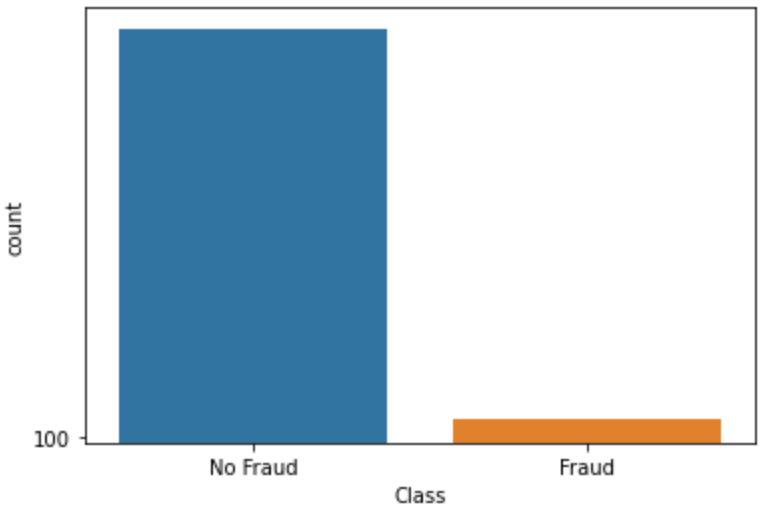
10 Techniques to deal with Imbalanced Classes in Machine Learning

Overview

- Get familiar with class imbalance
- Understand various techniques to treat imbalanced classes such as-
    - Random under-sampling
    - Random over-sampling
    - NearMiss
- You can check the implementation of the code in my GitHub repository here

Introduction

When observation in one class is higher than the observation in other classes then there exists a class imbalance. Example: To detect fraudulent credit card transactions. As you can see in the below graph fraudulent transaction is around 400 when compared with non-fraudulent transaction around 90000.



Class Imbalance is a common problem in machine learning, especially in classification problems. Imbalance data can hamper our model accuracy big time.

Class Imbalance appear in many domains, including:

- Fraud detection
- Spam filtering
- Disease screening
- SaaS subscription churn

- Advertising click-throughs

The Problem with Class Imbalance

Most machine learning algorithms work best when the number of samples in each class are about equal. This is because most algorithms are designed to maximize accuracy and reduce errors.

However, if the data set in imbalance then In such cases, you get a pretty high accuracy just by predicting the **majority class**, but you fail to capture the **minority class**, which is most often the point of creating the model in the first place.

Credit card fraud detection example

Let's say we have a dataset of credit card companies where we have to find out whether the credit card transaction was fraudulent or not.

But here's the catch... the fraud transaction is relatively rare, only 6% of the transaction is fraudulent.

Now, before you even start, do you see how the problem might break? Imagine if you didn't bother training a model at all. Instead, what if you just wrote a single line of code that always predicts 'no fraudulent transaction'

```
def transaction(transaction_data):
    return 'No fradulent transaction'
```

Well, guess what? Your "solution" would have 94% accuracy!

Unfortunately, that accuracy is misleading.

- All those **non-fraudulent** transactions, you'd have 100% accuracy.
- Those transactions which are **fraudulent**, you'd have 0% accuracy.
- Your overall **accuracy would be high** simply because the most transaction is not fraudulent(not because your model is any good).

This is clearly a problem because many machine learning algorithms are designed to maximize overall accuracy. In this article, we will see different techniques to handle the imbalanced data.

Data

We will use a credit card fraud detection dataset for this article you can find the dataset from here.

After loading the data display the first five-row of the data set.

