

Anish Almeida

Professor Sundeep Rangan

Machine Learning Project

12/16/18

A common occurrence in everyday life is the use of one's credit card. Most individuals make small purchases with their credit cards ranging from small groceries to lunch and dinner receipts. A vast majority of Americans have up to 2 to 3 credit cards. With multiple cards per person, keeping a check on one's day to day expenses is challenging enough- not to mention time consuming. Most individuals aren't even aware if there is fraudulent activity on their cards unless abnormally large purchases are made within a relatively close time frame to each other. Only when large transactions are made do the bank or credit card companies alert the individual. While these services are improving, and most credit card companies have dedicated fraud and identity theft prevention teams- it is certainly a growing topic of importance.

The scope of my project is aimed towards understanding how machine learning algorithms are being used to detect fraud. I will be basing my program from existing algorithms such as the Local Outlier Factor. Similar to how a credit card company tracks each purchase- the Local Outlier Factor in my program will make use of each purchase as a data point. The LOF is an unsupervised outlier detection method. Unsupervised learning makes the assumption that the vast majority of data points that are in clusters are the "normal" data points and are valid. The data points that are isolated are labeled as "abnormal." I also decided to use a second existing algorithm called the Isolation Forest Method to compare to the LOF. The Isolation Forest Method makes use of determining the anomaly score of each sample. The Isolation Forest 'isolates' observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature.

My dataset: <https://www.kaggle.com/mlg-ulb/creditcardfraud>

In this report, I will give a detailed explanation of each algorithm and the complete breakdown of my program.

While it is important to acknowledge that Credit card companies may have a faster response time and the benefit of having a personal representative for fraud detection- it is certainly important and beneficial to explore different programs and algorithms as this can improve existing code and perfect existing programs.

```
In [ ]: #Anish Almeida
        #Credit Card Fraud Project
        #Machine Learning
        #aa4170
```

```
In [14]: # importing all relevant libraries/packages
        # getting the library versions

import sys
import numpy
import pandas
import matplotlib
import seaborn
import scipy
import sklearn

print('Python: {}'.format(sys.version))
print('Numpy: {}'.format(numpy.__version__))
print('Pandas: {}'.format(pandas.__version__))
print('Matplotlib: {}'.format(matplotlib.__version__))
print('Seaborn: {}'.format(seaborn.__version__))
print('Scipy: {}'.format(scipy.__version__))
print('Sklearn: {}'.format(sklearn.__version__))

Python: 3.6.5 |Anaconda, Inc.| (default, Mar 29 2018, 13:32:41) [MSC v.1900 6
4 bit (AMD64)]
Numpy: 1.14.3
Pandas: 0.23.0
Matplotlib: 2.2.2
Seaborn: 0.8.1
Scipy: 1.1.0
Sklearn: 0.19.1
```

```
In [15]: # importing all relevant packages
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

- 1) I first begin with importing all necessary and relevant libraries and packages. Most importantly, I am using the sklearn package from Python.

```
In [16]: # Load the dataset from the csv file using pandas

import pandas as pd

dataset = pd.read_csv('creditcard.csv')
```

2) I also import the dataset.

```
In [17]: #Set up the data set

print(dataset.columns)

print(dataset.shape)

Index(['Time', 'V1', 'V2', 'V3', 'V4', 'V5', 'V6', 'V7', 'V8', 'V9', 'V10',
       'V11', 'V12', 'V13', 'V14', 'V15', 'V16', 'V17', 'V18', 'V19', 'V20',
       'V21', 'V22', 'V23', 'V24', 'V25', 'V26', 'V27', 'V28', 'Amount',
       'Class'],
      dtype='object')
(284807, 31)
```


3) I begin to format/set up the data set, so I can get a good idea of exploring the dataset. I see there is a 'class' column as well as 284,807 total transactions. The class column splits the transactions into two groups: 0 for valid and 1 for invalid.

