Machine Learning Engineer Nanodegree

Capstone Report

Ramakrishnan A

June 2019

Domain Background

(Copied from Proposal)

Credit card fraud is a wide-ranging term for theft and fraud committed using or involving a payment card, such as a credit card or debit card, as a fraudulent source of funds in a transaction. Although incidences of credit card fraud are limited to about 0.1% of all card transactions, they have resulted in huge financial losses as the fraudulent transactions have been large value transactions. In 1999, out of 12 billion transactions made annually, approximately 10 million—or one out of every 1200 transactions—turned out to be fraudulent. [1] The number of frauds and types of scams are increasing and it looks to become higher in the future. So, the need to detect fraudulent transactions is assuming higher importance than other activities.

The Kaggle site https://www.kaggle.com/mlg-ulb/creditcardfraud/kernels has a variety of solutions involving Deep Learning and Machine Learning techniques.

Problem Statement

(Copied from Proposal)

The objective of this model will be to identify whether a transaction is fraudulent or genuine. This should happen during the time the transactions is happening. The commonly used techniques for this problem, given that it involves categorization are Support Vector Machines (SVM) or Random Forest Classifier or Decision Trees. Every transaction has a set of data points that describe it and is classified as 'Fraud (1)' or 'Normal (0)'.

I am also curious about this problem as I want to try using Neural Networks instead of ML techniques and measure the performance.

Challenge - Dataset

Building models for fraud detection as a classification problem have the issue of unbalanced datasets. Given that majority of the data that will be available in the dataset will be genuine, we need to supplement the dataset with fraudulent data to build reasonably accurate models. Two of the popular methods of oversampling dataset(s) are SMOTE and ADASYN.

Synthetic Minority Over-Sampling Technique (SMOTE):

This method takes the n-nearest neighbours for the minority set and then draws a line between the neighbours. Then it generates random points on those lines and add those points to the dataset. This addition is synthetic in nature and hence the name. SMOTE is used during the pre-processing stage on the original dataset and balances the dataset.

Adaptive synthetic sampling approach (ADASYN):

This is an improved version of SMOTE where random values are added to the created points. This ensures that the samples are not linearly correlated to the parent point and have some variance. More information can be found at: https://github.com/scikit-learn-contrib/imbalanced-learn

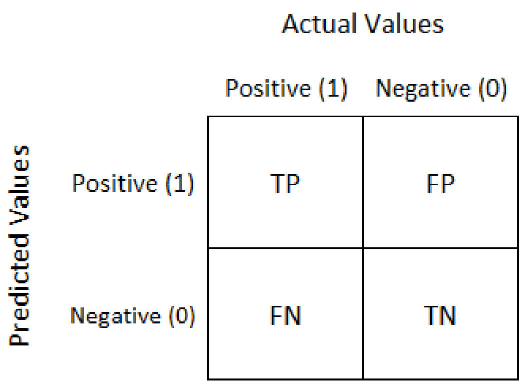
Evaluation Metrics

The common metrics used to compare models that predict probabilities for two-class problems are the ROC curve (Area under ROC Curve or AUROC) and the components of the confusion matrix.

Besides the AUROC, these metrics are also used for comparison

- Sensitivity
- Accuracy
- Specificity
-

Confusion Matrix:



		Actual Values	
		Positive (1)	Negative (0)
Predicted Values	Positive (1)	TP	FP
	Negative (0)	FN	TN

(Source: https://towardsdatascience.com/understanding-confusion-matrix-a9ad42dcfd62)

The Sensitivity of a Model is calculated as the number of true positives divided by the sum of the true positives and false negatives. It is a measure of how the model is at predicting the positive class when the actual result is positive.

Sensitivity = TP / (TP + FN)

The Specificity of a Model is calculated as the number of false positives divided by the sum of the false positives and true negatives. Specificity is inverted false positive rate which tell us how often a positive outcome is predicted with the actual result is negative.

Specificity = TN / (TN + FP)

Sensitivity and Specificity are inversely proportional to each other. When we increase Sensitivity, Specificity decreases and vice versa.

The other two metrics that I have shown in the code are Accuracy & error rate.

Accuracy = (TP + TN) / (Sum of all quadrants)
Error Rate = (FP + FN)/ (Sum of all quadrants)

Area under Receiver Operating Characteristic Curve (AUROC)

The AUROC curve is a useful tool when predicting the probability of a binary outcome. A model with an AUC score near 1 is good at separating the positive and negative outcomes, while a model which has a score of 0 has poor separability. A model which has a score of 0.5 has no capacity to separate the classes.

How is ROC defined?

ROC is a plot of False Positive Rate (FPR) on the x-axis against the True Positive Rate (TPR) on the y-axis for different threshold levels between 0 & 1.0. (False alarm rate vs. Hit rate). The curve provides an aggregated measure of performance across the thresholds. The higher the AUROC the better the model and vice-versa.

The objective of any fraud detection system is to identify the fraudulent transactions with minimum number of FN. It is acceptable that a genuine transaction be labelled as fraudulent, while the opposite is not acceptable. Sensitivity, Specificity and the AUROC values will help us determine these for a model and determine the performance.

Data Analysis

Datasets and Inputs

(Copied from Proposal)
I started working on this problem after I found the dataset in Kaggle [3]. (The next paragraph is copy-paste from Kaggle)

The datasets contain transactions made by credit cards in September 2013 by European cardholders. This dataset presents transactions that occurred in two days, where we have 492 frauds out of 284,807 transactions. The dataset is highly unbalanced, the positive class (frauds) account for 0.172% of all transactions. It contains only numerical input variables which are the result of a PCA transformation. Unfortunately, due to confidentiality issues, we cannot provide the original features and more background information about the data. Features V1, V2, ... V28 are the principal components obtained with PCA, the only features which have not been transformed with PCA are 'Time' and 'Amount'. Feature 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset. The feature 'Amount' is the transaction Amount, this feature can be used for example-dependant cost-sensitive learning. Feature 'Class' is the response variable and it takes value 1 in case of fraud and 0 otherwise.

The Independent variable is ‘Class’, while the other 30 columns represent the independent variables.

Exploration

The dataset as mentioned above is unbalanced. The number of genuine transactions outweigh the fraudulent transactions by a factor of 570.

