Imbalanced Target Categories

Module 2 | Chapter 3 | Notebook 5

In this notebook, you will learn strategies for dealing with a classification problem that you often see: target categories of different sizes. The reference category is very often smaller than the alternative. In our case too, there are far fewer people who have left the company than who are still there. As we saw in the last programming exercise, this makes it difficult for the classification model to correctly predict the smaller category. Therefore, in this lesson you'll learn three strategies for dealing with target categories of different sizes:

- Reduce the number of majority class data points (typically a random sample)
- Artificially increase the number of datapoints in the minority class (typically using what's called bootstrapping))
- Give more weight to the incorrect classification of minority data points when training the model

---

Reducing the size of the majority category or reducing the size of the minority category

Scenario: You work for an international global logistics company, which wants to limit the number of existing employees who leave the company. You should predict which employees are likely to want to leave the company so that measures can be taken to encourage them to stay.

Let's import the training and test data now in order to get started quickly.

In [1]:
```
import pandas as pd
import pickle

#load pipeline
pipeline = pickle.load(open("pipeline.p",'rb'))
col_names = pickle.load(open("col_names.p",'rb'))

#gather data
df_train = pd.read_csv('attrition_train.csv')
df_test = pd.read_csv('attrition_test.csv')

#extract features and target
features_train = df_train.drop('attrition', axis=1)
features_test = df_test.drop('attrition', axis=1)

target_train = df_train.loc[:,'attrition']
target_test = df_test.loc[:,'attrition']

#transform data
features_train = pd.DataFrame(pipeline.transform(features_train), columns=col_names)
```

```
features_test = pd.DataFrame(pipeline.transform(features_test), columns=col_names)

# Look at raw data
features_train.head()
```

Out[1]:

	pca_years_0	pca_years_1	age	gender	businesstravel	distancefromhome	education	joblevel	ma
0	0.385171	-0.156575	30.0	0.0	1.0	5.0	3.0	2.0	
1	-2.348248	-0.406330	33.0	0.0	1.0	5.0	3.0	1.0	
2	-0.781200	-0.233330	45.0	1.0	1.0	24.0	4.0	1.0	
3	-1.181156	-0.535303	28.0	1.0	1.0	15.0	2.0	1.0	
4	-1.447056	0.019780	30.0	1.0	1.0	1.0	3.0	1.0	

The code for that looks like this:

Column number	Column name	Type	Description
0	`'pca_years_0'`	continuous (`int`)	first principal component of the original columns `'totalworkingyears'`, `'years_atcompany'`, `'years_currentrole'`, `'years_lastpromotion'` and `'years_withmanager'`
1	`'pca_years_1'`	continuous (`int`)	second principal component of the original columns `'totalworkingyears'`, `'years_atcompany'`, `'years_currentrole'`, `'years_lastpromotion'` and `'years_withmanager'`
2	`'attrition'`	categorical	Whether the employee left the company (`1`) or not (`0`)
3	`'age'`	continuous (`int`)	The person's age in years
4	`'gender'`	categorical (nominal, `int`)	Gender: male (`1`) or female (`0`)
5	`'businesstravel'`	categorical (ordinal, `int`)	How often the employee is on a business trip: often (`2`), rarely (`1`) or never (`0`)
6	`'distancefromhome'`	continuous (`int`)	Distance from home address to work address in kilometers
7	`'education'`	categorical (ordinal, `int`)	Level of education: doctorate (`5`), master (`4`), bachelor (`3`), apprenticeship(`2`), Secondary school qualifications (`1`)
8	`'joblevel'`	categorical (ordinal,	Level of responsibility: Executive (`5`), Manager (`4`), Team leader (`3`), Senior employee (`2`),

