Literature survey on classification of Machine Learning techniques effective on imbalanced dataset

Sravan Kumar Pagolu
AI Tech Systems
University College of Engineering JNTUK Narsaraopet,
Narsaraopet, Andhra Pradesh, India.
sravansmart7733@gmail.com

Abstract— **Unbalanced data set, a problem often found in real world application, can cause seriously negative effect on classification performance of machine learning algorithms. There have been many attempts at dealing with classification of unbalanced data sets. In this paper we present a brief review of existing solutions to the class-imbalance problem proposed both at the data and algorithmic levels. Even though a common practice to handle the problem of imbalanced data is to rebalance them artificially by oversampling and/or under-sampling, some researchers proved that modified support vector machine, rough set-based minority class-oriented rule learning methods, cost sensitive classifier perform good on imbalanced data set. We observed that current research in imbalance data problem is moving to hybrid algorithms.**

***Keywords*— cost-sensitive learning, imbalanced data set, modified SVM, oversampling, under sampling.**

I. INTRODUCTION

A data set is called imbalanced if it contains many more samples from one class than from the rest of the classes. Data sets are unbalanced when at least one class is represented by only a small number of training examples (called the minority class) while other classes make up the majority. In this scenario, classifiers can have good accuracy on the majority class but very poor accuracy on the minority class(es) due to the influence that the larger majority class has on traditional training criteria. Most original classification algorithms pursue to minimize the error rate: the percentage of the incorrect prediction of class labels. They ignore the difference between types of misclassification errors. In particular, they implicitly assume that all misclassification errors cost equally. In many real-world applications, this assumption is not true. The differences between different misclassification errors can be quite large. For example, in medical diagnosis of a certain cancer, if the cancer is regarded as the positive class, and non-cancer (healthy) as negative, then missing a cancer (the patient is actually positive but is classified as negative; thus it is also called —false negative‖) is much more serious (thus expensive) than the false-positive error.
The patient could lose his/her life because of the delay in the correct diagnosis and treatment. Similarly, if carrying a bomb is positive, then it is much more expensive to miss a terrorist who carries a bomb to a flight than searching an innocent person. The unbalanced data set problem appears in many real-world applications like text categorization, fault detection, fraud detection, oil-spills detection in satellite images, toxicology, cultural modelling, medical diagnosis. Many research papers on imbalanced data sets have commonly agreed that because of this unequal class distribution, the performance of the existing classifiers tends to be biased towards the majority class. The reasons for poor performance of the existing classification algorithms on imbalanced data sets are: 1. They are accuracy driven i.e.; their goal is to minimize the overall error to which the minority class contributes very little. 2. They assume that there is equal distribution of data for all the classes. 3. They also assume that the errors coming from different classes have the same cost. With unbalanced data sets, data mining learning algorithms produce degenerated models that do not take into account the minority class as most data mining algorithms assume balanced data set. A number of solutions to the class-imbalance problem were previously proposed both at the data and algorithmic levels. At the data level, these solutions include many different forms of re-sampling such as random oversampling with replacement, random under sampling, directed oversampling (in which no new examples are created, but the choice of samples to replace is informed rather than random), directed under sampling (where, again, the choice of examples to eliminate is informed), oversampling with informed generation of new samples, and combinations of the above techniques. At the algorithmic level, solutions include adjusting the costs of the various classes so as to counter the class imbalance, adjusting the probabilistic estimate at the tree leaf (when working with decision trees), adjusting the decision threshold, and recognition-based (i.e., learning from one class) rather than discrimination-based (two class) learning. The most common techniques to deal with unbalanced data include resizing training datasets, cost-sensitive classifier, and snowball method. Recently, several methods have been proposed with good performance on unbalanced data. These approaches include modified SVMs, k nearest neighbor (kNN), neural networks, genetic programming, rough set-based algorithms, probabilistic decision tree and learning methods. The next sections focus on some of the method in detail.

II. UNDERSTANDING THE PROBLEM WITH CREDIT-CARD DATASET

Let's take a famous imbalance dataset called credit-card. In this data we have to classify the transactions as fraud or non-fraud. We already know that fraud transactions are very much lesser when compared to the Non-Fraud transactions. So, when we try to classify the transactions there's a great chance of falling of Fraud transactions into the Non-Fraud Transactions.

