Train In Data

Overcoming Class Imbalance with SMOTE: How to Tackle Imbalanced Datasets in Machine Learning

by Sole | Mar 28, 2023 | Imbalanced Data, Machine Learning, Python

In the field of data science and data mining, dealing with imbalanced datasets is a common challenge. Many real-world datasets are often imbalanced, meaning that the number of instances in one class is significantly higher than the other. This can lead to biased model performan models tend to classify the majority misclassifying the minority class.

In imbalanced classification, however, we are often interested in classifying the minority samples correctly since these minority samples represent the instances we are interested in, like fraud or rare diseases. However, machine learning algorithms used for binary classification or multi-class classification are designed to work with balanced datasets and hence optimize balanced metrics. This is true for all traditional machine learning models, including logistic regression, decision trees, bagging models like random forests, gradient boosting machines, and also SVMs, among others.

So what can we do to better classify the minority samples? Resampling the training data is often a useful way to tackle the class imbalance problem.

Oversampling and undersampling

Resampling is a data preprocessing step that aims to resolve the class imbalance problem by rebalancing the class distribution in the training data. Resampling methods are divided into undersampling and oversampling methods. Among the oversampling methods, we find random oversampling, which simply duplicates instances of the minority class, and SMOTE.

In this blog post, we will explore the works, and its benefits, and then see Python. Let's crack on!

For tutorials about undersampling, oversampling, and additional ways to work with imbalanced data, check our course Machine Learning with Imbalanced Data.

What is SMOTE?

SMOTE stands for Synthetic Minority Over-sampling Technique. It is an oversampling technique used to balance the class distribution of a dataset by creating synthetic minority class samples. SMOTE is a type of data augmentation technique that generates new synthetic samples by interpolating between existing minority-class samples.

SMOTE works by creating synthetic samples along the lines joining the nearest neighbors in the feature space. The basic idea behind SMOTE is to create new
taking small steps from one of the n

of its k nearest neighbors in the feature space, where k is a parameter of the algorithm.

The algorithm creates a new sample by randomly selecting one of the k nearest neighbors and then adding a small perturbation to the feature vector between the original sample and the selected neighbor. This creates new synthetic data that is similar to the minority class samples in the feature space but is not an exact copy of any of the existing samples.

How does SMOTE work?

The SMOTE algorithm works in the following way:

1. Select a minority class sample from the original dataset.

2. Find its k nearest minority class neighbors in the feature space.

3. Randomly select one of the k nearest neighbors.

4. Generate a new synthetic sample by interpolating between the selected minority class sample and the randomly selected neighbor.

5. Repeat steps 1-4 until the desired number of synthetic samples is generated.

The k parameter in SMOTE determi
neighbors to consider when generat
Typically, the value of k is set to 5, b

depending on the size of the dataset and the degree of imbalance.

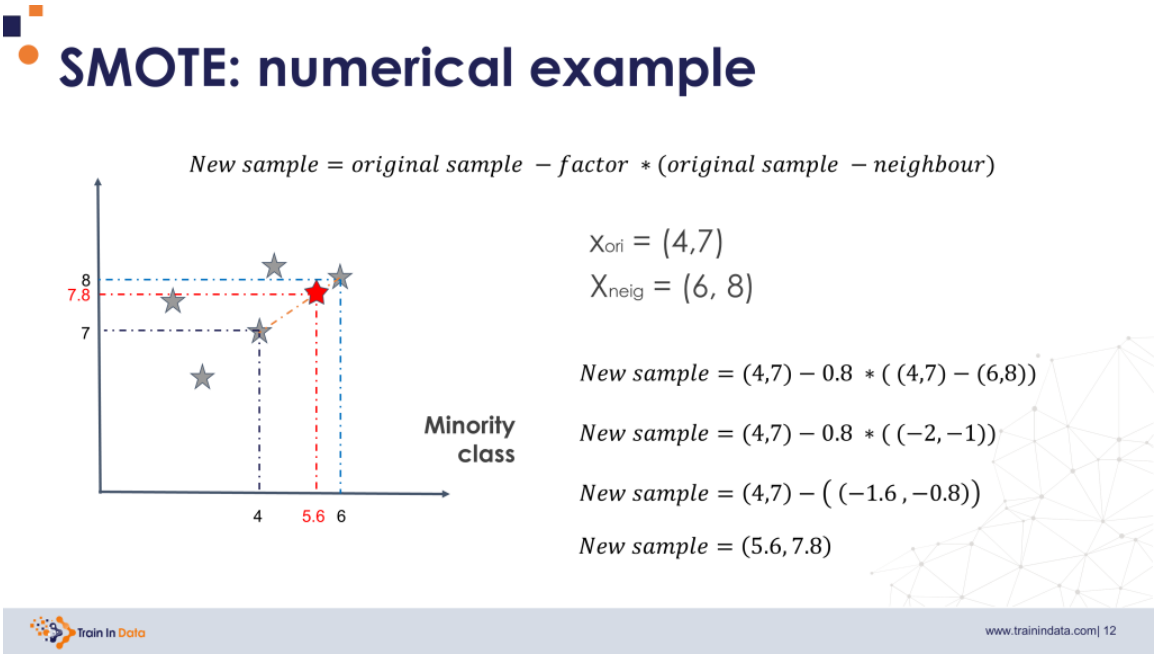
In fact, k could be a hyperparameter to optimize using cross-validation. However, SMOTE relies on the k nearest neighbors algorithm, which does not scale well, and therefore optimizing this value could be quite time and computing resource consuming.

