Machine Learning Engineer Nanodegree

Capstone Project

Credit Card Fraud Detection

Xueqiu Feng June 27th, 2018

I. Definition

Project Overview

- **Problem Domain**

  Wikipedia defines fraud as: *deliberate deception to secure unfair or unlawful gain, or to deprive a victim of a legal right*. Fraud or deliberate deception is a skill that every species has already mastered to perfection through evolution, which is especially true for human being. In the modern era, with the development of new technologies, new forms of fraud have also been invented. Although fraud appears to be rarely happening, it can result in an huge amount of loss. Therefore, fraud detection is a focal point for insurance, financial, retail and tele-communication sectors. It also has drawed attention from the academical

- **Project Origin**

  I used the Kaggle dataset Credit Card Fraud Detection (https://www.kaggle.com/mlg-ulb/creditcardfraud), which was collected and analysed during a research collaboration of Worldline and the Machine Learning Group of ULB (Université Libre de Bruxelles) (http://mlg.ulb.ac.be) on big data mining and fraud detection. More details on current and past projects on related topics are available on http://mlg.ulb.ac.be/BruFence (http://mlg.ulb.ac.be/BruFence) and http://mlg.ulb.ac.be/ARTML (http://mlg.ulb.ac.be/ARTML).

- **Dataset**

  The datasets contains 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 $V_1, V_2, \dots V_{28}$ 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-senstive learning. Feature 'Class' is the response variable and it takes value 1 in case of fraud and 0 otherwise.

Problem Statement

Credit card is almost a necessity to survive in the modern society. However, it associates with risk exposure because of physical or informational theft. Whereas a stolen card can be reported and frozen immediately to prevent unauthorised transactions, a compromised account can be abused without notice untill receiving bill statement, where before the frudulent use the account information can be hold for an arbitrary time, making the source even harder to trace. Because of the huge amount of transactions, it is nearly impossible to check them one by one manuelly, which will also inevitablly result in inacceptable delay of transactions.

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. Therefore it would be very meaningful to build a system which can automatically detect dubious transactions and freeze it for further inspection.

The creditcard fraud detection problem can be easily formulated as a supervised machine learning problem, with the binary labels *fraud* or *non-fraud*, i.e. binary classification. Therefore almost all the possible supervised machine learning algorithms can be applied. Where the *AUC* and *F score* can be very relevant in terms of determining the quality of the classifier, for the reason that the dataset is strongly imbalanced.

Neural network will not be considered because of the long training time and the difficulty of finding the best architecture. Moreover, my goal is to use the conventional machine learning algorithms to come as close as possible to be competitive against neural network, which will also be my benchmark model. At the end the classifiers will be compared such that the optimal one, depending on my own definition of optimality, will be chosen to tackle this particular problem.

Metrics

Due to the imbalanced nature of this dataset and the binary of the classification itself, following metrics are applied:

- **F Score**

It is defined as

$$F_\beta = (1+\beta^2)\frac{precisision \cdot recall}{\beta^2 \cdot precision + recall}, \quad \beta \in [0, +\infty)$$

where

$$precision = \frac{tp}{tp+fp}$$

i.e. out all all the transactions which are classified as fraud, the share of real fraud transactions.

$$recall = \frac{tp}{tp+fn}$$

i.e. the success rate of detection if a fraud transaction is being conducted.

For our dataset, where the positive (frauds) class only account for 0.172% of all transactions, the measure of *recall* shall be much more important than *precision*. Because it would be preferable to identify non-fraud as fraud then the other way around, i.e. it might be sensible to pay the price that some of the non-fraud transactions are classified as fraud, in order to identify all the fraud transactions.

However, in an extreme scenario where we assert indiscriminately that, all the samples are fraud, *recall* will be a perfect 100%, whereas *precision* will be nearly 0.

Thus we will use the $F_{0.5}$ score to give *recall* more weight over *precision* in the evaluation.

- **Recall & Precision**

Alternatively we could also simply use *recall* and *precision* as metric.

- **Area under the Receiver Operating Characteristic Curve (AUROC)**

A *receiver operating characteristic curve*, i.e. *ROC curve*, is created by plotting *recall* against *false positive rate (fpr)*, which is defined as

$$fpr = \frac{fp}{fp+tn}$$

i.e. the ratio between the number of normal transactions wrongly categorized as fraud and the total number of actual normal transactions, with thresholds from 0 to 1.