Column number	Column name	Type	Description
		`int`)	Junior employee (`1`)
9	`'maritalstatus'`	categorical (nominal, `int`)	Marital status: married (`2`), divorced (`1`), single (`0`)
10	`'monthlyincome'`	continuous (`int`)	Gross monthly salary in EUR
11	`'numcompaniesworked'`	continuous (`int`)	The number of enterprises where the employee worked before their current position
12	`'overtime'`	categorically (`int`)	Whether or not they have accumulated overtime in the past year (`1`) or not (`0`)
13	`'percentsalaryhike'`	continuous (`int`)	Salary increase in percent within the last twelve months
14	`'stock option levels'`	categorical (ordinal, `int`)	options on company shares: very many (`4`), many (`3`), few (`2`), very little (`1`), none (`0`)
15	`'trainingtimeslastyear'`	continuous (`int`)	Number of training courses taken in the last 12 months

Each row in `df_train` represents an employee

Data scientists often face the problem of generating predictions for imbalanced target categories. In the search for the best model, machine learning algorithms minimize the prediction error for all data points in the training set. Since the majority class has more data points, it usually has a stronger influence on the prediction error. For this reason, the model focuses particularly on the majority class, although we're usually particularly interested in the minority class. To address this problem, we need to mix up our data and either remove or add data points meaningfully. This is called resampling.

Now let's look at different resampling strategies. We'll use the `imblearn` module for this. It contains several *resamplers*, which are based on the syntax of `sklearn`. So you'll find your way around easily.

Resampling strategies

Let's start by concentrating on how to deal with different target category sizes:

- *Undersampling*: reducing the number of majority class data points (typically a random sample).

You can imagine this strategy by first counting how many minority class data points there are. In this case there are 167. Then you select the same number of data points of the majority class at random

`imblearn` offers this strategy in the `RandomUnderSampler` resample, which is located in `imblearn.under_sampling`. The following code shows you how to use it:

In [2]:
```
from imblearn.under_sampling import RandomUnderSampler

#1. initialize
undersampler = RandomUnderSampler(random_state=42)

#2. define Features and Target
# see first cell of notebook

#3. fit and resample
features_under, target_under = undersampler.fit_resample(features_train, target_train)
```

Check whether the numbers under the `1` and `0` categories in `target_under` are really the same. For example use the `pd.crosstab()` function.

In [3]:
```
pd.crosstab(index=target_under, columns='count')
```

Out[3]:

col_0	count
attrition	
0	167
1	167

The `'attrition'` categories are now actually the same size with 167 data points each. But the price we've paid for this is enormous: With a total of 892 people who haven't left the company, that means we're now ignoring the data of 695 people selected at random. A total of 81% of the training data with the `'attrition'` category `0` is not being used.

As a result, the training data set shrinks considerably, which can cause complications when training classification models. Therefore, we only recommend this strategy if you have an extremely large training data set and the smaller category is not too small.

So sometimes it's better to use the second strategy:

- *Oversampling*: artificially increasing the number of datapoints in the minority class (typically using what's called bootstrapping))

You can imagine this strategy by first counting how many minority class data points there are. Store this number in the variable `len_majority` and then print this.

In [4]:
```
mask_attrition_no = target_train==0
len_majority = mask_attrition_no.sum()
print(len_majority)

mask_attrition_no = target_train==1
len_majority = mask_attrition_no.sum()
print(len_majority)
```

```
862
167
```

In our there are 862. Now multiply the data points of the smaller category until there are just as many of them as there are in the bigger category. You use the bootstrap method, which means selecting a point, and putting it back in the container so it can possibly be selected again.

Think of this as selecting 862 new data points from the smaller category. Each one can be selected multiple times. This means that the same data point can be selected multiple times.

If you choose a number of balls (data points) in an container (smaller category) as a metaphor, as statisticians like to do, then you draw a ball 862 times. Each time you note down the details of the ball (data point values), put the selected ball (data point) back, shake the balls in the container and take another one. At the end you look at all the "new" data points you wrote down. These are now the training data for the smaller category.

The `RandomOverSampler` from `imblearn.over_sampling` can represent this behaviour easily. Use it to oversample the data. You can proceed just as you did with all the other resamplers. We've already done the second step, splitting the data into features and target. So you can skip this step. Make sure to set the parameter `random_state=42` again, so that your results match ours.