Genuine: 284315, Fraudulent: 492

Sample of the dataset (Please note this data is prior to normalization of the Amount column)
Data.head()

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928	0.128539	-0.189115	0.133558	-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	...	-0.225775	-0.638672	0.101288	-0.339846	0.167170	0.125895	-0.008983	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	...	0.247998	0.771679	0.909412	-0.689281	-0.327642	-0.139097	-0.055353	-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	...	-0.108300	0.005274	-0.190321	-1.175575	0.647376	-0.221929	0.062723	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	...	-0.009431	0.798278	-0.137458	0.141267	-0.206010	0.502292	0.219422	0.215153	69.99	

Some description about the Amount column.

genuine['Amount'].describe()

```
count    284315.000000
mean         88.291022
std         250.105092
min           0.000000
25%           5.650000
50%          22.000000
75%          77.050000
max       25691.160000
Name: Amount, dtype: float64
```

```
fraudulent['Amount'].describe()
count     492.000000
mean      122.211321
std       256.683288
min         0.000000
25%         1.000000
50%         9.250000
75%       105.890000
max      2125.870000
Name: Amount, dtype: float64
```

Just looking at the statistics above, it looks like the mean of fraudulent transactions is nearly 3 times genuine transactions and this itself can be a flag for identifying these transactions.

The next step was to identify if the data had any missing values and the output showed that there were no missing values. If there were missing values, then I would have used Imputer and used the ‘mean’ as the representative value.

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

```
Amount    False
Class     False
dtype: bool
```

The dataset has a value represented by 'Time' and this is different from the others (V1, V2….) as it represents the time elapsed from the first transaction. I wanted to check if this has any distribution either with respect to the Independent variable or it influences the 'Amount' variable. The plots are shown below.

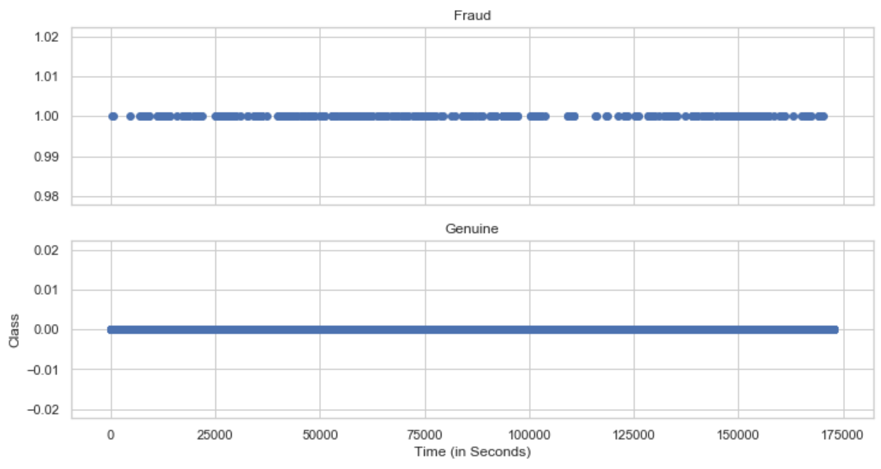


Figure 1 Time vs Class

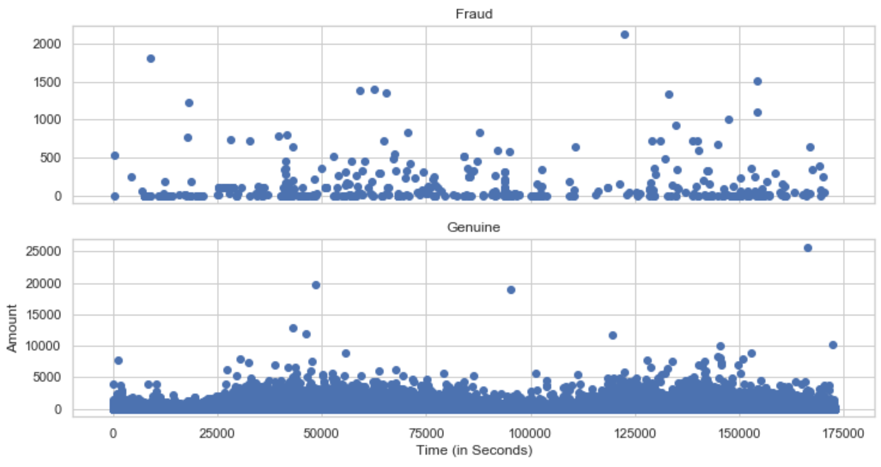


Figure 2 Time vs. Amount

Given the nature of distributions, I decided to drop the 'Time' feature as there is nothing that stands out in the above images.

The next step is to identify how the different variables (V1, V2, …) influence the 'Class' variable. To perform this, I computed the correlation matrix for the data.

correlation = data.corr()

The output is as follows:

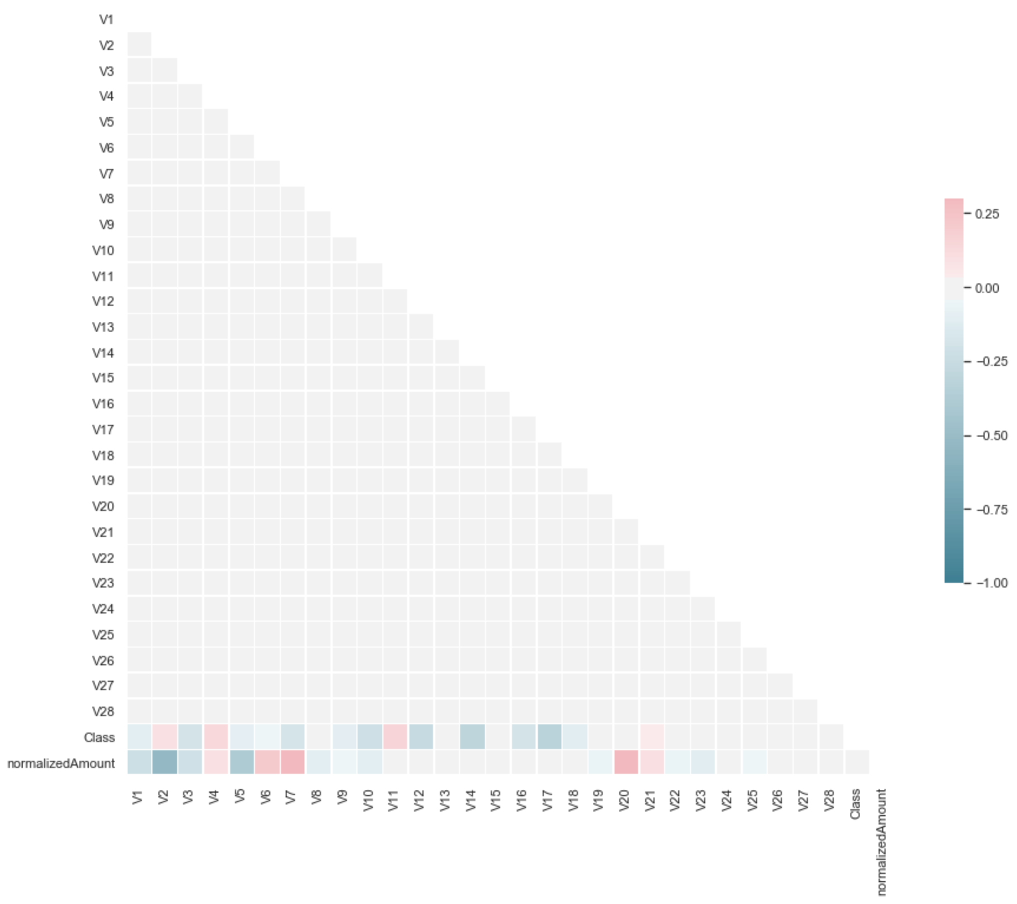


Figure 3 Correlation Matrix

As seen above, the correlation matrix clearly shows that V8, V13, V15, V20 & V22-28 have minimum influence on the 'Class' variable. I am dropping these columns before building the models as it would help in quicker and better performance of the model. Post this command, the dataset has 284807 rows and 19 columns including 'Class'.

Algorithms

Deep Neural Network is a method of information processing which is influenced by how the neurons work inside the human brain. In the human brain the systems adjust depending on the problem on hand and it varies from simple systems which involve a yes or no decision to complex analysis which might involve various levels of decision making. In the same way, neural networks can also adapt to different situations on the field provided they are trained for it. The network is also capable of adjusting the weights to ensure that the error in output is minimised.

A typical neural network consists of an input layer, followed by hidden layers and then an output layer. The output layer could have a single node or multiple nodes depending on the problem. Each layer in a neural network has a set of nodes, the associated weights and an activation function which provide the inputs to the next layer. The information flow is from input to output, while the learning can happen thru' back propagation.

Sequential Models

The sequential API allows you to create models layer-by-layer for most problems. It does not allow creation of models that share layers of have multiple inputs/outputs. However, it is the easiest way to build models using Keras. Each layer that is defined has weights that correspond to the layer the follows it.

At the minimum each model will have an input layer and an output layer. The input layer specifies the shape of data, nodes to use and the activation function. The output layer specifies the number of outputs and the activation function.

The most commonly used layer type is ‘Dense’. In a dense layer, all nodes in the previous layer connect to the nodes in the current layer. ‘Activation’ is the activation function for the layer. An activation function allows models to consider nonlinear relationships. The output activation function for classification problems is a ‘Sigmoid’ function.

My model:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 8)                 152
_________________________________________________________________
dense_2 (Dense)              (None, 24)                216
_________________________________________________________________
batch_normalization_1 (Batch (None, 24)                96
_________________________________________________________________
dropout_1 (Dropout)          (None, 24)                0
_________________________________________________________________
dense_3 (Dense)              (None, 32)                800
_________________________________________________________________
batch_normalization_2 (Batch (None, 32)                128
_________________________________________________________________
dropout_2 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_4 (Dense)              (None, 16)                528
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 17
=================================================================
Total params: 1,937
Trainable params: 1,825
Non-trainable params: 112
```

I started with a model that had 10 times more parameters and was producing really great results, but the time it took to compile and produce results made it unsustainable to be used. The initial model is listed below.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 16)                304
_________________________________________________________________
dense_2 (Dense)              (None, 64)                1088
_________________________________________________________________
batch_normalization_1 (Batch (None, 64)                256
_________________________________________________________________
dropout_1 (Dropout)          (None, 64)                0
_________________________________________________________________
dense_3 (Dense)              (None, 128)               8320
_________________________________________________________________
```

```
batch_normalization_2 (Batch (None, 128)           512
_________________________________________________________________
dropout_2 (Dropout)        (None, 128)            0
_________________________________________________________________
dense_4 (Dense)           (None, 64)            8256
_________________________________________________________________
batch_normalization_3 (Batch (None, 64)            256
_________________________________________________________________
dropout_3 (Dropout)        (None, 64)            0
_________________________________________________________________
dense_5 (Dense)           (None, 16)            1040
_________________________________________________________________
dense_6 (Dense)           (None, 1)             17
=================================================================
Total params: 20,049
Trainable params: 19,537
Non-trainable params: 512
_________________________________________________________________
```

Random Forest

The Random Forest algorithm is made up of Decision Trees. The decision trees in this case are formed by random sampling of data points when building trees. The other distinction from Decision trees is that the while splitting the nodes at every point, random subset of features is used. Beyond these differences, Random Forests and Decision Trees follow the same set of principles at a high level. I have used 50 (n_estimators) trees in the randomforest model. I started with 100 estimators, but there was no significant difference in any of the metrics between n_estimators of 50 or 100.

