Credit Card Fraud Detection

Shahenda Elsayed	3695
Dina Nabil	3677
Lina Gamal	3287
John Maged	3690

Overview

It is important that credit card companies are able to recognize fraudulent credit card transactions so that customers are not charged for items that they did not purchase.

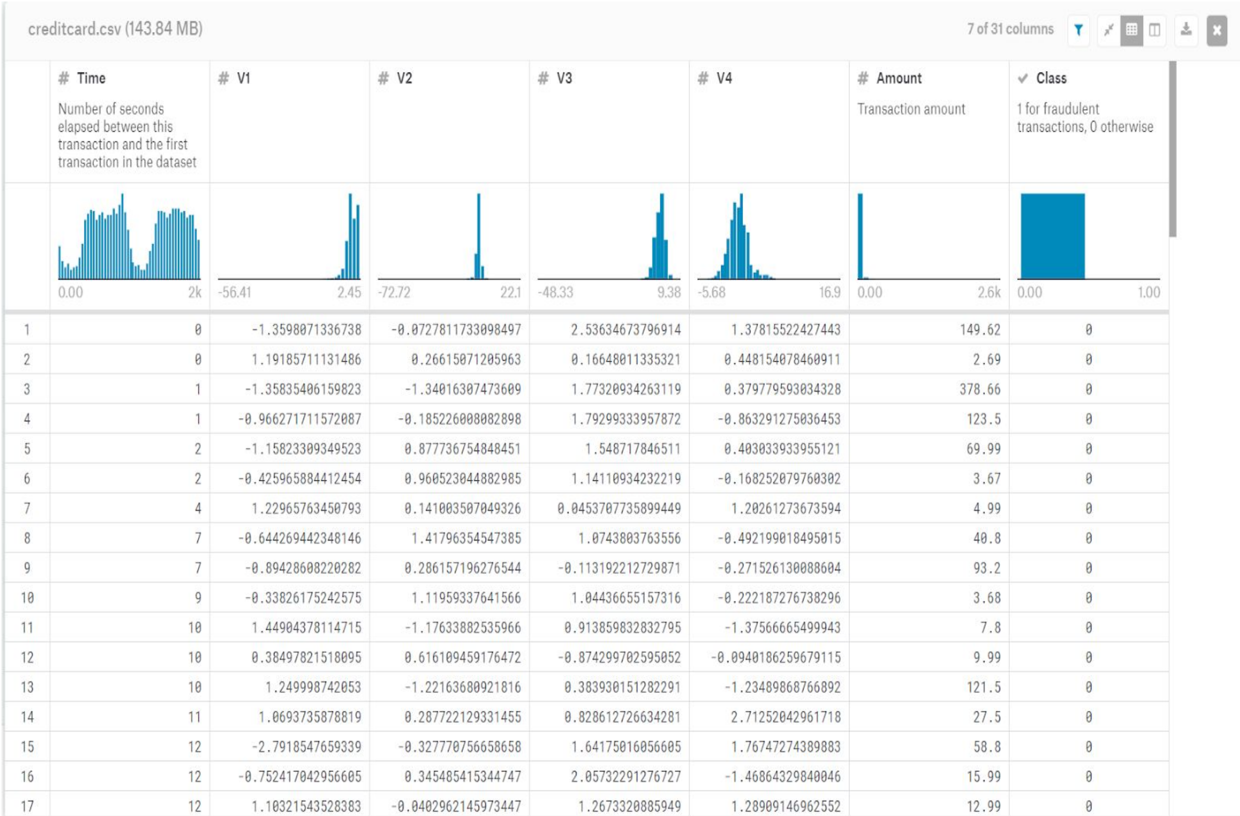
Problem Description

Using the dataset of transactions that occurred previously , Credit card transactions are labeled as fraudulent or genuine.

Dataset

- The datasets contains transactions made by credit cards in September 2013 by European cardholders. This dataset presents transactions that occurred in two days.
- We have 492 frauds out of 284,807 transactions. **The dataset is highly imbalanced**, the positive class (frauds) account for 0.172% of all transactions.
- The dataset contains only numeric input variables .Unfortunately, due to confidentiality issues, the original features cannot be provided.

➔ Features V1, V2, ... V28 are the principal components .
➔ Feature 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset.
➔ Feature 'Amount' is the transaction Amount.
➔ Feature 'Class' is the response variable and it takes value 1 in case of fraud and 0 otherwise.


creditcard.csv (143.84 MB)

7 of 31 columns

	Time	V1	V2	V3	V4	Amount	Class
	Number of seconds elapsed between this transaction and the first transaction in the dataset					Transaction amount	1 for fraudulent transactions, 0 otherwise
	0.00 – 2k	-56.41 – 2.45	-72.72 – 22.1	-48.33 – 9.38	-5.68 – 16.9	0.00 – 2.6k	0.00 – 1.00
1	0	-1.3598071336738	-0.0727811733098497	2.53634673796914	1.37815522427443	149.62	0
2	0	1.19185711131486	0.26615071205963	0.16648011335321	0.448154078460911	2.69	0
3	1	-1.35835406159823	-1.34016307473609	1.77320934263119	0.379779593034328	378.66	0
4	1	-0.966271711572087	-0.185226008082898	1.79299333957872	-0.863291275036453	123.5	0
5	2	-1.15823309349523	0.877736754848451	1.548717846511	0.403033933955121	69.99	0
6	2	-0.425965884412454	0.960523044882985	1.14110934232219	-0.168252079760302	3.67	0
7	4	1.22965763450793	0.141003507049326	0.0453707735899449	1.20261273673594	4.99	0
8	7	-0.644269442348146	1.41796354547385	1.0743803763556	-0.492199018495015	40.8	0
9	7	-0.89428608220282	0.286157196276544	-0.113192212729871	-0.271526130088604	93.2	0
10	9	-0.33826175242575	1.11959337641566	1.04436655157316	-0.222187276738296	3.68	0
11	10	1.44904378114715	-1.17633882535966	0.913859832832795	-1.37566665499943	7.8	0
12	10	0.38497821518095	0.616109459176472	-0.874299702595052	-0.0940186259679115	9.99	0
13	10	1.249998742053	-1.22163680921816	0.383930151282291	-1.23489868766892	121.5	0
14	11	1.0693735878819	0.287722129331455	0.828612726634281	2.71252042961718	27.5	0
15	12	-2.7918547659339	-0.327770756658658	1.64175016056605	1.76747274389883	58.8	0
16	12	-0.752417042956605	0.345485415344747	2.05732291276727	-1.46864329840046	15.99	0
17	12	1.10321543528383	-0.0402962145973447	1.2673320885949	1.28909146962552	12.99	0

Preprocessing

- ➔ Standardize “amount” column.
- ➔ Inspect features and identify features that isn’t much of help.
- ➔ Splitting the data into train and test sets.
- ➔ Balance the training set.

Standardizing the amount

We need to standardize the amount feature because it dominates other features in magnitude so the model will hardly pick the contribution of smaller scale features.

A common method for that is z-score standardization.

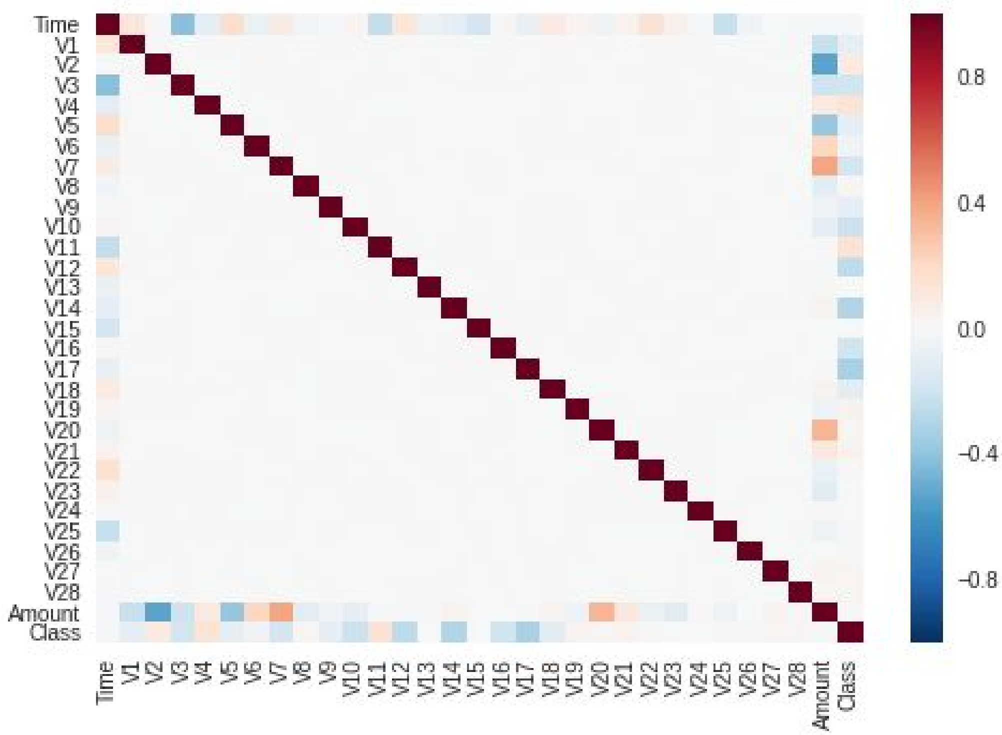
After processing “amount” column will have a mean of 0 and standard deviation of 1.

