STAT40800 – Data Programming with Python (Project Strand 1)
Damien Darcy – 15202834

Applying Random Forests to Imbalanced Classification – The case of card fraud detection

Rationale

Imbalanced classification refers to an strongly unequal distribution within a given class in a dataset. This is a significant problem in binary classification, and is evident in areas such as anomaly detection, medical diagnosis and fraud detection, amongst others. An example of this would be cases of a rare disease which occurs once per thousand observations. The dataset for this would show 999 negatives for every positive case . In datasets such as this, the performance of classifiers tends to become biased towards the majority class, thereby reducing the accuracy and predictive power of any given model when applied to the minority class. The issue is exacerbated by the fact that many algorithms emphasise overall accuracy, to which the minority class contributes very little. Similarly, algorithms frequently assume that the dataset has balanced class distributions. All of this can result in classification models which appear to show a high degree of accuracy (as they correctly identify cases of the majority class), but which are somewhat *less accurate* when their performance is applied solely to the minority class.

This assignment applies a number of classification techniques to an imbalanced dataset and discusses the relative accuracy and utility of each of the models in predicting occurences of the minority class. The dataset is initially examined through a random forest classifier. Following parameter tuning, a range of alternative approaches are then implemented and assessed. These are based on several re-sampling techniques derived from the 'imbalanced-learn' module in Python's 'scikit-learn'.

Dataset

The dataset was posted on Kaggle (https://www.kaggle.com/dalpozz/creditcardfraud) in mid-November 2016 and originates from an unnamed European bank. It shows customer transaction data for two days in September 2013. There are a total of 284,807 transactions – of which 492 were found to be fraudulant (or 0.172%). The dataset also contains a number of other numerical input variables which are the result of a PCA transformation. Further information about these features is not provided, however.

Data Description

To begin with the dataset was imported into Python. Both the components and the size of the dataset were verified, and a check was made for missing values. The distribution within the fraud class ('0' where fraud was absent and '1' were it was suspected) was confirmed. A simple correlation between the features of the dataset was implemented. This showed that the fraud class had a significant correlation with features V3, V7, V12 and V32. Interestingly, correlation between cases of fraud and the amount withdrawn, or the time of the withdrawl, was not significant.

Data Cleaning

The columns for 'Time' and 'Amount' were dropped from the dataframe. The response (Y / 'Fraud Class') and the explanatory variables (X / 'V1' – 'V28) were then set out. The data was split into training and test sets using the 'train_test_split' command in 'scikit learn'. A split of 70% / 30% between the sets was implemented, and the 'random_state' was set to '123'. Finally, the 'shape' command was used to verify the split between the two sets. This showed 199364 observations in the training set, with the remaining 85443 observations being assigned to the test set.

<u>Initial Random Forest Model</u>

A random forest classifier model was then fitted to the dataset. This was based on the default values of the command and no attempt was made to define features such the number or depth of the trees, or the class weighting, amongst others. A number of commands from 'scikit learn' were used to examine the results. The most prominent of these was the **'classification_report'**. This shows the '**precision**' (number of true positives divided by the number of true positives and false positives), the '**recall**' (number of true positives divided by the number of true positives and the number of false negatives), and the '**F1 score**' (which shows the balance between between the precision and the recall). The '**confusion_matrix**' command shows the relationship between the predicted and actual class occurrences. Finally, the '**accuracy_score**' command shows the overall classification score of the model. The results for each of these are summarised in ***figures 1 and 2*** below.

Whilst the model shows an accuracy score of **99.94%,** a significant number of false negatives (38) and false positives (9) can be discerned from the confusion matrix. Similarly, although the precision and recall criteria have values of '1' in both cases; the scores are somewhat lower for the minority class (being 0.76 and 0.93 respectively). The model struggles to classify the minority class correctly.