Which can be understood as how the classifier's performance varies with different thresholds, where the classifier has an output of probability predicting a transaction being fraud. And only if the probability passes the threshold, the transaction shall be categorized as fraud. Notice that this measure is therefore only suitable for classifiers that have probablistic outputs.

The area under the *ROC curve* (often referred to as simply the *AUC* or *AUROC*) is equal to the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one (assuming 'positive' ranks higher than 'negative')*(Fawcett 2006)*.

- **False positive Rate**

False positive rate is also an useful metric, it indicates what is the proportion of normal transactions being classified as fraudulent. Since there is an huge number of transactions occur every minute, it would be undesirable to constantly freeze normal transactions for further inspections.

Among the above mentioned metrics, I consider AUC, recall and false positive rate the most relevent metrics for evaluation the performance of classifiers for this problem.

II. Analysis

Data Exploration

- **Original Data**

	Time	V1	V2	V3	V4	...	V26	V27	V28	Amount	Class
0	0.00	-1.36	-0.07	2.54	1.38	...	-0.19	0.13	-0.02	149.62	0
1	0.00	1.19	0.27	0.17	0.45	...	0.13	-0.01	0.01	2.69	0
2	1.00	-1.36	-1.34	1.77	0.38	...	-0.14	-0.06	-0.06	378.66	0
3	1.00	-0.97	-0.19	1.79	-0.86	...	-0.22	0.06	0.06	123.50	0
4	2.00	-1.16	0.88	1.55	0.40	...	0.50	0.22	0.22	69.99	0

- **Descriptive Statistics**

	Time	V1	V2	V3	V4	...	V26	V27	V28	Amount	Class
count	284,807.00	284,807.00	284,807.00	284,807.00	284,807.00	...	284,807.00	284,807.00	284,807.00	284,807.00	284,807.00
mean	94,813.86	0.00	0.00	-0.00	0.00	...	0.00	-0.00	-0.00	88.35	0.00
std	47,488.15	1.96	1.65	1.52	1.42	...	0.48	0.40	0.33	250.12	0.04
min	0.00	-56.41	-72.72	-48.33	-5.68	...	-2.60	-22.57	-15.43	0.00	0.00
25%	54,201.50	-0.92	-0.60	-0.89	-0.85	...	-0.33	-0.07	-0.05	5.60	0.00
50%	84,692.00	0.02	0.07	0.18	-0.02	...	-0.05	0.00	0.01	22.00	0.00
75%	139,320.50	1.32	0.80	1.03	0.74	...	0.24	0.09	0.08	77.16	0.00
max	172,792.00	2.45	22.06	9.38	16.88	...	3.52	31.61	33.85	25,691.16	1.00

Obviously, the 28 anonamised features are centered already. Whereas the feature 'Amount' is not only not centered, but has also a totally different scale, which means I have some data pre-processing work to do later on.

- **Missing Values**

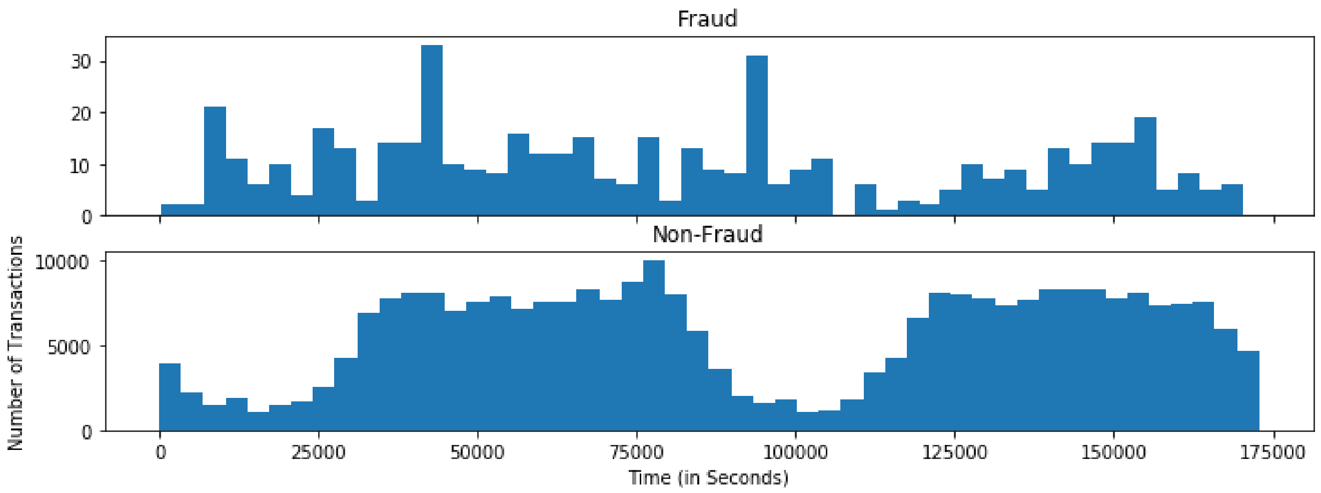
```
data.isnull().sum().sum()
```

```
0
```