Normalizing the amount column (To reduce its weight while learning) & splitting the data

```
[ ] original_data['Normalized_Amount'] = StandardScaler().fit_transform(original_data['Amount'].values.reshape(-1, 1))
    original_data = original_data.drop(['Time', 'Amount','V28','V27','V26','V25','V24','V23','V22','V20','V15','V13','V8'], axis=1)
    print(original_data.head())
```

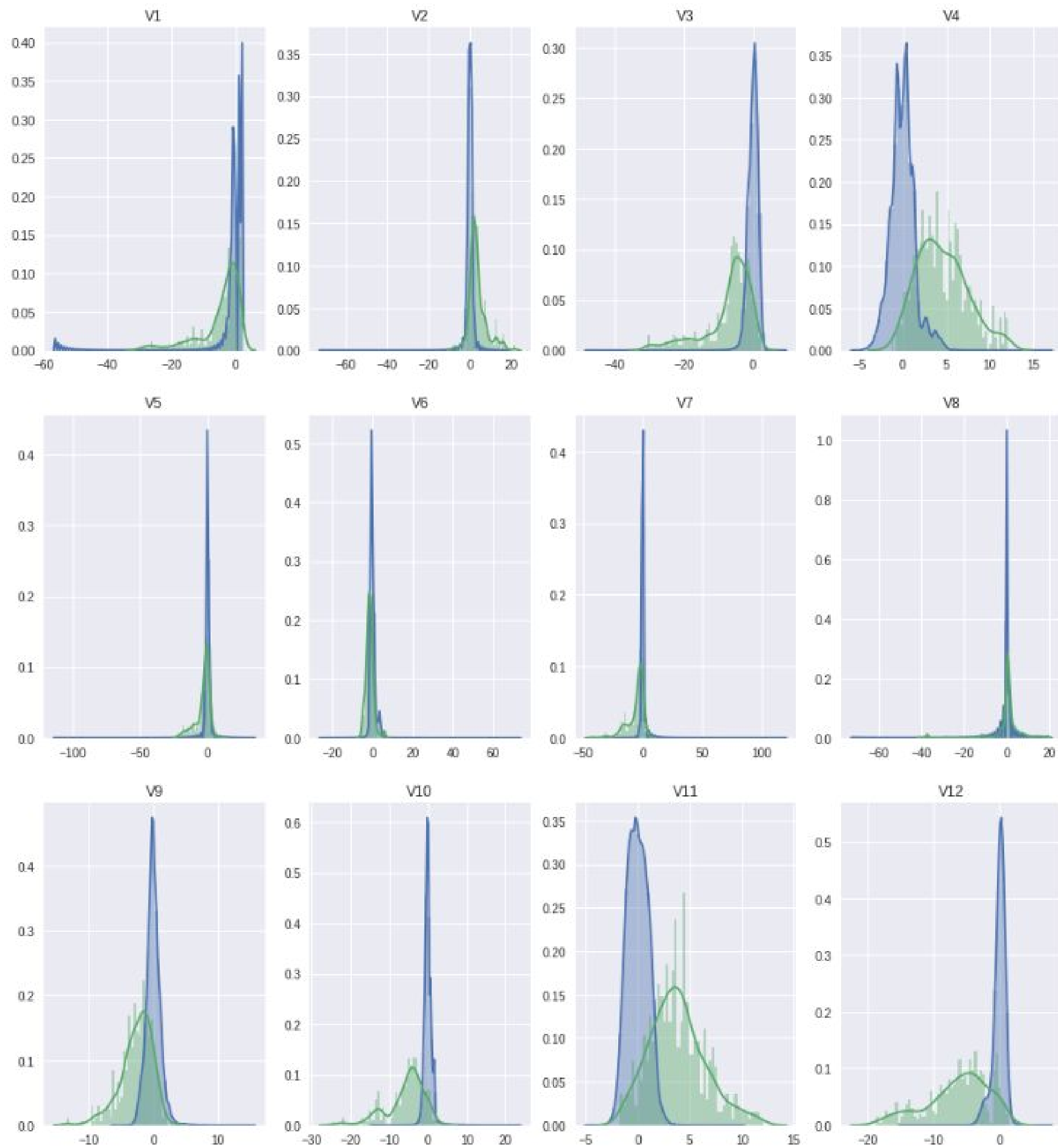
```
         V1        V2        V3        V4        V5        V6        V7  \
0 -1.359807 -0.072781  2.536347  1.378155 -0.338321  0.462388  0.239599
1  1.191857  0.266151  0.166480  0.448154  0.060018 -0.082361 -0.078803
2 -1.358354 -1.340163  1.773209  0.379780 -0.503198  1.800499  0.791461
3 -0.966272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

         V9       V10       V11       V12       V14       V16       V17  \
0  0.363787  0.090794 -0.551600 -0.617801 -0.311169 -0.470401  0.207971
1 -0.255425 -0.166974  1.612727  1.065235 -0.143772  0.463917 -0.114805
2 -1.514654  0.207643  0.624501  0.066084 -0.165946 -2.890083  1.109969
3 -1.387024 -0.054952 -0.226487  0.178228 -0.287924 -1.059647 -0.684093
4  0.817739  0.753074 -0.822843  0.538196 -1.119670 -0.451449 -0.237033

        V18       V19       V21  Class  Normalized_Amount
0  0.025791  0.403993 -0.018307      0           0.244964
1 -0.183361 -0.145783 -0.225775      0          -0.342475
2 -0.121359 -2.261857  0.247998      0           1.160686
3  1.965775 -1.232622 -0.108300      0           0.140534
4 -0.038195  0.803487 -0.009431      0          -0.073403
```

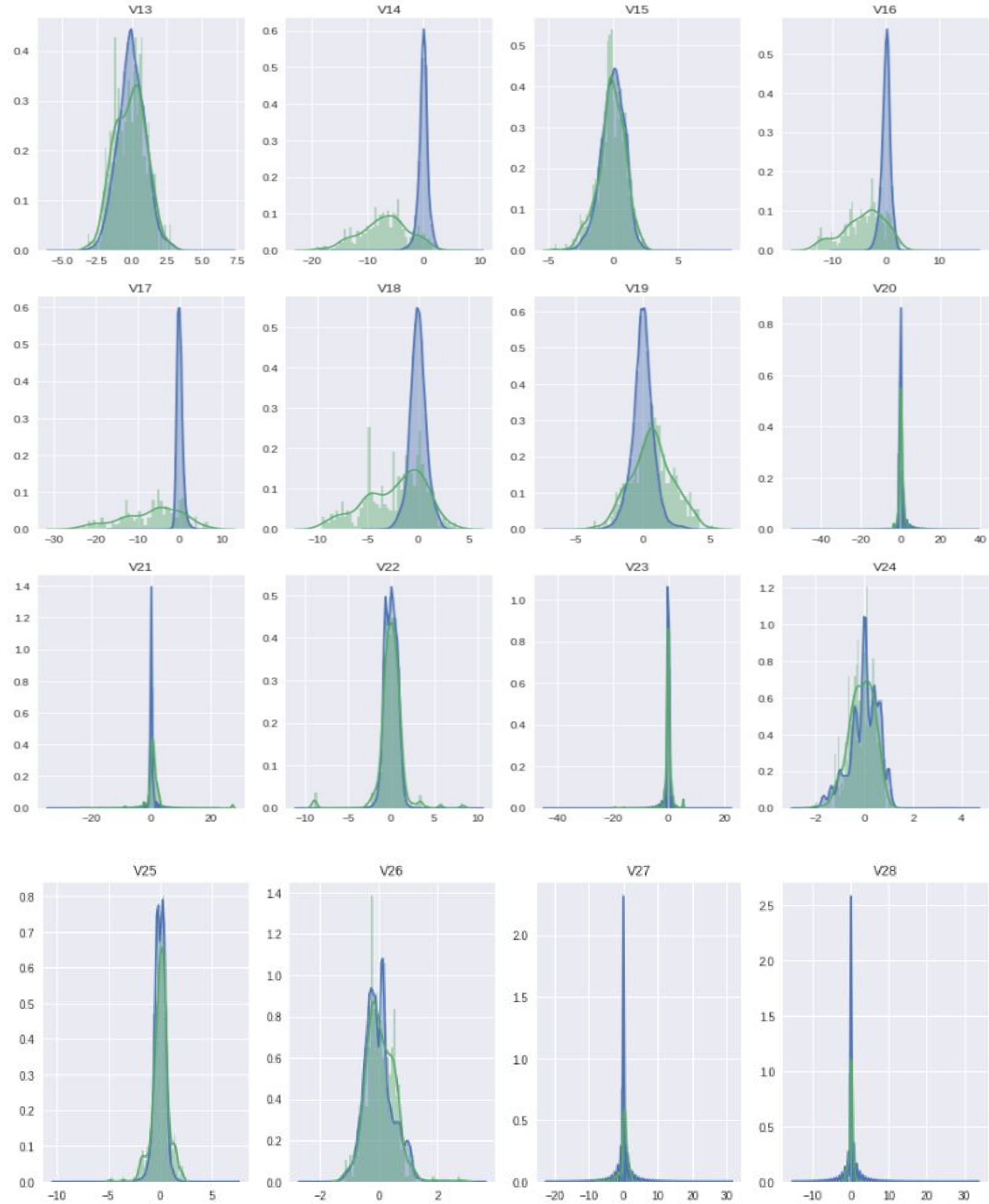
Dataset features correlation



The features are clearly linearly uncorrelated to each other.

Histogram of features:





From the shown histogram, a lot of features have very similar distributions between the two classes which make it very different to differentiate the classes. So we can drop those features from our dataset.