```
In [18]: dataset.head()
```

Out[18]:

	Time	V1	V2	V3	V4	V5	V6	V7	
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270

5 rows × 31 columns



4) Here I display the dataset, getting a better picture of what each transaction looked like and the time passed between each. V1-V28 is the results of PCA dimensionality to protect user identities and sensitive features. (PCA dimensionality reduces the number of random variables by obtaining a principal set of values.)

```
datasetnew=dataset.sample(frac=0.2,random_state=1)
print(datasetnew.shape)
print(datasetnew.describe())
```

- 5) Here I am printing the shape of the data, but I came to the realization that I would need to drastically shorten the sample size of the data set. 284,807 transactions would be too much in terms of runtime and computation. Originally, I took only 10% of the sample size. However, I realized that this was too small a sample size for good results. So I increased the sample size to 20%, which is 56,961 transactions. The shape of data is outputted below.

```
(56961, 31)
```

	Time	V1	V2	V3	V4
\ count	56961.000000	56961.000000	56961.000000	56961.000000	56961.000000
mean	94571.23811	0.007759	-0.007820	0.010548	0.004755
std	47566.88462	1.944402	1.654560	1.495860	1.415369
min	0.000000	-46.855047	-63.344698	-31.813586	-5.266509
25%	53809.000000	-0.915616	-0.607161	-0.883553	-0.843085
50%	84511.000000	0.033625	0.061363	0.185144	-0.012678
75%	139237.000000	1.318624	0.800511	1.031245	0.750016
max	172784.000000	2.411499	17.418649	4.069865	16.715537

	V5	V6	V7	V8	V9
\ count	56961.000000	56961.000000	56961.000000	56961.000000	56961.000000
mean	-0.011918	0.002191	-0.011845	0.000873	0.007915
std	1.379057	1.329111	1.209897	1.160905	1.093541