The model is

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
        max_depth=None, max_features='auto', max_leaf_nodes=None,
        min_impurity_decrease=0.0, min_impurity_split=None,
        min_samples_leaf=1, min_samples_split=2,
        min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=None,
        oob_score=False, random_state=None, verbose=0,
        warm_start=False)
```

Benchmark Model

I built a model with RandomForest as benchmark for this dataset as mentioned above. The ROC curve for this model on the test data is as follows

AUC Score: 0.957

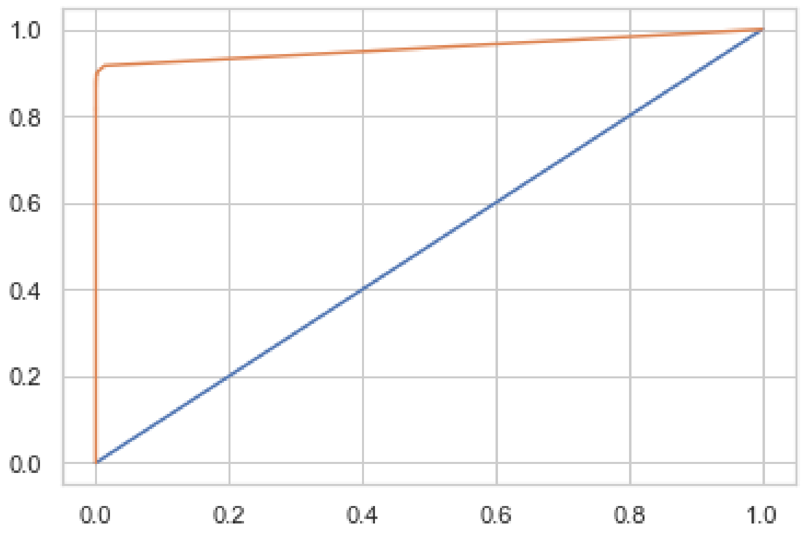


Figure 4 ROC Curve - RandomForest

The confusion matrix is as follows

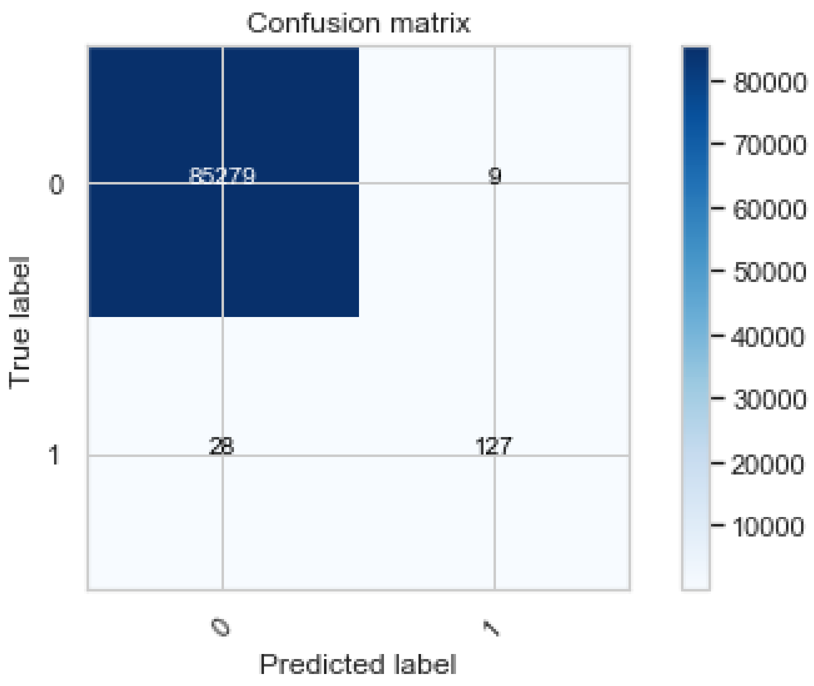
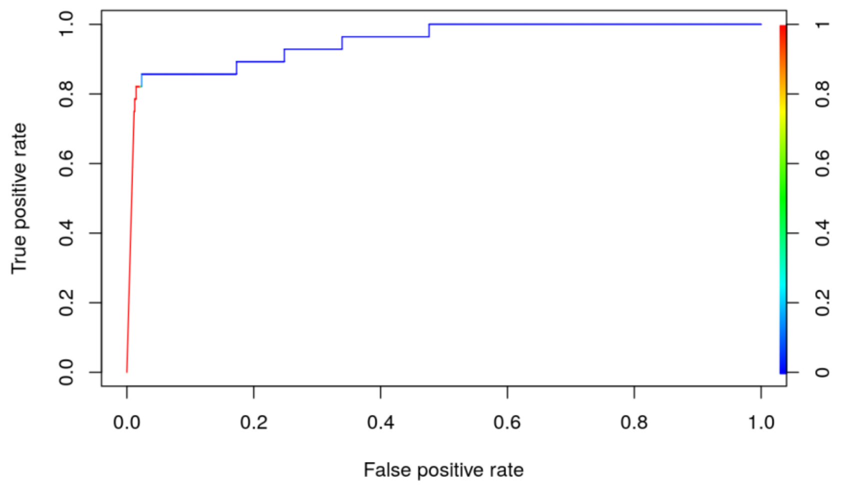


Figure 5 Confusion Matrix – RandomForest

(%)	Predicted Positives	Predicted Negatives
True Positives	99.80	0.010
True Negatives	0.032	0.148

The other model that I used for comparison is:

https://www.kaggle.com/yuridias/credit-card-fraud-detection-knn-naive-bayes/report, (KNN & Naïve Bayes), has an AUC of 0.949.



Building the Model

Data Pre-processing

As mentioned in the exploration state, I dropped the 'Time' column along with columns V8, V13, V15, V20 & V22-28 as they have minimum influence on the independent variable.

The other pre-processing step is to normalize the Amount data using the StandardScaler library. Besides this there are no more pre-processing steps that have been performed as we don't know about the nature of the other columns.

Building the Model

The model is built on the same lines as discussed above. The summary is also shown as part of the Algorithms section.
Libraries used:

- Keras
- Scikit Learn
- Matplotlib
- Seaborn
- Imblearn
- Pandas and Numpy

Python 3 with Jupyter notebook was used for execution.

1. Data is split into training and test set (done as part of building the randomforest model and repeated with SMOTE and ADASYN) with test size of 0.3 and random state of 7.
2. The sequential model is built with the Sequential function from Keras. I have used 5 Dense layers with the last layer using the 'Sigmoid' activation function. The other layers use 'Relu'.