	Precision	**Recall**	**F1 Score**	**# Observations**
0	1.0	1.0	1.0	85312
1	0.76	0.93	0.84	131
Avg / Total	1.0	1.0	1.0	85443

Figure 1 – Classification report for the initial Random Forest model

	Predicted '0'	**Predicted '1'**
Actual '0'	85274	9
Actual '1'	38	122

Figure 2 – Confusion matrix for the initial Random Forest model

<u>Parameter Tuning</u>

Techniques to refine a random forest classifier model include increasing the number of trees in the forest, changing the number of features to consider when looking for the best split, or modifying the maximal depth of the trees. However, such approaches are aimed at improving the overall accuracy of the model, which is already pretty high. Although a number of these approaches were tried out, they had a minimal effect on the models' accuracy for the minority class. A further refinement was to modify the '**class_weight**' variable in the random forest command, obliging the model to place a stronger emphasis on the minority class. The relative weights for 0:1 were set at 0.001:0.999. This reduced the number of false positive from 9 to 5 with an increase of only 1 in the number of false negatives (***figure 3***).

	Predicted '0'	**Predicted '1'**
Actual '0'	85278	5
Actual '1'	39	121

Figure 3 – Confusion matrix for the modified Random Forest model

Re-sampling

The remainder of this assignment contrasts a number of re-sampling techniques aimed at improving the predictive accuracy of the model for the minority class. The aim in all cases is to even up the classes either by deleting instances from the over-represented class (termed as **under-sampling**) or by adding copies of instances from the under-represented class (known as **over-sampling**). The 'imbalanced-learn' package sets out a number of possible techniques in each of these areas. The approach taken here, in each case, is to explain and implement the re-sampling technique prior to applying the random forest model again.

Under-Sampling – Random Independent and Cluster Centoids

The aim here is to reduce the number of observations from the majority class in an attempt to balance the dataset. One approach is '**random under-sampling'**, where the software under-samples the majority class at random (with replacement). An alternative under-sampling approach is to replace a cluster of majority samples by the cluster centroid of a KMeans algorithm – known as '**cluster centroid'** under-sampling.

The former approch results in a balanced sample of the class (332 cases of each). The performance of the model, when re-applied to the random forest, was less than impressive, however (***figures 4 and 5***). Whilst precision for the minority class rose (from 0.76 to 0.91), there was a large decline in recall (from 0.93 to 0.05). A glance at the confusion matrix confirms this. There is a very large number of false positives (3073), combined with a very small number of false negatives (14). The accurcy score of the model falls to 96.38%. A similar problem emerged with the 'cluster centroid' approach (***figure 6***), where the number of false positives grew to 3390. Under-sampling appears to produce acceptable accuracy results for the majority class, but struggles to do the same with the minority class.

	Precision	Recall	F1 Score	# Observations
0	0.96	1.0	0.98	82224
1	0.91	0.05	0.09	3219
Avg / Total	0.96	0.96	0.95	85443

Figure 4 – Classification report for the random under-sampling model

	Predicted '0'	Predicted '1'
Actual '0'	82210	3073
Actual '1'	14	146

Figure 5 – Classification report for the random under-sampling model

	Predicted '0'	Predicted '1'
Actual '0'	82217	3390
Actual '1'	11	125

Figure 6 – Confusion matrix for the cluster centroids under-sampling model

Over-Sampling – Random Independent and SMOTE

An alternative was to use over-sampling instead. In this case, new samples are generated from the minority case. Initially, **random over-sampling** (random replication of samples from the minority case) was carried out. The sample was balanced to have 199032 observations of '0' and an identical number of observations of the '1'. This resulted in an F1 score of 0.86 for the minority case. This represented a small improvement compared to the original random forest and a significant one compared to both under-sampling approaches. It was noticable that this included a relatively low 'precision' score, and a higher 'recall' score, compared to the initial models. The confusion matrix shows a relatively small number of false positives (7) and false negatives (33).

	Precision	Recall	F1 Score	# Observations
0	1.0	1.0	1.0	85309
1	0.79	0.95	0.86	134
Total / Avg	1.0	1.0	1.0	85443

Figure 7 – Classification report random over-sampling

	Predicted '0'	Predicted '1'
Actual '0'	85276	7
Actual '1'	33	127

Figure 8 – Confusion matrix for random over-sampling

A similar, and widely used, technique is known as **SMOTE** (synthetic minority oversampling technique). This creates artificial data based on feature space similarities from minority samples. The effect is to shift the classifier learning bias towards the minority class. This is done through a mixture of bootstrapping and k-nearest neighbours algorithms. The approach takes the difference between the sample and its nearest neighbours. This difference is then multiplied by a random number between 0 and 1, before being added to the original sample. As with random over-sampling, the generated observations were then assessed using a random forest classifier. The results show an improved balance between 'precision' and 'recall'. Both of these remain high for the minority class, but there is a small fall in the F1 score, down to 0.84. There was also an increase in the number of false positives.