There are no missing values in every feature, i.e. 'Time', V1, ..., V29, 'Amount' are missing value free.

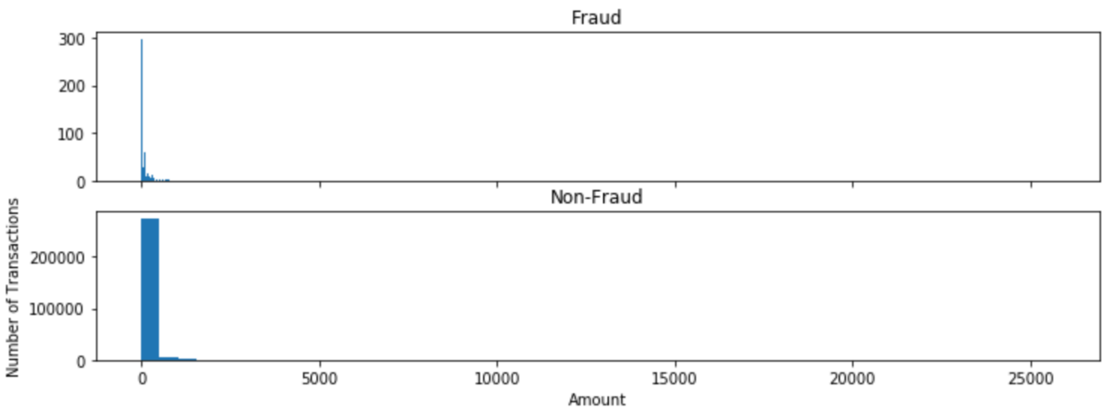
Exploratory Visualization

- **Time**



We observe that, the majority of transactions (non-fraud) were conducted periodically. There is also a vage pattern of fraud transactions, presumly occured during the night.

- **Amount**



Clearly, the amounts need to be transformed such that there is more discrimination.

Algorithms and Techniques

I aim to use simple and interpretable algorithms to tackle the problem of fraud detection, where at the end there are probability outputs of either class. Thus in my opion the following algorithms are potential candidates.

- **Decision Tree**
  This is the most intuitive algorithm, although there are some very technical details about how the tree should grow exactly. Basically it is just as the same as our daily decision makings, where we ask ourselves a sequence of simple questions which can be answered with 'yes' or 'no'. At the end we will reach a decision depending on our answers.

The algorithm was implemented with the default settings from sklearn except that the *random_state* is set to 0, i.e. *(criterion='gini', splitter='best', max_depth=None, min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_features=None, random_state=0, max_leaf_nodes=None, min_impurity_decrease=0.0, min_impurity_split=None, class_weight=None, presort=False)*

- **Gaussian Naive Bayes**
  Gaussian naive Bayes is a popular algorithm in classification problems with approximately Gaussian features, which is our case after some transformations. I will give it a chance and see if it can handle this

job well. Generally, we assume that given the class, the features are independently Gaussian distributed. Thus given features $\mathbf{x} = (x_1, \dots, x_p)$ for an arbitrary class $\mathbf{y}$ we have:

$$\begin{aligned}\mathbb{P}(\mathbf{y}|\mathbf{x}) &= \frac{\mathbb{P}(\mathbf{y})\mathbb{P}(\mathbf{x}|\mathbf{y})}{\mathbb{P}(\mathbf{x})} \\ &= \frac{\mathbb{P}(\mathbf{y})\prod_{i=1}^{p}\mathbb{P}(x_i|\mathbf{y})}{\mathbb{P}(\mathbf{x})} \propto \mathbb{P}(\mathbf{y})\prod_{i=1}^{p}\frac{1}{\sqrt{2\pi\sigma_{\mathbf{y}}^2}}exp\left(-\frac{(x_i-\mu_{\mathbf{y}})^2}{2\sigma_{\mathbf{y}}^2}\right)\end{aligned}$$

where the parameters $\sigma_{\mathbf{y}}$ and $\mu_{\mathbf{y}}$ are estimated using maximum likelihood.

