



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

ABSTRACT	2
TABLE OF CONTENTS	3
KDD Process	5
The Dataset	5
Description	5
Class Distribution	5
Preprocessing	6
Data Cleaning	6
Missing values	6
Duplicates	7
Dataset Rebalancing	7
Data Mining	9
Feature Selection	9
Cross Validation: Holdout Method	12
Classification	13
Random Forest	14
C4.5	14
Random Tree	15
Logistic Regression	15
AdaBoost	15
Naive Bayes	15
ZeroR	16
Model Selection	16
Implementation	17
Preprocessing	17
Cross-validation: Data split	18
Feature Selection	19
Random Forest Evaluation	20
Without rebalancing	20
With rebalancing	21
Anomaly Detection	22
Isolation Forest	22
Local Outlier Factor	23
K-Means Clustering	24
One Class Support Vector Machine (SVM)	24
Reflections	25

Application	26
Analysis Phase	26
Description	26
Main Actors	26
Requirements	26
Functional Requirements	26
Non-Functional Requirements	26
Use Case Diagram	27
Design Phase	27
Credit Card Fraud Detection Process Flow	27
Application Architecture	27
Implementation Phase	28
Test Phase	29
User Manual	31
Preliminary actions	31
How to use the Application	31
REFERENCES	32

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

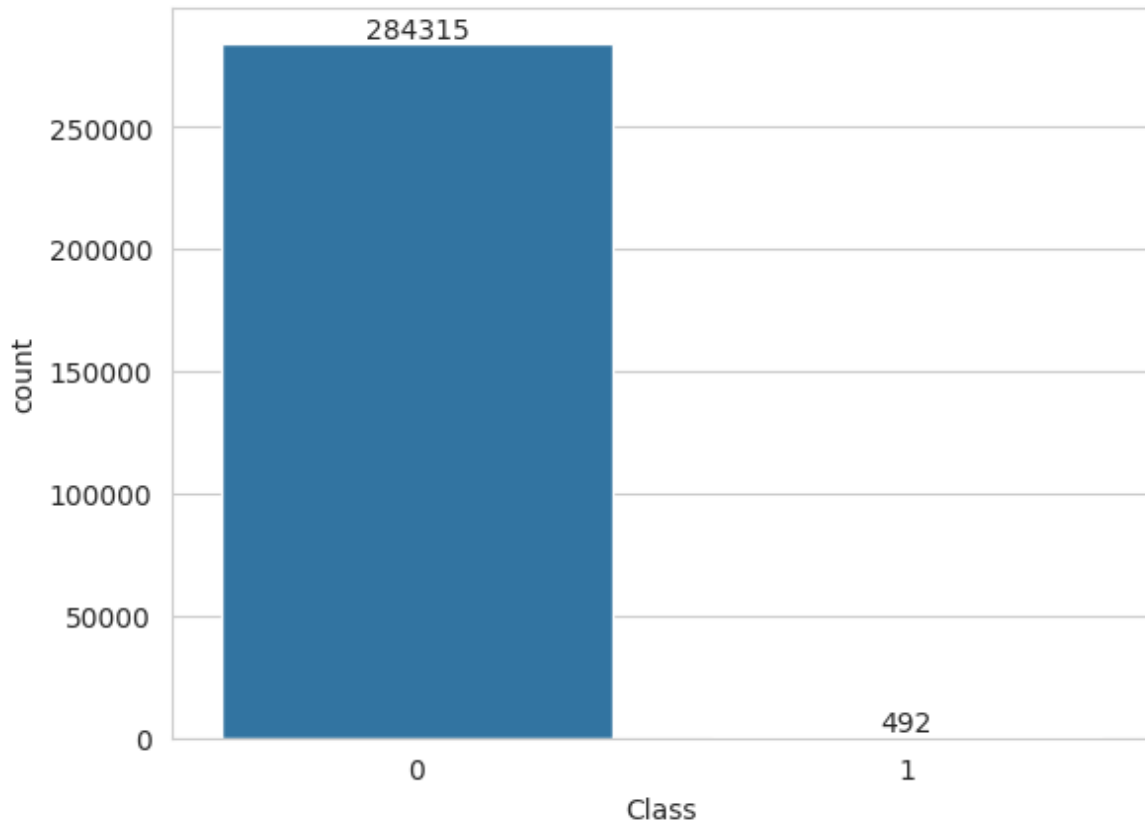
- Time (in s): relative to the duration between the current selected transaction and the first one recorded on that credit card.
- Amount (in €): The amount spent in each transaction.
- Class (binary): 0 for normal, 1 for fraudulent.
- V1 - V28: Features transformed with the Principal Component Analysis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

