our 34% testing data and labels using the *predict()* method, this only takes 0.8 seconds to finish.



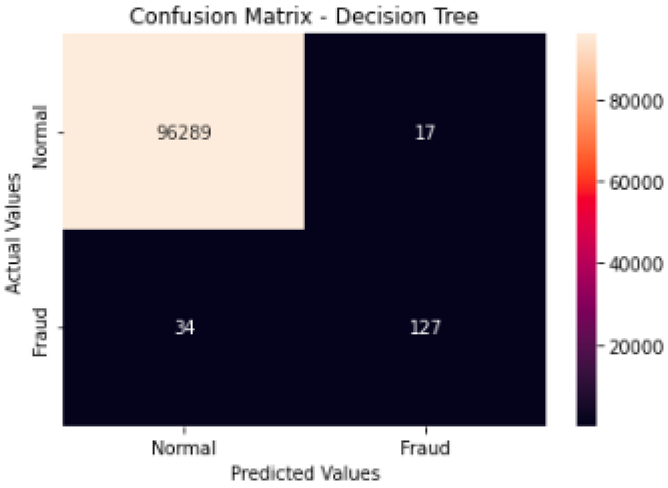
```
# Use the forest's predict method on the test data
predictions = model.predict(test_features)
```

```
[Parallel(n_jobs=2)]: Using backend ThreadingBackend with 2 concurrent workers.
[Parallel(n_jobs=2)]: Done  37 tasks      | elapsed:    0.3s
[Parallel(n_jobs=2)]: Done  80 out of  80 | elapsed:    0.5s finished
```

We have 0.017% misclassified normal transactions and 26.7% misclassified fraud transactions. Here are the results and confusion matrix we could achieve with this model.



	Metrics	Results
0	Accuracy	0.999471
1	Precision	0.881944
2	Recall	0.788820
3	F1_score	0.832787



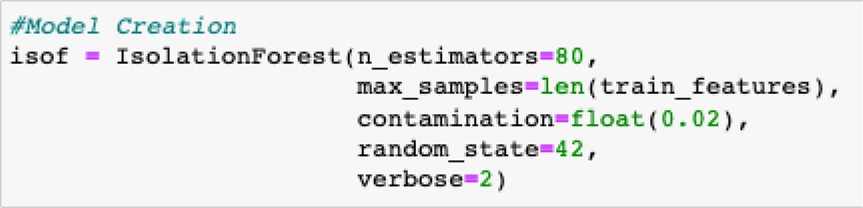
Anomaly Detection

Since the initial dataset was heavily imbalanced, fraudulent transactions could potentially be considered outliers or anomalies in comparison to the rest of the data, therefore, treating the problem as an anomaly detection problem should also yield some interesting results.

We tried a few outlier detection models on our dataset to compare their performance amongst themselves and especially with the classification that we chose to implement for the project, hoping to find out if it would have been more efficient to see this as an anomaly detection problem from the start.

Surely, in the preprocessing steps, only the duplicate elimination will be performed since the purpose is to leave the minority class instances scarce.

Isolation Forest

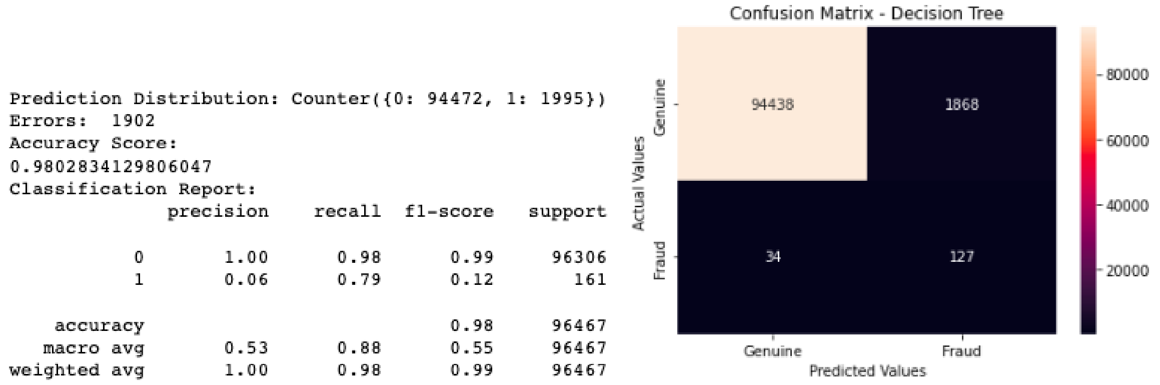


```
#Model Creation
isof = IsolationForest(n_estimators=80,
                       max_samples=len(train_features),
                       contamination=float(0.02),
                       random_state=42,
                       verbose=2)
```

