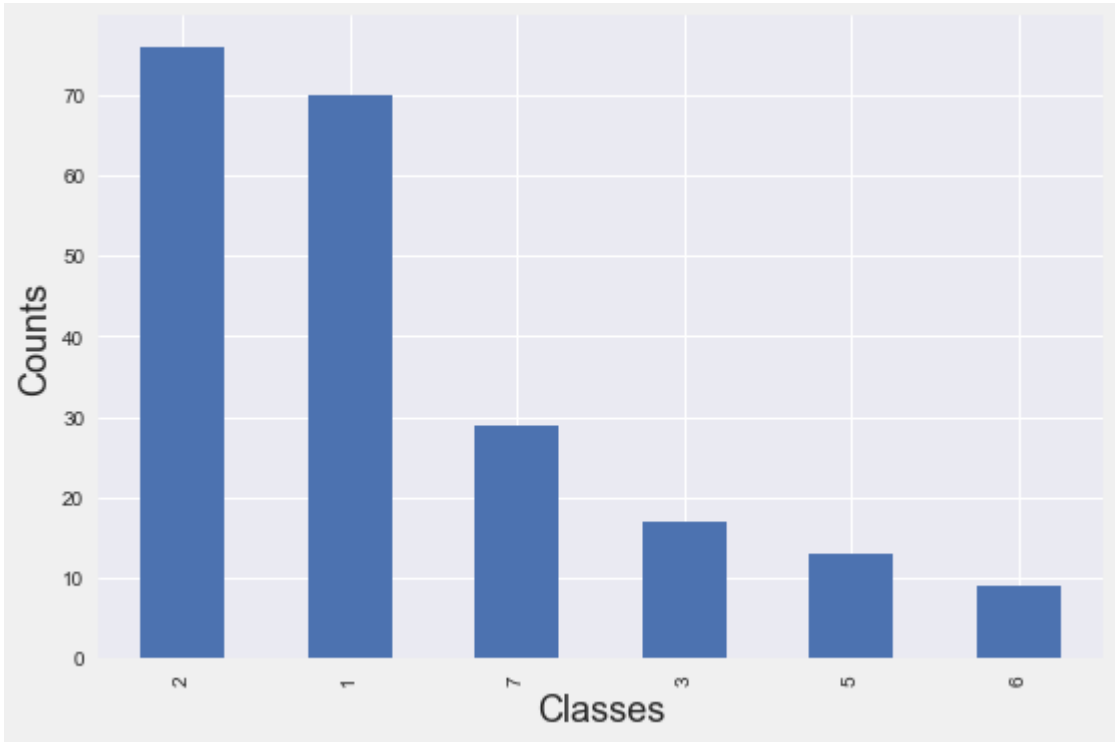
How to handle Multiclass Imbalanced Data?- Say No To SMOTE

towardsdatascience.com/how-to-handle-multiclass-imbalanced-data-say-no-to-smote-e9a7f393c310

August 31, 2020

No need of SMOTE anymore.



One of the common problems in Machine Learning is handling the imbalanced data, in which there is a highly disproportionate in the target classes.

Hello world, this is my second blog for the Data Science community. In this blog, we are going to see how to deal with the multiclass imbalanced data problem.

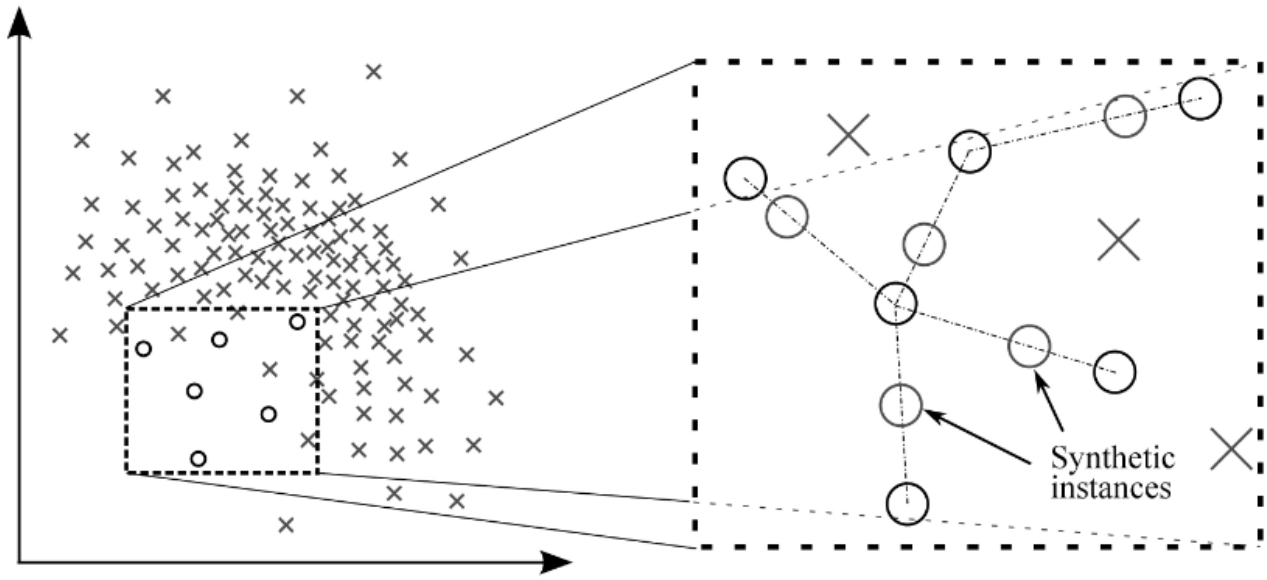
What is Multiclass Imbalanced Data?