Data Cleaning

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	

Duplicates

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

Dataset Rebalancing

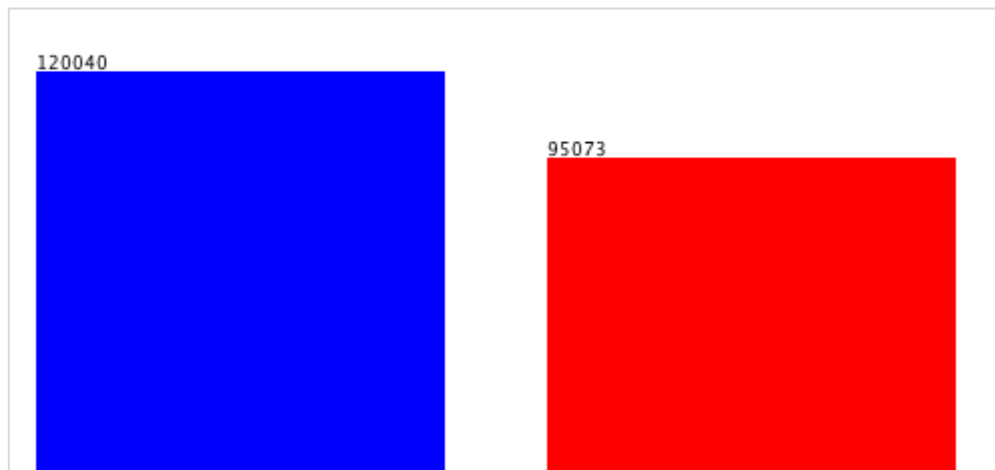
This step was achieved using two filters from Weka's instance-centered supervised algorithms for oversampling and undersampling.

Using The supervised algorithm "Resample" we were able to reduce the majority class to a total of 120 040 instances only, which is less than half. However, the logic of this algorithm makes it so that the minority class was also oversampled using replication methods.

We preferred to use 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.

First, we needed to remove the duplicates created by the “Resample” algorithm, then we applied the SMOTE filter. The results are as follows, the blue being the negative class, and red being the positive one.

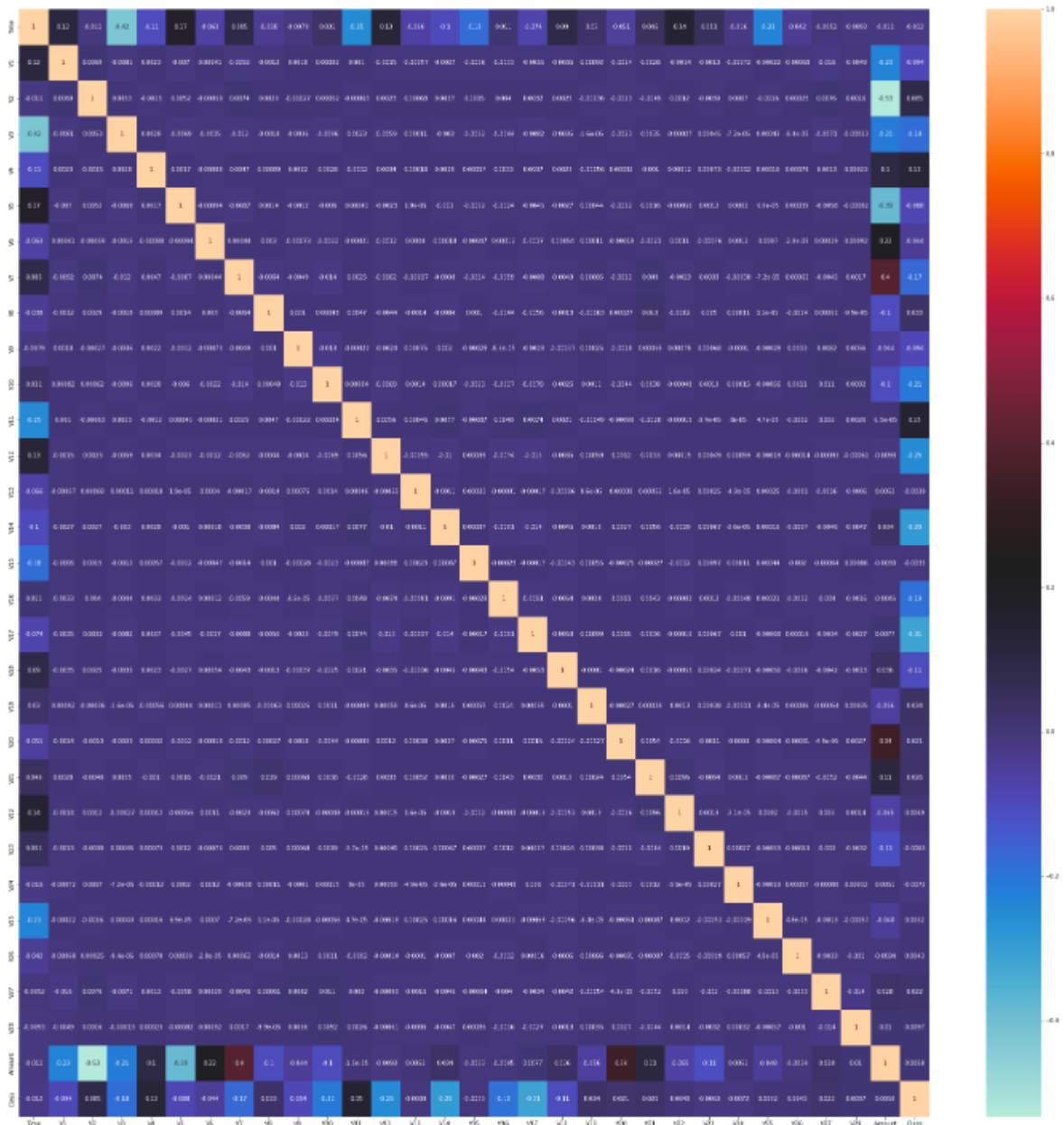


Data Mining

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.