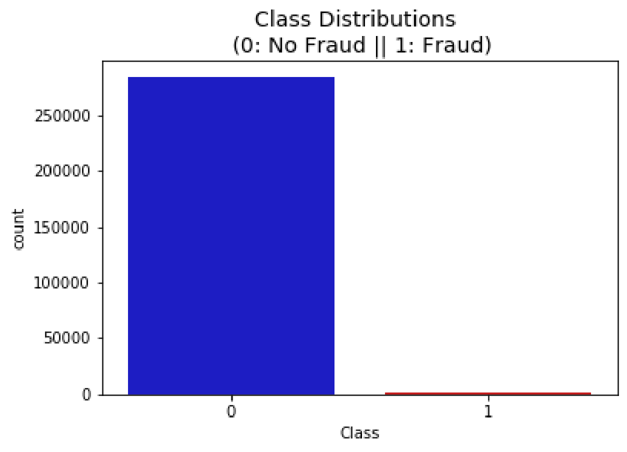
Let’s first see how many Fraud and Non-Fraud transactions in the data.

```
#Reading the creditcard.csv using
pandas
df = pd.read_csv('D:/creditcard.csv')
#Printing Non-Fraud Transactions
print('No Frauds',
round(df['Class'].value_counts()[0]/len
(df) * 100,2), '% of the dataset')
#Printing Fraud Transactions
print('Frauds',
round(df['Class'].value_counts()[1]/len
(df) * 100,2), '% of the dataset')
```

```
No Frauds 99.83 % of the dataset
Frauds 0.17 % of the dataset
```

From the above code snippet, we clearly say that **99.83%** of data is Non-Fraud and only **0.17%** of data is fraud.
Let’s Visualize the data in a bar chart.

```
colors = ["#0101DF", "#DF0101"]

sns.countplot('Class', data=df,
palette=colors)
plt.title('Class Distributions \n (0:
No Fraud || 1: Fraud)', fontsize=14)
```



By seeing the above graph, we can easily visualize how the data is distributed between Non-Fraud Transactions (0) and Fraud Transactions (1).
From here let’s check how accurate the classification algorithms work on the data.

Performance Metrics

- **Precision:** the number of true positives divided by all positive predictions. Precision is also called Positive Predictive Value. It is a measure of a classifier’s exactness. Low precision indicates a high number of false positives.

- **Recall:** the number of true positives divided by the number of positive values in the test data. Recall is also called Sensitivity or the True Positive Rate. It is a measure of a classifier’s completeness. Low recall indicates a high number of false negatives.

- **F1: Score:** the weighted average of precision and recall.

1.Logistic Regression

In statistics, the **logistic model** (or **logit model**) is used to model the probability of a certain class or event existing such as pass/fail, win/lose, alive/dead or healthy/sick. This can be extended to model several classes of events such as determining whether an image contains a cat, dog, lion, etc... Each object being detected in the image would be assigned a probability between 0 and 1 and the sum adding to one.
Let’s try Logistic Regression on the credit card dataset.
We have seen that the **accuracy** of the Logistic Regression is about **99.921%.**

```
# setting up testing and training sets
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.25,
random_state=27)
lr =
LogisticRegression(solver='liblinear').
fit(X_train, y_train)

# Predict on training set
lr_pred = lr.predict(X_test)

# Checking accuracy
a = accuracy_score(y_test, lr_pred)
print("Accuracy Score : ", a*100,"%")


# Checking unique values
predictions = pd.DataFrame(lr_pred)
predictions[0].value_counts()
```

```
Accuracy Score :  99.92135052386169 %
0    71108
1       94
Name: 0, dtype: int64
```

Checking the **f1, recall score** for the test data.

```
# f1 score
print("F1 Score : ", f1_score(y_test,
lr_pred))

# recall score
print("Recall Score : ",
recall_score(y_test, lr_pred))
```

```
F1 Score :  0.7522123893805309
Recall Score :  0.6439393939393939
```

2.RandomForestClassifier

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting.

```
from sklearn.ensemble import
RandomForestClassifier

# train model
rfc =
RandomForestClassifier(n_estimators=10)
.fit(X_train, y_train)

# predict on test set
rfc_pred = rfc.predict(X_test)
print("Accuracy Score :
",accuracy_score(y_test, rfc_pred)*100)
```

```
Accuracy Score : 99.9592708069998 %
```

