3.6 Handle Imbalanced Datasets

February 22, 2017

1 Handle Imbalanced Dataset

There are plenty of examples in real-world problems that deals with imbalanced target classes. Imagine medical data where there are only a few positive instances out of thousands of negatie (normal) ones. Another example might be analyzing fraud transaction, in which the actual frauds represent only a fraction of all available data.

> Imbalanced data refers to a classification problems where the classes are not equally distributed.

You can read good introduction about tackling imbalanced datasets here.

What you will learn: - what are the common approaches when dealing with class imbalance, - what is the *accuracy paradox*, - how to manually denote which samples are more important than others, - how to use scale_pos_weight parameter to do it automatically

1.0.1 General advices

These are some common tactics when approaching imbalanced datasets:

- collect more data,
- use better evaluation metric (that notices mistakes - ie. AUC, F1, Kappa, ...),
- try oversampling minority class or undersampling majority class,
- generate artificial samples of minority class (ie. SMOTE algorithm)

In XGBoost you can try to: - make sure that parameter `min_child_weight` is small (because leaf nodes can have smaller size groups), it is set to `min_child_weight=1` by default, - assign more weights to specific samples while initalizing `DMatrix`, - control the balance of positive and negative weights using `set_pos_weight` parameter, - use AUC for evaluation

1.0.2 Prepare data

Let's test it by generating an artificial dataset to perform some experiments. But first load essential libraries that will be used throughout the lecture.

```
In [1]: import numpy as np
        import pandas as pd

        import xgboost as xgb
```

```
from sklearn.datasets import make_classification
from sklearn.metrics import accuracy_score, precision_score, recall_score
from sklearn.cross_validation import train_test_split

# reproducibility
seed = 123
```

We'll use a function to generate dataset for binary classification. To assure that it's imbalanced use `weights` parameter. In this case there will be 200 samples each described by 5 features, but only 10% of them (about 20 samples) will be positive. That makes the problem harder.

```
In [2]: X, y = make_classification(
            n_samples=200,
            n_features=5,
            n_informative=3,
            n_classes=2,
            weights=[.9, .1],
            shuffle=True,
            random_state=seed
        )

        print('There are {} positive instances.'.format(y.sum()))
```

```
There are 21 positive instances.
```

Divide created data into train and test. Remember so that both datasets should be similiar in terms of distribution, so they need stratification.

```
In [3]: X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, stratify=y, ra

        print('Total number of postivie train instances: {}'.format(y_train.sum()))
        print('Total number of positive test instances: {}'.format(y_test.sum()))
```

```
Total number of postivie train instances: 14
Total number of positive test instances: 7
```

1.0.3 Baseline model

In this approach try to completely ignore the fact that classed are imbalanced and see how it will perform. Create `DMatrix` for train and test data.

```
In [4]: dtrain = xgb.DMatrix(X_train, label=y_train)
        dtest = xgb.DMatrix(X_test)
```

Assume that we will create 15 decision tree stumps, solving binary classification problem, where each next one will be train very aggressively.

These parameters will also be used in consecutive examples.

```
In [5]: params = {
            'objective':'binary:logistic',
            'max_depth':1,
            'silent':1,
            'eta':1
        }

        num_rounds = 15
```

Train the booster and make predictions.

```
In [6]: bst = xgb.train(params, dtrain, num_rounds)
        y_test_preds = (bst.predict(dtest) > 0.5).astype('int')
```

Let's see how the confusion matrix looks like.

```
In [7]: pd.crosstab(
            pd.Series(y_test, name='Actual'),
            pd.Series(y_test_preds, name='Predicted'),
            margins=True
        )
```

```
Out[7]: Predicted   0  1  All
        Actual
        0          59  0   59
        1           4  3    7
        All        63  3   66
```