min	-42.147898	-23.496714	-26.548144	-33.785407	-8.739670
25%	-0.707818	-0.765759	-0.563241	-0.206358	-0.632800
50%	-0.071228	-0.270147	0.030874	0.024519	-0.044122
75%	0.605042	0.403820	0.558736	0.327343	0.605625
max	34.099309	22.529298	36.677268	19.587773	10.370658

	...	V21	V22	V23	V24
\ count	...	56961.000000	56961.000000	56961.000000	56961.000000
mean	...	0.002932	0.003058	-0.001127	-0.001869
std	...	0.723614	0.723845	0.654303	0.603150
min	...	-16.640785	-10.933144	-36.666000	-2.836627
25%	...	-0.227125	-0.541228	-0.162431	-0.356802
50%	...	-0.029706	0.010522	-0.010926	0.038833
75%	...	0.186236	0.531158	0.148271	0.434270
max	...	22.588989	6.090514	18.946734	3.962197

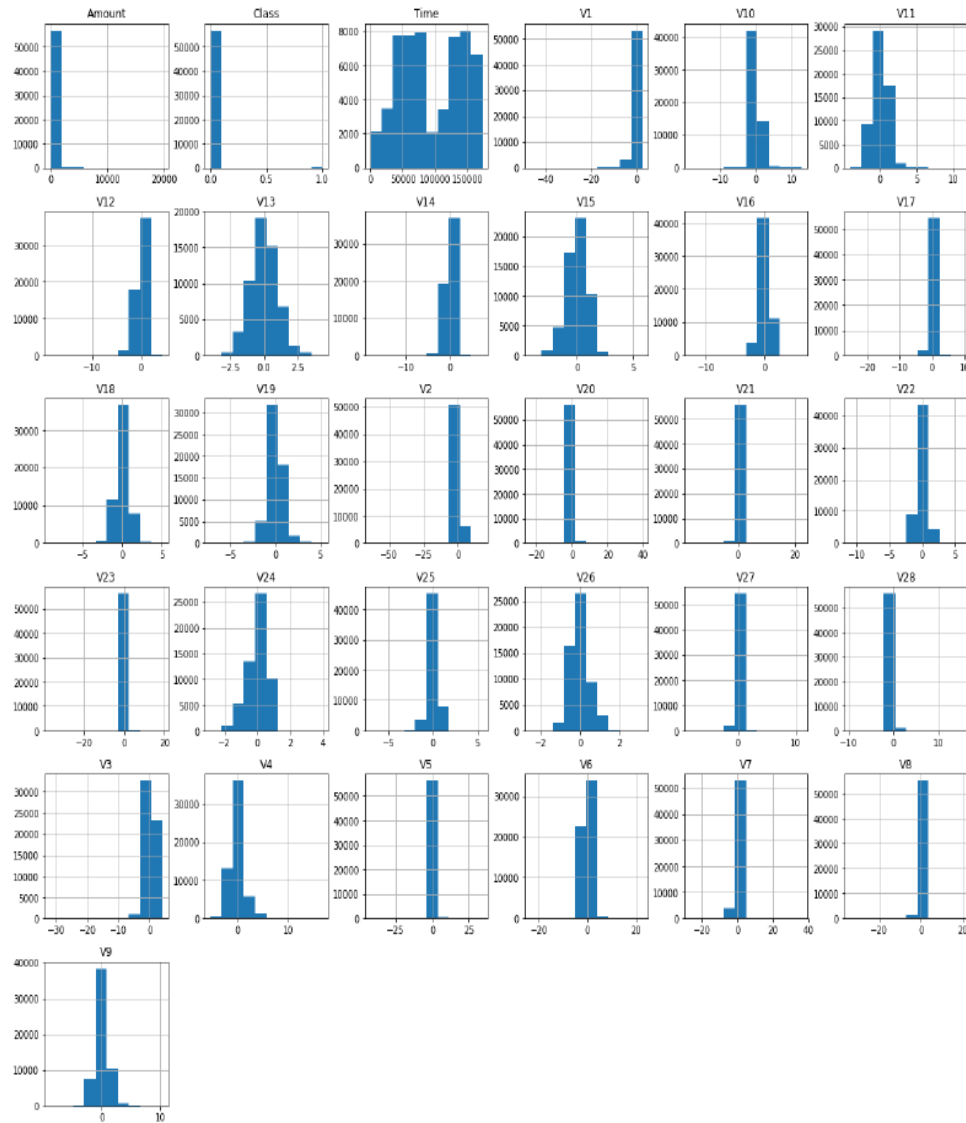
	V25	V26	V27	V28	Amount
count	56961.000000	56961.000000	56961.000000	56961.000000	56961.000000
mean	0.000662	0.002157	0.000632	-0.002841	88.750996
std	0.519287	0.481474	0.393477	0.302685	254.652038
min	-7.025783	-2.534330	-8.260909	-9.617915	0.000000
25%	-0.316805	-0.324992	-0.071111	-0.053391	5.950000
50%	0.017169	-0.049862	0.001250	0.010776	22.160000
75%	0.350835	0.243869	0.089448	0.076211	77.900000
max	5.541598	3.155327	11.135740	15.373170	19656.530000

	Class
count	56961.000000
mean	0.001527
std	0.039052
min	0.000000
25%	0.000000
50%	0.000000
75%	0.000000
max	1.000000

[8 rows x 31 columns]

In the class column, we see the mean is close to zero, which means most are valid transactions, as this is real life data, this isn't surprising. 1 max is fraud transactions and 0 is standard transactions.

```
In [32]: datasetnew.hist(figsize = (20, 20))
plt.show()
```



- 6) I plotted the histogram of each parameter, to better visualize the data. Most of the transactions are clustered around zero, with few outliers.