The optimal classification $\hat{\mathbf{y}}$ is determined by the maximum likelihood estimator:

$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y}}\mathbb{P}(\mathbf{y})\prod_{i=1}^{p}\frac{1}{\sqrt{2\pi\sigma_{\mathbf{y}}^2}}exp\left(-\frac{(x_i-\mu_{\mathbf{y}})^2}{2\sigma_{\mathbf{y}}^2}\right)$$

The *prior* $\mathbb{P}(\mathbf{y})$ can either be specified or automatically adjusted according to the heuristic distribution probability of each class in the training data. In my project the latter and default one in *sklearn* was adopted.

- **Logistic Regression**
  Logistic regression with L2 penalty and the hyperparameter $C \in (0, +\infty)$ which controls how important the penalty should be, can be formulated as the following optimization problem:
  $$\min_{w,b}\frac{1}{2}ww^T + C\sum_{i=1}^{n}log(exp(-y_i(X_iw^T+b))+1)$$
  where $X_1, \dots, X_n$ are X training samples.

The probability that a certain sample X_k will be classified as fraud is:

$$\mathbb{P}(Y_k = 1) = \frac{1}{1+e^{-X_kw^T-b}}$$

The algorithm is implemented with the default parameters from sklearn with the exception that *random_state* is set to 0, i.e. *(penalty='l2', dual=False, tol=0.0001, C=1.0, fit_intercept=True, intercept_scaling=1, class_weight=None, random_state=0, solver='liblinear', max_iter=100, multi_class='ovr', verbose=0, warm_start=False, n_jobs=1)*

Benchmark

As mentioned before, I plan to benchmark my model against a neural network, which is trained by Currie32 on Kaggle (https://www.kaggle.com/currie32/predicting-fraud-with-tensorflow) for exactly the same dataset. He claimed to have achieved a *recall* of 82.93% and a false positive rate of 0.10%, i.e. 82.92% of all fraudulent transitions were detected and 0.10% of the non-fraud transactions were falsely reported as fraud.