Save your results in `features_over` and `target_over`. Afterwards, return the class distribution in `target_over`.

In [5]:
```
from imblearn.over_sampling import RandomOverSampler

#1. initiatiate
oversampler = RandomOverSampler(random_state=42)

#2. fit and resample
features_over, target_over = oversampler.fit_resample(features_train, target_train)

#show classes
pd.crosstab(index=target_over, columns='count')
```

Out[5]:

col_0	count
attrition	
0	862
1	862

Both categories now have 862 data points - so they both have the same proportion.

Oversampling results in duplicated data points. This can lead to the problem with the k-Nearest-Neighbors algorithm, for example, that a data point has itself in its neighborhood so often that no other data points in the neighborhood are used for the classification. A classification based on only a single data point is very uncertain (high *variance*). It should therefore always be checked on a case-by-case basis whether *oversampling* improves the prediction quality.

One way to avoid duplicates during *oversampling* is to artificially create data points that resemble, but are not copies of points in the minority class, rather than using identical data points.

We call this procedure

- **SMOTE** : This name is an acronym and stands for ***S**ynthetic **M**inority **O**versampling **TE**chnique*.

Let's try it out now. Import the `SMOTE` resampler from `imblearn.over_sampling` and apply it to the training data. Name the object `smotesampler`. Store the results under `features_smote` and `target_smote`. Then return the class distribution in `target_smote`.

In [6]:
```
from imblearn.over_sampling import SMOTE
smotesampler = SMOTE()
smotesampler.fit(features_train,target_train)

features_smote, target_smote = smotesampler.fit_resample(features_train,target_train)

pd.crosstab(index=target_smote, columns='count')
```

Out[6]:

attrition	col_0 count
0	862
1	862

As you can see, you get the same class distribution as with *oversampling*. But are any of the points duplicates this time?

The easiest way to find out is to remove the duplicates in both *features* sets by using `my_df.drop_duplicates()` and set the `keep=False` parameter to remove any of the duplicate data points. The compare the shapes of the results for both sets.

In [7]:
```
print('Oversampling:', features_over.drop_duplicates(keep=False).shape)
print('SMOTE:', features_smote.drop_duplicates(keep=False).shape)
```

```
Oversampling: (865, 15)
SMOTE: (1724, 15)
```

You should receive 865 out of 1724 non-duplicated values for regular oversampling and 1724 of 1724 non-duplicated values for SMOTE. So SMOTE didn't just create duplicates. SMOTE has the advantage that algorithms such as k-Nearest Neighbors are no longer confused by countless duplicates, but now we have data points that are only approximations and are not real.

The most practical and probably most widespread strategy for dealing with unbalanced target categories is the following:

- Give more weight to the incorrect classification of minority data points when training the model.

This strategy doesn't start with the data, but with the classification algorithm itself. In our training data, the ratio of `'attrition'` people to staff who stayed is approximately 1:5. If you set the `class_weight` parameter to `balanced` when you instantiate the model, then in our case, the false classification of a data point in category `1` (leaves the company) is weighted five times more than the false classification of a data point in category `0` when training the model.

Import `DecisionTreeClassifier` from the `sklearn.tree` module and instantiate two different decision tree models, each with 12 decision lines:

- `model_unbalanced` should weight both target vector categories equally (the default setting)
- `model_balanced_by_class_weights` should weight the target vector categories in such a way that overall the smaller category gets as much weight as the larger category.

Tip: Use `random_state=42` again.

In [8]:
```
from sklearn.tree import DecisionTreeClassifier
model_unbalanced = DecisionTreeClassifier(random_state=42, max_depth=12)
model_balanced_by_class_weights = DecisionTreeClassifier(random_state=42, max_depth=12
```

Congratulations: You have learned four approaches to compensate for size differences in the target vector categories. Next, we'll look at how well these approaches perform by calculating the model quality measures from the test data.

