How to deal with imbalanced data in Python

The pythonic solution to balance the imbalanced

Jack Tan Nov 11, 2020 · 8 min read ⋆

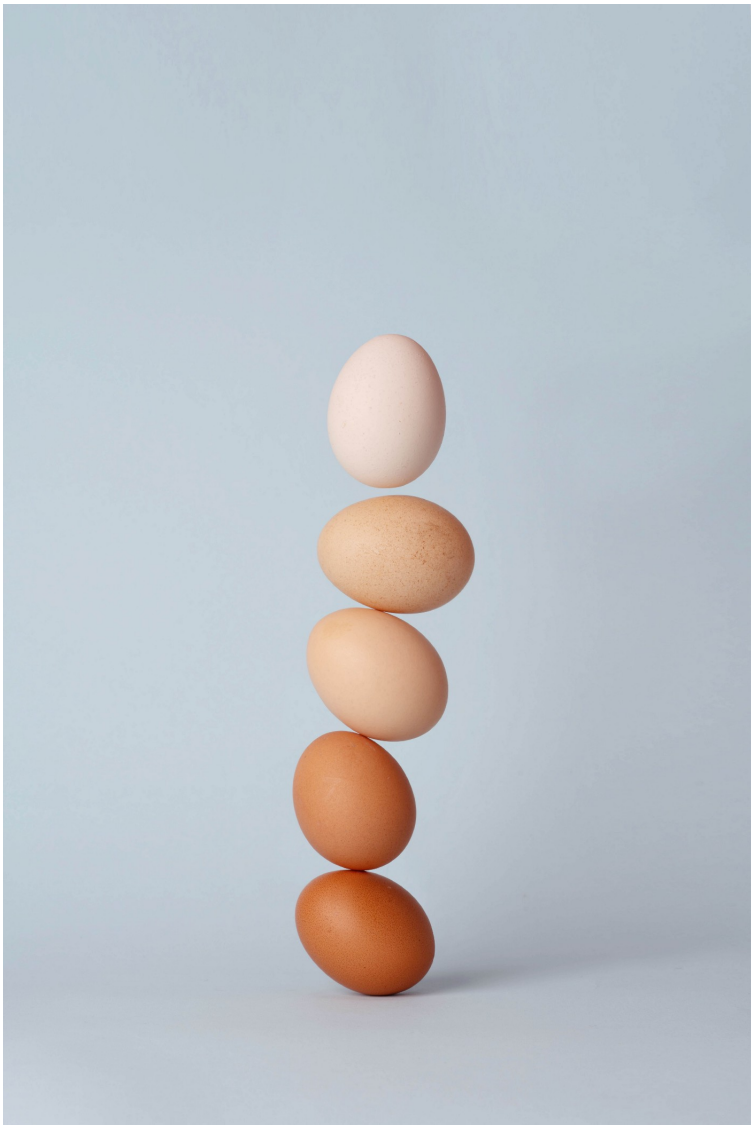


Photo by 青晨 on Unsplash

Imbalanced data is a very common occurrence in real-world domains, especially when the subject of interest for a decision-making system is a rare but important case. This can be a problem when a future decision is to be made based on insights from historical data. Inadequate data from the minority case can hamper the robustness of the new decision being made.

The case of imbalanced data exists almost to any real-life applications. For instance, the

range.

So, what can the problem be when using imbalanced data in Machine Learning (ML)? How can you deal with the imbalanced data? This article will walk you through all these questions and show you some practical examples in Python.

As a surprise, I will include a "one more thing" bonus at the end of this article on how to choose the best sampling ratio using a combination of oversampling and undersampling method.

What are imbalanced data precisely?

> ***Imbalanced data*** *sets are a special case for* ***classification problem*** *where the* ***class distribution is not uniform*** *among the classes. Typically, they are composed by two classes: The* ***majority (negative) class*** *and the* ***minority (positive) class*** *[1].*

Normally the minority class is what we hope the ML model would be able to correctly predict and thus known as the positive class. For instance, if a model is trying to train on a set of 10,000 customer data with 3% conversion rate, it might not have enough samples to be trained well enough to predict the probability of positive conversion in the future.