Determining Important Features:

Determine important features

```
[ ] X = original_data.iloc[:,1:29]
    y = original_data.iloc[:,30]
    rf = RandomForestClassifier()
    rf.fit(X, y)

    feature_importance = pd.DataFrame(rf.feature_importances_, index = X.columns, columns=['Importance']).sort_values('Importance', ascending=False)
```

```
[ ]   print(feature_importance)
```

```
       Importance
V12      0.161247
V17      0.134247
V11      0.104594
V14      0.093656
V10      0.079523
V16      0.061917
V4       0.045616
V9       0.045401
V6       0.024564
V7       0.021538
V26      0.021061
V21      0.017052
V1       0.016635
V27      0.015870
V3       0.014339
V22      0.014176
V19      0.013550
V5       0.013064
V2       0.013051
V15      0.011388
V24      0.011227
V28      0.010704
V25      0.010177
V18      0.010035
V8       0.009108
V13      0.008888
V23      0.008827
V20      0.008547
```

Solving Imbalanced Dataset Problem

Re-sampling techniques are divided in two categories:

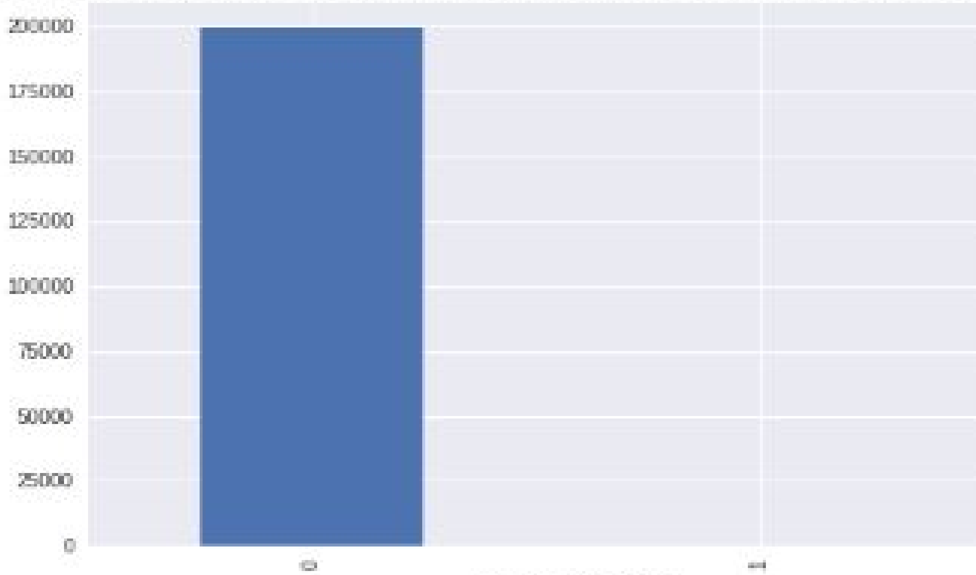
- Under-sampling the majority class.
- Over-sampling the minority class.
- Combining over- and under-sampling.

+ Balancing the data first using SMOTE (Synthetic Minority Over-sampling Technique) and then apply Tomek links (Undersampling).

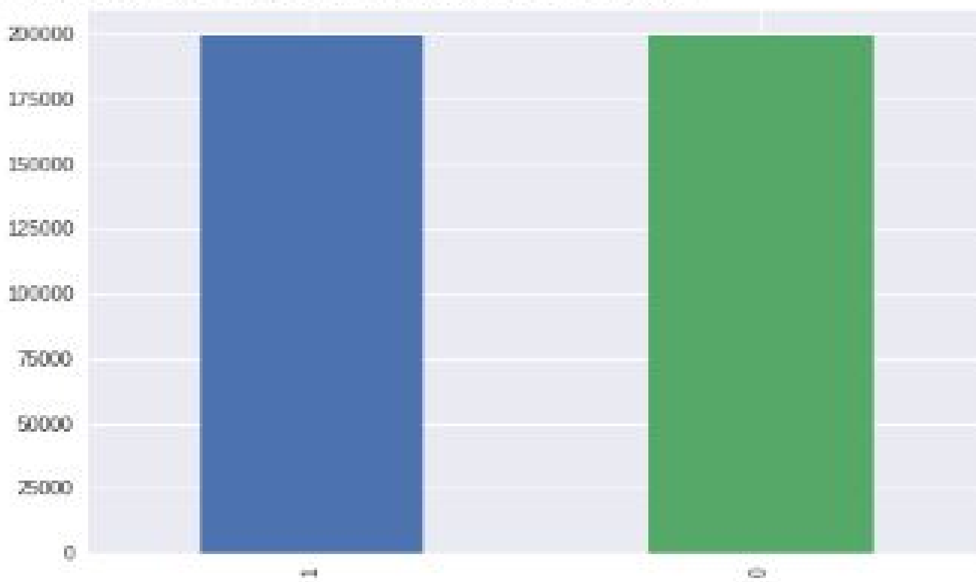
Method to balance the data using SMOTE then Tomek Links

```
[ ]  def balance_data(data):
         """Given an imbalanced dataset this method balances the data first using SMOTE (Synthetic Minority Over-sampling
         Technique) and then apply Tomek links (Undersampling)
         :param data: Original imbalanced dataset
         :return:
                 x_new: Array containing the new resampled data.
                 y_new: Array containing the corresponding labels for x_new
         """
         smote = imbcom.SMOTETomek()
         x = data.iloc[:, data.columns != 'Class']
         y = data.iloc[:, data.columns == 'Class']

         # Fit and resample the data directly
         x_new, y_new = smote.fit_sample(x, y.values.ravel())

         print(f'==================={str.center("ORIGINAL DATA", STRWIDTH)}===================')
         print(f'Total samples in the original dataset: {x.shape[0]}')
         original_fraud = len(data[data.Class == 1])
         print(f'Fraud in the original dataset: {original_fraud}')
         print(f'Percentage of fraud in original dataset {100*(original_fraud/x.shape[0])}%')
         pd.value_counts(y['Class']).plot(kind="bar")
         plt.show()

         print(f'==================={str.center("SAMPLED DATA", STRWIDTH)}===================')
         print(f'Total samples in the sampled data: {x_new.shape[0]}')
         sampled_fraud = len(y_new[y_new == 1])
         print(f'Fraud in sampled data: {sampled_fraud}')
         print(f'Percentage of fraud in sample data: {100*(sampled_fraud/x_new.shape[0])}%')
         y_new = pd.DataFrame(y_new, columns=['Class'])
         x_new = pd.DataFrame(x_new, columns=x.columns)

         pd.value_counts(y_new['Class']).plot(kind="bar")
         plt.show()

         return x_new, y_new
```

```
====================        ORIGINAL DATA        ====================
Total samples in the original dataset: 199365
Fraud in the original dataset: 344
Percentage of fraud in original dataset 0.17254783939006343%
```

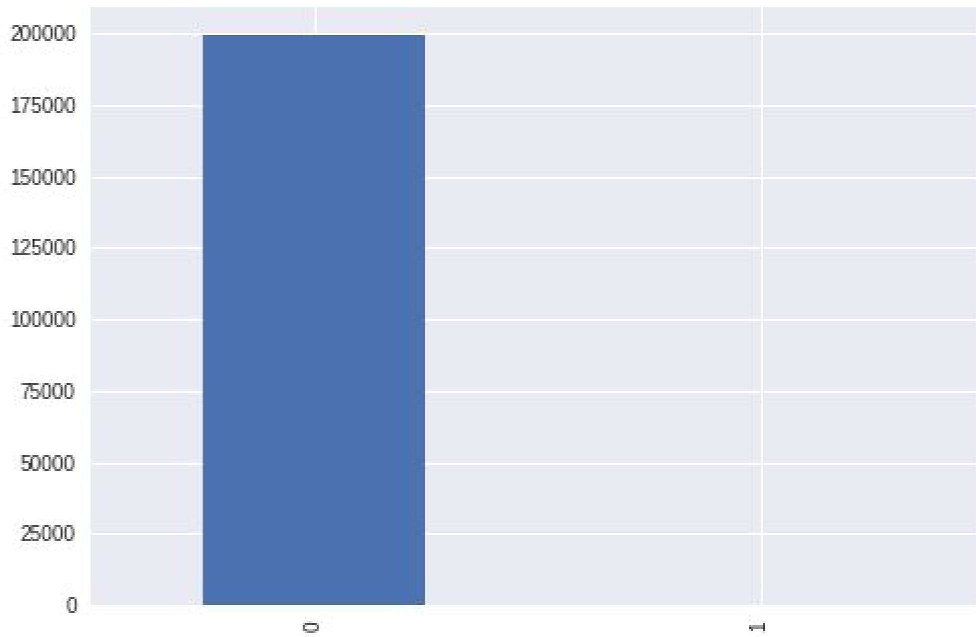



```
====================        SAMPLED DATA        ====================
Total samples in the sampled data: 398042
Fraud in sampled data: 199021
Percentage of fraud in sample data: 50.0%
```

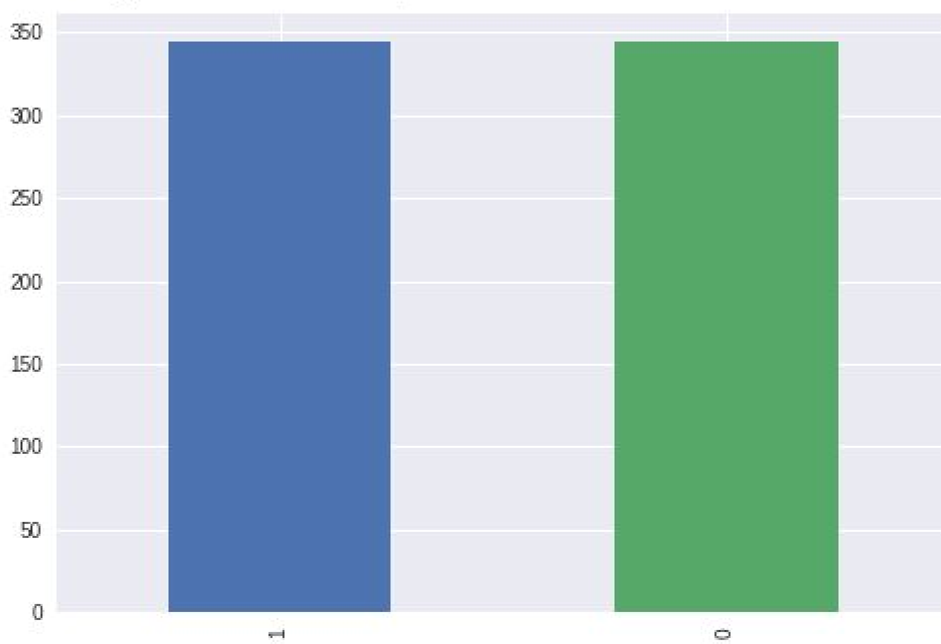


+ Balancing the data by undersampling:

```
===================          ORIGINAL DATA          ===================
Total samples in the original dataset: 199365
Fraud in the original dataset: 344
Percentage of fraud in original dataset 0.17254783939006343%
```




```
===================          SAMPLED DATA          ===================
Total samples in the sampled data: 688
Fraud in sampled data: 344
Percentage of fraud in sample data: 50.0%
```