3. I have also added Batchnormalization and Dropout functions as necessary.

I also used GridsearchCV to identify the batch-size and epochs that would be best with accuracy as the metric, with CV as 3.

parameters = {'batch_size': [8, 16, 32], 'epochs': [10, 30, 50]} grid_search = GridSearchCV(estimator = classifier, param_grid = parameters, scoring = 'accuracy', cv = 3)

Based on this the model was trained for 30 epochs with a batch size of 8.

The other parameters used for compiling the model are
Optimizer: adam
Loss: binary_crossentropy
Metrics: Accuracy.

A few points to note while building the model:

- It is important how the dataset is split for training and testing. I would recommend that we should use a factor of 0.25 or 0.3 or even higher, given the imbalance in the dataset. If we choose the normal default of 0.2, then the number of negative samples is less than 100 for this complete dataset. In case, of SMOTE or ADASYN, 0.2 will work fine.
- It is important to build the Sequential model with minimum nodes to start with and then proceed to build more complicated structures. I started with 2 hidden layers and then moved to having 5 hidden layers. But, with 5 hidden layers, the amount of time the model would take to compile was very high.
- It is important to identify the parameters around which the model will provide the lowest validation and training loss. As suggested this can be done by finding the best parameters model through GridsearchCV.
- With respect to SMOTE and ADASYN, it is important to note that we are changing the dataset characteristics by a great deal. However, given that it is only adding negative outcomes (with linear correlation in case of SMOTE), we should be careful in interpreting the values that are intrinsic to the dataset, like the mean of the Amount column.

Refinement(s)

Using SMOTE and ADASYN, I built a dataset that had more minority samples to see if there was a resulting improvement in performance. I used the same parameters for building the model besides the change in the dataset.

Model Evaluation

Deep Learning Model

AUROC Score: 0.976
The score indicates a high degree of learning and is better than the randomforest model.