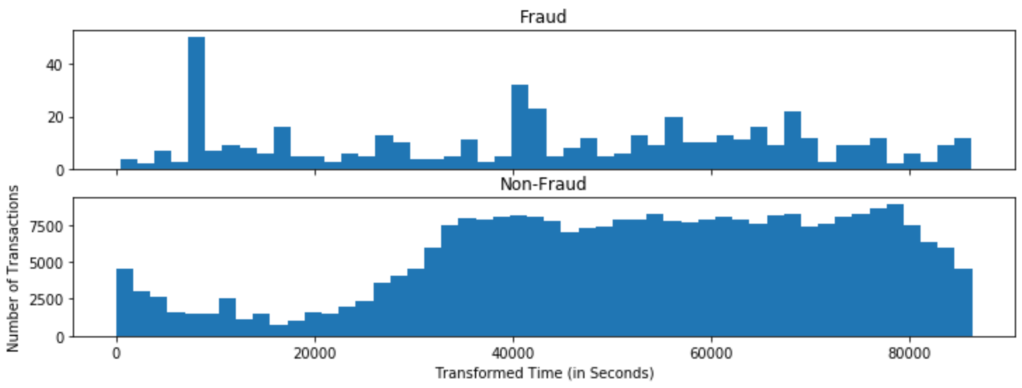
III. Methodology

Data Preprocessing

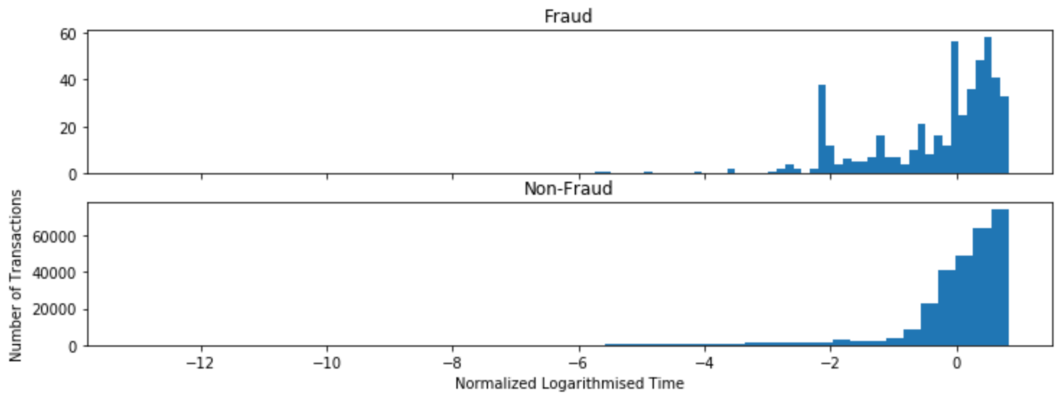
As already mentioned above, there is some pre-processing needed since the features 'Time' and 'Amount' are of much bigger scale than V1, ..., V28.

- **Time**

Clearly, the original feature 'Time' is the time eslapsed in second since the first transaction. Thus we can take the modulo 24x60x60 = 86,400, i.e. the total seconds in a day, to make it periodic. The result after the transformation is

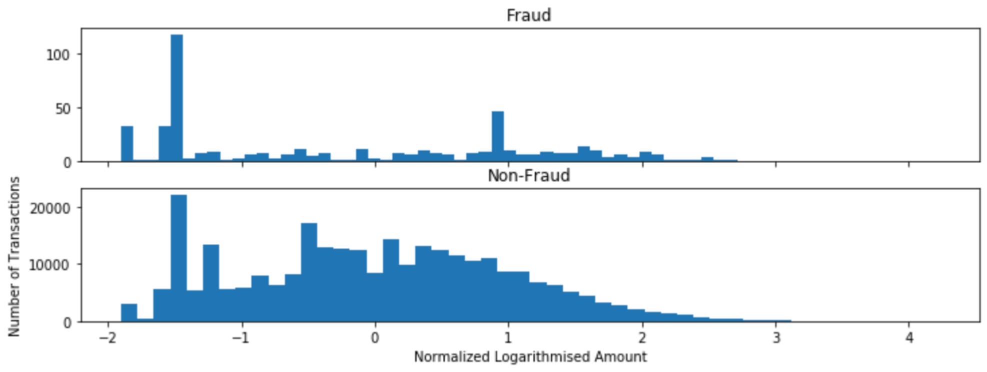


It then needs to be rescaled using logarithmization and then normalization, which results in



The transformed feature 'Time' can now be used for discriminate between *fraud* and *non-fraud*, since they have very different distribution.

- **Amount**
  The feature 'Amount' after logarithmization and normalization has the following distribution:



Implementation

- **Shuffle and Split**
  - Normal Dataset

    The dataset is splited with 80% training and 20% testing data. In order to make the results reproducible, a random state was set.

```
Class = data['Class']
Features = data.drop(['Class'], 1)

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(Features, Class, test_size = 0.2, random_state = 0)

print "Training set has {} samples.".format(X_train.shape[0])
print "Testing set has {} samples.".format(X_test.shape[0])
```

```
Training set has 227845 samples.
Testing set has 56962 samples.
```

- Undersampled Dataset

  Since the dataset is highly imbalanced, so is the training set, i.e. in the training data there is only a fraction of samples which are fraudulent. Which will presumably result in very low *recall* because the classifiers would not be 'well-educated' enough for identifying frauds. The easiest and most efficient way to tackle this problem is the so-called *undersampling*, i.e. the training set will be sampled in a way such that there are 50% *fraud* and 50% *non-fraud* training samples.

```
fraud_indices = data[data.Class == 1].index
non_fraud_indices = data[data.Class == 0].index

random_non_fraud_indices = np.random.choice(non_fraud_indices, len(fraud_indices), replace = False)

under_sample_indices = np.concatenate([fraud_indices, random_non_fraud_indices])

under_sample_data = data.iloc[under_sample_indices, :]

under_sample_Class = under_sample_data['Class']

under_sample_Features = under_sample_data.drop('Class', 1)

from sklearn.model_selection import train_test_split

X_train_under, X_test_under, y_train_under, y_test_under = train_test_split(under_sample_Features,
                                                                            under_sample_Class,
                                                                            test_size = 0.2,
                                                                            random_state = 0)

print "Training set has {} samples.".format(X_train_under.shape[0])
print "Testing set has {} samples.".format(X_test_under.shape[0])
```

```
Training set has 787 samples.
Testing set has 197 samples.
```