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 last combination was PrincipalComponents with Ranker. This algorithm performs a PCA transformation on the features (for the second time in the case of our dataset) to find the features that still preserve as much information from the complete data as possible.

```

Ranked attributes:
0.6221  1 0.274V10+0.271V17+0.27 V12+0.269V16+0.264V3...
0.5267  2 0.439V21-0.408V22+0.407V8+0.342V27-0.253V20...
0.471   3 0.356V25-0.33V5-0.321V1+0.284V14-0.256V26...
0.421   4 -0.428V24-0.393V19+0.308V25+0.25 V28+0.245V20...
0.38    5 0.791Amount+0.35 V20+0.224V27-0.192V2-0.19V23...
0.3416  6 0.494Time-0.445V26-0.369V13+0.312V28-0.303V15...
0.3046  7 0.495V28+0.455V15+0.396V23-0.378V13+0.256V24...
0.2724  8 -0.518V13-0.466V23+0.417V15-0.281V28-0.227Time...
0.2428  9 0.709V26+0.486V24-0.279V23-0.213V13+0.208Time...
0.214   10 -0.468Time-0.445V23+0.443V28+0.303V13+0.266V24...
0.1863  11 -0.35V13-0.319V15-0.312Time-0.305V20-0.287V28...
0.162   12 0.588V27+0.392V24-0.308V26-0.297V28+0.221V22...
0.1393  13 -0.402V19+0.387V15-0.366V25-0.339V6+0.266V24...
0.1183  14 -0.613V25+0.392V27-0.334V22+0.211V6-0.205Amount...
0.1007  15 0.446V20-0.412V6+0.411V19-0.392V27-0.308V22...
0.0847  16 0.452V22+0.379V8-0.368V6-0.275Time-0.273V25...
0.0712  17 -0.646V21-0.443V20+0.326V19-0.267V22+0.169Amount...
0.0594  18 0.418V18+0.342V19+0.335V9+0.315V21-0.309V8...
0.0481  19 0.619V8+0.44 V6+0.261V4+0.24 V18-0.195V27...

Selected attributes: 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19 : 19

```

This method as observed in the figure above, produces new attributes made of percentages of the original attributes, and calculates the variance, selecting features that were not picked by the other algorithms.

Cross Validation: Holdout 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 **holdout** method, which defines that the dataset is split into a training and testing set, based on a user-defined percentage.

We chose to feed our model 66% of the dataset during the training phase, then test its learning on the 34% remaining instances.

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 %	Mean Absolute Error	Root Mean Squared Error	Relative Absolute Error %	Root Relative Squared Error %	Precision	Recall	F-Measure
Random Forest	66% train – 34% test	-	99.831	0.012	0.048	2.451	9.801	0.998	0.998	0.998
C45 Pruned	66% train – 34% test	-	99.364	0.007	0.078	1.565	15.789	0.994	0.994	0.994
C45 Unpruned	66% train – 34% test	-	99.353	0.007	0.079	1.521	15.942	0.994	0.994	0.994
RandomTree	66% train – 34% test	-	98.937	0.011	0.103	2.154	20.764	0.989	0.989	0.989
logistic Function	66% train – 34% test	-	98.243	0.028	0.117	5.836	23.692	0.983	0.982	0.982
AdaBoost	66% train – 34% test	-	96.784	0.048	0.159	9.778	32.199	0.968	0.968	0.968
Naive Bayes	66% train – 34% test	-	94.176	0.058	0.236	11.841	47.718	0.943	0.942	0.941
ZeroR	66% train – 34% test	-	56.005	0.493	0.496	100	100	?	0.560	?
Random Forest	66% train – 34% test	Correlation AttributeEval+ Ranker	99.830	0.012	0.048	2.450	9.801	0.998	0.998	0.998
Random Forest	66% train – 34% test	CfSubsetEval + BestFirst	99.521	0.012	0.063	2.507	12.682	0.995	0.995	0.995
C45 pruned	66% train – 34% test	CorrelationAttributeEval + Ranker	99.364	0.007	0.0784	1.565	15.789	0.994	0.994	0.994
Random Tree	66% train – 34% test	CfSubsetEval + BestFirst	98.929	0.010	0.103	2.170	20.844	0.989	0.989	0.989
C45 pruned	66% train – 34% test	Principal Component + Ranker	98.586	0.016	0.115	3.355	23.304	0.986	0.986	0.986

The measure considered for evaluation are :

- **Accuracy:** The percentage of correctly classified instances compared to the total number of instances considered.
- **MAE (Mean Absolute Error):** A measure of errors between predictions and observations, where all individual differences have equal weight.
- **RMSE (Root Mean Squared Error):** Used to measure the differences between values predicted by a model or an estimator and the values observed, giving a higher weight to larger errors.
- **RAE (Relative Absolute Error):** Defined as the total absolute difference between the realized and predicted values. A good forecasting model will produce a ratio close to zero.
- **RRSE (Root Relative Square Error):** The Root Relative Squared Error indicates how well a model performs relative to the average of the true values. Therefore, when the RRSE is lower than one, the model performs better than the simple model.
- **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 accuracy with a dashing 99.83%, but there were other models that did better in other evaluation measures, namely the MAE (Mean Absolute Error) and RAE (Relative Absolute Error).