Splitting data into train and test sets

```
# Splitting data into test and train

#SMOTETomek

data = original_data.copy()

test_data = data[data.Class == 0].sample(frac=0.3)
test_data = pd.concat([test_data, data[data.Class == 1].sample(frac=0.3)])

# Untouched test data.

x_test = test_data.iloc[:, test_data.columns != 'Class']
y_test = test_data.iloc[:, test_data.columns == 'Class']

print("Testing data:")
print(f"# Class 0 (Legit): {len(test_data[test_data.Class == 0])}")
print(f"# Class 1 (Fraud): {len(test_data[test_data.Class == 1])}")

train_data = data.drop(test_data.index)

# SMOTETomek
sampled_train_data = balance_data(train_data.copy())
# Undersampling
underSampled_train_data = underSample(train_data.copy())

x_train_undersampled = underSampled_train_data[0]
y_train_undersampled = underSampled_train_data[1]

x_train = sampled_train_data[0]
y_train = sampled_train_data[1]
```

Apply the Algorithms

I. using Classification

Logistic Regression

Random Forest

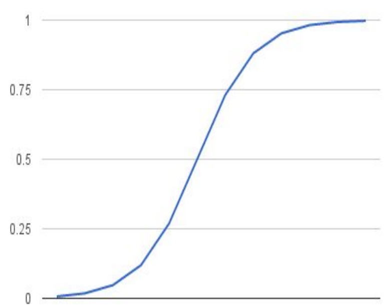
II. using Outlier Detection

Isolation Forest

Local Outlier Factor

Logistic Regression

- Logistic Regression is a type of classification algorithm named for the function used at the core of the method, the logistic function, also called the sigmoid function :



1 / (1 + e^-value)

- The output is a probability that the given input point belongs to a certain class.
- Assume having only two classes:
- >=Threshold , predict“+” class and < Threshold , predict“-” class.

A method to apply logistic regression algorithm on the train data, predict on train data and calculate accuracy

```
[ ]  def logistic_regression(X_train, X_test, y_train, y_test):

         penalty = ['l1', 'l2']
         C = [0.001, 0.01, 0.1, 1, 10, 100, 1000]
         parameters = dict(C=C, penalty=penalty)
         clf = GridSearchCV(LogisticRegression(),parameters)
         best_model = clf.fit(X_train, y_train.values.ravel())
         #print('Best Penalty:', best_model.best_estimator_.get_params()['penalty'])
         #print('Best C:', best_model.best_estimator_.get_params()['C'])
         prediction =best_model.predict(X_test)
         print("classification_report :  \n ", classification_report(y_test, prediction))
         accuracy = accuracy_score(y_test.values.ravel(), prediction)
         print(f'accuracy: {accuracy}')
         cm = pd.DataFrame(confusion_matrix(y_test, prediction))
         sb.heatmap(cm, annot=True)
```

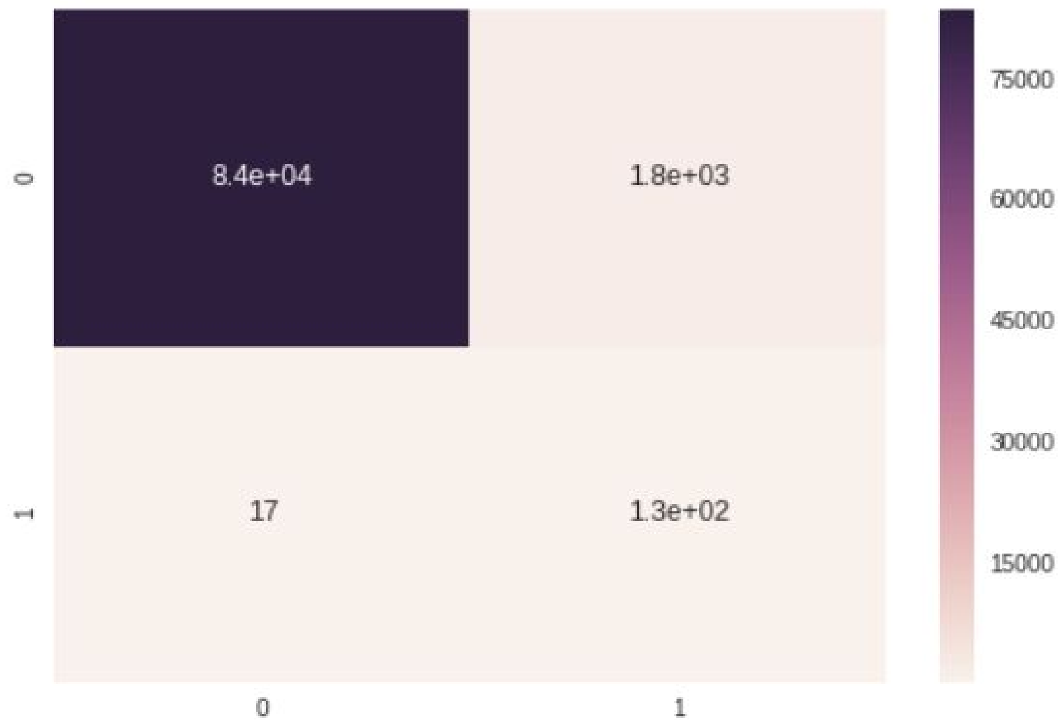
Using the function “ GridSearchCV “ for choosing the parameters [penalty and C] ,

the best penalty = l2 and the best C = 10.

Applying the function “LogisticRegression” using these parameters then evaluating the model , the results are as shown :

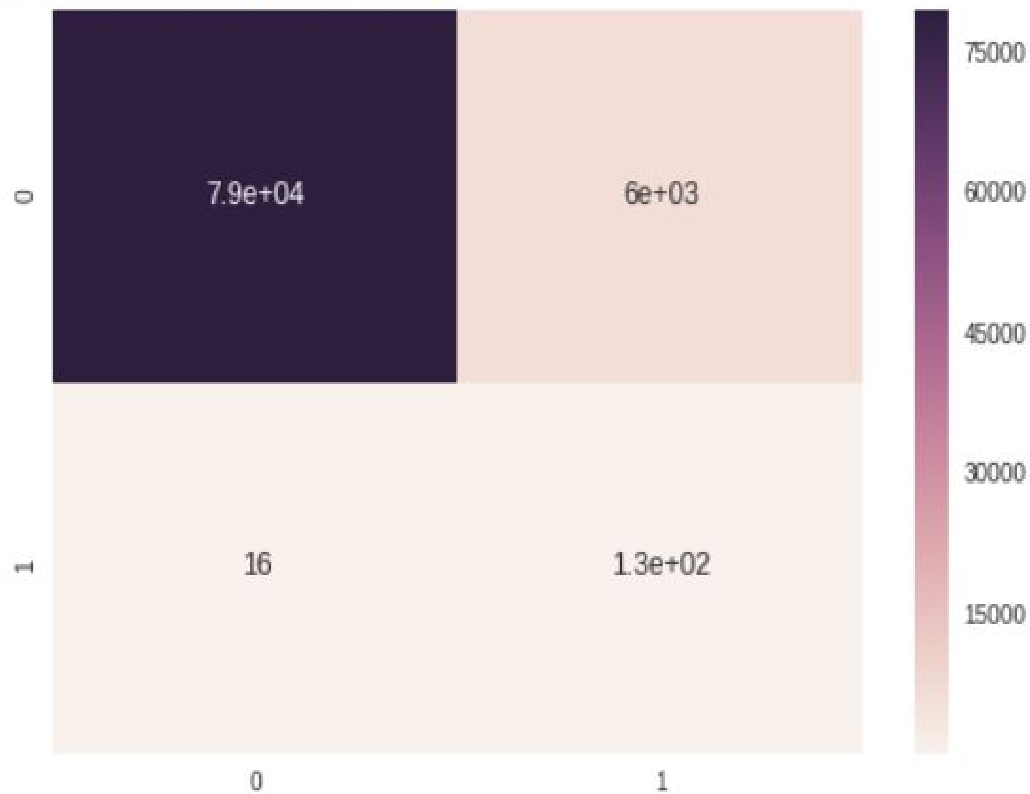
Results of over sampling and under sampling the data using the function SMOTETomek() :

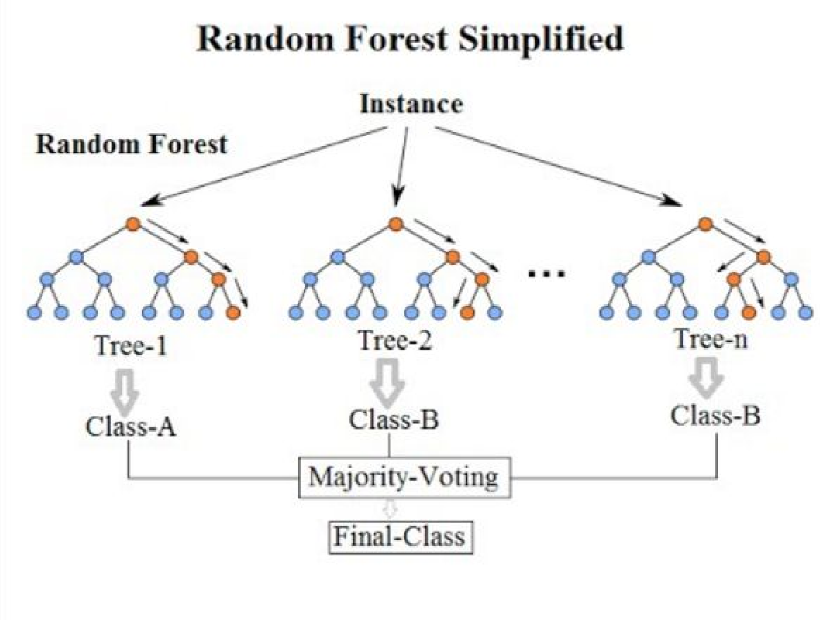
```
classification_report :
              precision    recall  f1-score   support

          0       1.00      0.98      0.99     85294
          1       0.07      0.89      0.13       148

avg / total       1.00      0.98      0.99     85442

accuracy: 0.9791320427892606
```



Results of only under sampling the data using the function RandomUnderSampler() :

```
classification_report :
             precision    recall  f1-score   support

          0       1.00      0.93      0.96     85294
          1       0.02      0.89      0.04       148

avg / total       1.00      0.93      0.96     85442

accuracy: 0.9300812246904333
```





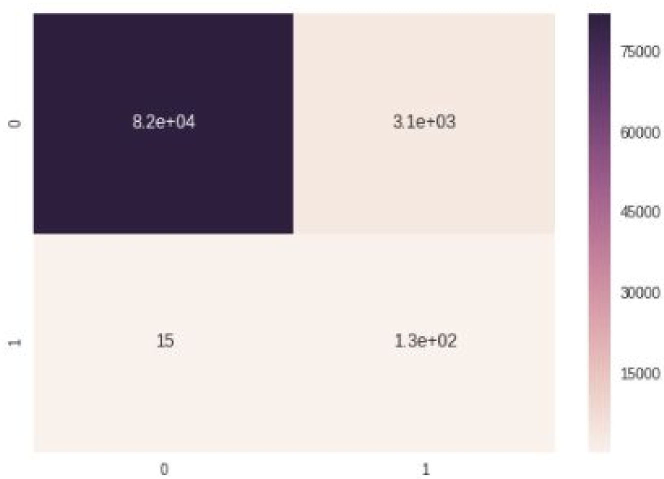
Random Forest

- Method for classification that operate by constructing multiple of decision trees.
- To classify a new object, each tree gives a classification, and we say the tree "votes" for that class. The forest chooses the classification having the most votes.