  Thus 80% of the fraudulent transactions and the same amount of normal transactions, i.e. 789 samples for each class, will be our training set in the undersampling trial. The testing set which is created here will not be used for the purpose of testing. The classifiers trained on this undersampled training set will be tested on the same testing set from the *normal dataset* above.

- **Pipeline**

I wrote a pipeline, such that I can train different classifiers and obtain the performance metrics on both the training and testing set in an on-line-code manner. The function has the inputs *classifier*, *training features*, *training labels*, *testing features* and *testing labels*. It starts with

```
def train_predict(learner, X_train, y_train, X_test, y_test):

    results = {}

    start = time()
    learner = learner.fit(X_train, y_train)
    end = time()

    results['train_time'] = end - start

    start = time()
    predictions_test = learner.predict(X_test)
    predictions_train = learner.predict(X_train)
    end = time()

    results['pred_time'] = end - start
```

such that we can document the training and prediction time of the classifier. Then it calculates the performance metrics without AUC on the training set

```
    tn, fp, fn, tp = confusion_matrix(y_train, predictions_train).ravel()
    results['fpr_train'] = fp/float(tn + fp)

    results['accuracy_train'] = accuracy_score(y_train, predictions_train)
    results['precision_train'] = precision_score(y_train, predictions_train)
    results['recall_train'] = recall_score(y_train, predictions_train)
    results['f_train'] = fbeta_score(y_train, predictions_train, beta = 0.5)
```

thereafter the performance metrics, all the *fpr* and *fpr* with different thresholds for the *ROC curve* and the *AUC* on the testing set

```
    tn, fp, fn, tp = confusion_matrix(y_test, predictions_test).ravel()
    results['fpr_test'] = fp/float(tn + fp)

    results['accuracy_test'] = accuracy_score(y_test, predictions_test)
    results['precision_test'] = precision_score(y_test, predictions_test)
    results['recall_test'] = recall_score(y_test, predictions_test)
    results['f_test'] = fbeta_score(y_test, predictions_test, beta = 0.5)

    probas_ = learner.predict_proba(X_test)
    fpr, tpr, thresholds = roc_curve(y_test, probas_[:, 1])
    results['AUROC'] = auc(fpr, tpr)
```

at the end the metrics and the *ROC* as output

```
    result = {'metrics': results}
    roc = {'fpr': fpr, 'tpr': tpr}

    results = dict(result = result, roc = roc)

    name = learner.__class__.__name__
    print "{} trained on {} samples.".format(name, X_train.shape[0])

    return results
```

Refinement

In this section, you will need to discuss the process of improvement you made upon the algorithms and techniques you used in your implementation. For example, adjusting parameters for certain models to acquire improved solutions would fall under the refinement category. Your initial and final solutions should be reported, as well as any significant intermediate results as necessary. Questions to ask yourself when writing this section:

- *Has an initial solution been found and clearly reported?*
- *Is the process of improvement clearly documented, such as what techniques were used?*
- *Are intermediate and final solutions clearly reported as the process is improved?*

IV. Results

Model Evaluation

- **Normal Dataset**

  The 3 different classifiers are trained on the normal training set and testing on the normal testing set.

```
from sklearn import linear_model
from sklearn.naive_bayes import GaussianNB
from sklearn import tree

clf_A = linear_model.LogisticRegression(random_state = 0)
clf_B = GaussianNB()
clf_C = tree.DecisionTreeClassifier(random_state = 0)

results = {}
for clf in [clf_A, clf_B, clf_C]:
    clf_name = clf.__class__.__name__
    results[clf_name] = train_predict(clf, X_train, y_train, X_test, y_test)['result']['metrics']
```

```
LogisticRegression trained on 227845 samples.
GaussianNB trained on 227845 samples.
DecisionTreeClassifier trained on 227845 samples.
```