Creating the synthetic data

How exactly does SMOTE create the new examples?

After selecting a template sample from the minority class and one of its closest neighbors (steps 1–3), SMOTE generates a new synthetic example by linearly interpolating between the feature values of the template sample and the neighbor. How?

SMOTE first computes the difference between each feature value of the neighbor and the template sample. It then multiplies this difference by a random number between 0 and 1, and adds the result to the corresponding feature value of the template sample. This generates a new feature value for the synthetic example that lies somewhere along the line connecting the template sample and its neighbor in the feature space.



This process is repeated for each feature in the template sample to generate a complete set of feature values for the synthetic example. The new example is then added to the training dataset as a synthetic example for the minority class.

Benefits of SMOTE

There are several benefits to using SMOTE for dealing with imbalanced datasets:

Improved model performance: SMOTE helps to balance the class distribution of the dataset, which can improve the performance of machine learning models.

Reduced risk of overfitting: By generating new synthetic samples, instead of simply duplicati
can help to reduce the risk of overfit

accompanies random oversampling. Hence, SMOTE is meant to be an improvement over random oversampling.

Easy to implement: SMOTE is a simple algorithm to implement to tackle classification problems. In fact, it can be applied out-of-the-box with the Python open source library imbalanced-learn.

Limitations of SMOTE

While SMOTE was introduced as a powerful technique for dealing with imbalanced datasets, it does have some limitations:

No consideration for the quality of synthetic samples: The SMOTE algorithm generates synthetic samples by linearly interpolating between minority class samples. However, this approach does not consider the quality or relevance of the synthetic samples generated. The synthetic samples may not accurately represent the underlying distribution of the minority class, which can negatively impact the model's performance.

SMOTE may not work well for datasets with overlapping classes: If the classes in the dataset overlap in the feature space, SMOTE may add noise and make the decision boundary blurry, which will negatively impact the performance of machine learning models.

Computational cost: SMOTE can be computationally expensive, especially for large datasets. Generating synthetic samples for all minority class instances can be time-consuming, which may limit its practical application in some cases. This is because SMOTE uses the k nearest neighbors algorithm under the hood, which does not scale well.

Sensitivity to the choice of k: The choice of k, the number of nearest neighbors to consider, can significantly impact the quality of the synthetic data.

SMOTE is only suitable for continuous variables: SMOTE is based on the k nearest neighbors algorithm, which uses euclidean distances to find the closes neighbors. The euclidean distance, however, does not make much sense for categorical or discrete variables. Hence, alternatives to SMOTE, like SMOTENC and SMOTEN were developed to work with training sets containing categorical variables.