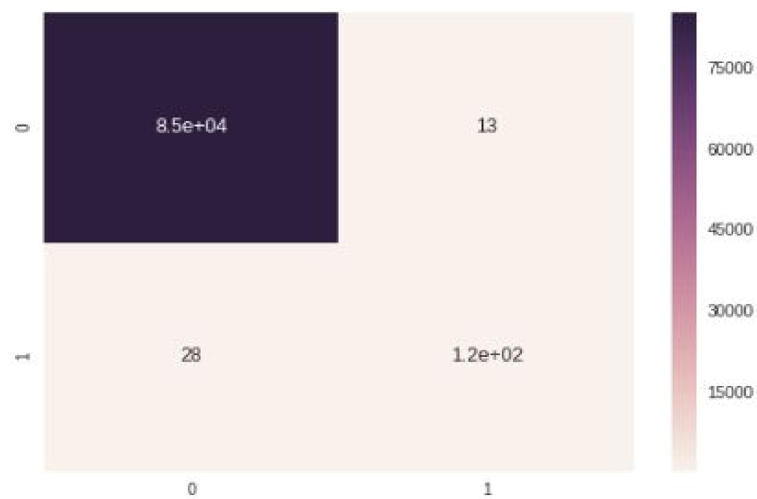
```
from sklearn.metrics import precision_score, recall_score, f1_score


def random_forest(X_train, X_test, y_train, y_test, n):

    rf = RandomForestClassifier(n_estimators=n, random_state=0, n_jobs=-1)
    rf.fit(X_train, y_train.values.ravel())
    prediction = rf.predict(X_test)
    accuracy = accuracy_score(y_test.values.ravel(), prediction)

    print(f'Mean accuracy score: {accuracy}')
    print("Precision: %1.3f" % precision_score(y_test, prediction))
    print("Recall: %1.3f" % recall_score(y_test, prediction))
    print("Classification Report: ")
    print(classification_report(y_test, prediction, target_names=['Class 0', 'Class 1']))
    # This should match the f1 score for class 1 in the classification report.
    print("F1: %1.3f\n" % f1_score(y_test, prediction))

    cm = pd.DataFrame(confusion_matrix(y_test, prediction))
    sb.heatmap(cm, annot=True)
    plt.show()
```

- ## Using 50 Trees:

```
Using Undersampling:
Mean accuracy score: 0.9630041431614429
Precision: 0.041
Recall: 0.899
Classification Report:
             precision    recall  f1-score   support

    Class 0       1.00      0.96      0.98     85294
    Class 1       0.04      0.90      0.08       148

avg / total       1.00      0.96      0.98     85442

F1: 0.078
```

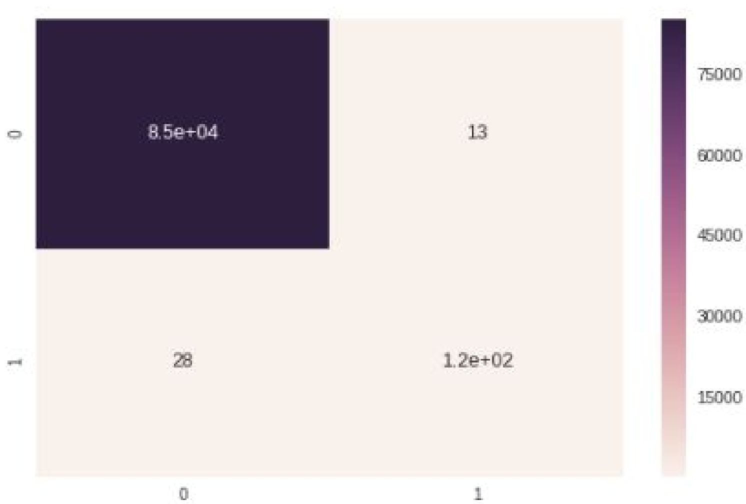



```
Using SMOTETomek sampling:
Mean accuracy score: 0.999520142318766
Precision: 0.902
Recall: 0.811
Classification Report:
             precision    recall  f1-score   support

    Class 0       1.00      1.00      1.00     85294
    Class 1       0.90      0.81      0.85       148

avg / total       1.00      1.00      1.00     85442

F1: 0.854
```

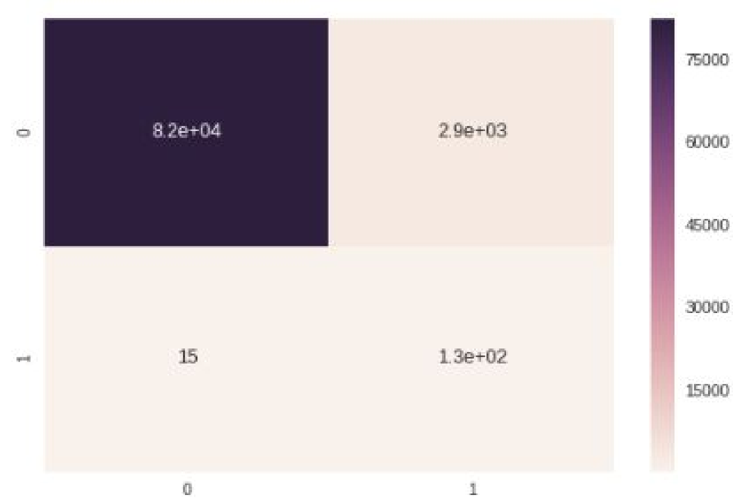


- ## Using 100 Trees:

```
Using SMOTETomek sampling:
Mean accuracy score: 0.999520142318766
Precision: 0.902
Recall: 0.811
Classification Report:
             precision    recall  f1-score   support

    Class 0       1.00      1.00      1.00     85294
    Class 1       0.90      0.81      0.85       148

avg / total       1.00      1.00      1.00     85442

F1: 0.854
```

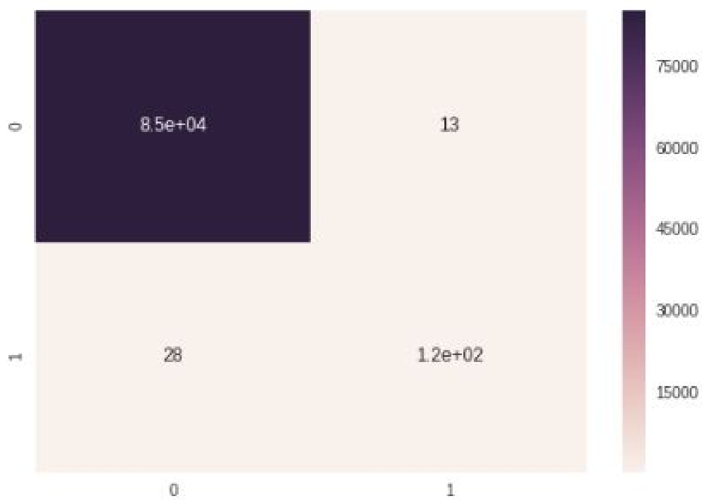



```
Using Undersampling:
Mean accuracy score: 0.9657662507900096
Precision: 0.044
Recall: 0.899
Classification Report:
             precision    recall  f1-score   support

    Class 0       1.00      0.97      0.98     85294
    Class 1       0.04      0.90      0.08       148

avg / total       1.00      0.97      0.98     85442

F1: 0.083
```

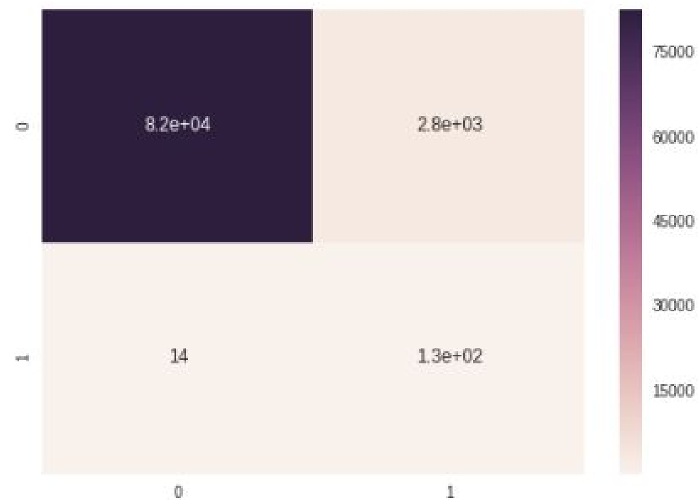


- ## **Using 150 Trees:**

```
Using SMOTETomek sampling:
Mean accuracy score: 0.999520142318766
Precision: 0.902
Recall: 0.811
Classification Report:
             precision    recall  f1-score   support

    Class 0       1.00      1.00      1.00     85294
    Class 1       0.90      0.81      0.85       148

avg / total       1.00      1.00      1.00     85442

F1: 0.854
```




```
Using Undersampling:
Mean accuracy score: 0.966901523840734
Precision: 0.045
Recall: 0.905
Classification Report:
             precision    recall  f1-score   support

    Class 0       1.00      0.97      0.98     85294
    Class 1       0.05      0.91      0.09       148

avg / total       1.00      0.97      0.98     85442

F1: 0.087
```

