Rare Event Modelling

Week 8 - Rare Event Modeling

Any classification problem where the minority event is less than 10% of the data is referred to as a **rare event problem**. Modeling rare events is challenging since the usual approach for modeling classification problems will tend to ignore the minority event. Since the percentage of this event is less than 10%, most classification models will tend show acceptable classification metrics, often with an F1 score above 90%. If the percent of misclassification is the only concern then almost any classification model produce acceptable results with misclassification less than 10% and F1 greater than 90%.

However, in most cases, the concern is over the loss associated with misclassification. That is, although there are only a few minority events, they can be costly, and it is the cost associated with these events that needs to be reduced. Often one of the misclassifications, either false positive or false negative is more costly than the other. The aim is to develop a model that reduces the most costly misclassifications.

This requires a calculation of loss for each case. In some cases the loss is constant such as:

Loss = $5 if the case is a false positive, or Loss = $3 if the case is a false negative.

In this case the total loss is:

Total Loss = $3 x (number of false negatives) + $5 x (number of false positives).

With is loss function, the model will be rewarded more for reducing false positives than false negatives. Unless this loss is made known to the model algorithm, the model will try to reduce the total number of false negatives and positives, without regard for the fact that false positives are more of a concern than false negatives.

In more complex applications, the loss is not constant across all cases. It varies by case, and it usually is calculated from part of the data. For example, if the loss is a function of not just the error type, but the amount of the transaction, the loss or each observation might look like the following:

Loss = $5 x Amount, if the case is a false positive, or Loss = $3 x Amount, if the case is a false negative. In this case, the total loss cannot be calculated directly from the total number of false positive and false negatives. It must be calculated for each observation, and then totaled over the entire dataset.

Analysis in this case, can be done using random under sampling, **RUS**, to build the best model that reduces the total loss, rather than the total misclassifications. In general, an RUS model will have a higher misclassification error rate, but the total loss will be lower because RUS will place more importance on reducing the misclassification of higher loss cases.

RUS involved building a model using two steps:

Step 1: Identify the best mixture of the minority and majority events, and

Step 2: Using the best mixture build an ensemble model for classification.

There are other issues. The model type needs to be identified and its parameters tuned. The usual choices are logistic regression, decision tree, random forest, and neural network. The selection and tuning are done before using RUS. Often this is done by first using hyperparameter 10-fold optimization to identify the model type and it's parameters that work best for these data. Best in this case is the model that lowers the total loss.

The following notes are an example of using RUS to classify loan applicants as 'good' or 'bad' credit risk. In this example, we assume that hyperparameter, 10-fold optimization identifed logistic regression as the best model for reducing loss with these data.

Data

The data consists of 10,500 credit applications, each classified as 'good' or 'bad' credit. However, there are only 500 'bad' credit applications. Since this is only 5% of the data, classifying applicants as 'good' or 'bad' credit is called a rare event problem.

In Python, there is a package that is available for constructing a random sample with user specified mixtures of the majority and minority events. However, this package is not installed in Anaconda. Install this package using the following **conda** command:

conda install -c glemaitre imbalanced-learn

This example uses the two-step approach to RUS described in class. First the best ratio is discoved by trying ratios between 50:50 to 90:10. Sometimes the original data is included as a ratio. In this case that would be approximately 95:5, and we would expect the loss from a model built using the original data to be higher than the others.

The second step is to build an ensemble model based on the selected ratio. This done my creating several datasets using the selected ratio, fitting a model to each, making classification probability predictions for each and then averaging those to get predicted classification probability. From that we can calculate the total loss over the enire dataset.

In this example, the base model is logistic regression with parameters designed to produce solutions similar to most other statistical software, software that does not use regularization. Hence the regularization parameter is set to C=1e+15.

Import Packages

After the **imblearn** package is installed using conda, it is imported along with the other packages need to read the data and fit logistic regression models.

In [1]:

```
# Install using Conda:
# conda install -c glemaitre imbalanced-learn
from imblearn.under_sampling import RandomUnderSampler

import pandas as pd
import numpy  as np
from Class_replace_impute_encode import ReplaceImputeEncode
from Class_regression import logreg
from sklearn.linear_model import LogisticRegression
```

Next we construct the following function to calculate loss and the confusion matrix for our models. These is useful since the loss calculations are a function of the **Amount** of the loan application. If the case is correctly classified by the model, the loss is zero. Otherwise the loss is a function of the loan amount, which is different for false positives and negatives.

Loss = Amount, if the case is a false positive, or **Loss** = 0.1 x Amount, if the case is a false negative.

False positives are applications that were classified as 'good' but the customer later defaulted on the loan. In that case, the entire amount of the loan is treated as the loss. In practice, this amount might be adjusted by the actual loss for the load which would be the loan amount minus payments, plus some overhead costs. Here, we are just using the unadjusted loan amount.