So, how do we define the threshold for imbalanced data exactly?

According to Google Developers, the answer can be divided into 3 degrees of imbalance — mild (20–40%), moderate (1–20%), and extreme (<1%).

Why does it matter?

The impacts of imbalanced data are implicit, i.e. it does not raise an immediate error when you build and run your model, but the results can be delusive. If the degree of class imbalance for the majority class is extreme, then a machine trained classifier may yield high overall prediction accuracy since the model is most likely to predict most samples belonging to the majority class. For example, in a two-class problem with a class distribution of 90:10, the performance of the classifier on majority-class examples will count nine times as much as the performance on minority-class examples.

How to deal with imbalanced data?

Several solutions have been suggested in the literature to address this problem, amongst which are:

- **Data-level techniques —** At the data level, solutions work by applying resampling techniques to balance the dataset. These can be done by oversampling the minority class, which is to synthetically create new instances from existing ones; or undersampling the majority class, which eliminates some instances in the majority class. However, both techniques can have their drawbacks. Oversampling new data can cause the classifier to overfit; whereas undersampling can discard essential information. A combination of both techniques with a heuristic approach can be found in specialized literature with excellent results.
- **Algorithmic-level techniques —**Algorithmic level solutions can be done by adjusting weighted costs accordingly to the number of training instances in each class. In parametric classifier like Support Vector Machine, grid search and cross-validation can be applied to optimise the C and gamma values. For non-parametric classifier like the decision tree, adjusting the probabilistic estimate at the tree leaf can improve the performance.
- **A combination of both —** A hybrid approach is also constantly being explored in various literature, including AdaOUBoost (adaptive over-sampling and undersampling boost) proposed by Peng and Yao and Learning By Recognition, using the concept of auto association-based classification approach proposed by Japkowicz.

One of the most popular libraries for sampling methods in Python is none other than the imbalanced-learn package. It provides several methods for both over- and undersampling, as well as some combinational methods. For this tutorial, we will explore one example for each of these 3 methods:

- Random undersampling with RandomUnderSampler
- Oversampling with SMOTE (Synthetic Minority Over-sampling Technique)
- A combination of both random undersampling and oversampling using pipeline

The dataset used in this tutorial is based on the bank marketing data from the UCI repo. It is a classification problem where the goal is to predict if the client will subscribe to a term deposit. All the categorical (text) feature columns are encoded into machine-friendly numerical forms. The encoded dataset is hosted on Github. In order to test how good the data perform after resampling, we train the *Support Vector Machine* model on the resampled data to check for the model performance. The complete Python codes can also be found in the same Github repository.

The reason why this dataset is chosen because it reflects the common imbalanced dataset experienced in daily applications. As expected, the data is highly imbalanced and only around 13% [5,289/(5,289+39,922)] of the contacted clients actually subscribed to a term deposit. In Python, the number of each predicted class can be printed using the method `value_counts()`.

```
[7] print(df['y'].value_counts())

0    39922
1     5289
Name: y, dtype: int64
```

Distribution of the targeted classes. 0 is the majority negative class aka clients who did not convert whereas 1 represents the minority positive class aka clients who subscribed to a term deposit.