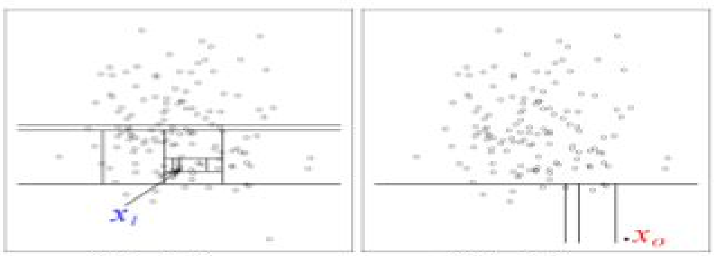


Isolation Forest

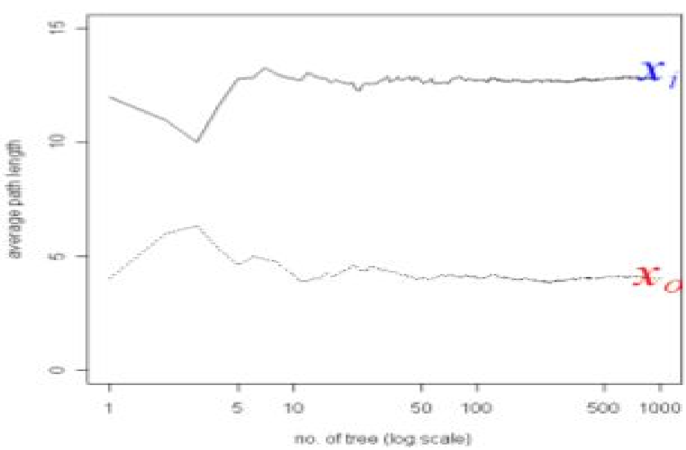
The algorithm is based on the fact that anomalies are data points that are few and different. As a result of these properties, anomalies are susceptible to a mechanism called isolation.

The Isolation Forest **Unsupervised anomaly detection** cause the observations build a model unlabeled .

The Isolation Forest algorithm isolates observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature. The logic argument goes: isolating anomaly observations is easier because only a few conditions are needed to separate those cases from the normal observations. On the other hand, isolating normal observations require more conditions. Therefore, an anomaly score can be calculated as the number of conditions required to separate a given observation.



(a) Isolating x_i (b) Isolating x_o



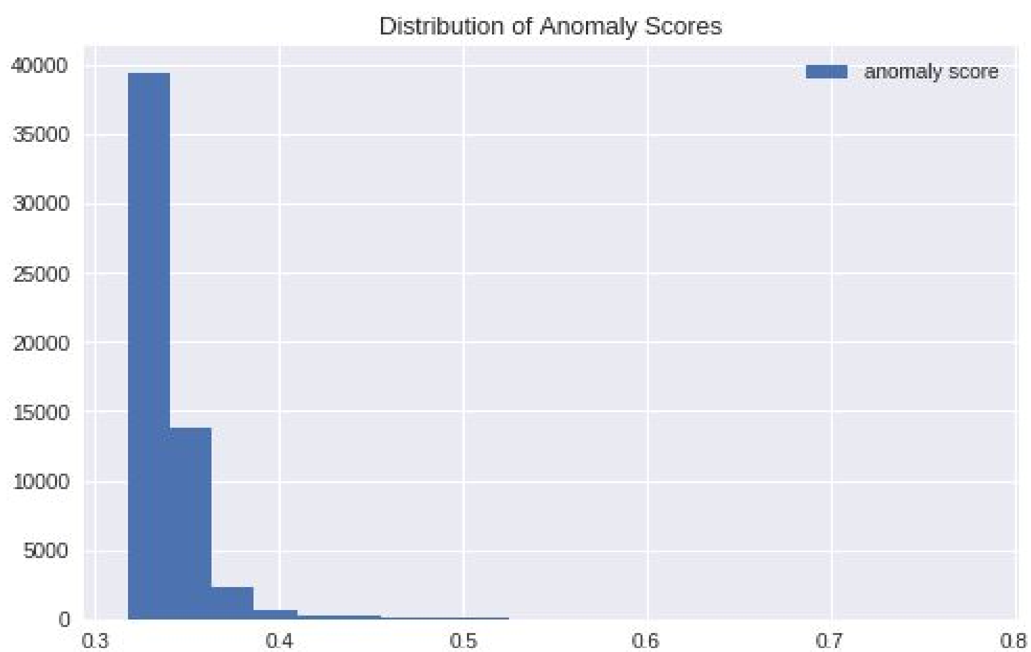
(c) Average path lengths converge

A method to apply Isolation Forest algorithm on the train data, predict on train data and calculate accuracy

```
def isolation_forest(X_train, X_test, Y_test, n):

  isf = IsolationForest(n_estimators=100, max_samples=len(X_train), contamination=n, random_state=0)
  isf.fit(X_train)
  prediction = isf.predict(X_test)   # For each observations, tells inlier or not (+1 or -1)
  prediction[prediction == 1] = 0
  prediction[prediction == -1] = 1

  anomaly_score = isf.decision_function(X_test)
  anomaly_score = 0.5 - anomaly_score
  #print("The predicted anomaly score: ", anomaly_score)

  plt.hist(anomaly_score, bins=40, label="anomaly score")
  plt.title("Distribution of Anomaly Scores")
  plt.legend()
  plt.show()
```

Calculate the outcome of **decision_function** to represent the anomaly score . The lower, the more abnormal .

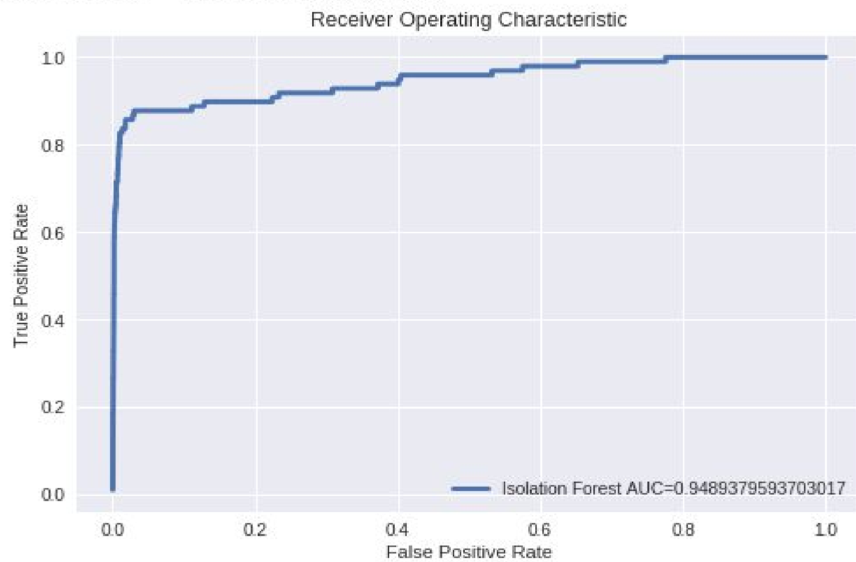
the histogram of the distribution of anomaly scores. from the chart that most data points are centered between 0.3 and 0.4, and only a very small fraction of data points have anomaly score over 0.5. This is what we expect to see because outliers only account for a very small proportion of total dataset.



Plot the ROC curve

```
auc = roc_auc_score(Y_test, anomaly_score)      # area under ROC curve
print("\nAUC score: ", auc)

fpr, tpr, thresholds = roc_curve(Y_test, anomaly_score, pos_label=1)
auc = np.trapz(tpr, fpr)
plt.plot(fpr, tpr, label="Isolation Forest AUC=" + str(auc), lw=3, color='C0')
plt.title("Receiver Operating Characteristic")
plt.legend()
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```

```
AUC score:  0.9489379593703017
```



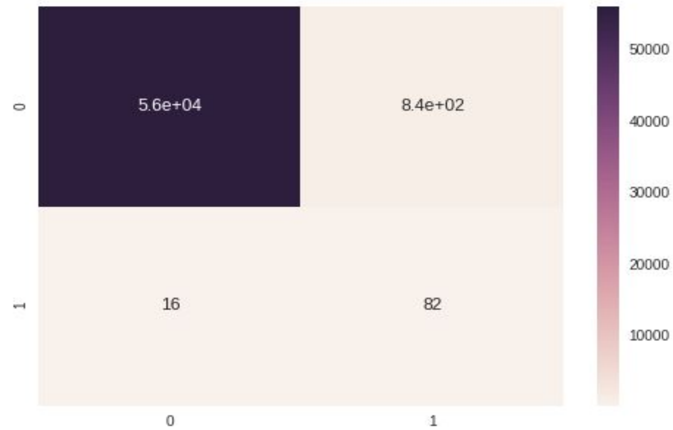

```
print("number of normal in test data: ", len(test_data[test_data.Class == 0]))
print("number of fraud in test data: ", len(test_data[test_data.Class == 1]))

cm = pd.DataFrame(confusion_matrix(Y_test, prediction))
#print(cm)
sb.heatmap(cm, annot=True)
plt.show()

accuracy = accuracy_score(Y_test, prediction)
print('accuracy score: ', accuracy)

precision, recall, fscore, support = score(Y_test, prediction, average='weighted')
print('precision: ', precision)
print('recall: ', recall)
print('f1-score: ', fscore)
```

```
[48] contamination = [0.015, 0.05, 0.1]
     for n in contamination:
       isolation_forest(x_train, x_test, y_test, n)
```

Tuning tha contamination (the proportion of outliers in the data set) and n_estimators (number of trees) .

When contamination = 0.015 and n_estimators = 500

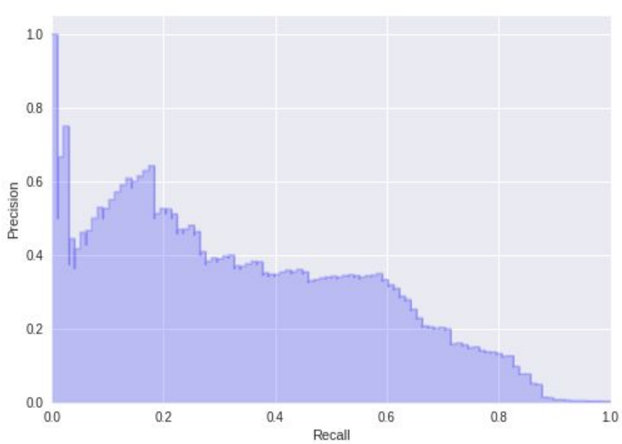
```
number of normal in test data:  56863
number of fraud in test data:  98
```