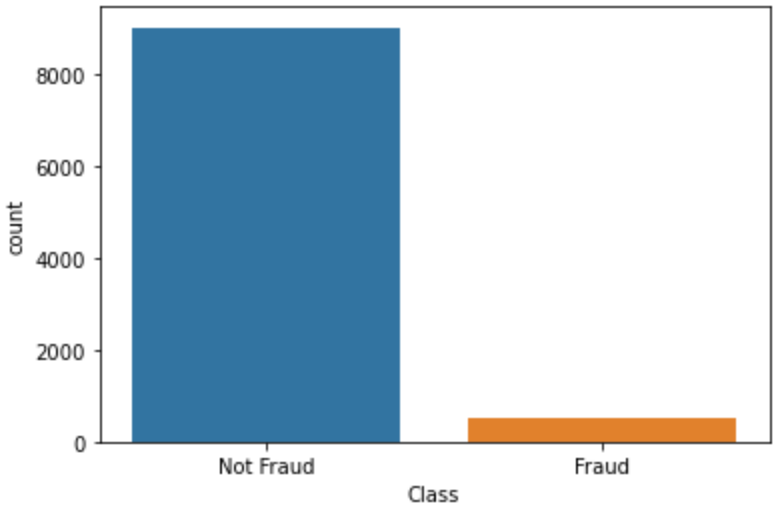
```
data = pd.read_csv(r'C:\\Users\\Benai\\Documents\\machin-learning\\imbalance-data\creditcard.csv')
```

```
data.head()
```

V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928	0.128539	-0.189115	0.133558	-0.021053	149.62	0
.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846	0.167170	0.125895	-0.008983	0.014724	2.69	0
.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281	-0.327642	-0.139097	-0.055353	-0.059752	378.66	0
.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.190321	-1.175575	0.647376	-0.221929	0.062723	0.061458	123.50	0
.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	-0.137458	0.141267	-0.206010	0.502292	0.219422	0.215153	69.99	0

```
# check the target variable that is fraudulet and not fradulent
transactiondata['Class'].value_counts()# 0 -> non fraudulent
# 1 -> fraudulent

# visualize the target variable
g = sns.countplot(data['Class'])
g.set_xticklabels(['Not Fraud','Fraud'])
plt.show()
```

```
0    9000
1     492
```



You can clearly see that there is a huge difference between the data set. 9000 non-fraudulent transactions and 492 fraudulent.

The Metric Trap

One of the major issues that new developer users fall into when dealing with unbalanced datasets relates to the metrics used to evaluate their model. Using simpler metrics like **accuracy score** can be misleading. In a dataset with highly unbalanced classes, the classifier will always "predicts" the most common class without performing any analysis of the features and it will have a high accuracy rate, obviously not the correct one.

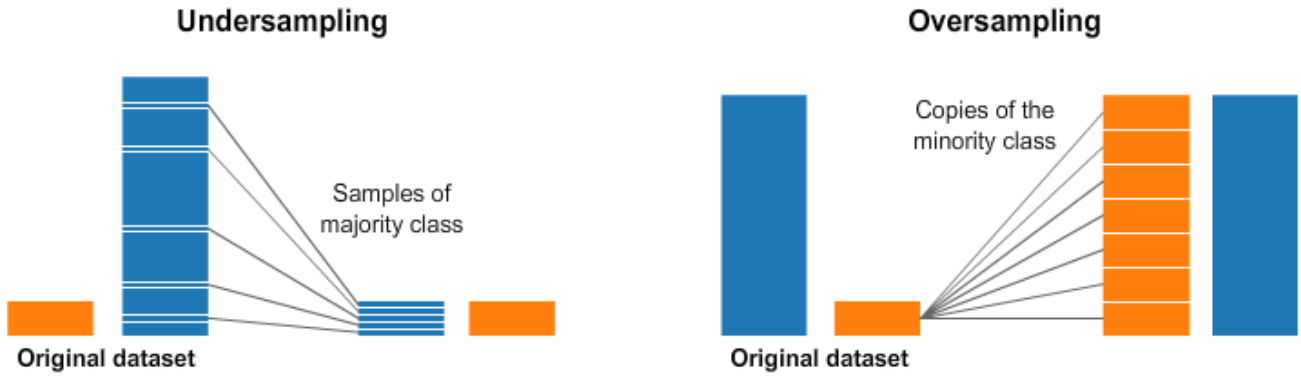
Let's do this experiment, using simple XGBClassifier and no feature engineering:

```
# import linrary
from xgboost import XGBClassifier

xgb_model = XGBClassifier().fit(x_train, y_train)

# predict
xgb_y_predict = xgb_model.predict(x_test)

# accuracy score
xgb_score = accuracy_score(xgb_y_predict, y_test)

print('Accuracy score is:', xbg_score)OUTPUT
Accuracy score is: 0.992
```

We can see 99% accuracy, we are getting very high accuracy because it is predicting mostly the **majority** class that is 0 (Non-fraudulent).

