Imbalanced Classification Problem

A credit card fraud detection example

Mahesh Naidu
Electrical and Computer Engineering
Texas A&M University
College Station, TX 77840 USA
mah12man@tamu.edu

Dhananjay patil
Industrial Engineering
Texas A&M University
College Station, TX 77840 USA
Patildp1992@gmail.com

***Abstract*—This report describes our research project in the course Statistical Computing in R & Python at Texas A&M University in which we took up a challenging topic of studying a classification problem wherein the data set chosen is highly imbalanced. Through this project, we have made a humble effort in understanding an unbalanced dataset, transforming it into a form that can be utilized for classification and deriving valuable insights from it. However, the primary aim of this project would be to try different linear and non- linear classification algorithms to detect fraudulent credit card transaction. We will also perform tuning of hyperparameters for each of the models. Eventually the project is focused on finding a model that could accurately predict fraudulent transactions which in turn would aid authorities/ institutions in keeping a check on suspicious banking activities.**

Keywords— Unbalanced Data, SVM, kNN, Logistic Regression, Hyperparameters

I. Introduction

With digitization of banking services coupled with latest advancements in technology, banking has become far more easy and convenient. However, as banking services have become more advanced and handle far more number of operations than they did before, they've become prone to a greater number of fraudulent and unlawful practices. One such problem that we've have focused on is fraudulent credit card transactions and we've sought a dataset (https://www.kaggle.com/mlg-ulb/creditcardfraud) to study this problem. Our objective is to obtain a classification model/ technique that can best detect such fraudulent transactions.

Machine Learning and classification techniques like Logistic Regression, Support Vector Regression (SVR), Random forest are widely used for data driven learning and prediction. With the immense success of machine learning and data mining approaches, many have been tempted to apply these methods in different settings. These techniques also allow for quick grasping of general trends and other aspects of the data.

Classification algorithms sometimes do not give the best results. One solution is to gather more data and perform feature engineering. However, gathering more data has usually has the greatest payoff in terms of time invested versus improved performance. Hence, modelling hyperparametric tuning takes centerstage. A Hyperparameter is a parameter whose value is set before the learning process begins. By contrast, the model parameters ate learned during training such as the slope and intercept in a linear regression. Scikit-Learn implements a set of sensible default Hyperparameters for all models. The best hyperparameters are usually impossible to determine ahead of time, and therefore the model hyperparameters are tuned through a trial-and-error based method. The optimal settings are determined by trying different combinations by evaluating the performance of each model. However, evaluating each model on the training set leads to the problem of overfitting. Overfitting occurs when the model performs well on the training set but poorly on the test set. The model knows the training set well but cannot be applied to newer problems.

To avoid this problem, hyperparametric optimization has been performed by fitting the models using cross validation. We have used the K-fold CV method for cross validation. In cross validation, the training data (obtained after splitting the original dataset into train and test) is split into K number of subsets called folds. We then iteratively fit the model K times, each time training the data on K-1 of the folds and evaluating on the Kth fold. At the very end of training, the performance on each of the folds is averaged in order to come up with final validation metrics for the model.

In this project, Hyperparametric tuning of 2 Linear and 2 non- Linear classification algorithms has been performed using cross validation. The optimal hyperparametric settings are retained which then are applied to the testing dataset.

II. Data Set

The dataset has been obtained from Kaggle. The dataset contains transactions made by credit cards in September 2013 in Europe. This dataset presents transactions that occurred in two days, in which there are 492 frauds out of 284,807 transactions. The dataset is highly unbalanced, the positive class (frauds) only account for 0.172% of all transactions.

It contains only numerical input variables which are the result of a PCA transformation. Due to confidentiality issues, the original feature and other sensitive information is hidden. 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. Feature 'Class' is the response variable or the target variable and it takes value 1 in case of fraud and 0 otherwise.

The unbalanced data is converted to balanced one using SMOTE. The pandas_ml package is used for this purpose. It is a package that integrates pandas, scikit- learn for easy handling of data and creation of machine learning models. The minority class has been 'over-sampled' to the size of the majority class. Each class have 284315 samples. However, since the total size of data goes over 400,000 points, for ease of computation, it has been reduced to 50,000 points using random sampling keeping the number of samples under each class (of the response variable) equal. The balanced data is now subjected to train & test split using the function *train_test_split*. The model is fitted on to the training data and tested on the test data.

III. Prediction Methodology

The following models were trained on the training data for prediction and subsequently tested on the test data. The module for logistic regression can be imported from scikit- learn package. The logistic regression model gave an accuracy of 94.4 percent. Accuracy measures the extent to which the set of labels predicted for a sample exactly match the corresponding set of labels having the true values. Its confusion matrix shows 7361 points of negative class (class 0) correctly matched and 636 values incorrectly matched. Similarly, for class 1, 6806 points are correctly matched, and 197 points are incorrectly matched.

