BBM467 - Small Data Science Project

Ladybug

21827497, Alperen Berk IŞILDAR
21827373, Uğurcan ERDOĞAN

Dealing with Imbalanced Data

If we should easily explain what imbalance data is, it is a dataset whose target classes are distributed not balanced. For example, assume that a dataset contains a class that includes a person's diabetes info, and it has 1000 rows. The number of diabetes rows is 9950 and the number of non-diabetes rows is 50, then which means this dataset is imbalanced.

In this post, we will discuss how we should deal with imbalanced datasets. This post's main topics are which way we should choose to convert an imbalanced dataset to a balanced dataset and how can we apply cross-validation to these datasets. But first, let us explain some terms and their meanings.

Some Pre-Information About Terms

First of all, we cannot train our program with that imbalanced data so we must arrange that data by using some algorithms. If we train our program with it, our classifier will be affected by this situation and the result will be biased.

While dealing with imbalanced datasets, 2 main methods are used. These are oversampling and undersampling.

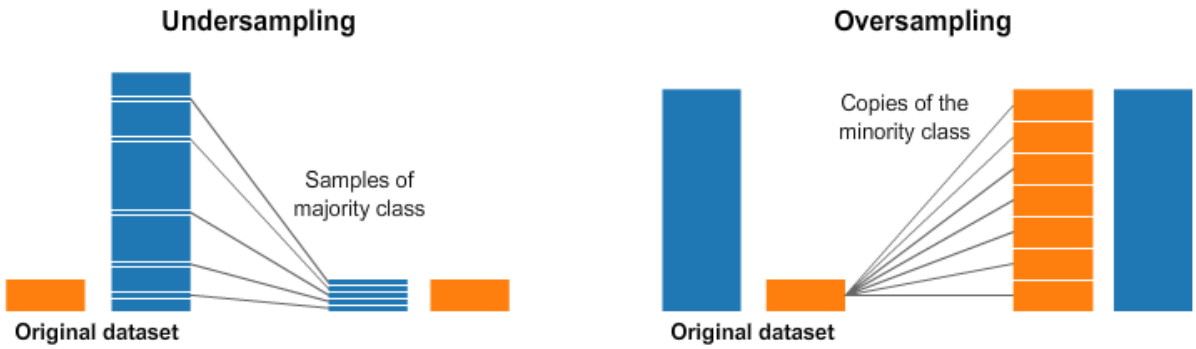

Figure 1: Undersampling and Oversampling visualized

The undersampling algorithm (Near Miss):

Firstly, the algorithm looks at the class distribution and it decides whether it is an imbalanced dataset. Then it randomly eliminates the data that has the majority of classes, until it is equal to the number of data that has the fewest classes. So, we now have a dataset that is evenly distributed i.e., 50% 50%.
Besides, there is one more undersampling algorithm except by Near miss called Random Sampling. We can shuffle the dataset and we can manually set the data up to a certain point to remain. This is not a common solution to this problem, but it is still an option.

The oversampling algorithm (known as Synthetic Minority Oversampling Technique (SMOTE)):

Firstly, this algorithm also looks at the class distribution and decides whether it is an imbalanced dataset. But then, it does the job completely opposite of the undersampling algorithms. It generates some new instances between existing instances from minority cases. These newly generated instances are not just copies of the existing instances, we can think that creating new neighbors next to the existing instances in the data that has the minority of the classes. This algorithm's thing is that does not change the number of majority instances.

We are going to use oversampling (SMOTE) while dealing because we do not want to miss any information or data points.