First we built an Isolation Forest model which 'isolates' observations by splitting the data space using lines that are orthogonal to the origin, and assigning higher anomaly scores to data points that need few splits to be isolated. (How far a data point is to the rest of the data).

This implementation of this model is included in the "sklearn.ensemble" module under the name IsolatonForest.

For contamination 0.02, the normal class had 1868 misclassified "Normal" transactions and 34 misclassified "Fraud" transactions (the same number as RF). When increasing the contamination value, we see a better assimilation of the Fraud class, but an increase of False Positives from the Normal class by thousands.



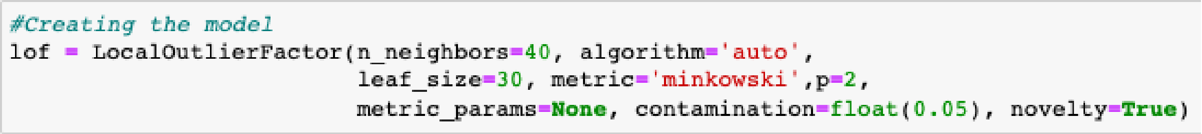
```
Prediction Distribution: Counter({0: 94472, 1: 1995})
Errors:  1902
Accuracy Score:
0.9802834129806047
Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.98      0.99     96306
           1       0.06      0.79      0.12       161

    accuracy                           0.98     96467
   macro avg       0.53      0.88      0.55     96467
weighted avg       1.00      0.98      0.99     96467
```

Local Outlier Factor

© DEPARTMENT OF INFORMATION ENGINEERING – UNIVERSITY OF PISA



```
#Creating the model
lof = LocalOutlierFactor(n_neighbors=40, algorithm='auto',
                         leaf_size=30, metric='minkowski',p=2,
                         metric_params=None, contamination=float(0.05), novelty=True)
```

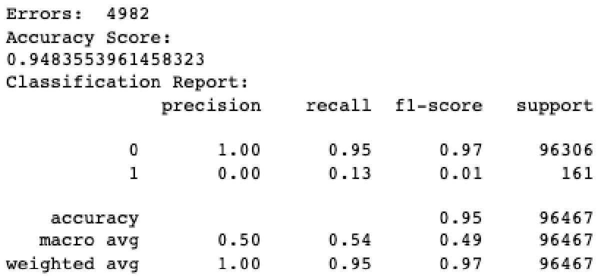
This density-based unsupervised anomaly detection method computes the local density deviation of a given data point with respect to its neighbors. It considers as outliers the samples that have a substantially lower density than their neighbors.

This implementation of this model is included in the “sklearn.neighbors” module under the name LocalOutlierFactor.

For contamination 0.05, the normal class had 4842 misclassified “Normal” transactions and 140 misclassified “Fraud” transactions (the same number as RF).

The smaller the contamination is the better the model learns the “Normal” class but the worse it learns the “Fraud” class and vice versa.

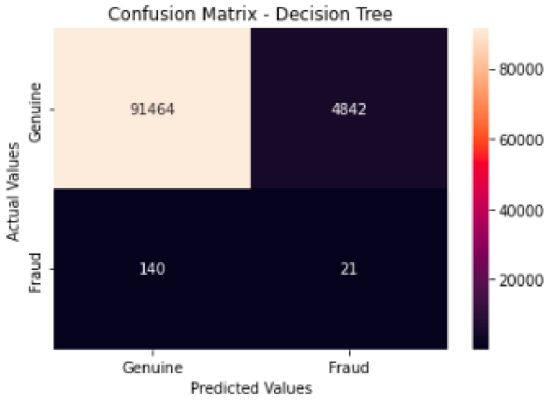
When contamination increases, we can lower the number of FP but the “Fraud” class is almost never identified.