While those two metrics may be important when performing regression because we'd want the closest prediction to the actual value, they don't rank the highest for binary classification. Either an instance was predicted correctly or not.

Therefore, we look at other measures such as Precision and recall which give an insight on the model's predictive performance, in this case 99.8% for both. The f-measure also being a high 99.8%.

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 very good accuracy, almost as good as the first results (CorrelationAttributeEval), 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 an accuracy level of 99.36%, which is very high as well. It did better than Random Forest in MAE and RAE with the lowest values in the table, but looking at the precision and recall which are more relevant, we see that it is lower by 0.4%
- **Unpruned tree:** Gave an accuracy percentage of 99.35%, slightly lower and most likely due to overfitting with larger subtrees.

Since the pruned version of the C4.5 decision tree performed better, we used it with attribute selection and found that with CorrelationAttributeEval it gave the exact same

result as without, and with the PrincipalComponent evaluator it was even a little worse.

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 the fourth highest accuracy with 98.94%, but still lower precision and recall than all first two. Again, the attribute selection produced very close results but not on-par and not better.

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 98.24% accuracy and 99.82% in PRF scores.

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 the rest 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 most likely the reason why it only achieved an accuracy of 96.78%.

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 lower 94.17% in accuracy; we can observe from the correlation matrix that there is a level of correlation between certain features, and that may have confused the classifier to an extent.

ZeroR

This classifier is rule-based, and it is the simplest classification model since it ignores all features only to predict the majority class, and that explains the 100% values in RAE and RRSE since those describe how much better or similar the model behaves compared to a simple model. The accuracy level is very low with 56% and takes the last place of the ranking.

Model Selection

After comparing all these results and evaluation metrics we can see that the best models are the Tree-based classifiers. They have very high accuracy and would be a good choice for readability and interpretability reasons.

Moreover, to narrow down the selection we can see that Random Forest and C4.5 have the closest results with over 99% accuracy. We can see that C4.5 has lower MAE and RAE which should be closest to zero for a good model, but as mentioned previously, the difference isn't large and those measures have more importance in the context of continuous numerical value prediction, rather than binary classification.

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

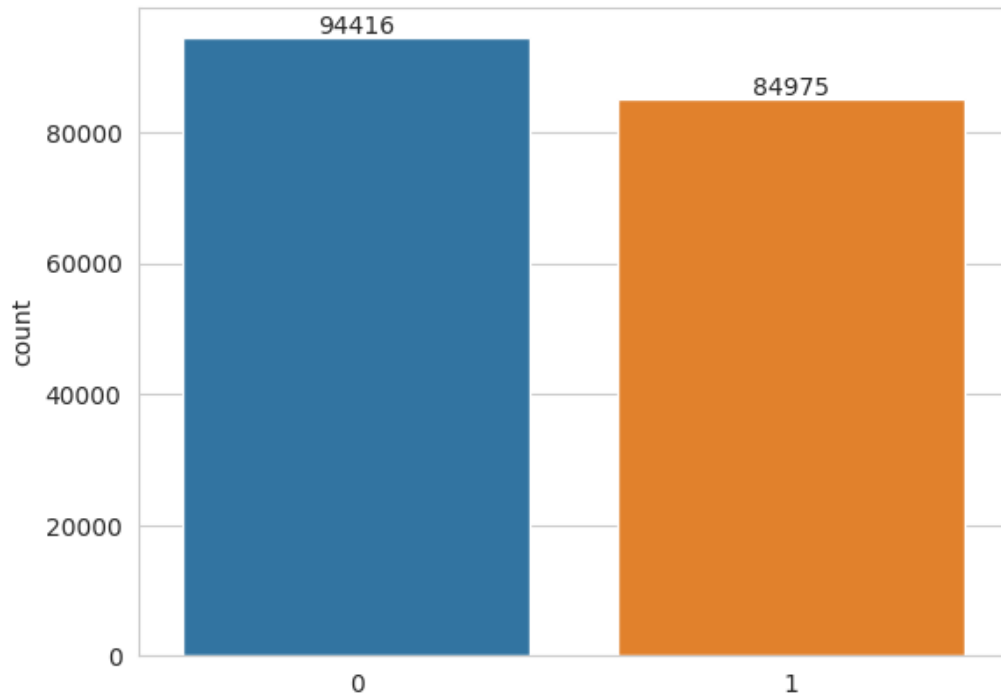
Implementation

Preprocessing

As mentioned, the same steps done on Weka were replicated in Python. For **duplicate** detection and removal, we used Panda's built-in method *duplicated()*.

```
## Finding and removing duplicates
data.duplicated().sum()
1081
data.drop_duplicates(inplace=True)
data['Class'].value_counts()
0    283253
1      473
Name: Class, dtype: int64
```

For dataset **rebalancing** In the python notebook this was done using an oversample/undersample combination or SMOTE and RandomUnderSampler, with sampling strategies of 0.3 and 0.9 respectively. Ending up with a class distribution of 94416 negatives and 84975 positives.