False negatives are applications that were classified as 'bad' by should have been classified as 'good'. That is the customer paid off the loan in a timely fashion, but the model is saying they should be denied a loan.

In this function, the numpy array **y** is the actual classification for each case. It is coded using zeros for the 'bad' classifications and one for the 'good' classifications. This happens by default since alphabetically 'bad' occurs before 'good'. If instead of 'bad' and 'good', the data used 'yes' and 'no', respectively, then the 'bad' classifications would have been coded as ones. This distinction is important and makes a difference in how this function is written and interpreted.

Function for Calculating Losses and Confusion Matrix

In [2]:

```
# Function for calculating loss and confusion matrix
def loss_cal(y, y_predict, fp_cost, fn_cost, display=True):
    loss     = [0, 0]       #False Neg Cost, False Pos Cost
    conf_mat = [0, 0, 0, 0] #tn, fp, fn, tp
    for j in range(len(y)):
        if y[j]==0:
            if y_predict[j]==0:
                conf_mat[0] += 1 #True Negative
            else:
                conf_mat[1] += 1 #False Positive
                loss[1] += fp_cost[j]
        else:
            if y_predict[j]==1:
                conf_mat[3] += 1 #True Positive
            else:
                conf_mat[2] += 1 #False Negative
                loss[0] += fn_cost[j]
    if display:
        fn_loss = loss[0]
        fp_loss = loss[1]
        total_loss = fn_loss + fp_loss
        misc    = conf_mat[1] + conf_mat[2]
        misc    = misc/len(y)
        print("{:.<23s}{:10.4f}".format("Misclassification Rate", misc))
        print("{:.<23s}{:10.0f}".format("False Negative Cost", fn_loss))
        print("{:.<23s}{:10.0f}".format("False Positive Cost", fp_loss))
        print("{:.<23s}{:10.0f}".format("Total Loss", total_loss))
    return loss, conf_mat
```

Data Map

The attribute map for these data was discussed in a previous example. These data are based upon the credit application used in previous examples. However, the attribute 'purpose' was removed since it has many levels and poor association with the credit target.

There are no outlier or missing values in these data, but the map is needed to properly encode the nominal attributes, including the target **good_bad**.

In [3]:

```
# Attribute Map for CreditData_RareEvent.xlsx, N=10,500
attribute_map = {
    'age':[0,(1, 120),[0,0]],
    'amount':[0,(0, 20000),[0,0]],
    'duration':[0,(1,100),[0,0]],
    'checking':[2,(1, 2, 3, 4),[0,0]],
    'coapp':[2,(1,2,3),[0,0]],
    'depends':[1,(1,2),[0,0]],
    'employed':[2,(1,2,3,4,5),[0,0]],
    'existcr':[2,(1,2,3,4),[0,0]],
    'foreign':[1,(1,2),[0,0]],
    'good_bad':[1,('bad', 'good'),[0,0]],
    'history':[2,(0,1,2,3,4),[0,0]],
    'housing':[2,(1, 2, 3), [0,0]],
    'installp':[2,(1,2,3,4),[0,0]],
    'job':[2,(1,2,3,4),[0,0]],
    'marital':[2,(1,2,3,4),[0,0]],
    'other':[2,(1,2,3),[0,0]],
    'property':[2,(1,2,3,4),[0,0]],
    'resident':[2,(1,2,3,4),[0,0]],
    'savings':[2,(1,2,3,4,5),[0,0]],
    'telephon':[1,(1,2),[0,0]] }
```

Data Preprocessing

Read the data using Pandas then encode the categorical attributes using one-hot encoding. Drop the last one-hot column since the base model is logistic regression.

In [4]:

```
file_path = '/Users/Home/Desktop/python/Excel/'
df = pd.read_excel(file_path+"CreditData_RareEvent.xlsx")
# Encode for Logistic Regression, drop last one-hot column
rie = ReplaceImputeEncode(data_map=attribute_map, nominal_encoding='one-hot', \
                          interval_scale = None, drop=True, display=False)
encoded_df = rie.fit_transform(df)
# Create X and y, numpy arrays
# bad=0 and good=1
```

```
y = np.asarray(encoded_df['good_bad']) # The target is not scaled or imputed
X = np.asarray(encoded_df.drop('good_bad',axis=1))
```

Calculate Potential Loss for Each Case

For each the potential loss is the false positive and false negative loss calculated assuming the case might be classifed as false positive or negative. In most cases the model will correctly classify the observation and the actual loss will be zero. The potential loss is calculated before any model is developed just in case that model incorrectly classifies the observation.