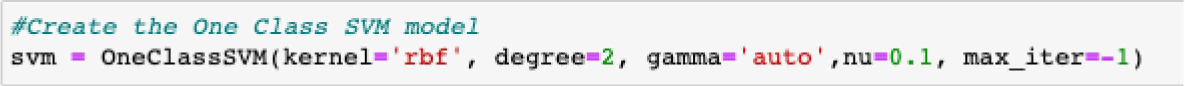
```
Errors:  4982
Accuracy Score:
0.9483553961458323
Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.95      0.97     96306
           1       0.00      0.13      0.01       161

    accuracy                           0.95     96467
   macro avg       0.50      0.54      0.49     96467
weighted avg       1.00      0.95      0.97     96467
```



One Class Support Vector Machine (SVM)



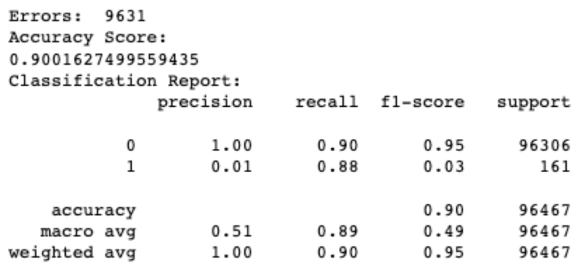
```
#Create the One Class SVM model
svm = OneClassSVM(kernel='rbf', degree=2, gamma='auto',nu=0.1, max_iter=-1)
```

The objective of the support vector machine algorithm is to find a hyperplane in an N-dimensional space (N — the number of features) that distinctly classifies the data points using the largest possible margin. Whichever side the point projects on will assign it into a class.

The one class SVM uses a (smallest possible) hypersphere instead of a hyperplane and assigns points to classes based on which are within the sphere and which are out.

This implementation of this model is included in the “sklearn.svm” module under the name OneClassSVM.

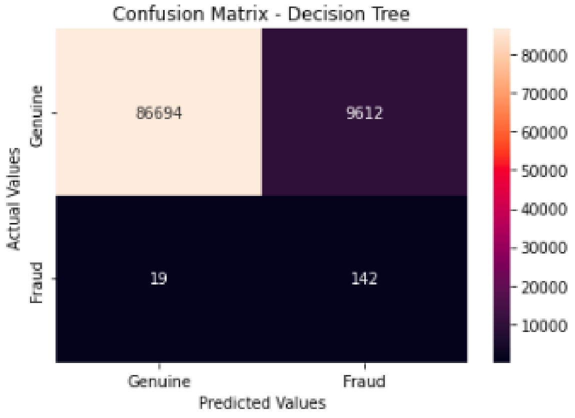
This model performed with 9631 errors. For nu(upper bound on fraction of training errors/ lower bound of the fraction of support vectors) is 0.5(default), the model classifies the Normal class perfectly, but never recognizes the Fraud class. When we decrease “nu” to 0.1 we get better results for the Fraud class and worse for the Normal class.



```
Errors:  9631
Accuracy Score:
0.9001627499559435
Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.90      0.95     96306
           1       0.01      0.88      0.03       161

    accuracy                           0.90     96467
   macro avg       0.51      0.89      0.49     96467
weighted avg       1.00      0.90      0.95     96467
```



Reflections

This goes to say that while conceptually anomaly detection does sound more appropriate for the nature of the data, the results weren’t very impressive and were far surpassed by classification.

Application

Analysis Phase

Description

The intent of this application is to be used as a middleware for financial institutions in detecting fraudulent and valid final transactions by their customers especially with online/ e-commerce purchases.

Main Actors

In the use-case of a real-time application for a financial company, the model will be deployed as part of an architecture which re-trains with Big Data configurations, Data Verification, Process management tools, Resource Management and others to recompute and flag transactions either as valid or fraud. For the purposes of our course and use-case, the input of the interface will be populated and validated against transactions flagged as Fraud or Normal.