Cross-validation: Data split

Firstly, we separated our labels (data to be predicted: Temperature) and the features (the rest of the attributes) in different variables.

The “sklearn” methods we used to build our models take data in arrays, so we transformed our temperature dataset into such.

Then, we needed to split the dataset into training data; the labeled data that the model will use during the learning process in order to find the necessary patterns, and testing data; the labeled data that it will test its own performance against.

We used Python’s “train_test_split()” from the library sklearn with a train-test ratio of 66%-34% respectively.

```

# 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,
                                                                              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,)

```

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, which as we saw led to a 99.53% accuracy in Random Forest, 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 100 decision trees and a 2-job parallelization. We then call the `fit()` method and pass our training features and appropriate labels. The model takes 1.8 minutes to build and learn the 66% training data.

```
# Train the model on training data
%time
model = RandomForestClassifier(n_estimators=100, verbose=2, n_jobs=2, oob_score=True).fit(train_features,
                                                                                          train_labels)
```

building tree 83 of 100
building tree 84 of 100
building tree 85 of 100
building tree 86 of 100
building tree 87 of 100
building tree 88 of 100
building tree 89 of 100
building tree 90 of 100
building tree 91 of 100
building tree 92 of 100
building tree 93 of 100
building tree 94 of 100
building tree 95 of 100
building tree 96 of 100
building tree 97 of 100
building tree 98 of 100
building tree 99 of 100
building tree 100 of 100

[Parallel(n_jobs=2)]: Done 100 out of 100 | elapsed: 1.8min finished

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: 0.2s
[Parallel(n_jobs=2)]: Done 100 out of 100 | elapsed: 0.6s finished

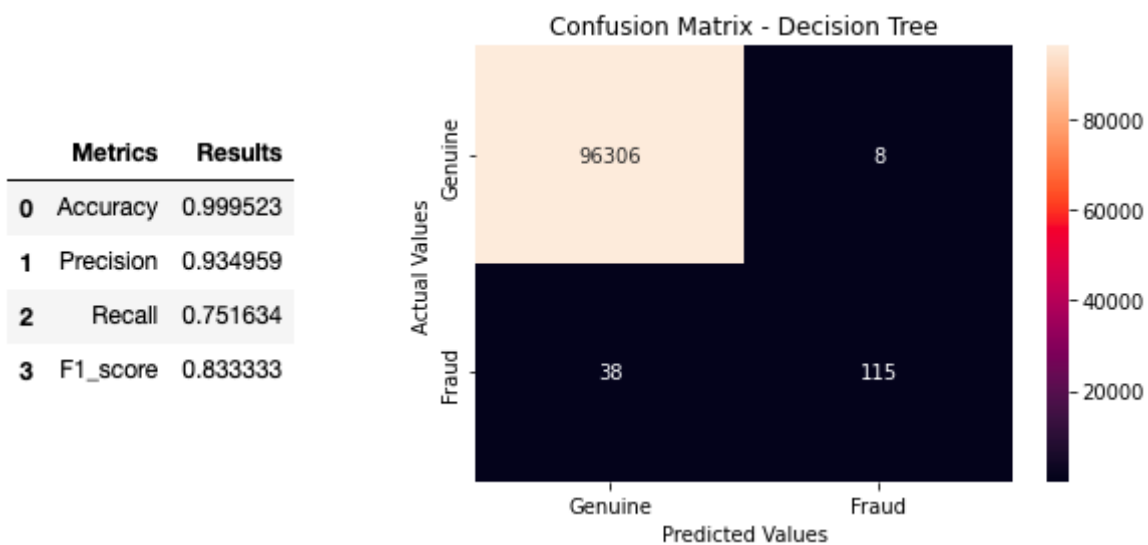
Without rebalancing

Without rebalancing the dataset, the performance of this model would have been 99.95% which is already very high. However when we look closely at the confusion matrix we can

notice that the ratio of correctly-incorrectly classified instances for the minority class isn't proportional to that of the majority class.

Which is why even if the accuracy is very high it was most likely skewed by the large number of instances in the majority class.

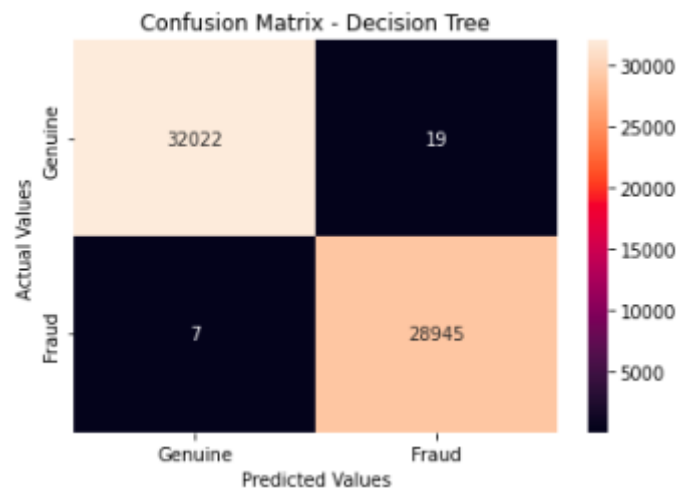
There are 38 instances misclassified in the minority class which is 25% of the total fraudulent transaction in the testing set. The Recall measure is 83.3% which is a little more telling on the performance but not too much.