Another potential problem with SMOTE is that it can create synthetic samples that are too similar to the existing minority class samples, and potentially far from the decision boundary. To address this issue, a modified version of SMOTE called Borderline-SMOTE was introduced. Borderline-SMOTE only generates synthetic samples for the are near the borderline between the classes. ADASYN (Adaptive Syntheti

sampling algorithm that creates more data at the decision boundary. In theory, creating more data at the decision boundary should help machine learning models better discriminate between the two classes.

Implementing SMOTE in Python

Let's see how we can implement SMOTE with the SMOTE transformer from imbalanced learn. First, we will carry out SMOTE in a toy dataset to visualize the synthetic data. After that, we will compare the performance of models trained after applying SMOTE and other data augmentation algorithms.

The examples can be found in our Github repository.

Let's make the imports:

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.datasets import make_blobs
from imblearn.over_sampling import SMOTE
```

Let's create a toy imbalanced datase


```
data, target = make_blobs(
    n_samples=1600,
```

```
    centers=[(0, 0), (5, 5)],
    n_features=2,
    cluster_std=1.5,
)
data = pd.DataFrame(data, columns=['VarA', 'Var
target = pd.Series(target)
data = pd.concat([
    data[target == 0],
    data[target == 1].sample(200, random_state=
], axis=0)
target = target.loc[data.index]
```

Let's display the toy dataset:

```
sns.scatterplot(
    data=data, x="VarA", y="VarB", hue=target,
)
plt.title('Toy dataset')
plt.show()
```

Below we see the toy dataset with th

Let’s apply SMOTE to resample the training set:

```
smote = SMOTE(
    sampling_strategy='auto',
    random_state=0,
    k_neighbors=5,
)
data_res, target_res = smote.fit_resample(data
```

With `resample()` we've created the synthetic data. Let's generate a scatter-plot for our new training set:

```
sns.scatterplot(
    data=data_res, x="VarA", y="VarB", hue=targ
)
plt.title('Over-sampled dataset')
plt.show()
```

Note in the following image the new observations in between the original data points of the minority class:

Let's now train random forests using various imbalanced datasets and compare the performance of the models trained on the original dataset and then on the data with the synthetic examples by using the roc-auc:

We being with the imports:

```
import pandas as pd
import matplotlib.pyplot a
from sklearn.ensemble impo
from sklearn.metrics impor
```

```
from sklearn.model_selection import train_test_
from sklearn.preprocessing import MinMaxScaler
from imblearn.datasets import fetch_datasets
from imblearn.over_sampling import (
    SMOTE,
    ADASYN,
    BorderlineSMOTE,
)
```

Now we create a dictionary with the transformers for the oversampling:

```
oversampler_dict = {
    'smote': SMOTE(
        sampling_strategy='auto'
        random_state=0,
        k_neighbors=5,
        n_jobs=4),
    'adasyn': ADASYN(
        sampling_strategy='auto',
        random_state=0,
        n_neighbors=5,
        n_jobs=4),
    'border': BorderlineSM
        sampling_strategy=
        random_state=0,
```

```
        k_neighbors=5,
        m_neighbors=10,
        kind='borderline-1',
        n_jobs=4),
}
```

Let's make a list containing different datasets:

```
datasets_ls = [
    'ecoli',
    'thyroid_sick',
    'arrhythmia',
]
```

Let's create a function to train a random forests classifier and evaluate the performance using the roc-auc:

```
def run_randomForests(X_train, X_test, y_train,
    rf = RandomForestClassifier(
        n_estimators=100, random_state=39, max_
    rf.fit(X_train, y_train)
    print('Train set')
    pred = rf.predict_proba(X_train)
    print(
        'Random Forests rc
    print('Test set')
    pred = rf.predict_prok
```

```
    print(
        'Random Forests roc-auc: {}'.format(roc
    return roc_auc_score(y_test, pred[:, 1])
```

Now let's train the random forests on the original datasets, and then on the training sets with the synthetic data and compare the model performance:

```
results_dict = {}
shapes_dict = {}
for dataset in datasets_ls:

    results_dict[dataset] = {}
    shapes_dict[dataset] = {}

    print(dataset)

    # load dataset
    data = fetch_datasets()[dataset]

    # separate train and test
    X_train, X_test, y_train, y_test = train_te
    data.data,
    data.target,
    test_size=0.3,
    random_state=0)
```

