Dealing with Imbalanced Data

A guide to effectively handling imbalanced datasets in Python

Tara Boyle Follow
Feb 4 · 5 min read

Photo by Ales Nesetril on Unsplash

Imbalanced classes are a common problem in machine learning classification where there are a disproportionate ratio of observations in each class. Class imbalance can be found in many different areas including medical diagnosis, spam filtering, and fraud detection.

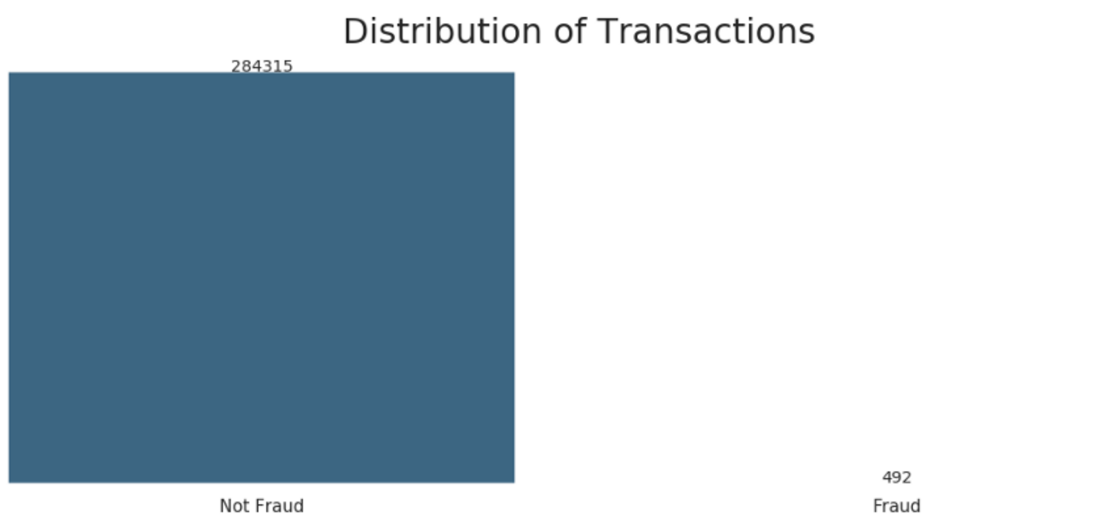
In this guide, we'll look at five possible ways to handle an imbalanced class problem.

Important Note: This guide will focus solely on addressing imbalanced classes and will not addressing other important machine learning steps including, but not limited to, feature selection or hyperparameter tuning.

Data

We will use the Credit Card Fraud Detection Dataset available on Kaggle. The dataset is high imbalanced, with only 0.17% of transactions being classified as fraudulent. The full notebook can be found here.

Our objective will be to correctly classify the minority class of fraudulent transactions.



The Problem with Imbalanced Classes

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 error.

The Problem with Accuracy

```python

# setting up testing and training sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=27)

# DummyClassifier to predict only target 0
dummy = DummyClassifier(strategy='most_frequent').fit(X_train, y_train)
dummy_pred = dummy.predict(X_test)

# checking unique labels
print('Unique predicted labels: ', (np.unique(dummy_pred)))

# checking accuracy
print('Test score: ', accuracy_score(y_test, dummy_pred))

Unique predicted labels:  [0]
Test score:  0.9981461194910255
```

dummy_classifier.py hosted with ♡ by GitHub view raw

We got an accuracy score of 99.8% — And without even training a model! Let's compare this to logistic regression, an actual trained classifier.

```python
# Modeling the data as is
# Train model
lr = LogisticRegression(solver='liblinear').fit(X_train, y_train)

# Predict on training set
lr_pred = lr.predict(X_test)

# Checking accuracy
accuracy_score(y_test, lr_pred)
    0.9992

# Checking unique values
predictions = pd.DataFrame(lr_pred)
predictions[0].value_counts()
    0    71108
    1       94
```

lr_baseline.py hosted with ♡ by GitHub view raw can be very misleading. Metrics that can provide better insight include:

- **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. 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.

Let's see what happens when we apply these F1 and recall scores to our logistic regression from above.

```
# f1 score
f1_score(y_test, lr_pred)
    0.7522

# recall score
recall_score(y_test, lr_pred)
    0.6439
```

change_metric.py hosted with ♡ by GitHub view raw

These scores don't look quite so impressive. Let's see what other methods we might try to improve our new metrics.

2. 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. They work by learning a hierarchy of if/else questions and this can force both classes to be addressed.

```
from sklearn.ensemble import RandomForestClassifier

# train model
rfc = RandomForestClassifier(n_estimators=10).fit(X_train, y_train)

# predict on test set
rfc_pred = rfc.predict(X_test)

accuracy_score(y_test, rfc_pred)
    0.9995

```

change_algorithm.py hosted with ♡ by GitHub view raw

While our accuracy score is slightly lower, both F1 and recall have increased as compared to logistic regression! It appears that for this specific problem, random forest may be a better choice of model.

3. Resampling Techniques — Oversample minority class

Our next method begins our resampling techniques.

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

We will use the resampling module from Scikit-Learn to randomly replicate samples from the minority class.