	Predicted -ve	Predicted +ve
Actual Negative	7361	636
Actual Positive	197	6806

The classification report metrics builds a text report showing the main classification metrics. It returns the following- 1) Precision value- The precision is the ratio *tp/(tp+fp)* where tp is the number of true positives and fp the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative**.** 2)Recall: The recall is the ratio *tp/(tp + fn)*. The recall is intuitively the ability of the classifier to find all the positive samples. 3)f1-score: The f1 score can be interpreted as the weighted harmonic mean of the precision and recall, where it reaches the best value at 1 and the worst at 0. The classification report of the logistic regression model is given in the following figure:

	precision	recall	F1 score
0	0.97	0.92	0.95
1	0.91	0.97	0.94
Avg/total	0.95	0.94	0.94

SVM: The module for SVM can be imported from sklearn. Since the data is tested on both linear and non-linear classification techniques, we have chosen the 'linear' kernel for SVM.

Random Forest (Non-Linear classification): The module for Random Forest is *RandomForestClassifier* belonging to the library- *sklearn.ensemble*. The accuracy of random forest classification is 99.4 percent. The parameter njobs=-1 describes that there is no bound on the number of processors that can be used for this computation.

kNN (Non- Linear classification): The module for kNN is KNeighborsClassifier belonging to the library sklearn.neighbors. The value of k (default) assumed by this function is 5. The accuracy of kNN is 99.2 percent.

The combined classification report matrices and confusion matrices for all the above classification modules are shown in the following tables-

Model		Predicted-ve	Predicted +ve
Logistic Reg.	Actual -ve	7361	636
	Actual +ve	197	6806
SVM	Actual -ve	7374	666
	Actual +ve	184	6776
R.Forest	Actual -ve	7541	60
	Actual +ve	17	7382
kNN	Actual -ve	7443	0
	Actual +ve	115	7442

The table below describes the classification report metrics of the discussed classifiers:

Model		Precision	Recall	F1- score
Log. Reg.	0	0.97	0.92	0.95
	1	0.91	0.97	0.94
	avg	0.95	0.94	0.94
SVM	0	0.98	0.92	0.95
	1	0.91	0.97	0.94

	Avg.	0.95	0.94	0.94
R. Forest	0	1.00	0.99	0.99
	1	0.99	1.00	0.99
	Avg.	0.99	0.99	0.99
kNN	0	0.98	1	0.99
	1	1	0.98	0.99
	Avg.	0.99	0.99	0.99

IV. TUNING OF HYPERPARAMETERS

The hyperparametric tuning of all models discussed before have been performed by fitting data using cross validation. Grid Search has been used to exhaustively generate options from a grid of parameter values specified within the *param_grid* parametric set. The Grid search is provided by *GridSearchCV.* On fitting a model, it evaluates all possible combinations of parameter values and the best one is retained. The model retaining the optimal combination is then used to predict the test variable.

Logistic regression modelling with Hyperparametric tuning: Logistic regression has a hyperparameter 'C' ,also known as regularization parameter, that controls the inverse of the regularization strength. This parameter has been tuned in this part. A large C can lead to an *overfit* model, while a small C can lead to an *underfit* model. Below is a snippet of the hyperparametric tuning implementation-

```
# Setup the hyperparameter grid
c_space = np.logspace(-5, 8, 15)
param_grid = {'C': c_space}

# Instantiate a logistic regression classifier: logreg
logreg = LogisticRegression()

# Instantiate the GridSearchCV object: logreg_cv
logreg_cv = GridSearchCV(logreg, param_grid, n_jobs=-1)
```

The hyperparametric grid *c_space* specifies the range of values that C can assume. The optimal value of C obtained is 31.622. The accuracy of the model so obtained is 94.47 percent.

SVM modelling with Hyperparametric tuning:
Below is the snippet describing hyperparametric tuning in SVM.

```
model=svm.SVC()
#Hyper Parameters Set
params = {'C': [6,7,8,9,10,11,12],
        'kernel': ['linear','rbf']}
#Making models with hyper parameters sets
model1 = GridSearchCV(model, param_grid=params, n_jobs=-1)
```

The soft margin constant 'C' here is the hyperparameter. In SVM, if the training data is linearly- separable, we can select two parallel hyperplanes that separate the two classes of data, so that the distance between them is as large as possible. The region bounded by these two hyperplanes is called the "margin", and the maximum-margin hyperplane is the hyperplane that lies halfway between them. A smaller 'C' will allow more errors and margin errors and usually produce a larger margin. When C goes to infinity, svm becomes a hard-margin with less errors. Besides C, we have the parameter 'Kernel' providing 2 options- linear & rbf. The best kernel is selected by the tuning process.
The best Hyperparameters chosen by the process are (see params in program above)- *{'C':12, 'kernel': 'rbf'}*. This combination is retained by the model and then used to predict the test variable, details of which will be shown in the following sections.