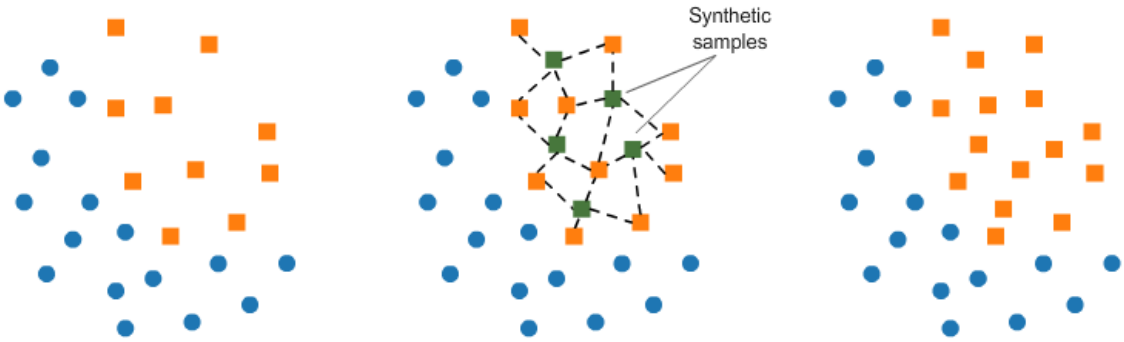

Figure 2: SMOTE visualized

We discussed the algorithms that convert the imbalanced data to balanced data and after selecting the algorithm (over or undersampling), the thing we should do very carefully is when to use this algorithm with cross-validation.

We should consider that applying this algorithm and cross-validation is not enough, the usage of them is at least as important. Because the wrong usage of them causes overfitting during cross-validation.

First, let us explain to you the wrong usage and visualize that:

While applying the algorithm, we should avoid “data leakage” which means generating nonsense data. If we apply oversampling before the cross-validation, the oversampled dataset will be considered during cross-validation, and it causes overfitting because oversampled data is already generated 100% accurate data so the result will be biased. Let’s visualize the wrong way to understand that clearly.

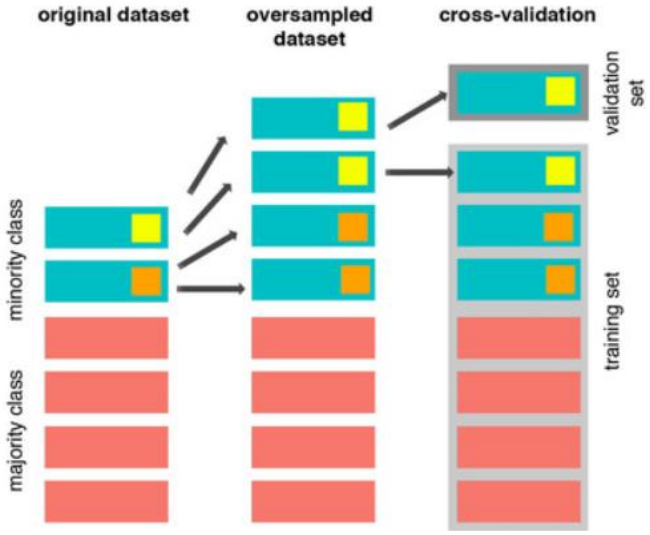

Figure 3: Cross-validation in a bad way

And now it’s time to explain the right way and visualize that:

To prevent bias and overfitting, we should apply the oversampling “during” the cross-validation. We can assume that the order of them will be cross-validation firstly, and then oversampling for the split training set on each cross-validation iteration.

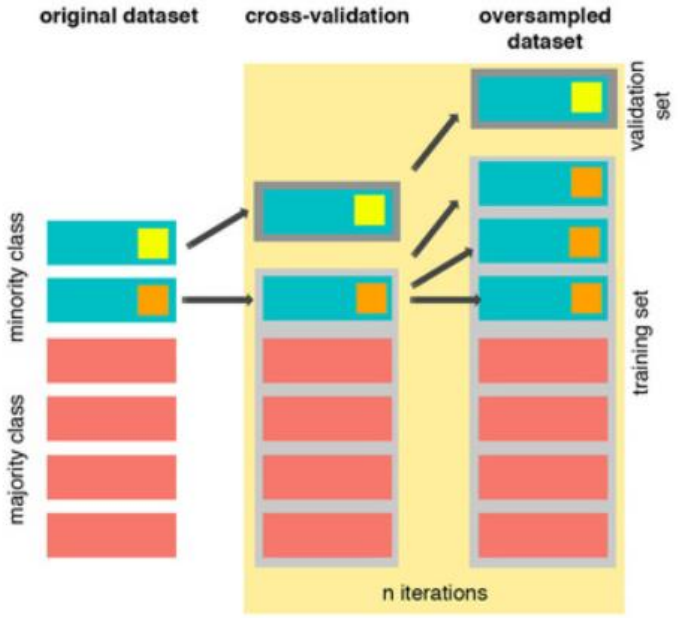

Figure 4: Cross-validation in a proper way

APPLICATION

About the data...