Evaluating compensation approaches

We'll evaluate all the *resampling* strategies we've learned about using model performances. However, we should first give some thought to where *resampling* should actually be used in the data science workflow. As mentioned earlier, *resampling* deletes or duplicates data and creates an artificial balance of target categories. However, we should only ever select data points from our training set for this and not from our test data set. After all, the test data set should create a test environment for our model that is as real as possible, and in reality we can't choose what kind of data our models have access to.

So remember to resample after you have split the training and test data! But how do we find out whether our model has any chance at all to do well in the test set? Very simply: By splitting off one or more validation sets from the training set and carrying out cross validation. But we also have to make sure that the validation set is similar to our test set and therefore to reality. This means that we can't resample the validation data either.

So the right time for *resampling* with cross-validation is **after** the **training-validation split** and should only be carried out on the data which the model actually trains with. Unfortunately, each iteration of the cross-validation process involves a different subset of the training set, so we cannot just *resample* our entire training set and use it directly for cross validation. So we need to include the resampling process with the *fitting* process of the model. The best way to achieve

this is with a pipeline. Unfortunately, `sklearn` doesn't (yet) have an implementation to include sampling strategies in its pipeline, because this would require resizing the target vector (*target*) when calling the `.fit()` method. But thankfully there is an implementation in `imblearn`. There you will find in `imblearn.pipeline` - a Pipeline class which is completely compatible with `sklearn`.

Now we'll build a pipeline that first resamples our data and then applies a decision tree to it. Then iterate through our four resampling strategies and use a *grid search* to find the best model. Using the evaluation metrics, we can then see which strategy works best with this data set.

Let's start by importing all the objects we need. The only thing that is new is to import `Pipeline` from `imblearn.pipeline` and not from sklearn.

In [9]:
```
from imblearn.over_sampling import SMOTE, RandomOverSampler
from imblearn.under_sampling import RandomUnderSampler

from imblearn.pipeline import Pipeline #our new pipeline builer

from sklearn.model_selection import GridSearchCV
from sklearn.metrics import precision_score, recall_score
```

Now we need a model that we can use for evaluation. Instantiate `tree_clf` as a simple `DecisionTreeClassifier`, setting only `random_state=42` to ensure that training is the same for each tree.

In [10]:
```
tree_clf = DecisionTreeClassifier(random_state=42)
```

Next, we'll prepare the grid search. We'll optimize the `max_depth` and `class_weight` hyperparameters in the search for the optimal model. We'll iterate through `max_depth` with every other number from 2 up to but not including 16 (2, 4, 6, 8, 10, 12, 14) so that the grid search doesn't take so long.

For `class_weight` we'll just check the options `[None, 'balanced']`. The name of the pipeline step, which will later refer to our model, will be `estimator`. Define a corresponding `search_space`.

In [11]:
```
search_space = {'estimator__max_depth': range(2, 16, 2), 'estimator__class_weight': [N
```

Unfortunately, we can only test different parameters during a grid search, but we can't exchange complete pipeline sections. So we have to use a little trick to accommodate our *sampler* objects. We'll simply create a collection of our samplers and iterate through that collection in a `for` loop. At each iteration we then create a new pipeline with the relevant sampler.

Now let's create this collection in the form of a list, so that we know afterwards which sampler object stands for which sampling method. In this list, for each sampler we store a tuple with the structure `('description_string',sampler-object)`.

We've prepared this collection for you:

In [12]:
```
samplers = [('oversampling', oversampler),
            ('undersampling', undersampler),
            ('class_weights', 'passthrough'),
            ('SMOTE', smotesampler)]
```

Note that the sampler in `class_weights` is not a real *sampler object*, but merely the `'passthrough'`, which both `sklearn` pipelines and the `imblearn` interpret "skip this step".

Let's now get down to business: evaluating the sampling strategies. Create a for loop with the following functionality:

For each `name`, `sampler` in `samplers`:

- Create a pipeline called `imb_pipe` with `sampler` in the first piece and `('estimator', tree_clf)` as the second piece.
- Optimize `imp_pipe` using a *grid search* which optimizes `f1` score with fivefold cross validation and runs with `n_jobs=-1`.
- Store the best model of the *grid search* as `model` and print it.
- Determine the `recall_score` and `precision_score` of the predictions of `model` based on the test data.
- Add `name`, `recall_score` and `precision_score` to a `results` `DataFrame`, or print this information.