Results are

	DecisionTreeClassifier	GaussianNB	LogisticRegression
AUROC	0.9057	0.9678	0.9783
accuracy_test	0.9992	0.9784	0.9992
accuracy_train	1.0000	0.9777	0.9992
f_test	0.7750	0.0811	0.8226
f_train	1.0000	0.0742	0.8080
fpr_test	0.0004	0.0214	0.0001
fpr_train	0.0000	0.0220	0.0001
precision_test	0.7664	0.0662	0.8889
precision_train	1.0000	0.0604	0.8787
pred_time	0.0344	0.1730	0.0099
recall_test	0.8119	0.8515	0.6337
recall_train	1.0000	0.8235	0.6113
train_time	15.1732	0.1373	2.3275

- ***Decision Tree***

  We observe that, *decision tree classifier* has been highly overfitted and is the most computationally expensive. It has only achieve a *recall* score of 0.8119 and an *AUROC* of 0.9057. Its *false positive rate* is 0.0004, which is decent but not the best.

- ***Gaussian Naive Bayes***

  Gaussian naive Bayes offers the best *recall* score of 0.8515, with the price that the *false positive rate* being the highest, which is 0.0214.

- ***Logistic Regression***

  Logistic Regression has the best performance on the testing set regarding *AUROC* and *false positive rate* with the scores of 0.9783 and 0.0001 respectively. Unfortunately, the *recall* score is only as low as 0.6337. Which means only 63.37% of the fraudulent transactions were successfully detected.

- **Undersampled Dataset**

The same classifiers are trained on the undersampled training set but testing on the normal testing set.

```
clf_A = linear_model.LogisticRegression(random_state = 0)
clf_B = GaussianNB()
clf_C = tree.DecisionTreeClassifier(random_state = 0)

results_under_sample = {}
for clf in [clf_A, clf_B, clf_C]:
    clf_name = clf.__class__.__name__
    results_under_sample[clf_name] = train_predict(clf,
                                                   X_train_under,
                                                   y_train_under,
                                                   X_test,
                                                   y_test)['result']['metrics']

display(pd.DataFrame(results_under_sample))
```

```
LogisticRegression trained on 787 samples.
GaussianNB trained on 787 samples.
DecisionTreeClassifier trained on 787 samples.
```

with the results

	DecisionTreeClassifier	GaussianNB	LogisticRegression
AUROC	0.9422	0.9618	0.9837
accuracy_test	0.8944	0.9617	0.9601
accuracy_train	1.0000	0.9161	0.9479
f_test	0.0204	0.0482	0.0502
f_train	1.0000	0.9473	0.9645
fpr_test	0.1057	0.0381	0.0399
fpr_train	0.0000	0.0259	0.0233
precision_test	0.0164	0.0390	0.0406
precision_train	1.0000	0.9718	0.9762
pred_time	0.0077	0.0332	0.0027
recall_test	0.9901	0.8713	0.9505
recall_train	1.0000	0.8603	0.9202
train_time	0.0121	0.0014	0.0069

- ***Decision Tree***

  This time *decision tree classifier* is still overfitted, however, it has achieve the best *recall* score of 0.9901 on the testing set. On the other hand, it has the worst performance on the testing set in terms of *false positive rate* and *AUROC*, 0.1057 and 0.9422 respectively, meaning that it has identified most frauds by paying the price of falsely categorizing most normal transactions.

- ***Gaussian Naive Bayes***

  Gaussian naive Bayes has achieved the best *false positive rate* of 0.0381 but also the worst *recall* score of 0.8713 on the testing set. It has an *AUROC* of 0.9618.

- ***Logistic Regression***

  Logistic regression performanced best on the testing set regarding *AUROC* with 0.9837. At the same time, its *false positive rate* is 0.0399, only a tick higher than that of Gaussian naive Bayes, while achieving a *recall* score of 0.9505.

- **The Optimal Solution**

  After considering all the metrics comprehensively, I concluded that *logistic regression* with *undersampling* has the best performance on this particular problem.