	Precision	Recall	F1 Score	# Observations
0	1.0	1.0	1.0	85289
1	0.82	0.86	0.84	154
Total / Avg	1.0	1.0	1.0	85443

Figure 9 – Classification report SMOTE

	Predicted '0'	Predicted '1'
Actual '0'	85261	22
Actual '1'	28	132

Figure 10 – Confusion matrix - SMOTE

A Combined Method – SMOTE ENN

The final approach taken was a modified version of SMOTE, known as **SMOTE-ENN**, again using the imblearn package in scikit-learn. This differs from SMOTE in terms of refinement. The approach removes outliers , defined, in this case, as examples that are misclassifided by their three nearest neighbours. In terms of results, there was a small increase in both 'recall' and the F1 score compared to the conventional SMOTE model.

	Precision	**Recall**	**F1 Score**	**# Observations**
0	1.00	1.00	1.00	85294
1	0.82	0.88	0.85	149
Total / Avg	1.00	1.00	1.00	85443

Figure 11 – Classification report SMOTE-ENN

	Predicted '0'	**Predicted '1'**
Actual '0'	85265	18
Actual '1'	29	131

Figure 12 – Confusion matrix SMOTE-ENN

Conclusion

The aim of this project has been to assess the accuracy of a number of modelling approaches when applied to classifying the minority case of an imbalanced dataset. The emphasis has been on measures such as presicion, recall and the F1 score, more than on the overall accuracy score of each of the approaches. The results have been mixed. The initial random forest approach reached an F1 score of 0.84 with little modification. The models based on under-sampling performed quite poorly, with large numbers of false positives and accordingly low recall scores. The over-sampling approaches did somewhat better, raising both precision and recall compared to the initial model. However, there was little difference in the relative performance of random over-sampling, SMOTE and SMOTEENN – with the first of these models recording the highest F1 score of 0.86. The overall effect of over-sampling was to increase precision (which was a low 0.76) in the initial model and to ensure a better distribution between precision and recall, leading to a small increase in the F1 score. Its' primary effect was to fine tune the original random forest model.