It is important that credit card companies are able to recognize fraudulent credit card transactions so that customers are not charged for items that they did not purchase. In this dataset, there are pre-transformed transaction data, time and amount values as features. As a result, there is a label according to whether the relevant example is fraud or non fraud. This information is also included in the class column.

- 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'. We also know that the full meaning of these features is not given in terms of privacy.

- Feature 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset.

- Feature 'Amount' is the transaction Amount, this feature can be used for example-dependant cost-sensitive learning.

You can access the dataset from this link : https://www.kaggle.com/mlg-ulb/creditcardfraud

```
# Data statistics
df.describe()
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8
count	284807.000000	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05	2.848070e+05
mean	94813.859575	3.918649e-15	5.682686e-16	-8.761736e-15	2.811118e-15	-1.552103e-15	2.040130e-15	-1.698953e-15	-1.893285e-16
std	47488.145955	1.958696e+00	1.651309e+00	1.516255e+00	1.415869e+00	1.380247e+00	1.332271e+00	1.237094e+00	1.194353e+00
min	0.000000	-5.640751e+01	-7.271573e+01	-4.832559e+01	-5.683171e+00	-1.137433e+02	-2.616051e+01	-4.355724e+01	-7.321672e+01
25%	54201.500000	-9.203734e-01	-5.985499e-01	-8.903648e-01	-8.486401e-01	-6.915971e-01	-7.682956e-01	-5.540759e-01	-2.086297e-01
50%	84692.000000	1.810880e-02	6.548556e-02	1.798463e-01	-1.984653e-02	-5.433583e-02	-2.741871e-01	4.010308e-02	2.235804e-02
75%	139320.500000	1.315642e+00	8.037239e-01	1.027196e+00	7.433413e-01	6.119264e-01	3.985649e-01	5.704361e-01	3.273459e-01
max	172792.000000	2.454930e+00	2.205773e+01	9.382558e+00	1.687534e+01	3.480167e+01	7.330163e+01	1.205895e+02	2.000721e+01

8 rows × 31 columns Open in new tab

Figure 5: Data and details

And as we can see in figure 6, there is a huge imbalance between classes.

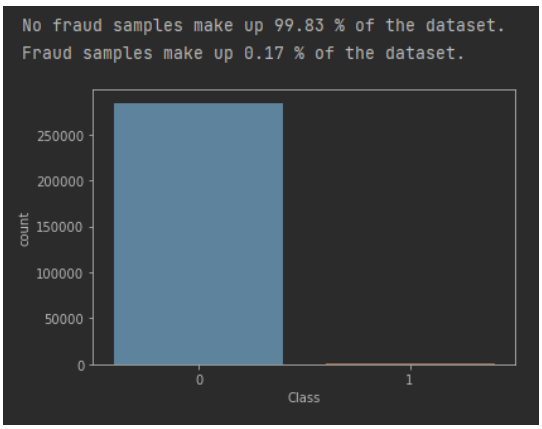

Figure 6: Class distribution visualized before SMOTE

We skip showing data preprocessing (cleaning/normalizing) steps to focus on the purpose of our blog post. After scaling our feature values with Robust Scaler, we split our dataset.

```
from sklearn.model_selection import train_test_split
# Splitting test set before doing CV and SMOTE.
X_train, X_test, y_train, y_test = train_test_split(df.drop("Class",
                                                            axis =1),
                                                    df["Class"],
                                                    test_size=0.2,
                                                    stratify=df["Class"],
                                                    random_state=11)
```

Before and after SMOTE operation and with the help of the Dimension Reduction algorithm (PCA), our data points can be seen in 2D plot space like figure 7 and 8.