1. **Random Undersampling**

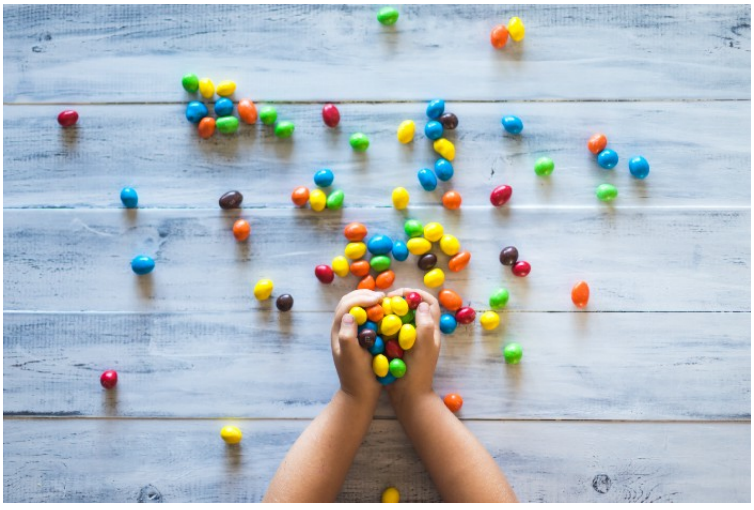


Photo by Patrick Fore on Unsplash

As the name suggests, random undersampling reduces the number of majority class randomly down to the desired ratio against the minority class. This is probably the easiest way to undersample and can actually yield good results if there are a lot of the majority class instances that are close to each other. For a more statistical way of reducing majority class, you can refer to Condensed nearest neighbour (aka. CNN, based

techniques are available in the same imbalanced-learn package which you can easily import to try them out.

The code snippet is split into two parts: PART 1 shows you the codes to random undersample the majority class; PART 2 trains the resampled data with *Support Vector Machine* and output the ROC AUC score.

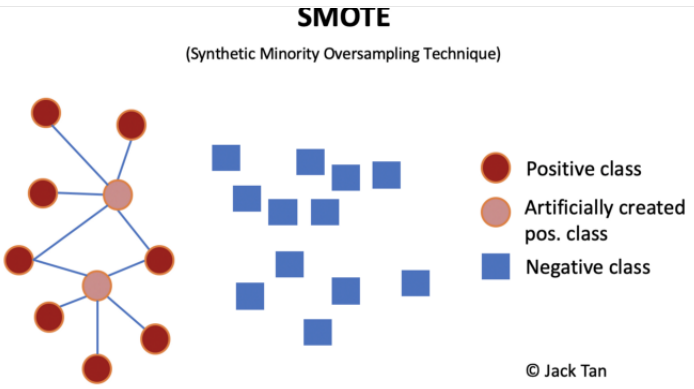
```
#PART 1
# import random undersampling and other necessary libraries
from collections import Counter
from imblearn.under_sampling import RandomUnderSampler
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

#import data
url = "https://raw.githubusercontent.com/jackty9/Handling_Imbalanced_Data_in_Python/mas
df = pd.read_csv(url)

# Separating the independent variables from dependent variables
X = df.iloc[:,:-1]
y = df.iloc[:,-1]

#Split train-test data
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.30)

# summarize class distribution
print("Before undersampling: ", Counter(y_train))

# define undersampling strategy
undersample = RandomUnderSampler(sampling_strategy='majority')

# fit and apply the transform
X_train_under, y_train_under = undersample.fit_resample(X_train, y_train)

# summarize class distribution
print("After undersampling: ", Counter(y_train_under))

#PART 2
# import SVM libraries
from sklearn.svm import SVC
from sklearn.metrics import classification_report, roc_auc_score

model=SVC()
clf_under = model.fit(X_train_under, y_train_under)
pred_under = clf_under.predict(X_test)

print("ROC AUC score for undersampled data: ", roc_auc_score(y_test, pred_under))
```

random_undersampling_imblearn.py hosted with ❤ by **GitHub** **view raw**

```
Before undersampling:  Counter({0: 27962, 1: 3685})
After undersampling:  Counter({0: 3685, 1: 3685})
ROC AUC score for undersampled data:  0.7388684434398952
```

The output of the code snippet: 1st line shows the original data distribution; 2nd line shows the undersampled distribution; 3rd line shows the model performance using the undersampled data

2. Oversampling with SMOTE



An illustration of oversampling with SMOTE using 5 as k nearest neighbours. Self-illustrated by the author.

For over-sampling techniques, SMOTE (Synthetic Minority Oversampling Technique) is considered as one of the most popular and influential data sampling algorithms in ML and data mining. With SMOTE, the minority class is over-sampled by creating "synthetic" examples rather than by over-sampling with replacement [2]. These introduced synthetic examples are based along the line segments joining a defined number of k minority class nearest neighbours, which is in the `imblearn` package is set at five by default.