Appendix

```
# STAT 40800 - Student 15202834 - Project / Strand 1


# Import Libraries

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import sklearn

from sklearn.cross_validation import train_test_split

from sklearn.metrics import confusion_matrix

from sklearn.metrics import classification_report

from sklearn.metrics import accuracy_score

from sklearn.ensemble import RandomForestClassifier

from imblearn.under_sampling import ClusterCentroids

from imblearn.under_sampling import RandomUnderSampler #undersampling

from imblearn.over_sampling import SMOTE

from imblearn.over_sampling import RandomOverSampler  #oversampling

from imblearn.combine import SMOTEENN

from sklearn.metrics import roc_curve, auc


#Data Cleaning / Description


card=pd.read_csv('C:/Users/ITS/Desktop/Data_Fraud.csv')

card.head()

card.describe()


# How imbalanced is the dataset?
```

```
card['Class'].value_counts()


# Check for missing values

card.isnull().values.any()


# Simple correlation

columns = ['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',

    'Class', 'Time', 'Amount']

correlation = card[columns].corr(method='pearson')


# Test and Training


card.drop(['Time','Amount'], axis=1, inplace=True)


Y = card['Class']

X = card.drop("Class",axis=1)


# Usoing test_train_split for test and training sets - 70/30 split. Random number generator

X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3, random_state=123)


# Check the relative number of observations


print(X_train.shape, X_test.shape)

print(Y_train.shape, Y_test.shape)
```

```
#Random Forest - Attempt 1


rfc = RandomForestClassifier()

rfc.fit(X_train,Y_train)

Y_pred = rfc.predict(X_test)


# Show classification report, confusion matrix, accuracy score

print(classification_report(Y_pred,Y_test))

print(confusion_matrix(Y_test, Y_pred))

print(accuracy_score(Y_test,Y_pred))


#Parameter Tuning - Random Forest - Attempt 2


# Weighting in favour of the minority class in an attempt to improve accuracy

rfc = RandomForestClassifier(class_weight={0:0.001,1:0.999})

rfc.fit(X_train,Y_train)

Y_pred = rfc.predict(X_test)


print(classification_report(Y_pred,Y_test))

print(confusion_matrix(Y_test, Y_pred))

print(accuracy_score(Y_test,Y_pred))


#Random Re-Sampling (under-sampling)


rus = RandomUnderSampler()

rux, ruy = rus.fit_sample(X_train, Y_train)
```

```
# Sample counts following under-sampling

unique, counts = np.unique(ruy, return_counts=True)

print (np.asarray((unique, counts)).T)


# Apply re-sampled data to RFC

rfc = RandomForestClassifier()

rfc.fit(rux,ruy)

Y_rus_pred = rfc.predict(X_test)


# Performance not so good on minority class

print(classification_report(Y_rus_pred,Y_test))

print(confusion_matrix(Y_test, Y_rus_pred))

print(accuracy_score(Y_test,Y_rus_pred))


#Under-sampling via ClusterCentroids


CC = ClusterCentroids()

ccx, ccy = CC.fit_sample(X_train, Y_train)


# Again - size / shape of resampled data

unique, counts = np.unique(ccy, return_counts=True)

print (np.asarray((unique, counts)).T)


# It takes a long time to run ....

rfc = RandomForestClassifier()

rfc.fit(ccx,ccy)

Y_cc_pred = rfc.predict(X_test)
```

```
print(classification_report(Y_cc_pred,Y_test))

print(confusion_matrix(Y_test, Y_cc_pred))

print(accuracy_score(Y_test,Y_cc_pred))


#Random over-sampling - with replacement

ros = RandomOverSampler()

rox, roy = ros.fit_sample(X_train, Y_train)


# What is the size of the re-sampled dataset?

unique, counts = np.unique(roy, return_counts=True)

print (np.asarray((unique, counts)).T)


rfc = RandomForestClassifier()

rfc.fit(rox,roy)

Y_ros_pred = rfc.predict(X_test)


print(classification_report(Y_ros_pred,Y_test))

print(confusion_matrix(Y_test, Y_ros_pred))

print(accuracy_score(Y_test,Y_ros_pred))


#SMOTE / SMOTEENN


#SMOTE

smote = SMOTE(ratio='auto', kind='regular')

smox, smoy = smote.fit_sample(X_train, Y_train)
```

```
unique_smote, counts_smote = np.unique(smoy, return_counts=True)

print (np.asarray((unique_smote, counts_smote)).T)


rfc.fit(smox,smoy)

y_smote_pred = rfc.predict(X_test)

print(classification_report(y_smote_pred,Y_test))

print(confusion_matrix(Y_test, y_smote_pred))

print(accuracy_score(Y_test,y_smote_pred))


#SMOTEENN


# This has an additional data cleaning step to remove outliers

SENN = SMOTEENN(ratio = 'auto')

ennx, enny = SENN.fit_sample(X_train, Y_train)

unique_enny, counts_enny = np.unique(enny, return_counts=True)

print (np.asarray((unique_enny, counts_enny)).T)


rfc.fit(ennx, enny)

y_senn_pred = rfc.predict(X_test)

print(classification_report(y_senn_pred,Y_test))

print(confusion_matrix(Y_test, y_senn_pred))

print(accuracy_score(Y_test,y_senn_pred))



#Finally - ROC / AUC for the three over-sampling approaches


false_positive_rate, true_positive_rate, thresholds = roc_curve(Y_test,Y_ros_pred)
```

```
roc_auc=auc(false_positive_rate, true_positive_rate)

print(roc_auc)


false_positive_rate, true_positive_rate, thresholds = roc_curve(Y_test,y_smote_pred)

roc_auc=auc(false_positive_rate, true_positive_rate)

print(roc_auc)


false_positive_rate, true_positive_rate, thresholds = roc_curve(Y_test,y_senn_pred)

roc_auc=auc(false_positive_rate, true_positive_rate)

print(roc_auc)


# Plot of the SMOTEENN version


plt.title('Receiver Operating Characteristic')

plt.plot(false_positive_rate, true_positive_rate, 'b', label='AUC=%0.2f'% roc_auc)

plt.legend(loc='lower right')

plt.plot([0,1],[0,1],'r--')

plt.xlim([-0.1,1.2])

plt.ylim([-0.1,1.2])

plt.ylabel('True Positive Rate')

plt.xlabel('False Positive Rate')

plt.show()
```