In [13]:
```
# storage container for results
results = []

# go through every sampler
for name, sampler in samplers:
    #sampling
    imb_pipe = Pipeline([('sampler', sampler),
                         ('estimator', tree_clf)])

    #gridsearch and CV
    grid = GridSearchCV(estimator=imb_pipe,
                        param_grid=search_space,
                        n_jobs=-1,
                        cv=5,
                        scoring='f1')

    grid.fit(features_train, target_train)

    #evaluation
    model = grid.best_estimator_.named_steps['estimator']
    recall = recall_score(target_test, model.predict(features_test))
    precision = precision_score(target_test, model.predict(features_test))

    #verbose
    print(name.upper())
    print(grid.scoring, 'on Validationset:', grid.best_score_ )
    print("precision :", precision)
```

```
    print("recall :", recall)
    print(model)
    print('#'*11)

    #save
    scores = {'name': name,
              'precision': precision,
              'recall': recall}
    results.append(scores)

#show results
pd.DataFrame(results)
```

```
OVERSAMPLING
f1 on Validationset: 0.44496255288980063
precision : 0.2753623188405797
recall : 0.5428571428571428
DecisionTreeClassifier(max_depth=4, random_state=42)
############
UNDERSAMPLING
f1 on Validationset: 0.38931415617182213
precision : 0.2883435582822086
recall : 0.6714285714285714
DecisionTreeClassifier(max_depth=2, random_state=42)
############
CLASS_WEIGHTS
f1 on Validationset: 0.413245991624491
precision : 0.30597014925373134
recall : 0.5857142857142857
DecisionTreeClassifier(class_weight='balanced', max_depth=4, random_state=42)
############
SMOTE
f1 on Validationset: 0.4432957965466156
precision : 0.4
recall : 0.5142857142857142
DecisionTreeClassifier(max_depth=4, random_state=42)
############
```

Out[13]:

	name	precision	recall
0	oversampling	0.275362	0.542857
1	undersampling	0.288344	0.671429
2	class_weights	0.305970	0.585714
3	SMOTE	0.400000	0.514286

You should get the following values:

Your results might differ slightly if you haven't set `random_state=42` everywhere.

	name	precision	recall
0	oversampling	0,275	0,543
1	undersampling	0,288	0,671
2	class_weights	0,306	0,586
3	SMOTE	0,413	0,371

Balancing the `'attrition'` categories during the model fitting using the `class_weights` parameter results in the second best precision and second best recall. You should be extremely careful with the extremely good recall value `undersampling` receives. In this case, the training data set is so small that we can expect these values to depend heavily on which data is used in the training data. The model with the `SMOTE` resampling strategy has the greatest relevance, but it also has the worst recall.

Important: You can find the `class_weights` parameter in almost all `sklearn` classification algorithms. Unfortunately, the most important exception is `NeighborsClassifier`, which currently doesn't offer this functionality. It's generally a good idea to set `class_weights` to `class_weight='balanced'`, even if the differences between the target vector categories are not that big. This ensures that even the smaller category, whose classification is typically of particular interest, is well predicted.

Congratulations: You have learned how to use the `class_weights` parameter to train classification models so that both target vector categories are equally important. Alternatively, you can reduce the larger category or increase the smaller category by duplicating or inserting synthetic data (SMOTE). But proceed with caution - especially with undersampling. Usually we recommend the solution with `class_weights` or `SMOTE`.

But thankfully you now know the bootstrap procedure, which we can use in the next lesson to progress from decision trees to decision forests, known as random forests. Random forests are a very popular classification approach. We'll look at them in the next lesson.

Remember:

- The bootstrap procedure is like taking a data point and putting it back so it can be taken again.
- For classification algorithms in general, specify `class_weight='balanced'` or resample the data with `SMOTE`.
- With *undersampling* and *oversampling* the number of data points in the classes is adjusted.