With rebalancing

When we rebalance the dataset to the distribution mentioned above, we see that the ratios are close. We have 0.06% misclassified normal transactions and 0.02% misclassified fraud transactions. The confusion matrix is in this case insightful and valid. Moreover we see that the recall and f-score measures have increased.

	Metrics	Results
0	Accuracy	0.999574
1	Precision	0.999344
2	Recall	0.999758
3	F1_score	0.999551



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

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).

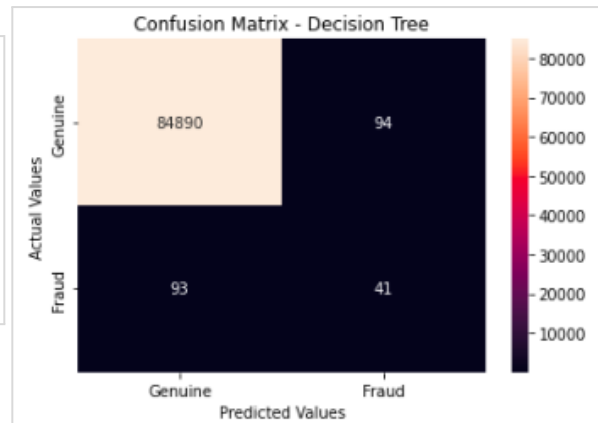
This implementation of this model is included in the “sklearn.ensemble” module under the name IsolationForest.

As we can see in the results, the Isolation Forest made 187 errors, and has an accuracy of 99.78%, most likely due to the unbalanced dataset, so it is necessary to

look at the precision and recall: it was able to recognize all the normal transactions perfectly with 100% PRF, but only 31% of the fraudulent ones.

Though considering the number of errors, PRF cannot be 100 for the normal class, so assuming that something is wrong with sklearn's *classification_report()* method, we made use of a confusion matrix to understand the classification better.

Errors: 187					
Accuracy Score:					
0.9978030498836908					
Classification Report:					
	precision	recall	f1-score	support	
0	1.00	1.00	1.00	84984	
1	0.30	0.31	0.30	134	
accuracy			1.00	85118	
macro avg	0.65	0.65	0.65	85118	
weighted avg	1.00	1.00	1.00	85118	



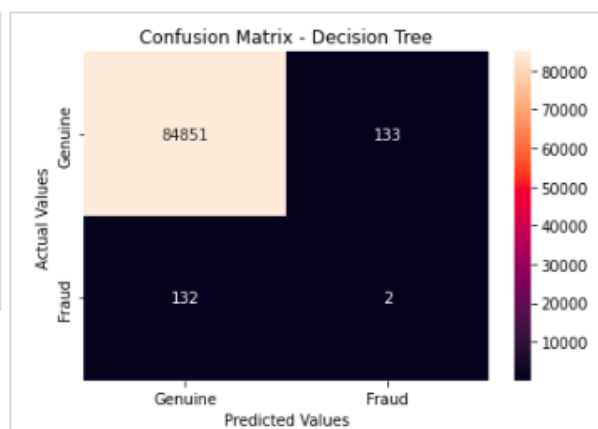
Local Outlier Factor

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.

Here we can see that it made 265 errors, more than the last, and has an accuracy of 99.68% while only being able to detect 1% of the fraudulent transactions.

Errors: 265					
Accuracy Score:					
0.9968866749688667					
Classification Report:					
	precision	recall	f1-score	support	
0	1.00	1.00	1.00	84984	
1	0.01	0.01	0.01	134	
accuracy			1.00	85118	
macro avg	0.51	0.51	0.51	85118	
weighted avg	1.00	1.00	1.00	85118	



K-Means Clustering

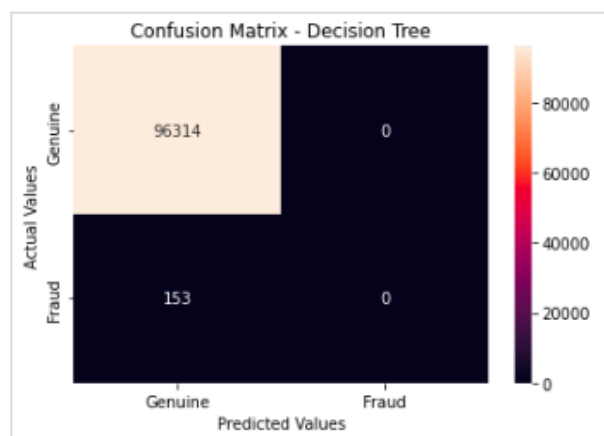
This algorithm aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean (cluster centers or cluster centroid), serving as a prototype of the cluster.