```

fraudulent_cases = datasetnew[datasetnew['Class'] == 1]
valid_cases = datasetnew[datasetnew['Class'] == 0]

percent_fraud = len(fraudulent_cases)/float(len(valid_cases))
print(percent_fraud)

print('Fraudulent Cases: {}'.format(len(datasetnew[datasetnew['Class'] == 1])))
print('Valid/Correct Transactions: {}'.format(len(datasetnew[datasetnew['Class'] == 0])))

0.0015296972254457222
Fraudulent Cases: 87
Valid/Correct Transactions: 56874

```

- 7) Now that I had a good visual of the dataset, I decided to classify the data and to determine what the number of fraud cases in the dataset was. I indexed the data to class as 1 for the fraudulent case and did the same for valid case as 0. I then calculated the percent of fraud cases to valid cases, the percent fraud. The percent fraud will be needed later on.

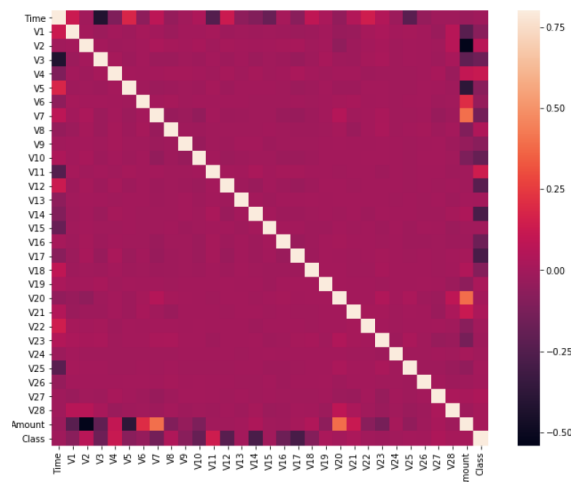
There is a large disparity between fraud cases and valid transactions, so this will be harder to predict.

```

In [34]: # Correlation matrix
corrmat = datasetnew.corr()
fig = plt.figure(figsize = (12, 9))

sns.heatmap(corrmat, vmax = .8, square = True)
plt.show()

```



- 8) I also built a correlation matrix which tells me if there is a strong correlation between variables, or if I need to remove any variables I don't need. It also shows me which features are important for overall classification.

I can see on the y-axis, V1-V28 is mostly 0. I don't see any one to one correlations, so I do not need to remove any columns.

```
In [36]: # Get all the columns from the dataframe
columns = datasetnew.columns.tolist()

# Filter the columns to remove data we do not want
columns = [c for c in columns if c not in ["Class"]]

# Store the variable we'll be predicting on
target = "Class"

X = datasetnew[columns]
Y = datasetnew[target]

# Print shapes
print(X.shape)
print(Y.shape)

(56961, 30)
(56961,)
```

- 9) I get all data variables in columns, and I remove the ones I don't want. I am removing because it would be too easy if I simply told my neural network what was fraudulent! I specifically don't want the labels with 'valid' and 'fraud'. I store it on the variable I'll be predicting on, target= "Class"

The output reads 30 columns now, as we got rid of 'Class' in X.

Y is a one dimensional array which has all class labels for all 28,481 samples.



```
In [37]: from sklearn.metrics import classification_report, accuracy_score
from sklearn.ensemble import IsolationForest
from sklearn.neighbors import LocalOutlierFactor

# define random states
rand_state = 1

# define outlier detection tools to be compared
classifiers = {
    "Isolation Forest": IsolationForest(max_samples=len(X),
                                         contamination=outlier_fraction,
                                         random_state=rand_state),
    "Local Outlier Factor": LocalOutlierFactor(
        n_neighbors=20,
        contamination=percent_fraud)}

#class sklearn.neighbors.LocalOutlierFactor(n_neighbors=20, algorithm='auto',
#leaf_size=30, metric='minkowski', p=2, metric_params=None, contamination='legacy', novelty=False, n_jobs=None)
#referenced LOF Method and code from:
#https://github.com/scikit-learn/scikit-learn/blob/55bf5d9/sklearn/neighbors/lof.py#L19