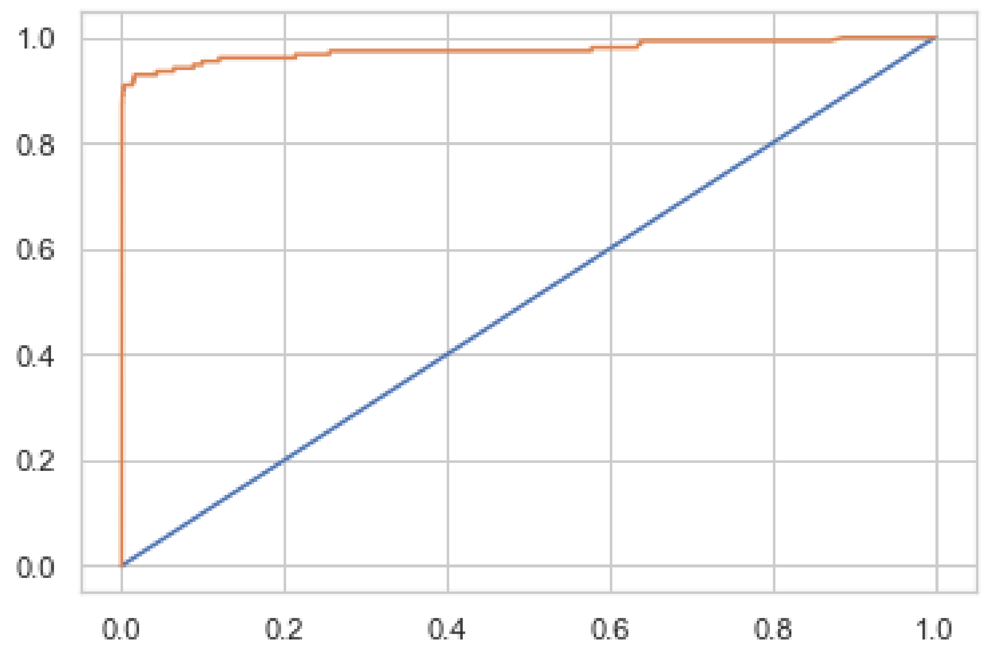


Figure 6 AUROC Curve - Deep Learning

Confusion Matrix and Loss Plot

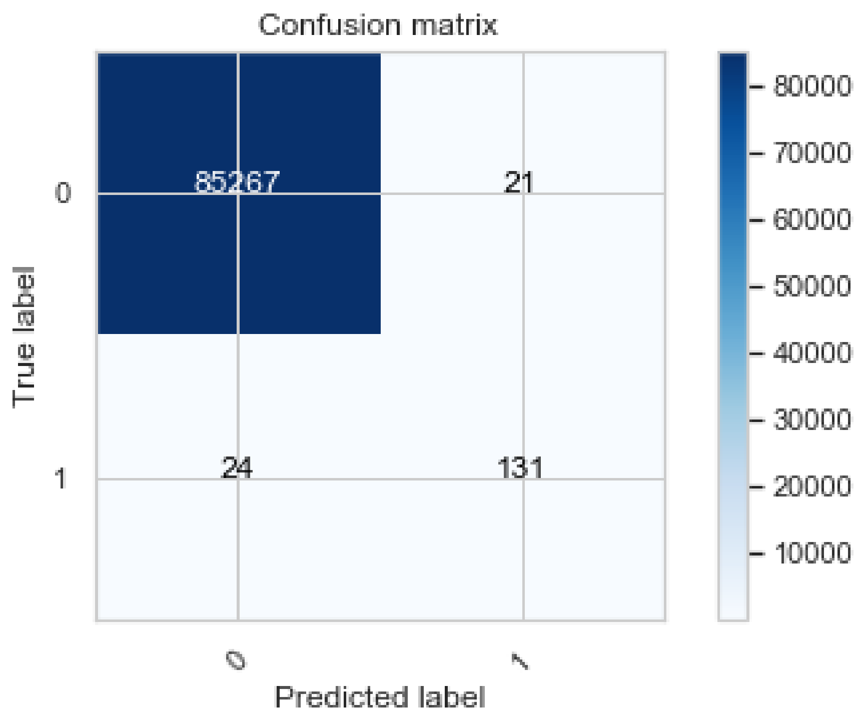


Figure 7 Confusion Matrix - Deep Learning

(%)	Predicted Positives	Predicted Negatives
True Positives	99.79	0.02
True Negatives	0.02	0.15

The number of misclassified values can be compared to the randomforest model (45 vs. 37) and over the sheer number of datapoints the statistical significance is limited.

The loss curve shown below indicates that the model has learnt the data well enough to make predictions as the validation error is varying more than the training error. The model isn't remembering what it learnt and is making predictions properly.

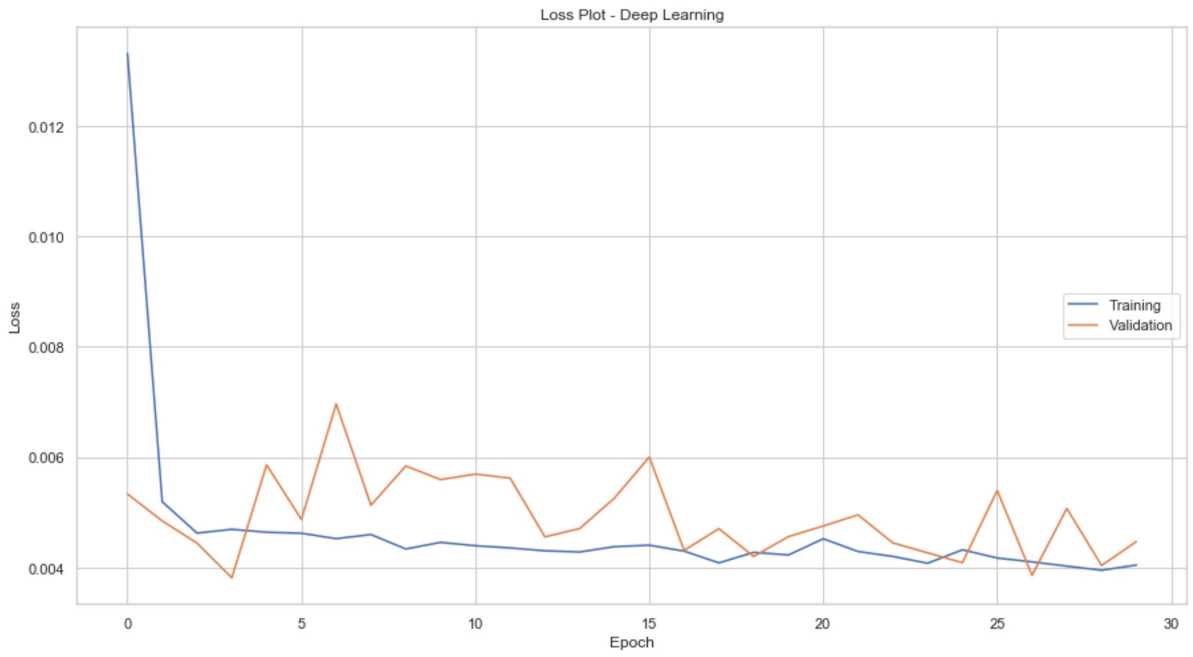


Figure 8 Loss Plot - Deep Learning

AUROC is on the higher side for a Deep Learning model. I have optimized for accuracy rather than speed. This is fine to do for test scenarios, but in the real world there needs to be a balance between the time taken for output and the accuracy.

Using SMOTE

The size of data once SMOTE algorithm was applied is (568630, 18).
Size of test data: 170589

AUROC – Smote: 0.999

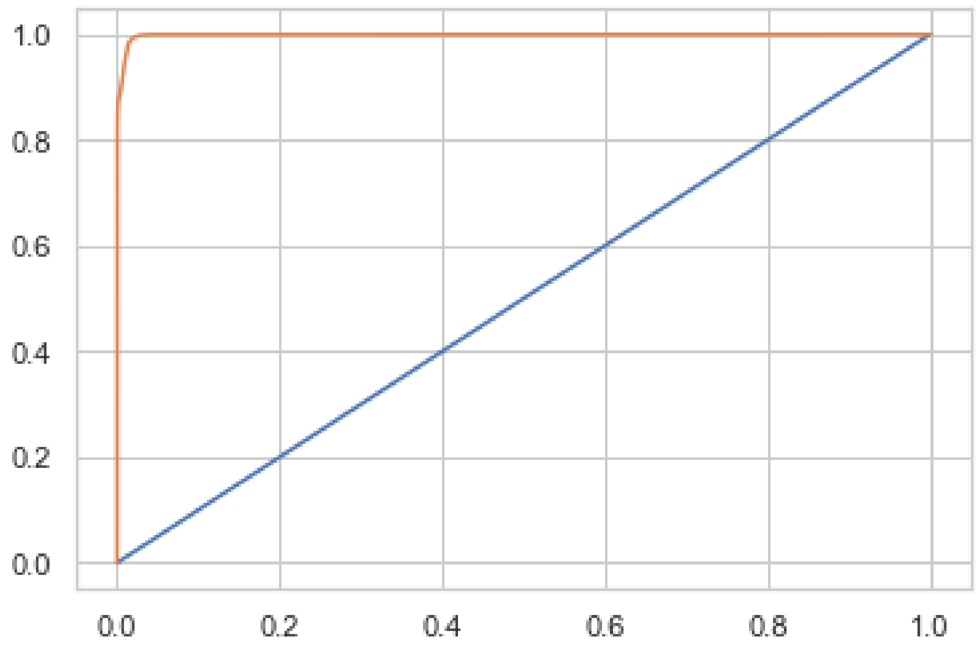


Figure 9 AUROC – SMOTE

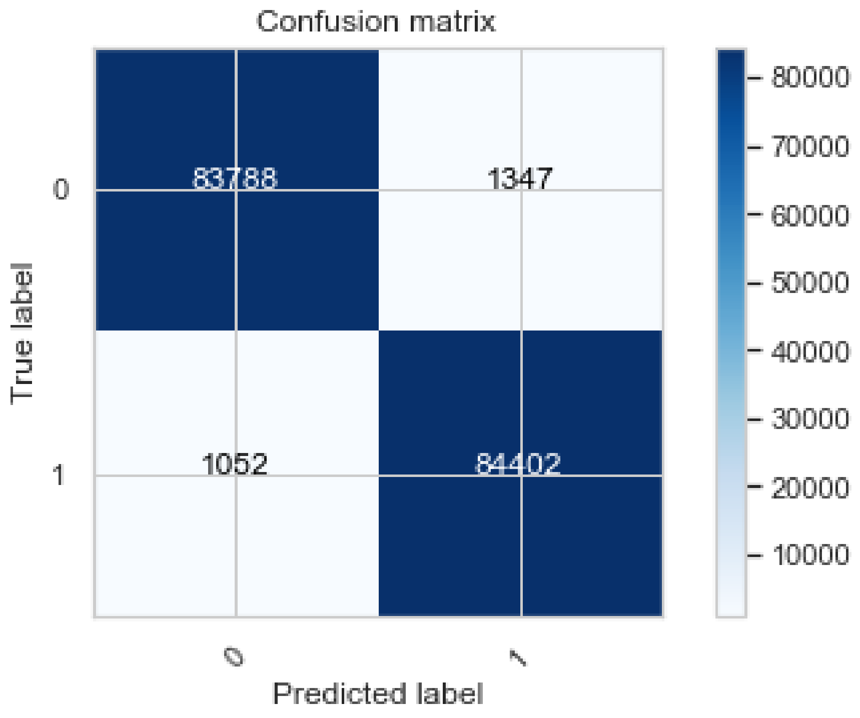


Figure 10 Confusion Matrix – SMOTE

(%)	Predicted Positives	Predicted Negatives
True Positives	49.11	0.78
True Negatives	0.61	49.47

The above matrix clearly indicates that the algorithm is working as needed. The misclassified values are ~1.3% of the total values compared to 0.04% for the Deep Learning model. This represents 32 times increase for the model to learn about False Positives and False Negatives.

The loss plot is available below.

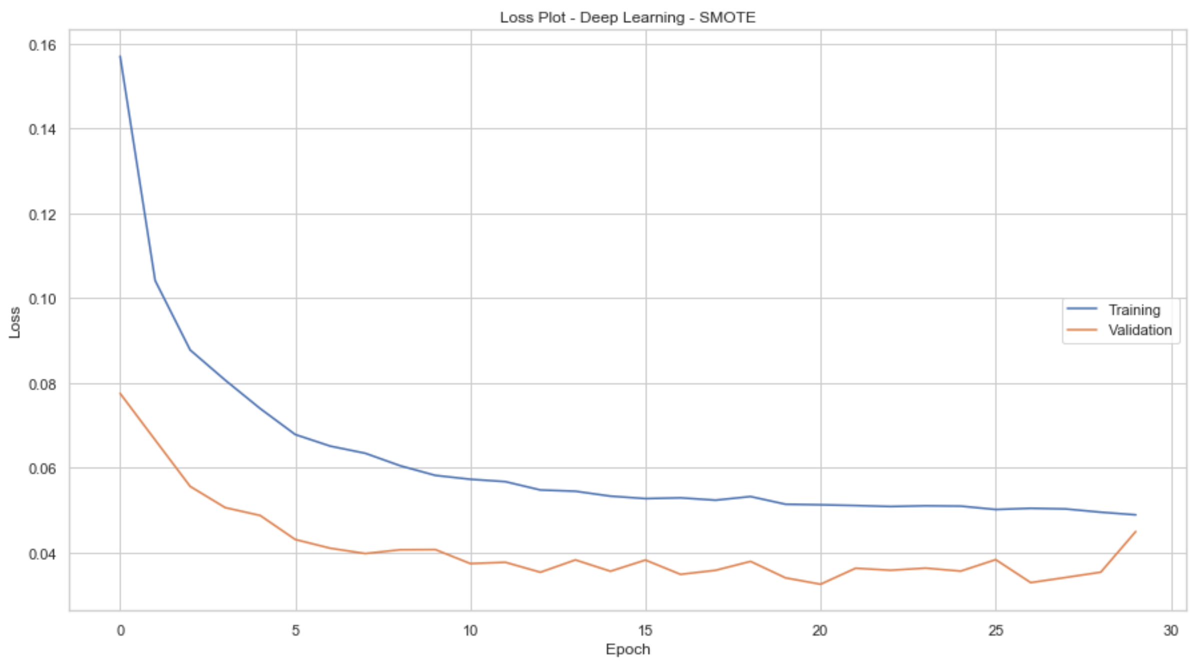


Figure 11 Loss Plot - Deep Learning - SMOTE

The loss plot indicates that the validation error is much higher than what Deep Learning loss plot indicated. The validation error is nearly 10 times higher and should be the case as additional number of negative outcomes have been added.

Using ADASYN

The size of data once ADASYN algorithm was applied is (568613, 18).
Size of test data: 175984