Resampling Technique

A widely adopted technique for dealing with highly unbalanced datasets is called resampling. It consists of removing samples from the majority class (under-sampling) and/or adding more examples from the minority class (over-sampling).



Despite the advantage of balancing classes, these techniques also have their weaknesses (there is no free lunch).

The simplest implementation of **over-sampling** is to duplicate random records from the minority class, which can cause overfishing.

In **under-sampling**, the simplest technique involves removing random records from the majority class, which can cause loss of information.

Let's implement this with the credit card fraud detection example.

We will start by separating the class that will be 0 and class 1.

```
# class count
class_count_0, class_count_1 = data['Class'].value_counts()

# Separate class
class_0 = data[data['Class'] == 0]
class_1 = data[data['Class'] == 1]# print the shape of the class
print('class 0:', class_0.shape)
print('class 1:', class_1.shape
```

```
class 0: (9000, 31)

class 1: (492, 31)
```

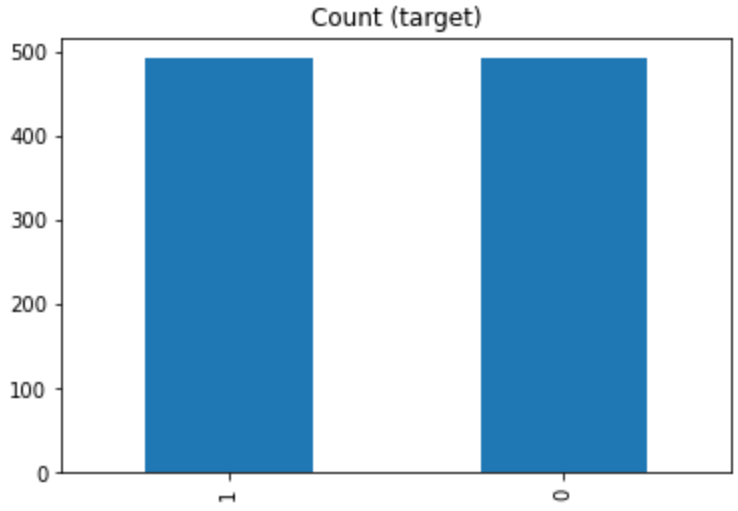
1. Random Under-Sampling

Undersampling can be defined as **removing some observations of the majority class**. This is done until the majority and minority class is balanced out.

Undersampling can be a good choice when you have a ton of data -think millions of rows. But a drawback to undersampling is that we are removing information that may be valuable.

```
class_0_under = class_0.sample(class_count_1)

test_under = pd.concat([class_0_under, class_1], axis=0)

print("total class of 1 and0:",test_under['Class'].value_counts())# plot the count
after under-sampeling
test_under['Class'].value_counts().plot(kind='bar', title='count (target)')
```

```
total class of 1 and 0:
 1    492
0    492
Name: Class, dtype: int64
```