#class sklearn.ensemble.IsolationForest(n_estimators=100, max_samples='auto',
#contamination='legacy', max_features=1.0, bootstrap=False, n_jobs=None, behavior='old', random_state=None, verbose=0)[source]
#referenced Isolation Forest Method and code from:
#https://github.com/scikit-learn/scikit-learn/blob/55bf5d9/sklearn/ensemble/isolation_forest.py#L27
```

- 10) From sklearn, I import the Isolation Forest Method and the LOF. I chose the number of neighbors for the LOF to be twenty since this is the value that works the best. However it is possible to adjust and increase/decrease the numbers for a slightly different result. I would also like to add that one can use SVMs to do this as well, but would take a much longer time. The citations are included where I was able to understand these algorithms and then implement them directly into my program.

```

In [38]: # Fit the model
plt.figure(figsize=(9, 7))
n_outliers = len(fraudulent_cases)

for i, (clf_name, clf) in enumerate(classifiers.items()):

    # fit the data and tag outliers
    if clf_name == "Local Outlier Factor":
        y_pred = clf.fit_predict(X)
        scores_pred = clf.negative_outlier_factor_
    else:
        clf.fit(X)
        scores_pred = clf.decision_function(X)
        y_pred = clf.predict(X)

    # Reshape the prediction values to 0 for valid, 1 for fraud.
    y_pred[y_pred == 1] = 0
    y_pred[y_pred == -1] = 1

    n_errors = (y_pred != Y).sum()

    # Run classification metrics
    print('{:}: {}'.format(clf_name, n_errors))
    print(accuracy_score(Y, y_pred))
    print(classification_report(Y, y_pred))

```

11) I start with a for loop through two different classifiers. Enumerate() cycles through list of classifiers. The if statement is for the LOF and the else statement contains Isolation Forest Method. And lastly, I want the results to compare to my class labels, 0 for valid and 1 for invalid.

y\_pred[y\_pred == 1] = 0 : pulls up the y prediction and indexes them, based on y prediction and reassigns to zero.

y\_pred[y\_pred == -1] = 1 : pulls up the y prediction and indexes them, based on y prediction and reassigns to one.

n\_errors calculates the number of errors.

```

Isolation Forest: 136
0.9976124014676708
      precision    recall  f1-score   support

     0       1.00      1.00      1.00     56874
     1       0.25      0.29      0.27         87

 avg / total       1.00      1.00      1.00     56961

Local Outlier Factor: 173
0.9969628342199048
      precision    recall  f1-score   support

     0       1.00      1.00      1.00     56874
     1       0.01      0.01      0.01         87

 avg / total       1.00      1.00      1.00     56961

```

<Figure size 648x504 with 0 Axes>

```
]: # We can see Isolation Forest Method yeilded better results
```

12) So with a sample size of 20% of the full dataset- we can see the results. Both programs were extremely accurate, with Isolation Forest Method holding a slightly higher accuracy of 99.76%.

For Isolation Forest Method flagged 136 errors, the precision of valid transactions was 100% precise and for invalid transactions was 25% precise. It flagged 100% false positives and 29% false negatives.

For Local Outlier Factor flagged 173 errors, the precision of valid transactions was 100% precise and for invalid transactions was 1% precise. It flagged 100% false positives and 1% false negatives.

Note: I increased the sample size to 50% and yielded mostly similar results

```

Isolation Forest: 317
0.9977739389342996
      precision    recall  f1-score   support

     0       1.00      1.00      1.00    142177
     1       0.30      0.30      0.30         227

 avg / total       1.00      1.00      1.00    142404

Local Outlier Factor: 439
0.9969172214263644
      precision    recall  f1-score   support

     0       1.00      1.00      1.00    142177
     1       0.04      0.04      0.04         227

 avg / total       1.00      1.00      1.00    142404

```