AUROC – ADASYN: 0.995

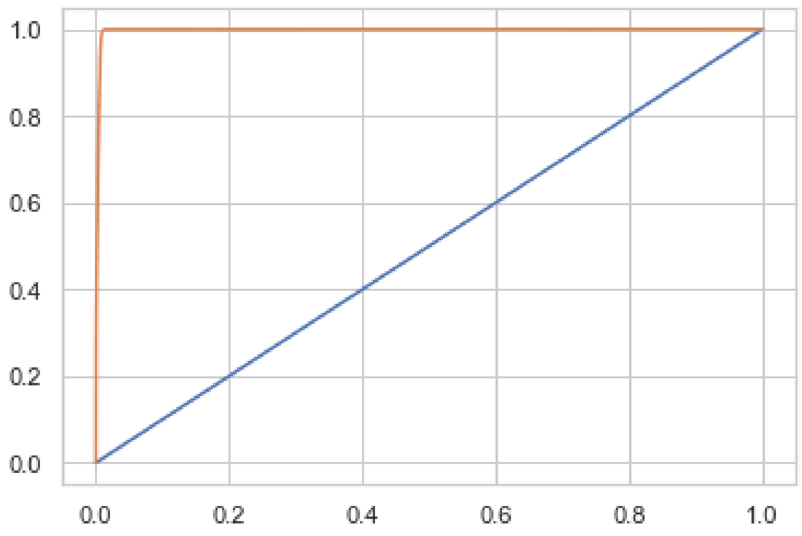


Figure 12 AUROC – ADASYN

The confusion Matrix is as follows:

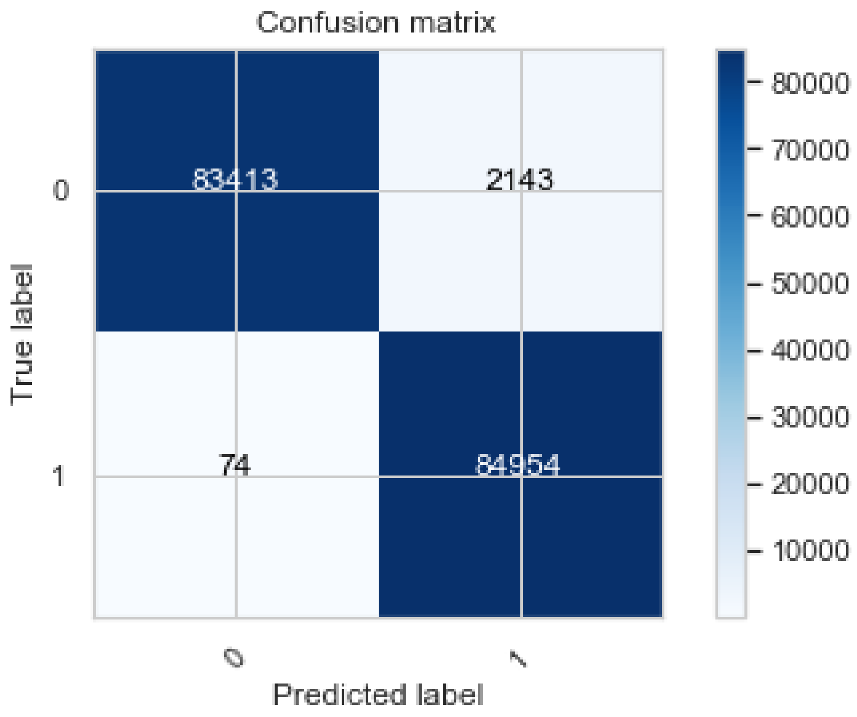


Figure 13 Confusion Matrix – ADASYN

(%)	Predicted Positives	Predicted Negatives
True Positives	48.89	1.25
True Negatives	0.04	49.8

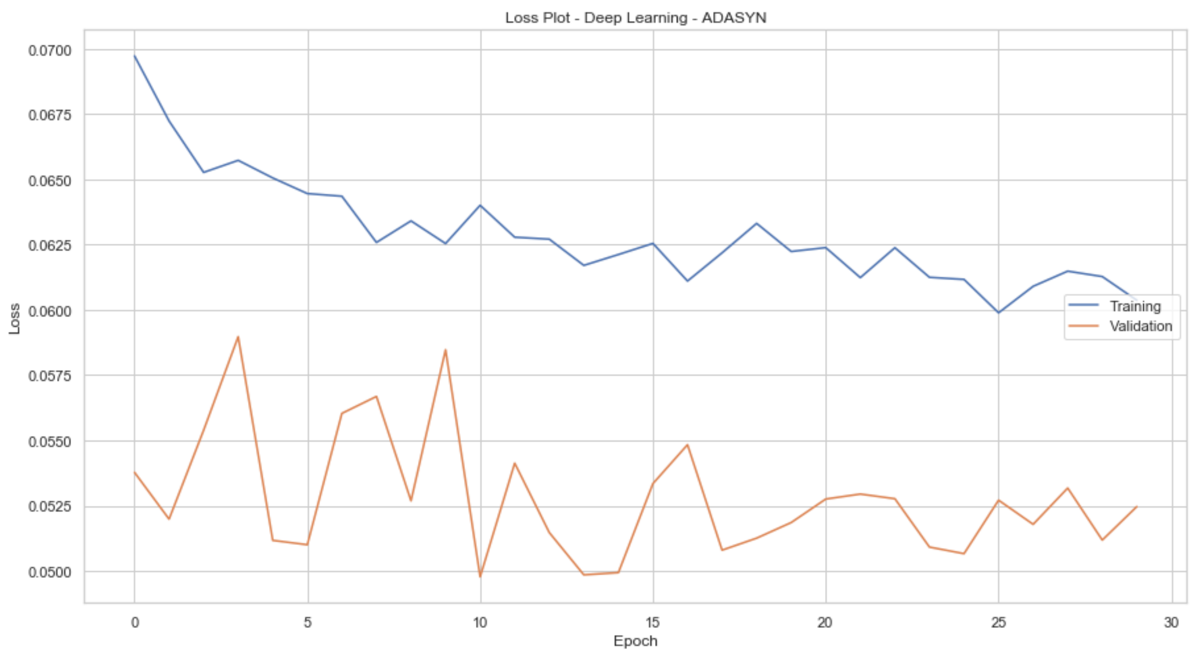


Figure 14 Loss Plot – ADASYN

The loss plot for ADASYN and SMOTE are similar and settle around a loss of 0.05.

Both ADASYN and SMOTE have higher AUROC scores than the Deep Learning model as the model has learnt better with more negative outcomes in the dataset.

This shows that though the Deep Learning model looks better in terms of accuracy scores in the confusion matrix, it will not work well with real data.

The metrics for all the models built are available below:

	Model	**Accuracy**	**Error Rate**	**Specificity**	**Sensitivity**	**AUC Score**
0	Random Forest - Test Data (n=100)	99.956696	0.043304	93.382353	99.967177	0.957357
1	Random Forest - Complete Data (n=100)	99.986658	0.013342	98.093220	99.989801	0.986651
2	Deep Learning Model - Test Data	99.947333	0.052667	86.184211	99.971861	0.976260
3	Deep Learning Model - Complete Data	99.936799	0.063201	83.766234	99.963073	0.983860
4	DL Model - Using SMOTE - Test Data	98.593696	1.406304	98.429136	98.760019	0.998671
5	DL Model - Using SMOTE - Complete Data	98.583613	1.416387	98.401797	98.766801	0.998671
6	DL Model - Using ADASYN - Test Data	98.700347	1.299653	97.539525	99.911363	0.995364
7	DL Model - Using ADASYN - Complete Data	98.769462	1.230538	97.681776	99.907857	0.995748

All models have been tested with the test data that was split and complete dataset. This is a useful comparison to have as the metrics for both datasets should be comparable. If the complete dataset has significantly higher values for all the metrics, then the model is not good. However, if the test data has better metrics then the model has understood the dataset properly.

Conclusion

All the models learned to recognize frauds. The Deep Learning & (SMOTE & ADASYN) models learnt to identify fraud better than the randomforest model(s) as shown in the metrics above.