```
accuracy score:  0.9850599533013816
precision:  0.9981483741944738
recall:  0.9850599533013816
f1-score:  0.991033303645214
```

Plot precision_recall_curve

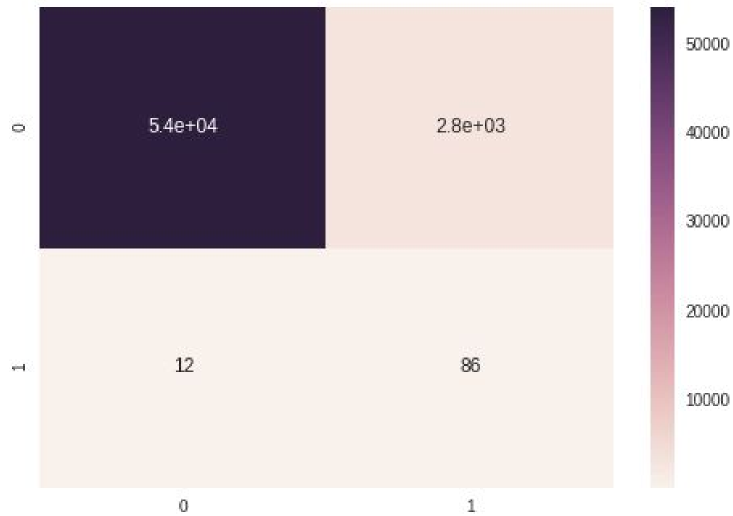
```
average_precision = average_precision_score(Y_test, anomaly_score)
#print('Average precision-recall score: {0:0.2f}'.format(average_precision))
precision, recall, _ = precision_recall_curve(Y_test, anomaly_score)
plt.step(recall, precision, color='b', alpha=0.2, where='post')
plt.fill_between(recall, precision, step='post', alpha=0.2,color='b')
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.ylim([0.0, 1.05])
plt.xlim([0.0, 1.0])
```



The precision-recall curve shows the tradeoff between precision and recall for different threshold. A high area under the curve represents both high recall and high precision, where high precision relates to a low false positive rate, and high recall relates to a low false negative rate. High scores for both show that the classifier is returning accurate results (high precision), as well as returning a majority of all positive results (high recall).

When contamination = 0.05 and n_estimators = 100

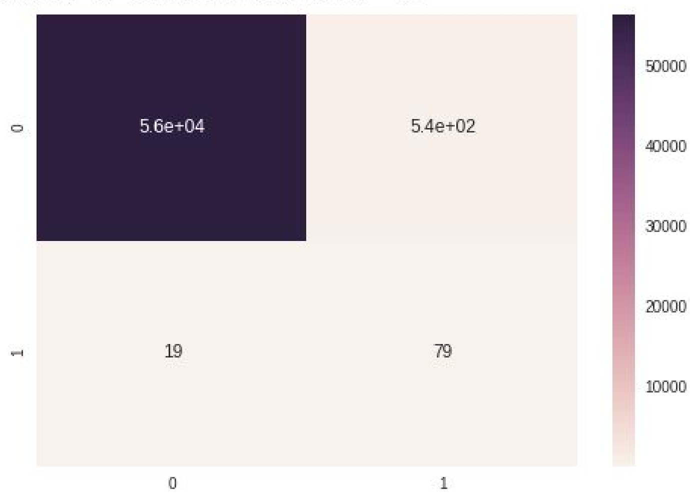
```
number of normal in test data:  56863
number of fraud in test data:  98
```




```
accuracy score:  0.950281771738558
precision:  0.9981088260630653
recall:  0.950281771738558
f1-score:  0.9728895863873523
```

When contamination = 0.01 and n_estimators = 100

```
number of normal in test data:  56863
number of fraud in test data:  98
```

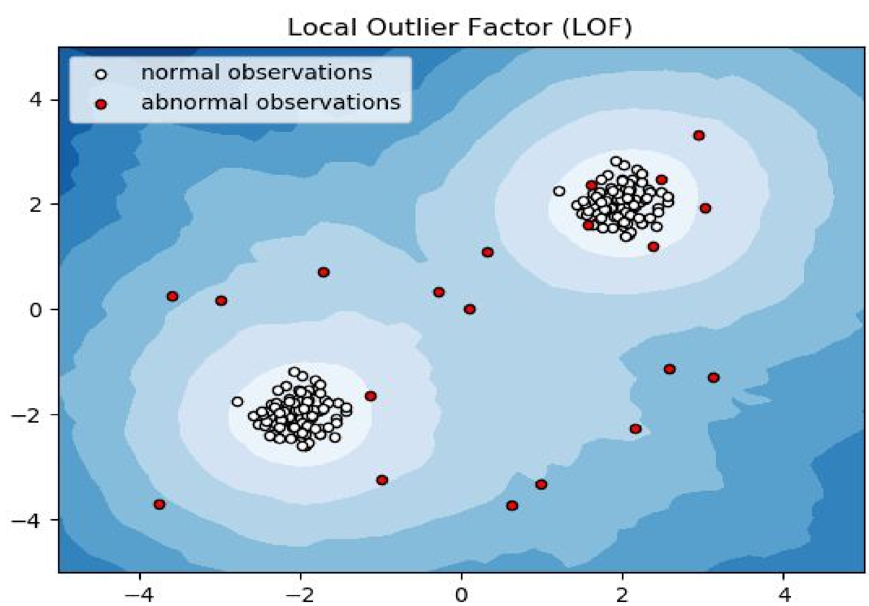



```
accuracy score:  0.9901511560541423
precision:  0.9981617354532224
recall:  0.9901511560541423
f1-score:  0.9937104255652516
```

Local Outlier Factor

The local outlier factor is based on a concept of a local density, where locality is given by ***K*** nearest neighbors, whose distance is used to estimate the density. By comparing the local density of an object to the local densities of its neighbors, one can identify regions of similar density, and points that have a substantially lower density than their neighbors. These are considered to be outliers.

The local density is estimated by the typical distance at which a point can be "reached" from its neighbors. The definition of "reachability distance" used in LOF is an additional measure to produce more stable results within clusters.



Function used to apply it is :-

LocalOutlierFactor(*n_neighbors=20*, *algorithm='auto'*, *leaf_size=30*, *metric='minkowski'*, *p=2*, *metric_params=None*, *contamination=0.1*, *n_jobs=1*

Used code :

```
import matplotlib.pyplot as plt
from sklearn.neighbors import LocalOutlierFactor
def local_outlier(X_train, X_test, Y_train, Y_test, n, os):

    X = X_train
    n_outliers = len(Y_test.Class == 1)
    ground_truth = np.ones(len(Y_test), dtype=int)
    ground_truth[-n_outliers:] = -1

    clf = LocalOutlierFactor(n_neighbors= n, contamination= os)
    clf.fit(X_train)
    y_pred=clf.fit_predict(X_test)

    n_errors = (y_pred != ground_truth).sum()
    X_scores = clf.negative_outlier_factor_
    y_pred[y_pred == 1] = 0
    y_pred[y_pred == -1] = 1

    accuracy = accuracy_score(Y_test, y_pred)

    print('accuracy score : ', accuracy)
    print("classification_report :  \n ", classification_report(Y_test, y_pred ))

    cm = pd.DataFrame(confusion_matrix(Y_test, y_pred))
    sb.heatmap(cm, annot=True)
    plt.show()


    precision, recall, fscore, support = score(Y_test, y_pred, average='weighted')
    print('precision: ', precision)
    print('recall: ', recall)
    print('f1-score: ', fscore)
```