We can also present the performance using 3 different evaluation metrics: - accuracy, - precision (the ability of the classifier not to label as positive a sample that is negative), - recall (the ability of the classifier to find all the positive samples).

```
In [8]: print('Accuracy: {0:.2f}'.format(accuracy_score(y_test, y_test_preds)))
        print('Precision: {0:.2f}'.format(precision_score(y_test, y_test_preds)))
        print('Recall: {0:.2f}'.format(recall_score(y_test, y_test_preds)))
```

```
Accuracy: 0.94
Precision: 1.00
Recall: 0.43
```

Intuitively we know that the foucs should be on finding positive samples. First results are very promising (94% accuracy - wow), but deeper analysis show that the results are biased towards majority class - we are very poor at predicting the actual label of positive instances. That is called an accuracy paradox.

1.0.4 Custom weights

Try to explicitly tell the algorithm what important using relative instance weights. Let's specify that positive instances have 5x more weight and add this information while creating `DMatrix`.

```
In [9]: weights = np.zeros(len(y_train))
        weights[y_train == 0] = 1
        weights[y_train == 1] = 5

        dtrain = xgb.DMatrix(X_train, label=y_train, weight=weights) # weights added
        dtest = xgb.DMatrix(X_test)
```

Train the classifier and get predictions (same as in baseline):

```
In [10]: bst = xgb.train(params, dtrain, num_rounds)
         y_test_preds = (bst.predict(dtest) > 0.5).astype('int')
```

Inspect the confusion matrix, and obtained evaluation metrics:

```
In [11]: pd.crosstab(
             pd.Series(y_test, name='Actual'),
             pd.Series(y_test_preds, name='Predicted'),
             margins=True
         )
```

Out[11]:

Predicted	0	1	All
Actual			
0	52	7	59
1	2	5	7
All	54	12	66

```
In [12]: print('Accuracy: {0:.2f}'.format(accuracy_score(y_test, y_test_preds)))
         print('Precision: {0:.2f}'.format(precision_score(y_test, y_test_preds)))
         print('Recall: {0:.2f}'.format(recall_score(y_test, y_test_preds)))
```

```
Accuracy: 0.86
Precision: 0.42
Recall: 0.71
```

You see that we made a trade-off here. We are now able to better classify the minority class, but the overall accuracy and precision decreased. Test multiple weights combinations and see which one works best.

1.0.5 Use `scale_pos_weight` parameter

You can automate the process of assigning weights manually by calculating the proportion between negative and positive instances and setting it to `scale_pos_weight` parameter.

Let's reinitialize datasets.

```
In [13]: dtrain = xgb.DMatrix(X_train, label=y_train)
         dtest = xgb.DMatrix(X_test)
```

Calculate the ratio between both classes and assign it to a parameter.

```
In [14]: train_labels = dtrain.get_label()

         ratio = float(np.sum(train_labels == 0)) / np.sum(train_labels == 1)
         params['scale_pos_weight'] = ratio
```

And like before, analyze the confusion matrix and obtained metrics

```
In [15]: bst = xgb.train(params, dtrain, num_rounds)
         y_test_preds = (bst.predict(dtest) > 0.5).astype('int')

         pd.crosstab(
             pd.Series(y_test, name='Actual'),
             pd.Series(y_test_preds, name='Predicted'),
             margins=True
         )
```

```
Out[15]: Predicted   0   1  All
         Actual
         0          51   8   59
         1           0   7    7
         All        51  15   66
```

```
In [16]: print('Accuracy: {0:.2f}'.format(accuracy_score(y_test, y_test_preds)))
         print('Precision: {0:.2f}'.format(precision_score(y_test, y_test_preds)))
         print('Recall: {0:.2f}'.format(recall_score(y_test, y_test_preds)))
```

```
Accuracy: 0.88
Precision: 0.47
Recall: 1.00
```

You can see that scalling weight by using `scale_pos_weights` in this case gives better results that doing it manually. We are now able to perfectly classify all posivie classes (focusing on the real problem). On the other hand the classifier sometimes makes a mistake by wrongly classifing the negative case into positive (producing so called *false positives*).