Given that fraud detection needs be accurate and fast, we need to identify anomalies quickly without compromising on detection quality. Basically, identify all the fraudulent transactions, with minimum false positives. The Deep Learning model works well with the test and training data but the improvement with SMOTE and ADASYN is evident.

Improvement(s)

The Deep Learning model with SMOTE improvement on the dataset works the best in terms of accuracy and speed. More experimentation is needed to identify if this model will work with the same level of performance with other dataset(s) given that the negative outcomes generated are related to the original negative outcomes.

We can try to use Penalized model(s) on the original dataset and compare the performance against ADASYN and SMOTE. Penalized classification imposes additional cost on the model for making mistakes on the negative outcomes in training. These penalties can also bias the model to pay more attention to the negative outcomes and understand the negative outcomes carefully.

On the original dataset we can use Mathews Correlation Coefficient (MCC) to predict how good the model is performing with respect to the original class and improve, if required. We can also try to use SMOTEBoost to improve sampling. https://www3.nd.edu/~nchawla/papers/ECML03.pdf. There are improvements to SMOTE that have been suggested, but I haven't tried them to comment on their performances.

Adding data with marginalized corrupted features (as suggested in http://proceedings.mlr.press/v28/vandermaaten13.pdf) can also be tried. Though it isn't directly related to this problem, the general approach can be used.

References

[1] https://en.wikipedia.org/wiki/Credit_card_fraud
[2] https://pdfs.semanticscholar.org/0419/c275f05841d87ab9a4c9767a4f997b61a50e.pdf
[3] https://www.kaggle.com/mlg-ulb/creditcardfraud
[4] https://www.kaggle.com/yuridias/credit-card-fraud-detection-knn-naive-bayes
[5] https://machinelearningmastery.com/roc-curves-and-precision-recall-curves-for-classification-in-python/
[6] https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html
[7] https://www.kaggle.com/sid321axn/fraud-detection-deep-learning-with-smote