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

It made 134 errors, most likely as all fraud transaction have been labeled normal, since the PRF scores of the minority class are all 0%

```
Errors: 134
Accuracy Score:
0.9984257148899175
Classification Report:
```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	84984
1	0.00	0.00	0.00	134
accuracy			1.00	85118
macro avg	0.50	0.50	0.50	85118
weighted avg	1.00	1.00	1.00	85118



One Class Support Vector Machine (SVM)

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 the worst out of all, with 34054 error and 59.99% accuracy. The model has less than 0% precision on the fraudulent transactions, and not only is it unable to detect outliers, it is also unable to identify the normal class as opposed the the first two which did it perfectly

Errors: 34054				
Accuracy Score:				
0.5999201109048615				
Classification Report:				
	precision	recall	f1-score	support
0	1.00	0.60	0.75	84984
1	0.00	0.41	0.00	134
accuracy			0.60	85118
macro avg	0.50	0.51	0.38	85118
weighted avg	1.00	0.60	0.75	85118

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. Perhaps that is due to the fact that in many features the outliers were not necessarily distinct from the normal transactions, and so it was difficult to use density/distance based methods to find them.

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

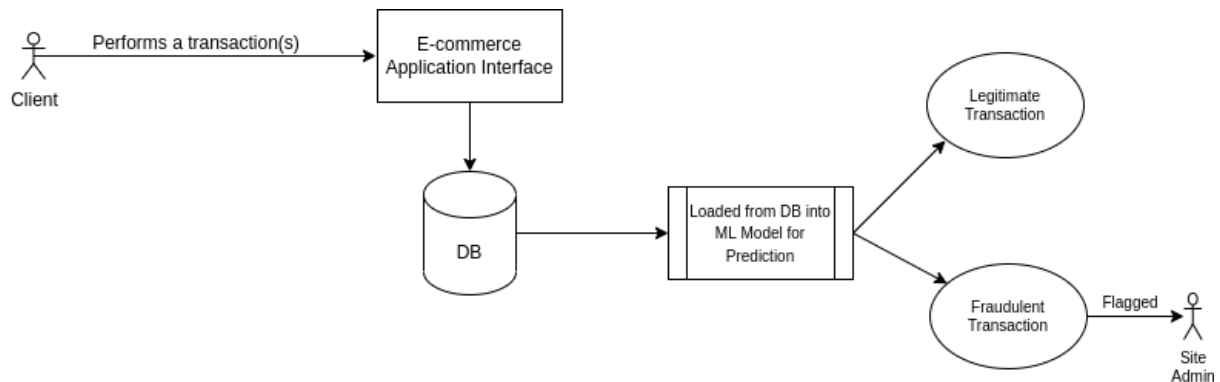
The application must relate to the information populated on the interface to classify a transaction as fraud or normal. The application consumes a Comma Separated Values (CSV) file chosen by the user, and converts it to a Python list sent to the model. After the test data is submitted, it is run through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



Design Phase

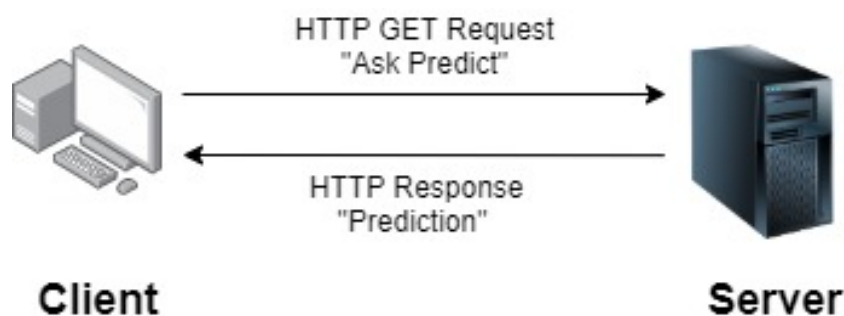
This phase highlights the application's main scenario. Pseudocode and mock-ups are used to explain the application flow and logic.

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

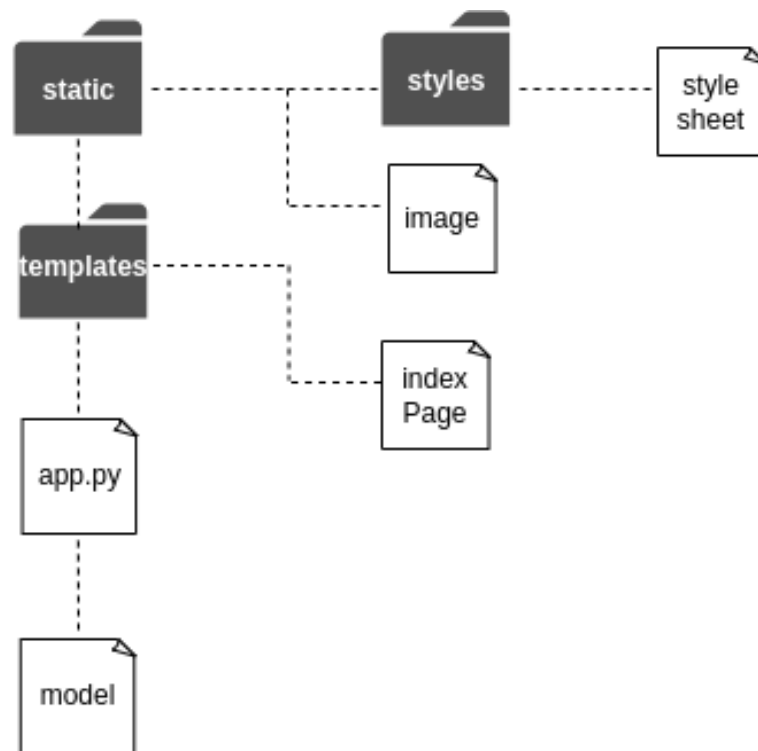
In order to satisfy the non-functional requirements for the application, specifically, the ease-of-use and portability of the application, a client-server architecture was implemented. The client hits a POST request by making a call to a predict function and the server responds with the prediction value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