Model Validation

Now I would like to validate *logistic regression* with 5 different *undersampled training sets* to see if it can be generalized. It is tested on the same normal testing set from before.

```
cv = {}

for i in [1,2,3,4,5]:
    random_non_fraud_indices = np.random.choice(non_fraud_indices, len(fraud_indices), replace = False)

    under_sample_indices = np.concatenate([fraud_indices, random_non_fraud_indices])

    under_sample_data = data.iloc[under_sample_indices, :]

    under_sample_Class = under_sample_data['Class']

    under_sample_Features = under_sample_data.drop('Class', 1)

    X_train_under, X_test_under, y_train_under, y_test_under = train_test_split(under_sample_Features,
                                                                                under_sample_Class,
                                                                                test_size = 0.2)

    cv[i] = train_predict(linear_model.LogisticRegression(random_state = 0),
                          X_train_under,
                              y_train_under,
                                  X_test,
                                     y_test)['result']['metrics']
```

```
LogisticRegression trained on 787 samples.
LogisticRegression trained on 787 samples.
LogisticRegression trained on 787 samples.
LogisticRegression trained on 787 samples.
LogisticRegression trained on 787 samples.
```

The results are:

	1	2	3	4	5	mean	std
AUROC	0.9876	0.9924	0.9901	0.9898	0.9923	0.9905	0.0020
accuracy_test	0.9563	0.9705	0.9653	0.9729	0.9711	0.9672	0.0067
accuracy_train	0.9543	0.9581	0.9441	0.9466	0.9466	0.9499	0.0059
f_test	0.0465	0.0668	0.0579	0.0716	0.0681	0.0622	0.0101
f_train	0.9691	0.9727	0.9583	0.9611	0.9631	0.9649	0.0059
fpr_test	0.0437	0.0295	0.0347	0.0271	0.0289	0.0327	0.0067
fpr_train	0.0209	0.0151	0.0275	0.0251	0.0228	0.0223	0.0047
precision_test	0.0376	0.0542	0.0469	0.0581	0.0553	0.0504	0.0083
precision_train	0.9792	0.9837	0.9699	0.9727	0.9756	0.9762	0.0054
pred_time	0.0021	0.0022	0.0022	0.0022	0.0024	0.0022	0.0001
recall_test	0.9604	0.9505	0.9604	0.9406	0.9505	0.9525	0.0083
recall_train	0.9307	0.9308	0.9147	0.9175	0.9160	0.9219	0.0081
train_time	0.0054	0.0057	0.0058	0.0055	0.0057	0.0056	0.0002

It confirms that *logistic regression* offers very stable performances with different *undersampled training sets*.

Justification

Comparing to the benchmark results, my model offers better *recall* with around 95.25% against 82.92%. However, it has much worse score on *false positive rate* with around 3.27% against 0.10%. I would consider this even with the benchmark results, because although *recall* is improved from 82.92% to 95.25%, the *false positive rate* increased more than 300%. Depending on how important *recall* is, we can now therefore adapt different strategies.

V. Conclusion

Reflection

In my opinion, the most interesting aspect of this project is that we can even achieve better results with less training data because of the skewness in the normal training set. With this project, I am convinced that *undersampling* is indeed an useful technique to deal with imbalanced classification problem.

It is also very important that in the data exploration, we should consider which features need to be transformed such that it makes more sense for the algorithms.

The major difficulty of this project is defining the metrics in a meaningful way. Some metrics, e.g. *accuracy* would not be suitable for imbalanced classification problem. While *AUROC* and *recall* would make more sense.

Improvement

There is an improvement potential if we consider the feature 'Amount' not as a normal feature just like the other, but as a penalty for the mis-classification in the training phase. To be more concrete, we could use the untransformed original amount $\alpha \in [0, +\inf]$ to re-define the optimization problem of *logistic regression* as:

$$\min_{w,b} \frac{1}{2} ww^T + C \sum_{i=1}^{n} log(exp(-\alpha_i \cdot d(y_i, X_i w^T + b)) + 1)$$

i.e. transactions with bigger amount will be weighted more heavily, where $d(.,.)$ is some distance metrics.

Another possibility would be that we fine tune the hyperparameter C using cross validation. Or even we can try different optimization strategies to solve the above problem.

Bibliography

Fawcett, Tom. 2006. “An Introduction to ROC Analysis.” *Pattern Recognition Letters*, ROC Analysis in Pattern Recognition, 27 (8): 861–74. doi:10.1016/j.patrec.2005.10.010.