Random Forest with Hyperparametric tuning: Below is the code snippet of hyperparametric tuning in Random Forest.
(For the complete code, please refer to the Github repository)

```
#making the instance
model=RandomForestClassifier()
#hyper parameters set
params = {'criterion':['gini','entropy'],
        'n_estimators':[10,15,20,25,30],
        'min_samples_leaf':[1,2,3],
        'min_samples_split':[3,4,5,6,7],
        'random_state':[123],
        'n_jobs':[-1]}
#Making models with hyper parameters sets
model1 = GridSearchCV(model, param_grid=params, n_jobs=-1)
```

Hyperparameters of Random Forest: 1) *n_estimators*: This is the number of trees one wants to build before taking the maximum voting or averages of predictions. Higher number of trees give better performance but makes the code slower. 2) *min_samples_leaf*: The minimum number of samples required to be at a leaf node. 3) *min_samples_split:* The minimum number of samples required to split an internal node. 4) *Criterion*: The function to measure the quality of the split.
The best hyperparameters obtained are- *{'criterion': 'entropy', 'min_samples_leaf': 1, 'min_samples_split': 3, 'n_estimators': 30, 'n_jobs': -1, 'random_state': 123}*

kNN with Hyperparametric tuning:

Code snippet:

```
#making the instance
model = KNeighborsClassifier(n_jobs=-1)
```

```
#Hyper Parameters Set
params = {'n_neighbors':[5,6,7,8,9,10],
          'leaf_size':[1,2,3,5],
          'weights':['uniform', 'distance'],
          'algorithm':['auto', 'ball_tree','kd_tree'],
          'n_jobs':[-1]}
#Making models with hyper parameters sets
model1 = GridSearchCV(model, param_grid=params,
n_jobs=1)
```

Hyperparameters of kNN: 1) n_neighbors: Number of nearest neighbours (K). A smaller value of K provides the best fit with low bias.

The best hyperparameters obtained are- *{'algorithm': 'auto', 'leaf_size': 1, 'n_jobs': -1, 'n_neighbors': 6, 'weights': 'distance'}*

V. Prediction after Tuning

After tuning the hyperparameters, the models are then tested to predict the test variable (Class). The results are shown below:

Confusion Matrix:

Model		Predicted-ve	Predicted +ve
Logistic Reg.	Actual -ve	7361	636
	Actual +ve	197	6806
SVM	Actual -ve	7374	666
	Actual +ve	184	6776
R.Forest	Actual -ve	7541	60
	Actual +ve	17	7382
kNN	Actual -ve	7443	0
	Actual +ve	115	7442

Classification report metrics:

Model		Precision	Recall	F1- score
Log. Reg.	0	0.97	0.92	0.95
	1	0.91	0.97	0.94
	avg	0.95	0.94	0.94
SVM	0	1.00	1.00	1.00
	1	1.00	1.00	1.00
	Avg.	1.00	1.00	1.00
R. Forest	0	1.00	1.00	1.00
	1	1.00	1.00	1.00
	Avg.	1.00	1.00	1.00
kNN	0	0.99	1.00	0.99
	1	1.00	0.99	0.99
	Avg.	0.99	0.99	0.99

VI. Analysis & Conclusion

Model	Without Tuning			
	Accuracy	Precision	Recall value	F1- score
L.R	0.944	0.95	0.94	0.94
SVM	0.943	0.95	0.94	0.94
R.Forest	0.994	0.99	0.99	0.99
kNN	0.992	0.99	0.99	0.99

Table 1

Model	With Tuning			
	Accuracy	Precision	Recall value	F1- score
L.R	0.938	0.95	0.94	0.94
SVM	0.998	1.00	1.00	1.00
R.Forest	0.998	1.00	1.00	1.00
kNN	0.992	0.99	0.99	0.99

Table 2

(The above values are average values for both the classes)

1) From table 1, it is observed that out of the four models, the non-linear classification algorithms Random Forest and kNN provide with the maximum accuracy and hence, they are best for prediction.

2) From table 2, it is observed that after hyperparametric tuning, there is no significant change in the accuracy of Logistic regression model. The Accuracy of prediction for R.Forest and kNN also do not change. Only the accuracy of SVM changes significantly for the better, from 94 percent to 99.8 percent.

References

[1] (https://www.kaggle.com/mlg-ulb/creditcardfraud)

[2] https://www.analyticsvidhya.com/blog/2015/06/tuning-random-forest-model/

[3] https://campus.datacamp.com/courses/supervised-learning-with-scikit-learn/fine-tuning-your-model?ex=11

[4] https://kevinzakka.github.io/2016/07/13/k-nearest-neighbor/

[5] https://campus.datacamp.com/courses/supervised-learning-with-scikit-learn/fine-tuning-your-model?ex=11

[6] https://chrisalbon.com/machine_learning/model_evaluation/cross_validation_parameter_tuning_grid_search/

[7] http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html