The accuracy is increased from the Logistic Regression by **0.03%.**
Checking the **f1, recall score** for the test data.

```
print("F1 score : ",f1_score(y_test,
rfc_pred))

print("Recall score : ",
recall_score(y_test, rfc_pred))
```

```
F1 score :  0.879668049792531
Recall score :  0.803030303030303
```

3.Gradient Boosting Classifier

Gradient boosting is a machine learning technique for regression and classification problems, which produces a prediction model in the form of an ensemble of weak prediction models, typically decision trees. It builds the model in a stage-wise fashion like other boosting methods do, and it generalizes them by allowing optimization of an arbitrary differentiable loss function.

```
from sklearn.ensemble import
GradientBoostingClassifier

gb_clf =
GradientBoostingClassifier(n_estimators
=20, max_features=2, max_depth=2,
random_state=0)
gb = gb_clf.fit(X_train, y_train)
gb_pred = gb.predict(X_test)
```

```
Accuracy Score :  99.7415802926884
```

The Accuracy Score is increased than that of above two classifiers.

Checking the **f1, recall score** for the test data

```
print("F1 score : ",f1_score(y_test,
gb_pred))

print("Recall score : ",
recall_score(y_test, gb_pred))
```

```
F1 score :  0.17117117117117117
Recall score :  0.14393939393939395
```

III. SOLUTIONS FOR THE IMBALANCE PROBLEM

UNDERSAMPLING

In this phase of the project we will implement *"Random Under Sampling"* which basically consists of removing data in order to have a more **balanced dataset** and thus avoiding our models to overfitting.

Steps:

The first thing we have to do is determine how **imbalanced** is our class (use "value_counts()" on the class column to determine the amount for each label)

Once we determine how many instances are considered **fraud transactions** (Fraud = "1") , we should bring the **non-fraud transactions** to the same amount as fraud transactions (assuming we want a 50/50 ratio), this will be equivalent to 492 cases of fraud and 492 cases of non-fraud transactions.

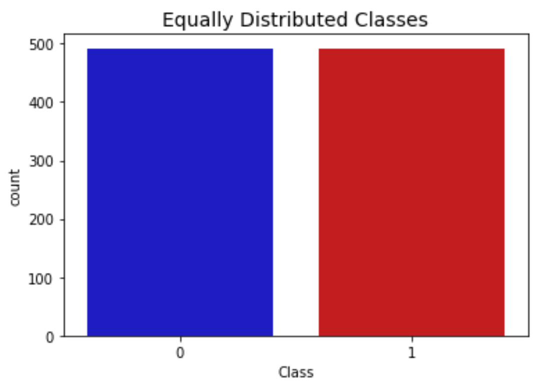
After implementing this technique, we have a sub-sample of our dataframe with a 50/50 ratio with regards to our classes. Then the next step we will implement is to **shuffle the data** to see if our models can maintain a certain accuracy every time we run this script.

Note: The main issue with "Random Under-Sampling" is that we run the risk that our classification models will not perform as accurate as we would like to since there is a great deal of **information loss** (bringing 492 non-fraud transaction from 284,315 non-fraud transaction)

```
df = df.sample(frac=1)

# amount of fraud classes 492 rows.
fraud_df = df.loc[df['Class'] == 1]
non_fraud_df = df.loc[df['Class'] ==
0][:492]

normal_distributed_df =
pd.concat([fraud_df, non_fraud_df])
```

```
# Shuffle dataframe rows
new_df =
normal_distributed_df.sample(frac=1,
random_state=42) # Since our classes are
highly skewed we should make them
equivalent in order to have a normal
distribution of the classes.

# Lets shuffle the data before creating
the subsamples

df = df.sample(frac=1)

# amount of fraud classes 492 rows.
fraud_df = df.loc[df['Class'] == 1]
non_fraud_df = df.loc[df['Class'] ==
0][:492]

normal_distributed_df =
pd.concat([fraud_df, non_fraud_df])

# Shuffle dataframe rows
new_df =
normal_distributed_df.sample(frac=1,
random_state=42)
print('Distribution of the Classes in
the subsample dataset')
print(new_df['Class'].value_counts()/le
n(new_df))



sns.countplot('Class', data=new_df,
palette=colors)
plt.title('Equally Distributed
Classes', fontsize=14)
plt.show()
```

```
Distribution of the Classes in the subs
ample dataset
1    0.5
0    0.5
Name: Class, dtype: float64
```