In this example the potential losses are calculated using the following formulas, one for false positive and another for false negatives:

fp_cost = Amount, and

fn_cost = 0.1 x Amount

In [5]:

```
# Setup false positive and false negative costs for each transaction
fp_cost = np.array(df['amount'])
fn_cost = np.array(0.1*df['amount'])
```

Calculate Total Loss without RUS

As a benchmark, it is good to know the results from fitting the entire dataset without using RUS. The following code evaluates fitting the entire dataset using a standard logistic regression model built using all predictors.

In [6]:

```
lgr = LogisticRegression()
lgr.fit(X, y)
print("\nLogistic Regression Model using Entire Dataset")
col = rie.col
col.remove('good_bad')
logreg.display_binary_metrics(lgr, X, y)
print("\nLoss Calculations from a Logistic Regression")
print("Model fitted to the entire dataset:")
loss, conf_mat = loss_cal(y, lgr.predict(X), fp_cost, fn_cost)
```

```
Logistic Regression Model using Entire Dataset

Model Metrics
Observations..............     10500
Coefficients..............        46
DF Error..................     10454
Mean Absolute Error.......    0.0848
Avg Squared Error.........    0.0408
Accuracy..................    0.9530
Precision.................    0.9529
Recall (Sensitivity)......    1.0000
F1-Score..................    0.9759
MISC (Misclassification)...      4.7%
     class 0..............     98.8%
     class 1..............      0.0%


     Confusion
       Matrix     Class 0   Class 1
Class 0.....         6       494
Class 1.....         0     10000

Loss Calculations from a Logistic Regression
Model fitted to the entire dataset:
Misclassification Rate.    0.0470
False Negative Cost....         0
False Positive Cost....   2027463
Total Loss.............   2027463
```

In this example, the logistic regression model fitted to the entire dataset, has great quality metrics for handling misclassifications. The misclassification rate if 4.7%. The F1-score is 97.6%, and the accuracy is 95%! However, notice that the 'bad' credit applicants are being ignored. Almost all of the 500 applicants with 'bad' credit are being classified as 'good'. The model is classifing all but 6 applicants as 'good' credit risks.

From the prespective of the quality metrics this is an excellent model, but from the prespective of a banker who needs to reduce loss from bad loans, this is a terrible model.

The estimate total loss from this model is over $2 million, and it is all from false positives, classifying applicants with 'bad' credit as 'good'.

RUS - Step 1

Building a model that decreases this loss involves using **RUS**. The first step is to identify the best mixture of majority and minority event applicants. The data represents a mixture of 95%:5%, approximately. This example considers mixtures of 50:50, 60:40, 70:30, 80:20 and 90:10.

In Python, instead of passing these ratios into the RUS routine, it is necessary to pass the actual number of observations represented by these mixtures. For example, the 50:50 ratio is a sample constructed using all 500 'bad' applicants and a random sample of an additional 500 'good' applications. The total sample is 1,000 applications evently divided between 'bad' and 'good'.

A mixture of 80:20 would have 80% 'good' and 20% 'bad'. Since each RUS sample is constructed using all of the minority data, all 500 of the 'bad' applicants, the number of randomly selected 'good' applicants will need to be 4 times larger. That is, the number of randomly selected good applications to achieve an 80:20 ratio of 'good' to 'bad' cases is calculated by: (0.8/0.2) x 500 = 2,000.

After making these calculations for each ratio, we create list containing the random seeds we would like to use, **rand_val**, a list of ratios, **ratio**, and a tuple, **rus_ratio**, containing dictionaries that describe the number of observations for each ratio.

In [7]:

```
# Setup random number seeds
rand_val = np.array([1, 12, 123, 1234, 12345, 654321, 54321, 4321, 321, 21])
# Ratios of Majority:Minority Events
ratio = [ '50:50', '60:40', '70:30', '80:20', '90:10' ]
# Dictionaries contains number of minority and majority events in each ratio sample
# n_majority = ratio x n_minority
rus_ratio = ({0:500, 1:500}, {0:500, 1:750}, {0:500, 1:1167}, {0:500, 1:2000}, {0:500, 1:4500})
```

Now evaluate 10 randomly selected samples for each ratio. Keep track of the best ratio using the variable **best_ratio** and the minimum loss in **min_loss**.