Important Note

Always split into test and train sets BEFORE trying oversampling techniques! Oversampling before splitting the data can allow the exact same observations to be present in both the test and train sets. This can allow our model to simply memorize specific data points and cause overfitting and poor generalization to the test data.

```
from sklearn.utils import resample

# Separate input features and target
y = df.Class
X = df.drop('Class', axis=1)

# setting up testing and training sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=27)

# concatenate our training data back together
X = pd.concat([X_train, y_train], axis=1)

# separate minority and majority classes
not_fraud = X[X.Class==0]
fraud = X[X.Class==1]

# upsample minority
fraud_upsampled = resample(fraud,
                          replace=True, # sample with replacement
                          n_samples=len(not_fraud), # match number in majority class
                          random_state=27) # reproducible results

# combine majority and upsampled minority
```

```
    0    213245
```

upsampled.py hosted with ♡ by GitHub view raw

After resampling we have an equal ratio of data points for each class! Let's try our logistic regression again with the balanced training data.

```
# trying logistic regression again with the balanced dataset
y_train = upsampled.Class
X_train = upsampled.drop('Class', axis=1)

upsampled = LogisticRegression(solver='liblinear').fit(X_train, y_train)

upsampled_pred = upsampled.predict(X_test)

# Checking accuracy
accuracy_score(y_test, upsampled_pred)
    0.9807

# f1 score
f1_score(y_test, upsampled_pred)
    0.1437

recall_score(y_test, upsampled_pred)
    0.8712
```

upsampled_lr.py hosted with ♡ by GitHub view raw

Our recall score increased, but F1 is much lower than with either our baseline logistic regression or random forest from above. Let's see if undersampling might perform better here.

4. Resampling techniques — Undersample majority class

Undersampling can be defined as removing some observations of the majority class. Undersampling can be a good choice when you have a ton of data -think millions of rows. But a drawback is that we are removing information that may be valuable. This could lead to underfitting and poor generalization to the test set.

We will again use the resampling module from Scikit-Learn to randomly remove samples from the majority class.

```
# still using our separated classes fraud and not_fraud from above

# downsample majority
not_fraud_downsampled = resample(not_fraud,
                                replace = False, # sample without replacement
```

```
# checking counts
downsampled.Class.value_counts()
    1    360
    0    360
```

downsampled.py hosted with ♡ by GitHub view raw

Again, we have an equal ratio of fraud to not fraud data points, but in this case a much smaller quantity of data to train the model on. Let's again apply our logistic regression.

```
# trying logistic regression again with the undersampled dataset

y_train = downsampled.Class
X_train = downsampled.drop('Class', axis=1)

undersampled = LogisticRegression(solver='liblinear').fit(X_train, y_train)

undersampled_pred = undersampled.predict(X_test)

# Checking accuracy
accuracy_score(y_test, undersampled_pred)
    0.9758

# f1 score
f1_score(y_test, undersampled_pred)
    0.1171

recall_score(y_test, undersampled_pred)
    0.8636
```

downsampled_lr.py hosted with ♡ by GitHub view raw

Undersampling underperformed oversampling in this case. Let's try one more method for handling imbalanced data.

5. Generate synthetic samples

A technique similar to upsampling is to create synthetic samples. Here we will use imblearn's SMOTE or Synthetic Minority Oversampling Technique. SMOTE uses a nearest neighbors algorithm to generate new and synthetic data we can use for training our model.

Again, it's important to generate the new samples only in the training set to ensure our model generalizes well to unseen data.

```
from imblearn.over_sampling import SMOTE
```

```python
# setting up testing and training sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=27)

sm = SMOTE(random_state=27, ratio=1.0)
X_train, y_train = sm.fit_sample(X_train, y_train)
```

smote.py hosted with ♡ by GitHub view raw

After generating our synthetic data points, let's see how our logistic regression performs.

```python
smote = LogisticRegression(solver='liblinear').fit(X_train, y_train)

smote_pred = smote.predict(X_test)

# Checking accuracy
accuracy_score(y_test, smote_pred)
    0.9858

# f1 score
f1_score(y_test, smote_pred)
    0.1846

recall_score(y_test, smote_pred)
    0.8636
```

smote_lr.py hosted with ♡ by GitHub view raw

Our F1 score is increased and recall is similar to the upsampled model above and for our data here outperforms undersampling.

Conclusion

We explored 5 different methods for dealing with imbalanced datasets:

1. Change the performance metric
2. Change the algorithm
3. Oversample minority class
4. Undersample majority class
5. Generate synthetic samples

It appears for this particular dataset random forest and SMOTE are among the best of the options we tried here.

These are just some of the many possible methods to try when dealing with imbalanced datasets, and not an exhaustive list. Some others methods to consider are collecting more data or choosing different resampling ratios — you don't have to have exactly a 1:1 ratio!

You should always try several approaches and then decide which is best for your problem.

WRITTEN BY

Tara Boyle

I'm passionate about all things data! I'm interested in applying machine learning to create business solutions.

Towards Data Science

Sharing concepts, ideas, and codes.

Towards Data Science

Sharing concepts, ideas, and codes.

Following

327

Learning Engineer

Daniel Bourke in Towar... Jul 6 · 11 min read ★ 5.6K

Susan Li in Towards Dat... Jul 17 · 10 min read 896

Admond Lee in Towards... Jul 13 · 11 min read ★ 908