Comparing the accuracy, f1 and recall score for the classification algorithms:

```
# Our data is already scaled we should
split our training and test sets
from sklearn.model_selection import
train_test_split
# This is explicitly used for
undersampling.
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.2,
random_state=42)
# Turn the values into an array for
feeding the classification algorithms.
X_train = X_train.values
X_test = X_test.values
y_train = y_train.values
y_test = y_test.values
# Let's implement simple classifiers
classifiers = {
    "LogisiticRegression":
LogisticRegression(),
    "RandomTreeClassifier":
RandomForestClassifier(),
    "GradientBoostingClassifier":
GradientBoostingClassifier()
}

# Wow our scores are getting even high
scores even when applying cross
validation.
from sklearn.model_selection import
cross_val_score
```

```
for key, classifier in
classifiers.items():
    train = classifier.fit(X_train,
y_train)
    pred = train.predict(X_test)
    print("Classifiers: ",
classifier.__class__.__name__)
    print("Accuracy Score : ",
accuracy_score(y_test, pred)*100)
    print("F1 score :
",f1_score(y_test, pred))
    print("Recall score : ",
recall_score(y_test, pred))
    print("-------------------------------
----------------")
```

```
Classifiers:  LogisticRegression
Accuracy Score :  99.89993328885924
F1 score :  0.6459627329192548
Recall score :  0.5306122448979592
-------------------------------------
-------
Classifiers:  RandomForestClassifier
Accuracy Score :  99.95611109160492
F1 score :  0.8587570621468926
Recall score :  0.7755102040816326
-------------------------------------
-------
Classifiers:  GradientBoostingClassif
ier
Accuracy Score :  99.89466661985183
F1 score :  0.6629213483146067
Recall score :  0.6020408163265306
-------------------------------------
-------
```

SMOTE(OVERSAMPLING)

SMOTE stands for Synthetic Minority Over-sampling Technique. Unlike Random Under Sampling, SMOTE creates new synthetic points in order to have an equal balance of the classes. This is another alternative for solving the "class imbalance problems".

Understanding SMOTE:

Solving the Class Imbalance: SMOTE creates synthetic points from the minority class in order to reach an equal balance between the minority and majority class.

Location of the synthetic points: SMOTE picks the distance between the closest neighbours of the minority class, in between these distances it creates synthetic points.

Final Effect: More information is retained since we didn't have to delete any rows unlike in random under sampling.

```
from imblearn.over_sampling import
SMOTE
# SMOTE Technique (OverSampling) After
splitting and Cross Validating
sm = SMOTE(ratio='minority',
random_state=42)
# Xsm_train, ysm_train =
sm.fit_sample(X_train, y_train)

# This will be the data were we are
going to
Xsm_train, ysm_train =
sm.fit_sample(X_train, y_train)
classifiers = {
    "LogisiticRegression":
LogisticRegression(),
    "RandomTreeClassifier":
RandomForestClassifier(),
    "GradientBoostingClassifier":
GradientBoostingClassifier()
}
```

```
# Wow our scores are getting even high
scores even when applying cross
validation.
from sklearn.model_selection import
cross_val_score
for key, classifier in
classifiers.items():
    train = classifier.fit(Xsm_train,
ysm_train)
    pred = train.predict(X_test)
    print("Classifiers: ",
classifier.__class__.__name__)
    print("Accuracy Score : ",
accuracy_score(y_test, pred)*100)
    print("F1 score :
",f1_score(y_test, pred))
    print("Recall score : ",
recall_score(y_test, pred))
    print("-------------------------")
```

```
Classifiers:  LogisticRegression
Accuracy Score :  98.71492373809726
F1 score :  0.16893732970027248
Recall score :  0.8773584905660378
-------------------------------------
Classifiers:  RandomForestClassifier
Accuracy Score :  99.96067526193085
F1 score :  0.8600000000000001
Recall score :  0.8113207547169812
-------------------------------------
Classifiers:  GradientBoostingClassif
ier
Accuracy Score :  99.47332940085953
F1 score :  0.3267504488330341
Recall score :  0.8584905660377359
-------------------------------------
```

IV. CONCLUSION

As most of the datasets in the present world are like the above example, we have to achieve a solution for clearing the imbalance problem and to find the classification algorithm which is effective on the imbalance dataset. The above are only some of the algorithms. There are many classification algorithms. So, applying the above solutions and comparing the accuracy and scores, we can find the effective algorithm on the desired dataset.