```
from imblearn.over_sampling import SMOTE

# Original train set distribution
plot_2d_space(X_train, y_train, "Classes before SMOTE")

oversampler = SMOTE()
X_smote, y_smote = oversampler.fit_resample(X_train, y_train)

# Oversampled train set distribution (it won't look like this during cv)
plot_2d_space(X_smote,y_smote, "Classes after SMOTE")
```

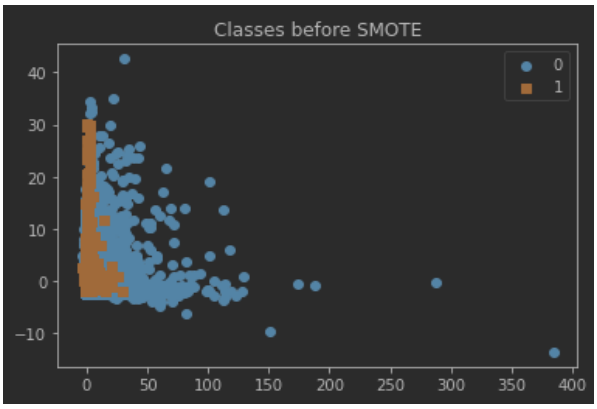

Figure 7: Data points before SMOTE

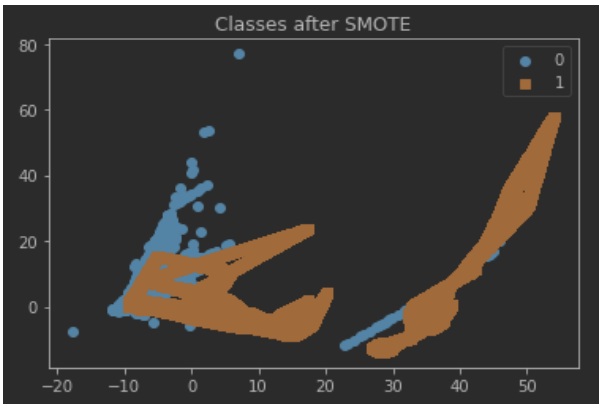

Figure 8: Data points after SMOTE

The SMOTE operation seems to have added extra data points to the minority class. Now, our class distribution looks pretty even and equal.

Figure 9: Class distribution visualized after SMOTE

Let's start with the wrong way of doing SMOTE and CV. First of all, after importing the relevant sklearn modules, we create the object that will perform cross-validation with Stratified KFold. After choosing the K value of 10 for the K-Fold CV, we can consider Logistic Regression for classifier selection. It is important to pay attention to the content of the validation set while making the most appropriate hyperparameter selections for the classifier. As in the first part of our blog post, if there are data points in the training set in the validation set, we will observe that our model is overfitting as a result of the operations.

Above, **we have already done the SMOTE process on the train set. When KFold is done in GridsearchCV, the train set will be split and traces of the train set will be found in the validation set.** This is an undesirable situation.

```
# This cross-validation object is a variation of KFold that returns
stratified folds. The folds are made by preserving the percentage of
samples for each class.
stratified_kfold = StratifiedKFold(n_splits=10,
                                   shuffle=True,
                                   random_state=11)
# Here, there is only a regressor in our pipeline. In this way, after
performing random undersampling only once, we will be able to send the
relevant subsample data into fit.
pipeline1 = Pipeline([('classifier', LogisticRegression(random_state=11,
max_iter=1000))])

# Possible hyper parameters for LogReg.
param_grid = {"classifier__penalty": ['l1', 'l2'],
              'classifier__C':[0.01, 0.1, 1, 10]}

# Choosing best Logistic Regression model with the best hyperparameters.
oversample before cv grid search = GridSearchCV(estimator=pipeline1,
                                                param_grid=param_grid,
                                                scoring='roc_auc',
                                                cv=stratified_kfold,
                                                n_jobs=-1)
```

Let’s see the score metrics:

```
undersample_before_cv_grid_search.fit(X_train, y_train)
cv_score = undersample_before_cv_grid_search.best_score_
test_score = undersample_before_cv_grid_search.score(X_test, y_test)

print_metric(y_test,undersample_before_cv_grid_search.predict(X_test))
print(f'roc-auc Cross-validation score: {round(cv_score,3)}\nroc-auc Test
score: {round(test_score,3)}')
```

```
Precision: 0.868, Recall: 0.602, F1: 0.711, Accuracy: 0.999
roc-auc Cross-validation score: 0.982
roc-auc Test score: 0.986
```

Figure 10: Scoring metrics for the overfitted model

The first thing that draws our attention in figure 10 is the accuracy score. It is almost a hundred percent value. If we look at the status of roc-auc scores and other metrics, it is obvious that the model is overfitting.

Now let's observe how to perform SMOTE and CV operations in the correct order. We create a Stratified KFold object as we did in the previous experiment. **In this pipeline, we have a SMOTE object that will run on the train-set again and again in each CV iteration this time and will not affect the validation set.** Then our LogReg classifier will be set with optimal hyperparameters.

```
# This cross-validation object is a variation of KFold that returns
stratified folds. The folds are made by preserving the percentage of
samples for each class.
stratified_kfold = StratifiedKFold(n_splits=10,
                                   shuffle=True,
                                   random_state=11)

# Here, we prepare the estimator that we will send into GridSearchCV using
imbpipeline.
# Using the pipeline, a different subsample will be created by gridsearch
in each iteration.
pipeline2 = imbpipeline([('smote', SMOTE()),
                         ('classifier',
LogisticRegression(random_state=11,max_iter=1000))])

# Possible hyper parameters for LogReg.
param_grid = {"classifier__penalty": ['l1', 'l2'],
              'classifier__C':[0.01, 0.1, 1, 10]}

# Choosing best Logistic Regression model with the best hyperparameters.
oversample_during_cv_grid_search = GridSearchCV(estimator=pipeline2,
                                                param_grid=param_grid,
                                                scoring='roc_auc',
                                                cv=stratified_kfold,
                                                n_jobs=-1)
```

Let's see the score metrics now:

```
oversample_during_cv_grid_search.fit(X_train, y_train)
cv_score = oversample_during_cv_grid_search.best_score_
test_score = oversample_during_cv_grid_search.score(X_test, y_test)

print_metric(y_test,oversample_during_cv_grid_search.predict(X_test))
print(f'roc-auc Cross-validation score: {round(cv_score,3)}\nroc-auc Test
score: {round(test_score,3)}')
```

```
Precision: 0.062, Recall: 0.898, F1: 0.116, Accuracy: 0.976
roc-auc Cross-validation score: 0.979
roc-auc Test score: 0.955
```

Figure 11: Scoring metrics for the normal model

As can be seen in figure 11, all scores, especially the accuracy score, decreased. The main reason for such high scores in the previous experiment was overfitting. Now, while our accuracy value has decreased, other metrics show very different results from the previous step.

Logistic Regression Classifier

SCORES / SMOTE…	Accuracy	Precision	Recall	F1	ROC-AUC CV	ROC-AUC TEST
before CV	0.999	0.868	0.602	0.711	0.982	0.986
during CV	0.976	0.062	0.898	0.116	0.98	0.955

CONCLUSION

In our experiments, we examined the correct steps of oversampling and cross-validation. We have observed how our model is affected by this situation if it is not done correctly.

- While tuning the model, there should be no oversampled data in the validation set. This is also true for the test set.

- While balancing the dataset, we should separate our test and validation sets from the train set to oversample the minority class and we should do the SMOTE operation for the train set.

- The train set should be used in the creation of the classifier model. Validation and test sets should be used for their intended purpose and should not be interfered with.

References

1. https://www.marcoaltini.com/blog/dealing-with-imbalanced-data-undersampling-oversampling-and-proper-cross-validation

2. https://www.kaggle.com/code/rafjaa/resampling-strategies-for-imbalanced-datasets/notebook

3. https://scikit-learn.org/stable/modules/generated/sklearn.pipeline.Pipeline.html

4. https://imbalanced-learn.org/stable/references/generated/imblearn.pipeline.Pipeline.html