The code snippet is split into two parts: PART 1 shows you the codes to oversample the minority class with SMOTE; PART 2 trains the resampled data with *Support Vector Machin*e and output the ROC AUC score.

```python
# import SMOTE oversampling and other necessary libraries
from collections import Counter
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

#import data
url = "https://raw.githubusercontent.com/jackty9/Handling_Imbalanced_Data_in_Python/mas
df = pd.read_csv(url)

# Separating the independent variables from dependent variables
X = df.iloc[:,:-1]
y = df.iloc[:,-1]

#Split train-test data
X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.30)

# summarize class distribution
print("Before oversampling: ",Counter(y_train))

# define oversampling strategy
SMOTE = SMOTE()

# fit and apply the transform
X_train_SMOTE, y_train_SMOTE = SMOTE.fit_resample(X_train, y_train)

# summarize class distribution
print("After oversampling: ",Counter(y_train_SMOTE))

#PART 2
# import SVM libraries
from sklearn.svm import SVC
from sklearn.metrics import classification_report, roc_auc_score

model=SVC()
clf_SMOTE = model.fit(X_train_SMOTE, y_train_SMOTE)
pred_SMOTE = clf_SMOTE.predict(X_test)

print("ROC AUC score for oversampled SMOTE data: ", roc_auc_score(y_test, pred_SMOTE))
```

SMOTE_oversampling_imblearn.py hosted with ❤ by **GitHub** **view raw**

```
Before oversampling:  Counter({0: 27943, 1: 3704})
After oversampling:  Counter({1: 27943, 0: 27943})
ROC AUC score for oversampled SMOTE data:  0.7493411314174148
```

The output of the code snippet: 1st line shows the original data distribution; 2nd line shows the oversampled distribution; 3rd line shows the model performance using the oversampled data

3. A combination of under- and oversampling method using pipeline

Photo by The Creative Exchange on Unsplash

The code snippet is split into two parts: PART 1 shows you the codes to create a pipeline to resample the data using both under- and oversampling method; PART 2 trains the resampled data with *Support Vector Machin*e and output the mean ROC AUC score after cross-validation.

```
#PART 1
# import sampling and other necessary libraries
from collections import Counter
from imblearn.over_sampling import SMOTE
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)
from sklearn.svm import SVC
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from imblearn.pipeline import Pipeline

#import data
url = "https://raw.githubusercontent.com/jackty9/Handling_Imbalanced_Data_in_Python/mas
df = pd.read_csv(url)

# Separating the independent variables from dependent variables
X = df.iloc[:,:-1]
y = df.iloc[:,-1]

# define pipeline
model = SVC()
over = SMOTE(sampling_strategy=0.4)
under = RandomUnderSampler(sampling_strategy=0.5)
steps = [('o', over), ('u', under), ('model', model)]
pipeline = Pipeline(steps=steps)

#PART 2
# import libraries for evaluation
from sklearn.model_selection import cross_val_score
from sklearn.metrics import roc_auc_score
from numpy import mean

# evaluate pipeline
scores = cross_val_score(pipeline, X, y, scoring='roc_auc', cv=5, n_jobs=-1)
score = mean(scores)
print('ROC AUC score for the combined sampling method: %.3f' % score)
```

A_combination_of_over_and_under_sampling_imblearn.py hosted with ❤ by **GitHub** **view raw**

```
ROC AUC score for the combined sampling method: 0.820
```

The output of the code snippet showing the ROC AUC scores using a combined sampling method.

Photo by Ion Tyson on Unsplash

technique (random or SMOTE) to use. However, most researches have pointed out that undersampling in most cases does produce better results than oversampling. In industrial practice, it has been reported that tech giants like Facebook and Microsoft also tend to use undersampling method when it comes to classifying their ad performance. The common argument is that undersampling is generally "cheaper" than oversampling and since the class of interest is the minority positive class, reducing some information of the majority negative class is acceptable. But there is no guarantee that the same can always be replicated in your case.