When the target classes (two or more) of classification problems are not equally distributed, then we call it Imbalanced data. If we failed to handle this problem then the model will become a disaster because modeling using class-imbalanced data is biased in favor of the majority class.

There are different methods of handling imbalanced data, the most common methods are Oversampling and creating synthetic samples.

What is SMOTE?

SMOTE is an oversampling technique that generates synthetic samples from the dataset which increases the predictive power for minority classes. Even though there is no loss of information but it has a few limitations.



Synthetic Samples

Limitations:

1. SMOTE is not very good for high dimensionality data
2. Overlapping of classes may happen and can introduce more noise to the data.

So, to skip this problem, we can assign weights for the class manually with the '**class_weight**' parameter.

Why use Class weight?

Class weights modify the loss function directly by giving a penalty to the classes with different weights. It means purposely increasing the power of the minority class and reducing the power of the majority class. Therefore, it gives better results than SMOTE.

Overview:

I aim to keep this blog very simple. We have a few most preferred techniques for getting the weights for the data which worked for my Imbalanced learning problems.

1. Sklearn utils.
2. Counts to Length.
3. Smoothen Weights.
4. Sample Weight Strategy.

1. Sklearn utils:

We can get class weights using sklearn to compute the class weight. By adding those weight to the minority classes while training the model, can help the performance while classifying the classes.

```
from sklearn.utils import class_weightclass_weight =
class_weight.compute_class_weight('balanced,
                                                np.unique(target_Y),
                                                target_Y)model =
LogisticRegression(class_weight = class_weight)
model.fit(X,target_Y)# ['balanced', 'calculated balanced', 'normalized'] are
hyperpaameters whic we can play with.
```

We have a class_weight parameter for almost all the classification algorithms from Logistic regression to Catboost. But XGboost has scale_pos_weight for binary classification and sample_weights (refer 4) for both binary and multiclass problems.

2. Counts to Length Ratio:

Very simple and straightforward! Dividing the no. of counts of each class with the no. of rows. Then

```
weights = df[target_Y].value_counts()/len(df)
model = LGBMClassifier(class_weight = weights)
model.fit(X,target_Y)
```

3. Smoothen Weights Technique:

This is one of the preferable methods of choosing weights.

labels_dict is the dictionary object contains counts of each class.

The log function smooths the weights for the imbalanced class.

```
def class_weight(labels_dict,mu=0.15):
    total = np.sum(labels_dict.values())
    keys = labels_dict.keys()
    weight = dict()for i in keys:
        score = np.log(mu*total/float(labels_dict[i]))
        weight[i] = score if score > 1 else 1return weight# random labels_dict
labels_dict = df[target_Y].value_counts().to_dict()weights =
class_weight(labels_dict)model = RandomForestClassifier(class_weight = weights)
model.fit(X,target_Y)
```

4. Sample Weight Strategy:

This below function is different from the class_weight parameter which is used to get sample weights for the XGboost algorithm. It returns different weights for each training sample.

> Sample_weight is an array of the same length as data, containing weights to apply to the model's loss for each sample.

```
def BalancedSampleWeights(y_train,class_weight_coef):
    classes = np.unique(y_train, axis = 0)
    classes.sort()
    class_samples = np.bincount(y_train)
    total_samples = class_samples.sum()
    n_classes = len(class_samples)
    weights = total_samples / (n_classes * class_samples * 1.0)
    class_weight_dict = {key : value for (key, value) in              zip(classes,
weights)}
    class_weight_dict[classes[1]] = class_weight_dict[classes[1]] *
    class_weight_coef
    sample_weights = [class_weight_dict[i] for i in y_train]
    return sample_weights#Usage
weight=BalancedSampleWeights(target_Y,class_weight_coef)
model = XGBClassifier(sample_weight = weight)
model.fit(X, target_Y)
```

class_weights vs sample_weight:

`sample_weights` is used to give weights for each training sample. That means that you should pass a one-dimensional array with the exact same number of elements as your training samples.

`class_weights` is used to give weights for each target class. This means you should pass a weight for each class that you are trying to classify.

Conclusion:

The above are few methods of finding class weights and sample weights for your classifier. I mention almost all the techniques which worked well for my project.

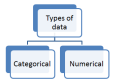
Sign up for The Variable

By Towards Data Science

Custom face mask from the Copenhagen Center for Social Data Science (SODAS), photo by author.

The inaugural cohort of the University of Copenhagen's MSc of Social Data Science (SODAS)

More From Medium



Medium