For this, a set of transactions that adhere to the input format of the training data was generated. Within the flask app, a list of labels are displayed to match the input transactions based on the output predictions from the model.

Requirements

Functional Requirements

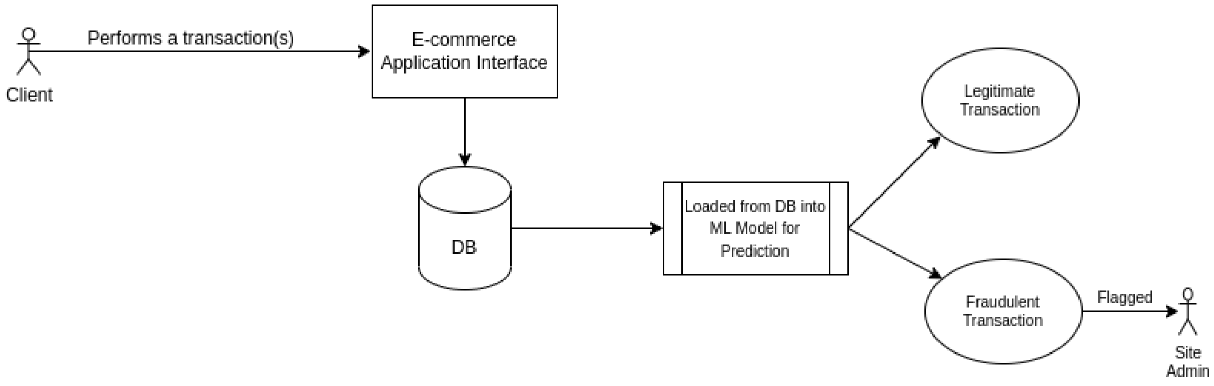
The application must relate to the information populated on the interface to classify a transaction as fraud or normal. The application consumes a Comma Separated Values (CSV) file chosen by the user, and converts it to a Python list sent to the model. After the test data is submitted, it is run through the model and the classification output is sent back to the user.

Non-Functional Requirements

The application is a simple but nicely designed interface with the Bootstrap library. The application has an embedded ML classification model to make predictions. The user should not wait too much time to get a prediction for the values entered. The model should predict results which limit the amount of False Positives that can be misleading, and also be able to adapt well to unseen data. The code is easily composed and can be integrated into a real-world infrastructure as a service.

Use Case Diagram

Below is a simple use-case diagram of the application functionalities:


Performs a transaction(s)
Client
E-commerce Application Interface
DB
Loaded from DB into ML Model for Prediction
Legitimate Transaction
Fraudulent Transaction
Flagged
Site Admin

Design Phase

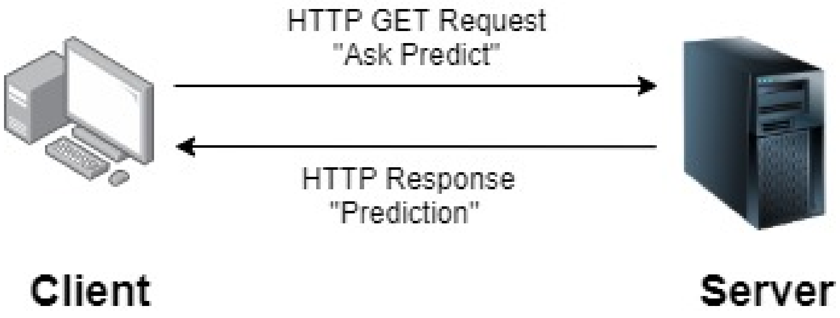
This phase highlights the application's main scenario. Pseudocode and mock-ups are used to explain the application flow and logic.

Credit Card Fraud Detection Process Flow

- Transactions are inputted through a .csv file and converted into a python list, then sent to the model by pressing the "Test" button.
- Model computes the values and predicts either a Fraud or Normal which within the Class, represents either a Fraud or Valid transaction respectively.