My advice is to decide based on the time and resources you have when handling imbalanced data. If you have a limited amount of time and resources to investigate which method works best for you, then you can opt for random undersampling from the get-go. However, if you have a bit of luxury to try out different sampling methods and techniques, I will highly recommend the latter. Once you have a well-balanced data, your model will potentially learn to classify different classes better for a more reliable prediction.

In the tutorial above, we either under- or/and oversample the data to have a balanced 50:50 distribution of both classes. But what if we want to find out the best sampling ratio based on the model performance? We can do this by implementing a `for` loops in Python to iterate a possible list of sampling ratio and see which combination has the best performance based on the mean ROC AUC score (or other performance metrics of choice) after cross-validation.

The code snippet shows you the codes to create a pipeline to resample the data using both under- and oversampling method. The list of possible ratios for oversampling is `[0.3, 0.4, 0.5]` while for undersampling `[0.7, 0.6, 0.5]`. These ratios are selected randomly for demonstration purposes and can be replaced with other values after some trial and error. The *Support Vector Machine* is then trained with the resampled data for every possible pair of the ratios combination before outputting the mean ROC AUC score after cross-validation.

The next step after dealing with imbalanced data will probably be to select the best features for your model, for that you are welcome to check out my article in Feature Selection.

**Disclaimer: Due to the stochastic nature of the train-test split and cross-validation, you might not get the exact output as shown.*

Reference:

[1] Imbalanced data set, Keel 2018

[2] SMOTE: Synthetic Minority Over-sampling Technique

Data Science Programming Python Machine Learning Imbalanced Data

```python
import numpy as np
from sklearn.svm import SVC
from imblearn.over_sampling import SMOTE
from imblearn.under_sampling import RandomUnderSampler
from imblearn.pipeline import Pipeline
from sklearn.model_selection import cross_val_score
from sklearn.metrics import roc_auc_score
from numpy import mean

#import data
url = "https://raw.githubusercontent.com/jackty9/Handling_Imbalanced_Data_in_Python/mas
df = pd.read_csv(url)

# Separating the independent variables from dependent variables
X = df.iloc[:,:-1]
y = df.iloc[:,-1]

# values to evaluate
over_values = [0.3,0.4,0.5]
under_values = [0.7,0.6,0.5]
for o in over_values:
  for u in under_values:
    # define pipeline
    model = SVC()
    over = SMOTE(sampling_strategy=o)
    under = RandomUnderSampler(sampling_strategy=u)
    steps = [('over', over), ('under', under), ('model', model)]
    pipeline = Pipeline(steps=steps)
    # evaluate pipeline
    scores = cross_val_score(pipeline, X, y, scoring='roc_auc', cv=5, n_jobs=-1)
    score = mean(scores)
    print('SMOTE oversampling rate:%.1f, Random undersampling rate:%.1f , Mean ROC AUC:
```

Finding_the_best_sampling_ratio.py hosted with ❤ by **GitHub** **view raw**

```
SMOTE oversampling rate:0.3, Random undersampling rate:0.6 , Me
SMOTE oversampling rate:0.3, Random undersampling rate:0.5 , Me
SMOTE oversampling rate:0.4, Random undersampling rate:0.7 , Me
SMOTE oversampling rate:0.4, Random undersampling rate:0.6 , Me
SMOTE oversampling rate:0.4, Random undersampling rate:0.5 , Me
SMOTE oversampling rate:0.5, Random undersampling rate:0.7 , Me
SMOTE oversampling rate:0.5, Random undersampling rate:0.6 , Me
SMOTE oversampling rate:0.5, Random undersampling rate:0.5 , Me
```

The output of the code snippet for the mean ROC AUC scoring for every possible pair of ratios combination.

As seen in the output snapshot, the best sampling ratio in this example is to have an oversampling ratio of 0.5, follows by an under mean ROC AUC score at 82.1%.