In [8]:

```
# Best model is one that minimizes the loss
min_loss   = 9e+15
best_ratio = 0
for k in range(len(rus_ratio)):
    rand_vals = (k+1)*rand_val
    print("\nLogistic Regression Model using " + ratio[k] + " RUS")
    fn_loss = np.zeros(len(rand_vals))
    fp_loss = np.zeros(len(rand_vals))
    misc    = np.zeros(len(rand_vals))
    for i in range(len(rand_vals)):
        rus = RandomUnderSampler(ratio=rus_ratio[k], \
                random_state=rand_vals[i], return_indices=False, \
                replacement=False)
        X_rus, y_rus = rus.fit_sample(X, y)
        lgr = LogisticRegression(C=1e+15, tol=1e-8, \
                                 max_iter=300, solver='lbfgs')
        lgr.fit(X_rus, y_rus)
        loss, conf_mat = loss_cal(y, lgr.predict(X), fp_cost, fn_cost,\
                                  display=False)
        fn_loss[i] = loss[0]
        fp_loss[i] = loss[1]
        misc[i]    = conf_mat[1] + conf_mat[2]
    misc = np.sum(misc)/(10500 * len(rand_vals))
    fn_avg_loss = np.average(fn_loss)
    fp_avg_loss = np.average(fp_loss)
    total_loss  = fn_loss + fp_loss
    avg_loss    = np.average(total_loss)
    std_loss    = np.std(total_loss)
    print("{:.<23s}{:10.4f}".format("Misclassification Rate", misc))
    print("{:.<23s}{:10.0f}".format("False Negative Cost", fn_avg_loss))
    print("{:.<23s}{:10.0f}".format("False Positive Cost", fp_avg_loss))
    print("{:.<23s}{:10.0f}{:5s}{:<10.2f}".format("Total Loss", avg_loss, \
                  " +/- ", std_loss))
    if avg_loss < min_loss:
        min_loss   = avg_loss
        best_ratio = k
```

```
Logistic Regression Model using 50:50 RUS
Misclassification Rate.    0.2795
False Negative Cost....   1056193
False Positive Cost....    397936
Total Loss.............   1454129 +/- 26010.38

Logistic Regression Model using 60:40 RUS
Misclassification Rate.    0.1942
False Negative Cost....    738964
False Positive Cost....    551239
Total Loss.............   1290203 +/- 47674.11

Logistic Regression Model using 70:30 RUS
Misclassification Rate.    0.1351
False Negative Cost....    506224
False Positive Cost....    770320
Total Loss.............   1276544 +/- 60951.42

Logistic Regression Model using 80:20 RUS
Misclassification Rate.    0.0802
False Negative Cost....    234642
False Positive Cost....   1201360
Total Loss.............   1436002 +/- 113935.79

Logistic Regression Model using 90:10 RUS
Misclassification Rate.    0.0495
False Negative Cost....     60124
False Positive Cost....   1612461
Total Loss.............   1672585 +/- 34589.45
```

RUS - Step 2 - Ensemble Model

From the first analysis, it is clear that the best ratio identified is 70:30 with an estimated loss of $1.3 million. The final step in RUS modeling is to build an ensemble model using the best ratio, and to calculate a better estimate of the total loss from this model.

An ensemble model, in this case, is the average of several models, each developed using the same best ratio, but with different random samples. Each 70:30 ratio uses all 500 case from the 'bad' applicants and then an additional 1,167 cases randomly selected from the remaining 10,000 'good' applicants. Since these are randomly selected, each sample created produces different estimates of the logistic regression model.

In this example, 100 separate samples will be created. Each will produce different estimates of for the probability of the probably that the applicant is a 'bad' credit risk. The ensemble model average the 100 estimates for each of the 10,500 cases in the data. These average probabilities are used to classify the data and finally to evaluate the misclassification rate and total loss.

In [9]:

```
# Ensemble Modeling - Averaging Classification Probabilities
avg_prob = np.zeros((len(y),2))
# Setup 100 random number seeds for use in creating random samples
np.random.seed(12345)
max_seed = 2**32 - 1
rand_value = np.random.randint(1, high=max_seed, size=100)
# Model 100 random samples, each with a 70:30 ratio
for i in range(len(rand_value)):
    rus = RandomUnderSampler(ratio=rus_ratio[best_ratio], \
                    random_state=rand_value[i], return_indices=False, \
                    replacement=False)
    X_rus, y_rus = rus.fit_sample(X, y)
    lgr = LogisticRegression(C=1e+15, tol=1e-8, \
                             max_iter=300, solver='lbfgs')
    lgr.fit(X_rus, y_rus)
    avg_prob += lgr.predict_proba(X)
avg_prob = avg_prob/len(rand_value)
# Set y_pred equal to the predicted classification
y_pred = avg_prob[0:,0] < 0.5
y_pred.astype(np.int)
# Calculate loss from using the ensemble predictions
print("\nEnsemble Estimates based on averaging",len(rand_value), "Models")
loss, conf_mat = loss_cal(y, y_pred,fp_cost,fn_cost)
```

```
Ensemble Estimates based on averaging 100 Models
Misclassification Rate.    0.1325
False Negative Cost....    487478
False Positive Cost....    766540
Total Loss.............   1254018
```