```
n=200
os=0.001
local_outlier(x_train, x_test, y_train, y_test, n, os)
```

Two factors were to be changed to achieve best results :

- **Neighbours number**

  It represents the number of neighbours desired to be scanned for around each point , through which the algorithm can decide whether this point of low density (fraud) or of high density (legit) .

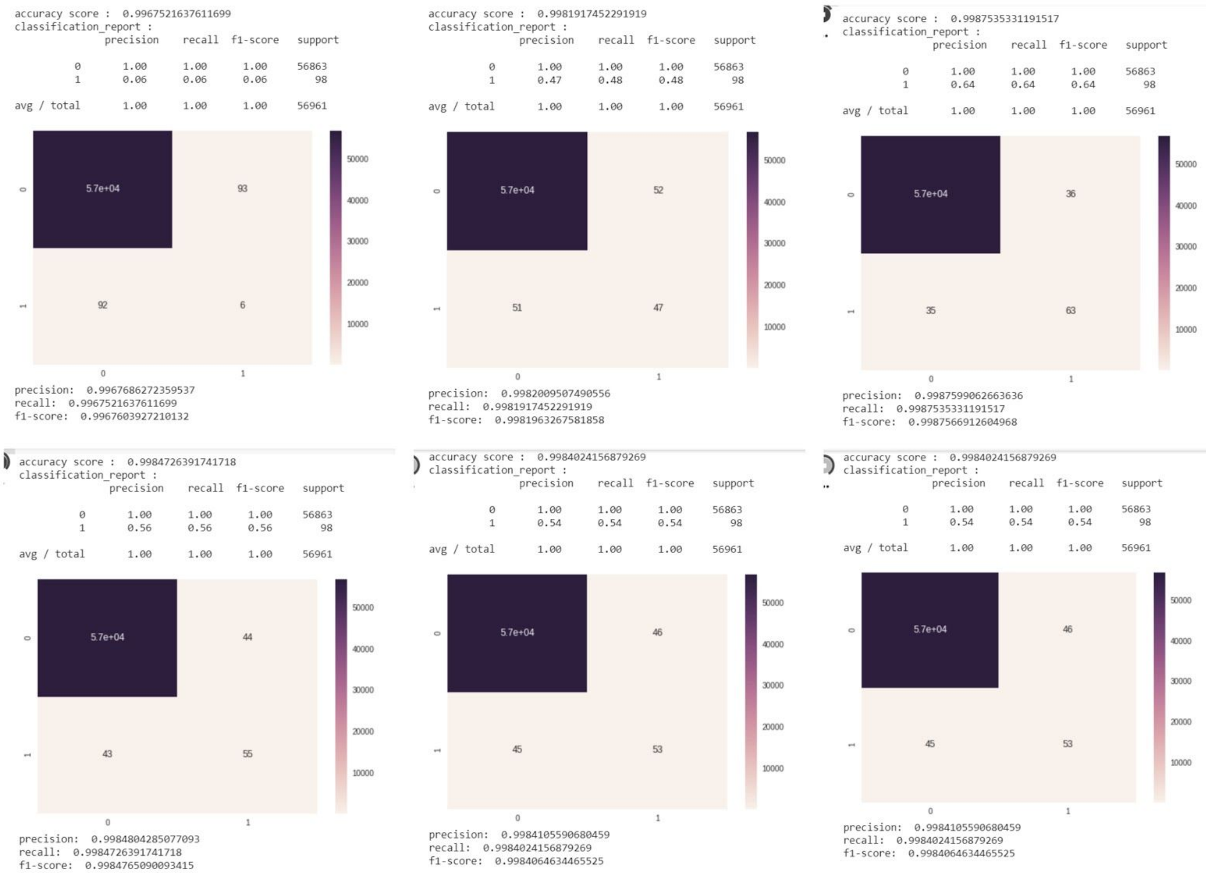
- **Contamination**

  Contamination here is a factor requested by the function LOF that represents roughly the ratio of outliers to those not .

First the neighbours number

Various values can be applied , the picked few were (50 , 100, 200, 500, 700, 900)

At a fixed value of contamination for comparison .



	50	100	200	500	700	900
Accuracy TP+TN/TP+FP+FN+TN	0.9967	0.9981	0.9987	0.9984	0.9984	0.9984
Precision TP/TP+FP	0.9967	0.9982	0.9987	0.9984	0.9984	0.9984
F1-score	0.9967	0.9989	0.9987	0.9984	0.9984	0.9984

Observation :

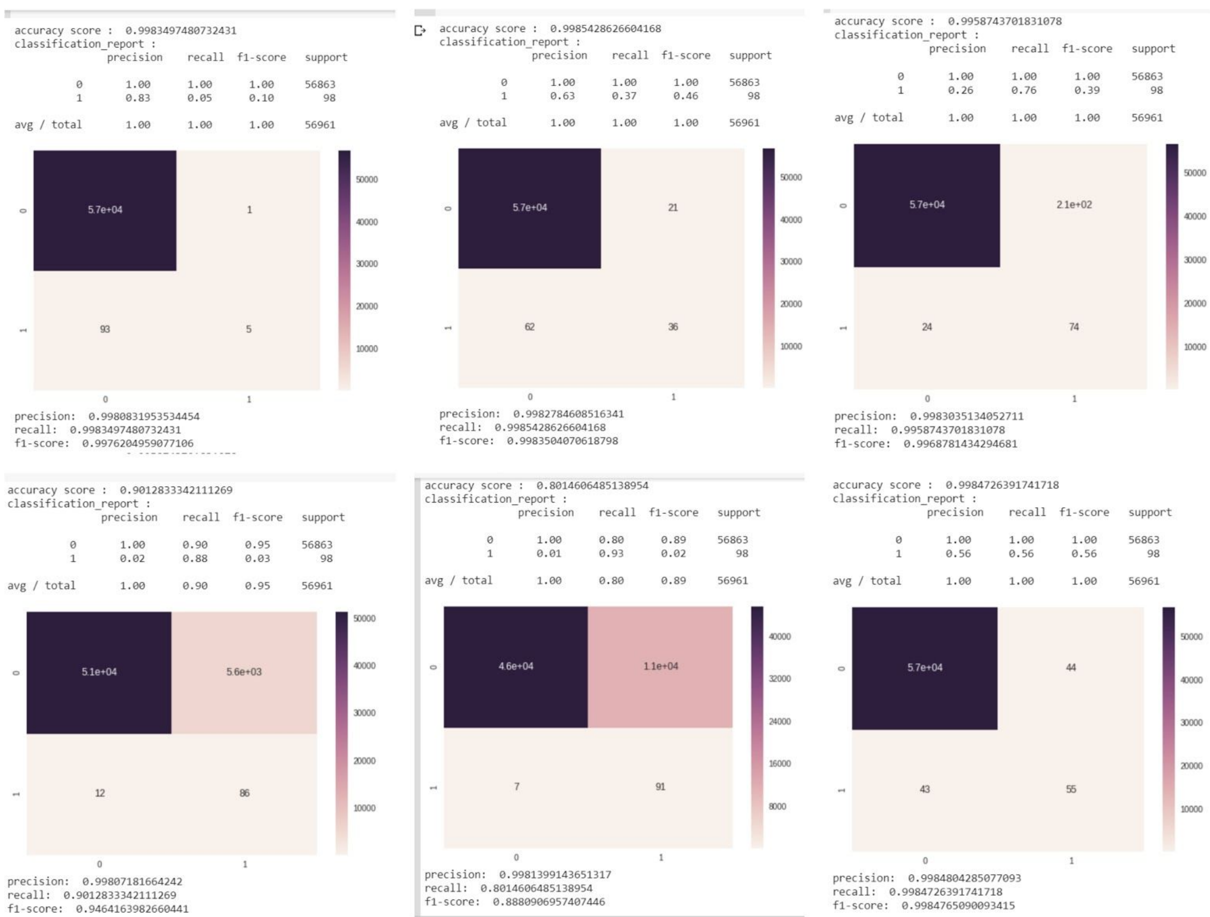
the slope of effect of the changing values according to the results showed that the peak where at the value of (200) at all aspects accuracy , precision , F1 .

Second the contamination

Various values can be applied , the picked few were (0.0001,0.001,0.005,0.1,0.2)

(And an additional value equal to 0.001727 which is the actual ratio of the outliers to the data calculated for experimenting)

At a fixed value for the neighbours number for comparison .



	0.0001	0.001	0.005	0.1	0.2	Outliers Actual ratio (0.0001727)
Accuracy TP+TN/TP+FP+FN+TN	0.9983	0.9985	0.9958	0.90	0.80	0.9984
Precision TP/TP+FP	0.9980	0.9982	0.9983	0.9980	0.9981	0.9984
F1-score	0.9976	0.9983	0.9968	0.9464	0.888	0.9984

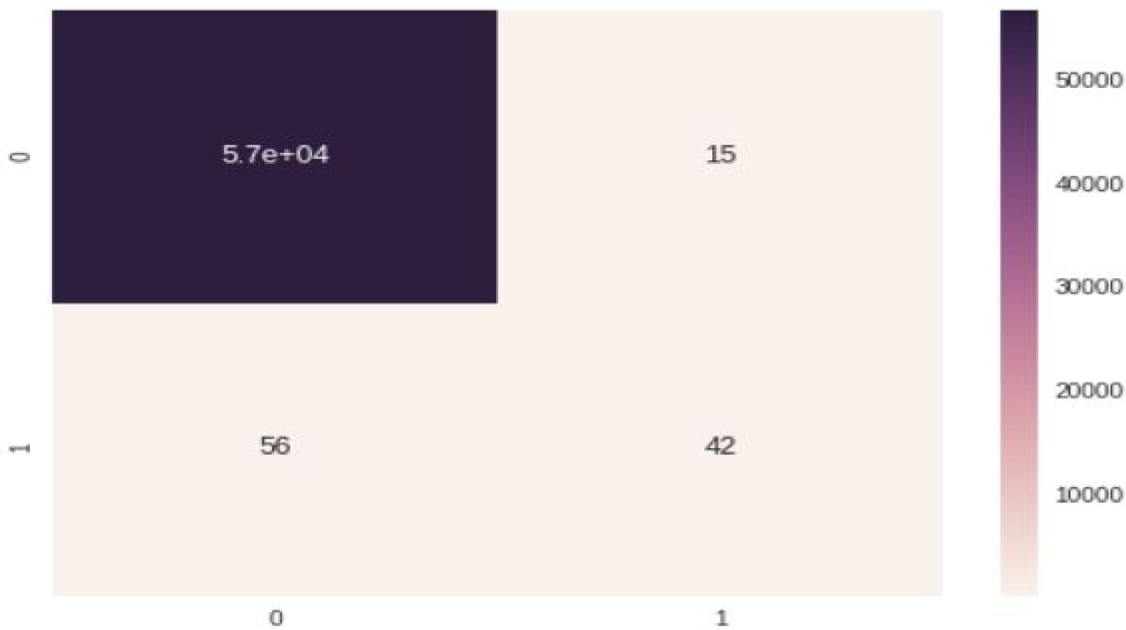
Observation :

the slope of effect of the changing values according to the results showed that the peak where at the value of (0.001) at all aspects accuracy , precision , F1 .

Thus make the best pair approximate to (200, 0.001).

Which gives the following results :

```
accuracy score :  0.9987535331191517
classification_report :
             precision    recall  f1-score   support

          0       1.00      1.00      1.00     56863
          1       0.74      0.43      0.54        98

avg / total       1.00      1.00      1.00     56961
```




```
precision:  0.998564822860785
recall:  0.9987535331191517
f1-score:  0.9985889024264273
```

Probably better results could be achieved at precise values but the ending result is close enough to the peak compared to the other values

Other than those two parameters ,other parameters were of no interest , However feature selection can produce perfect results but it’s far too complex for the code and the application, which makes the ending result here considerably satisfying .

Results

	Isolation Forest	LOF	Logistic Regression	Random Forest
Accuracy TP+TN/TP+FP+FN+TN	0.98	0.99	0.97	0.99
Precision TP/TP+FP	0.99	0.99	0.07	0.90
Recall TP/TP+FN	0.98	0.99	0.89	0.81
F1-score 2*(Recall * Precision) / (Recall + Precision)	0.99	0.99	0.13	0.85

References

https://blog.easysol.net/using-isolation-forests-anamoly-detection/

https://towardsdatascience.com/outlier-detection-with-isolation-forest-3d190448d45e