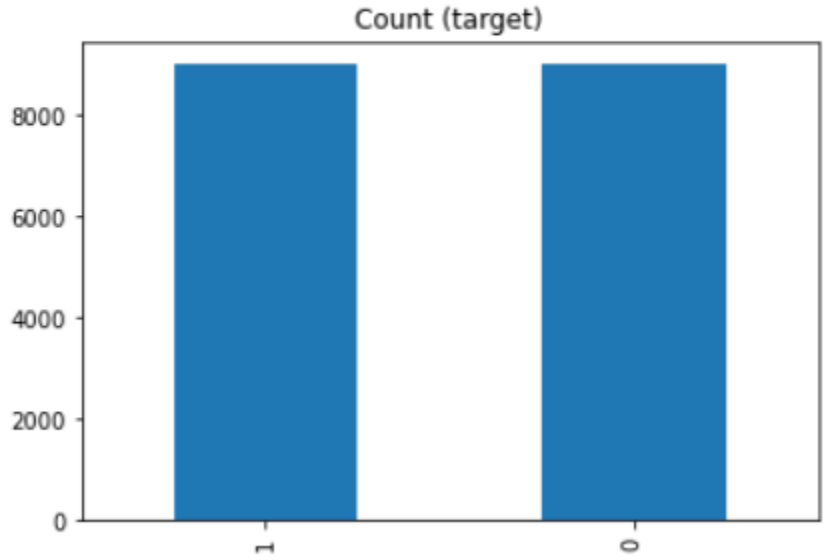
2. Random Over-Sampling

Oversampling can be defined as adding more copies to the minority class. Oversampling can be a good choice when you don't have a ton of data to work with.

A con to consider when undersampling is that it can cause overfitting and poor generalization to your test set.

```
class_1_over = class_1.sample(class_count_0, replace=True)

test_over = pd.concat([class_1_over, class_0], axis=0)

print("total class of 1 and 0:",test_under['Class'].value_counts())# plot the count
after under-sampeling
test_over['Class'].value_counts().plot(kind='bar', title='count (target)')
```

```
class count of 1 and 0:
 1    9000
0    9000
Name: Class, dtype: int64
```



Balance data with the imbalanced-learn python module

A number of more sophisticated resampling techniques have been proposed in the scientific literature.

For example, we can cluster the records of the majority class, and do the under-sampling by removing records from each cluster, thus seeking to preserve information. In over-sampling, instead of creating exact copies of the minority class records, we can introduce small variations into those copies, creating more diverse synthetic samples.

Let's apply some of these resampling techniques, using the Python library imbalanced-learn. It is compatible with scikit-learn and is part of scikit-learn-contrib projects.

```
import imblearn
```

3. Random under-sampling with imblearn

`RandomUnderSampler` is a fast and easy way to balance the data by randomly selecting a subset of data for the targeted classes. Under-sample the majority class(es) by randomly picking samples with or without replacement.

```
# import library
from imblearn.under_sampling import RandomUnderSampler

rus = RandomUnderSampler(random_state=42, replacement=True)# fit predictor and target
variable
x_rus, y_rus = rus.fit_resample(x, y)

print('original dataset shape:', Counter(y))
print('Resample dataset shape', Counter(y_rus))
```

```
original dataset shape: Counter({0: 9000, 1: 492})
Resample dataset shape Counter({0: 492, 1: 492})
```

4. Random over-sampling with imblearn

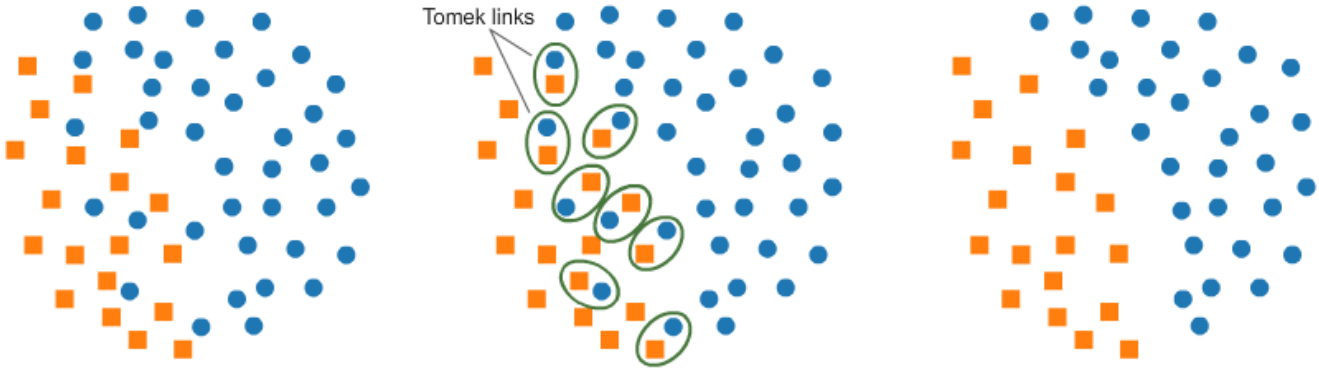
One way to fight imbalance data is to **generate new samples** in the minority classes. The most naive strategy is to generate new samples by randomly sampling with replacement of the currently available samples. The `RandomOverSampler` offers such a scheme.

```
# import library
from imblearn.over_sampling import RandomOverSampler

ros = RandomOverSampler(random_state=42)

# fit predictor and target variablex_ros, y_ros = ros.fit_resample(x, y)

print('Original dataset shape', Counter(y))
print('Resample dataset shape', Counter(y_ros))
```