```
# as the oversampling techniques use KNN
# we scale the variables
scaler = MinMaxScaler().fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

roc = run_randomForests(X_train, X_test, y_

# store results
results_dict[dataset]['full_data'] = roc
shapes_dict[dataset]['full_data'] = len(X_t

print()

for oversampler in oversampler_dict.keys():

    print(oversampler)

    # resample
    X_resampled, y_resampled = oversampler_

    # evaluate performance
    roc = run_randomFc

    #store results
```

```
            results_dict[dataset][oversampler] = ro
            shapes_dict[dataset][oversampler] = len
            print()

    print()
```

The model performance on the different dataset is displayed below.

We see that SMOTE was useful to improve the performance in the ecoli dataset, and Borderline SMOTE was even better.

```
ecoli
Train set
Random Forests roc-auc: 0.9716599190283401
Test set
Random Forests roc-auc: 0.9408212560386474
smote
Train set
Random Forests roc-auc: 0.9773356837068748
Test set
Random Forests roc-auc: 0.9601449275362319
adasyn
Train set
Random Forests roc-auc: 0.
Test set
```

```
Random Forests roc-auc: 0.967391304347826
border
Train set
Random Forests roc-auc: 0.9769121586044276
Test set
Random Forests roc-auc: 0.9806763285024154
```

On the other hand, neither SMOTE nor ADASYN or Borderline SMOTE improved the model performance in the thyroid dataset.

```
thyroid_sick
Train set
Random Forests roc-auc: 0.9646448684059303
Test set
Random Forests roc-auc: 0.9521203914568843
smote
Train set
Random Forests roc-auc: 0.9605968807461769
Test set
Random Forests roc-auc: 0.9487843909644857
adasyn
Train set
Random Forests roc-auc: 0.9582371083546223
Test set
Random Forests roc-auc: 0.
border
```

```
Train set
Random Forests roc-auc: 0.9617160200097215
Test set
Random Forests roc-auc: 0.9432572167169323
```

Finally, SMOTE massively improved performance in the arrhythmia dataset, and ADASYN was even better. But borderline SMOTE did not help.

```
arrhythmia
Train set
Random Forests roc-auc: 0.9931143025772182
Test set
Random Forests roc-auc: 0.8515625
smote
Train set
Random Forests roc-auc: 0.9987919598214785
Test set
Random Forests roc-auc: 0.908203125
adasyn
Train set
Random Forests roc-auc: 0.9996085054977014
Test set
Random Forests roc-auc: 0.
border
Train set
```

```
Random Forests roc-auc: 0.9985123208912652
Test set
Random Forests roc-auc: 0.796875
```

Conclusion

SMOTE is a powerful technique for learning from imbalanced data. It helps to balance the class distribution of the original dataset by generating synthetic samples for the minority class. However, it has some limitations, such as no consideration for the quality of synthetic samples and computational cost. It is important to choose the right value of k to ensure that the synthetic samples generated are of high quality.

Despite its limitations, SMOTE is a valuable tool in the machine learning toolkit for dealing with imbalanced datasets, as we saw in our code examples. And should SMOTE not work, there are alternative oversampling methods, as well as undersampling algorithms to choose from to tackle the class imbalance problem.

To know more about how to tackle class imbalance, check out our course Machine Learning with Imbalanced Data and the references below.

Citations

N. V. Chawla, K. W. Bowyer, L. O.Hall, W. P. Kegelmeyer, "SMOTE: synthetic minority over-sampling technique," Journal of artificial intelligence research, 321-357, 2002.

H. Han, W. Wen-Yuan, M. Bing-Huan, "Borderline-SMOTE: a new over-sampling method in imbalanced data sets learning," Advances in intelligent computing, 878-887, 2005.

He, Haibo, Yang Bai, Edwardo A. Garcia, and Shutao Li. "ADASYN: Adaptive synthetic sampling approach for imbalanced learning," In IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence), pp. 1322-1328, 2008.

Additional resources

- Machine Learning with Imbalanced Data online course
- The Role of Undersampling in Tackling Imbalanced Datasets in Machine Learning
- Exploring Oversampling Techniques for Imbalanced Datasets
- Dealing with Imbalanced Datasets in Machine Learning: Techniques and Best Practices