Application Architecture

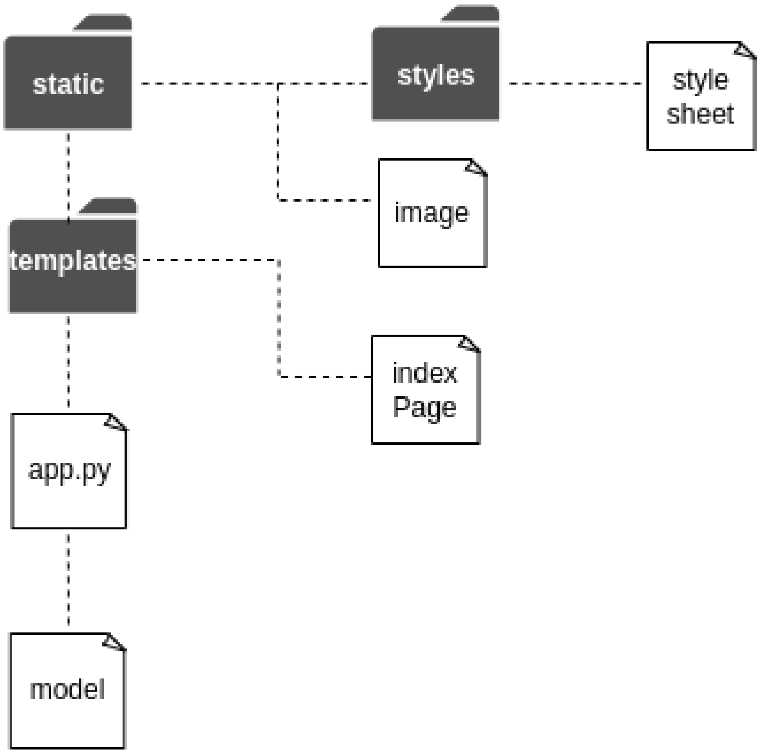
In order to satisfy the non-functional requirements for the application, specifically, the ease-of-use and portability of the application, a client-server architecture was implemented. The client hits a POST request by making a call to a predict function and the server responds with the prediction value after consuming the model saved in a pickle file.



Implementation Phase

Python was used as the Programming language for the model and the data preprocessing phase because of its simplicity and reduced implementation tasks with dynamic modules and libraries. With the Sklearn library several classification algorithms and implementation techniques with Cross Validation exist which can thus be used to obtain an optimized set of parameters for the project scope. It also allows the use of other libraries to visualize charts and plots to make inferences such as analyzing skewness, kurtosis, correlation, margin of error and others.

The front-end application is built with Bootstrap HTML5 and consumed by a Flask API, a micro-framework written in Python. Since Flask enables the re-usability of design scripts and images, the application structure is visualized below:



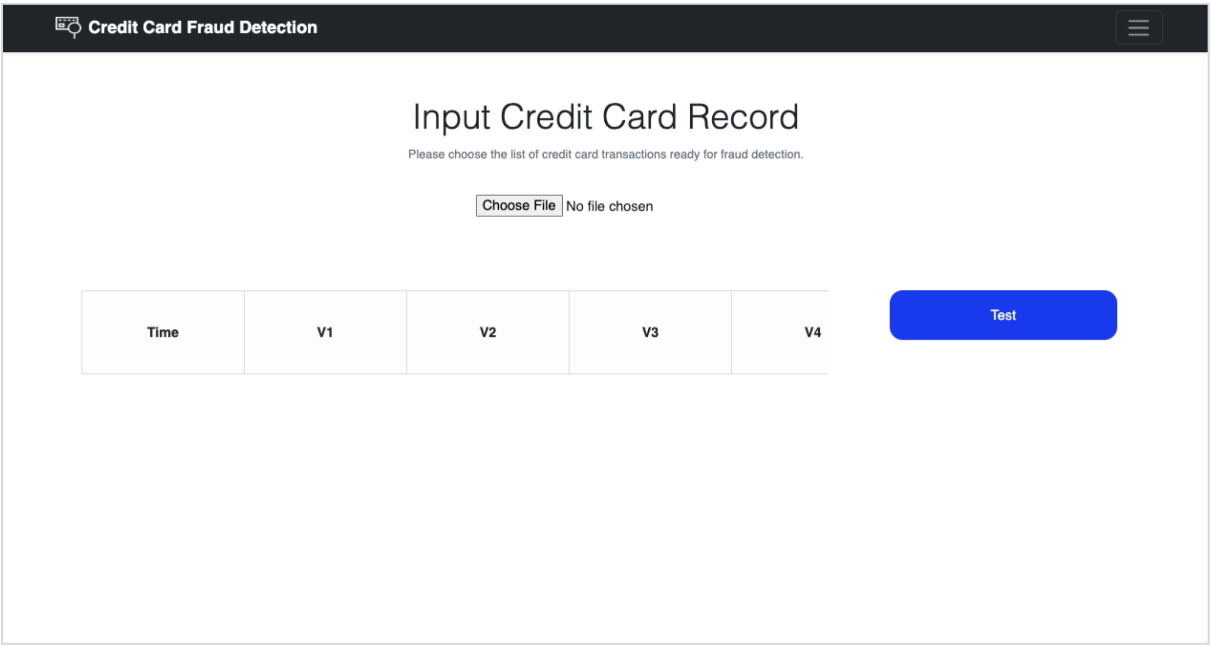
The application directory is organized into:

- **static:** The directory which contains the stylesheet for the interface and images used.
- **templates:** Contains the actual html markup language for the loaded page where the user interacts with the application.
- **app.py:** This file contains the API which takes user input and submits it to the model. An SMTP library was also included to trigger an auto-email on each prediction.
- **ModelTesting:** This directory contains the saved pickle image of the Random Forest model, along with the test transactions to be used on the web app.

Test Phase

A test phase of the application was done by choosing the “demo_transactions.csv” file in order to show the display of the application functionalities with regards to the classification.

The front page only shows a file chooser button, the features of the transactions to be inputted into the model, and a Test button which will trigger the model predictions.



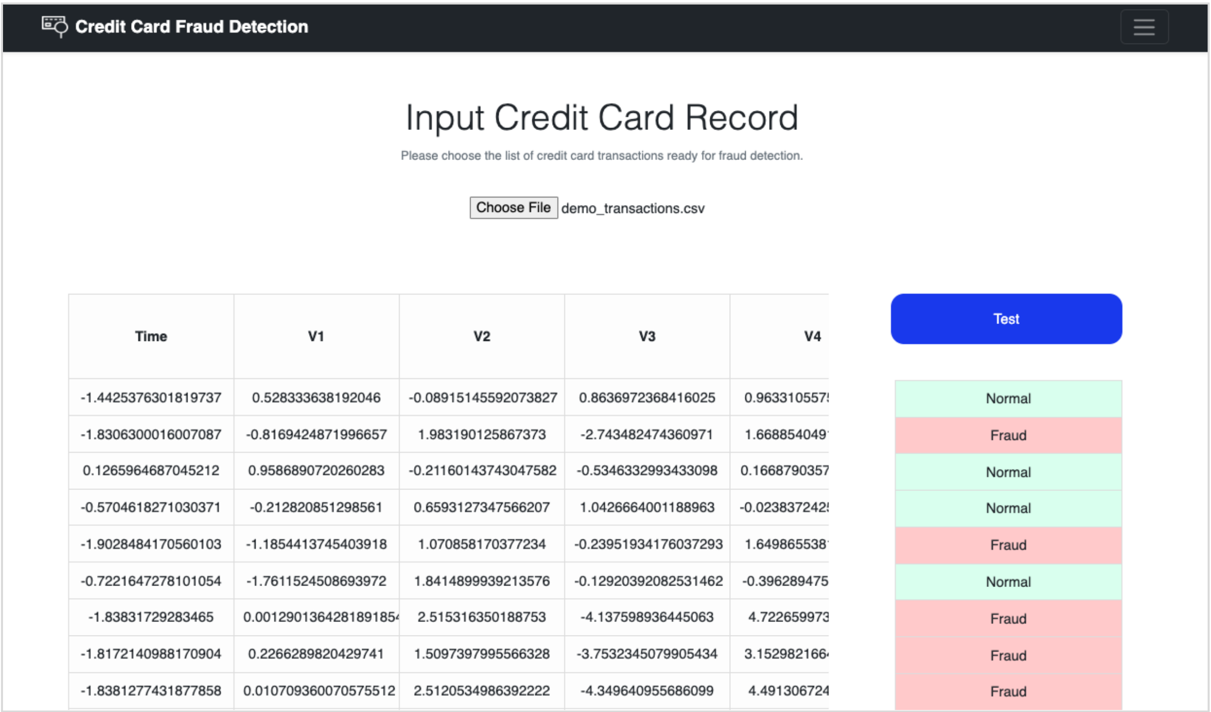
Input Credit Card Record

Please choose the list of credit card transactions ready for fraud detection.

Time	V1	V2	V3	V4

After loading the .csv file, the empty HTML table should be populated with the contents of the file. Next is pressing the “Test” button, which will display the predictions of the model alongside the appropriate transactions.

The goal of this app is to flag fraud to a client of the financial organization, as such, if fraudulent transactions have been detected at this stage, the organization must be alerted and subsequently the client themself.



Credit Card Fraud Detection

Input Credit Card Record

Please choose the list of credit card transactions ready for fraud detection.

Choose File demo_transactions.csv

Test

Time	V1	V2	V3	V4	
-1.4425376301819737	0.528333638192046	-0.08915145592073827	0.8636972368416025	0.963310557	Normal
-1.8306300016007087	-0.8169424871996657	1.983190125867373	-2.743482474360971	1.668854049	Fraud