5. Under-sampling: Tomek links

Tomek links are pairs of very close instances but of opposite classes. Removing the instances of the majority class of each pair increases the space between the two classes, facilitating the classification process.

Tomek's link exists if the two samples are the nearest neighbors of each other



In the code below, we'll use `ratio='majority'` to resample the majority class.

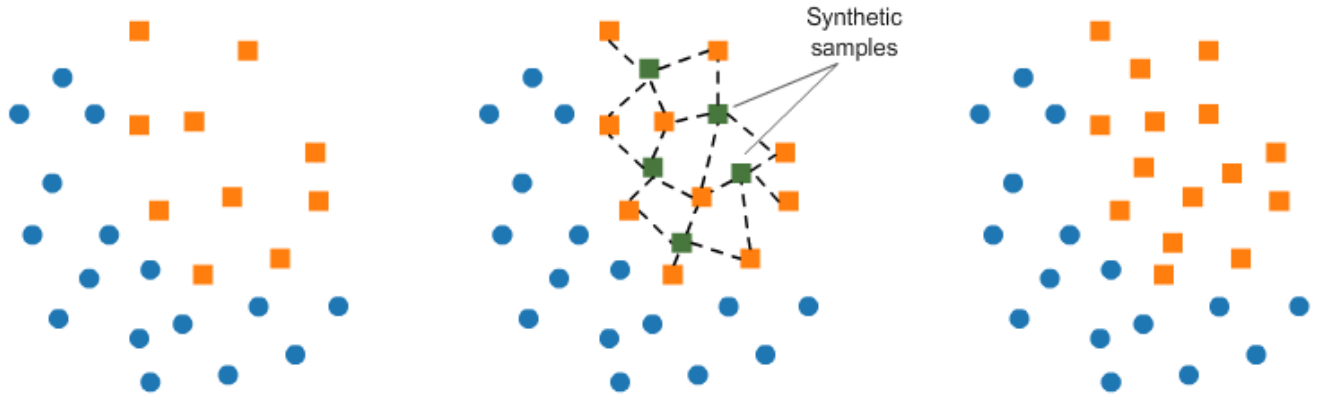
```
# import library
from imblearn.under_sampling import TomekLinks

tl = RandomOverSampler(sampling_strategy='majority')

# fit predictor and target variable
x_tl, y_tl = ros.fit_resample(x, y)

print('Original dataset shape', Counter(y))
print('Resample dataset shape', Counter(y_ros))
```

```
Original dataset shape: Counter({0: 9000, 1: 492})
Resample dataset shape: Counter({0: 8839, 1: 492})
```

6. Synthetic Minority Oversampling Technique (SMOTE)

This technique generates synthetic data for the minority class.

SMOTE (Synthetic Minority Oversampling Technique) works by randomly picking a point from the minority class and computing the k-nearest neighbors for this point. The **synthetic points are added** between the chosen point and its neighbors.



SMOTE algorithm works in 4 simple steps:

1. Choose a minority class as the input vector
2. Find its k nearest neighbors (***k_neighbors*** is specified as an argument in the ***SMOTE()*** function)
3. Choose one of these neighbors and place a synthetic point anywhere on the line joining the point under consideration and its chosen neighbor
4. Repeat the steps until data is balanced

```
# import library
from imblearn.over_sampling import SMOTE

smote = SMOTE()

# fit predictor and target variable
x_smote, y_smote = smote.fit_resample(x, y)

print('Original dataset shape', Counter(y))
print('Resample dataset shape', Counter(y_ros))
```

7. NearMiss

NearMiss is an under-sampling technique. Instead of resampling the Minority class, using a distance, this will make the majority class equal to the minority class.

```
from imblearn.under_sampling import NearMiss

nm = NearMiss()

x_nm, y_nm = nm.fit_resample(x, y)

print('Original dataset shape:', Counter(y))
print('Resample dataset shape:', Counter(y_nm))
```

```
Original dataset shape: Counter({0: 16000, 1: 492})
Resample dataset shape: Counter({0: 492, 1: 492})
```

8. Change the performance metric

Accuracy is not the best metric to use when evaluating imbalanced datasets as it can be misleading.