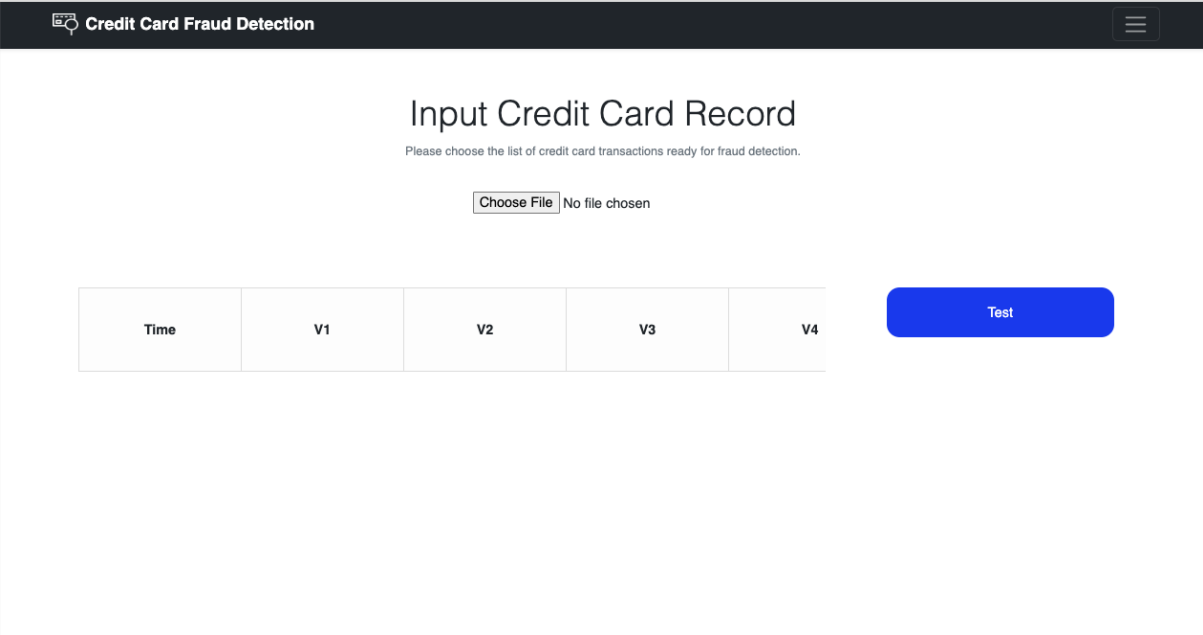
The application directory is organized into:

- **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.



The screenshot shows the web interface of the "Credit Card Fraud Detection" application. At the top, there is a dark header with the application name and a menu icon. The main content area has a title "Input Credit Card Record" and a subtitle "Please choose the list of credit card transactions ready for fraud detection." Below this is a file upload section with a "Choose File" button and the text "No file chosen". Underneath is a table with five columns: "Time", "V1", "V2", "V3", and "V4". To the right of the table is a prominent blue "Test" button.

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 themselves.

Input Credit Card Record

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

demo_transactions.csv

Time	V1	V2	V3	V4
-1.4425376301819737	0.528333638192046	-0.08915145592073827	0.8636972368416025	0.9633105571
-1.8306300016007087	-0.8169424871996657	1.983190125867373	-2.743482474360971	1.668854049
0.1265964687045212	0.9586890720260283	-0.21160143743047582	-0.5346332993433098	0.1668790357
-0.5704618271030371	-0.212820851298561	0.6593127347566207	1.0426664001188963	-0.0238372421
-1.9028484170560103	-1.1854413745403918	1.070858170377234	-0.23951934176037293	1.649865538
-0.7221647278101054	-1.7611524508693972	1.8414899939213576	-0.12920392082531462	-0.396289475
-1.83831729283465	0.0012901364281891854	2.515316350188753	-4.137598936445063	4.722659973
-1.8172140988170904	0.2266289820429741	1.5097397995566328	-3.7532345079905434	3.152982166
-1.8381277431877858	0.010709360070575512	2.5120534986392222	-4.349640955686099	4.491306724

Test

Normal

Fraud

Normal

Normal

Fraud

Normal

Fraud

Fraud

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