0.1265964687045212	0.9586890720260283	-0.21160143743047582	-0.5346332993433098	0.1668790357	Normal
-0.5704618271030371	-0.212820851298561	0.6593127347566207	1.0426664001188963	-0.023837242	Normal
-1.9028484170560103	-1.1854413745403918	1.070858170377234	-0.23951934176037293	1.649865538	Fraud
-0.7221647278101054	-1.7611524508693972	1.8414899939213576	-0.12920392082531462	-0.396289475	Normal
-1.83831729283465	0.001290136428189185	2.515316350188753	-4.137598936445063	4.722659973	Fraud
-1.8172140988170904	0.2266289820429741	1.5097397995566328	-3.7532345079905434	3.152982166	Fraud
-1.8381277431877858	0.010709360070575512	2.5120534986392222	-4.349640955686099	4.491306724	Fraud

User Manual

Preliminary actions

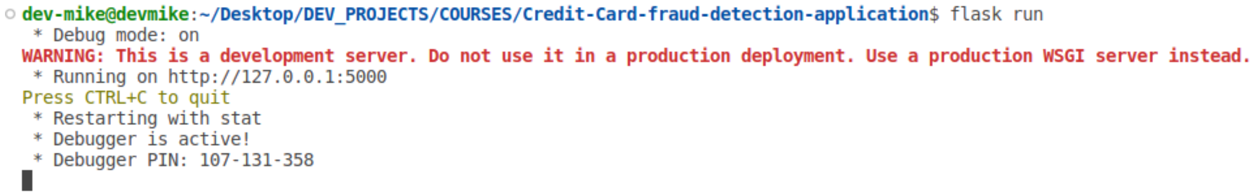
The first thing to do is to install the requirements.txt file for the libraries of the application. Open the command prompt and with a Python version > 3.x.x and type the following command. Note that this command might be different if you use Python 2.x.x. :

pip install -r requirements.txt

Afterwards, open the project directory in Terminal or Command Line depending on your operating system. Type the following command:

flask run

You should see an image like this showing that server is running on port 5000



```
○ dev-mike@devmike:~/Desktop/DEV_PROJECTS/COURSES/Credit-Card-fraud-detection-application$ flask run
   * Debug mode: on
  WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
   * Running on http://127.0.0.1:5000
  Press CTRL+C to quit
   * Restarting with stat
   * Debugger is active!
   * Debugger PIN: 107-131-358
  █
```

How to use the Application

Open a browser and type the url:

http://localhost:5000

Enter the values within the fields and click on the submit button to check the output from the prediction..

REFERENCES

Dataset: https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud?resource=download

Github Repository: https://github.com/aidahimm/CreditCard_FraudDetection