Metrics that can provide better insight are:

- **Confusion Matrix:** a table showing correct predictions and types of incorrect predictions.
- **Precision:** the number of true positives divided by all positive predictions. Precision is also called Positive Predictive Value. It is a measure of a classifier's exactness. Low precision indicates a high number of false positives.

- **Recall:** the number of true positives divided by the number of positive values in the test data. The recall is also called Sensitivity or the True Positive Rate. It is a measure of a classifier's completeness. Low recall indicates a high number of false negatives.
- **F1: Score:** the weighted average of precision and recall.
- **Area Under ROC Curve** (AUROC): AUROC represents the likelihood of your model distinguishing observations from two classes.
  In other words, if you randomly select one observation from each class, what's the probability that your model will be able to "rank" them correctly?

9. Penalize Algorithms (Cost-Sensitive Training)

The next tactic is to use penalized learning algorithms that increase the cost of classification mistakes on the minority class.

A popular algorithm for this technique is Penalized-SVM.

During training, we can use the argument `class_weight='balanced'` to penalize mistakes on the minority class by an amount proportional to how under-represented it is.

We also want to include the argument `probability=True` if we want to enable probability estimates for SVM algorithms.

Let's train a model using Penalized-SVM on the original imbalanced dataset:

```
# load library
from sklearn.svm import SVC

# we can add class_weight='balanced' to add panalize mistake
svc_model = SVC(class_weight='balanced', probability=True)

svc_model.fit(x_train, y_train)

svc_predict = svc_model.predict(x_test)# check performance
print('ROCAUC score:',roc_auc_score(y_test, svc_predict))
print('Accuracy score:',accuracy_score(y_test, svc_predict))
print('F1 score:',f1_score(y_test, svc_predict))
```

```
ROCAUC score: 0.548660218804503
Accuracy score: 0.4807517429524098
F1 score: 0.07952713594841485
```

10. Change the algorithm

While in every machine learning problem, it's a good rule of thumb to try a variety of algorithms, it can be especially beneficial with imbalanced datasets.

Decision trees frequently perform well on imbalanced data. In modern machine learning, tree ensembles (Random Forests, Gradient Boosted Trees, etc.) almost always outperform singular decision trees, so we'll jump right into those:

Tree base algorithm work by learning a hierarchy of if/else questions. This can force both classes to be addressed.

```
# load library
from sklearn.ensemble import RandomForestClassifier

rfc = RandomForestClassifier()

# fit the predictor and target
rfc.fit(x_train, y_train)

# predict
rfc_predict = rfc.predict(x_test)# check performance
print('ROCAUC score:',roc_auc_score(y_test, rfc_predict))
print('Accuracy score:',accuracy_score(y_test, rfc_predict))
print('F1 score:',f1_score(y_test, rfc_predict))
```

```
ROCAUC score: 0.9074057925056814
Accuracy score: 0.9930281903607153
F1 score: 0.8940092165898618
```

Advantage and disadvantages of Under-sampling

Advantages

It can help improve run time and storage problems by reducing the number of training data samples when the training data set is huge.

Disadvantages

- It can discard potentially useful information which could be important for building rule classifiers.
- The sample chosen by random under-sampling may be a biased sample. And it will not be an accurate representation of the population. Thereby, resulting in inaccurate results with the actual test data set.

Advantages and Disadvantage of over-sampling

Advantages

- Unlike under-sampling, this method leads to no information loss.
- Outperforms under sampling

Disadvantages

It increases the likelihood of overfitting since it replicates the minority class events.