Imbalanced Classification with Python

7-Day Crash-Course

Jason Brownlee

MACHINE LEARNING MASTERY

Disclaimer

Imbalanced Classification with Python Crash Course

Edition: v1.2

Find the latest version of this guide online at: http://MachineLearningMastery.com

Contents

Before We Get Started...

Classification predictive modeling is the task of assigning a label to an example. Imbalanced classification are those classification tasks where the distribution of examples across the classes is not equal. Practical imbalanced classification requires the use of a suite of specialized techniques, data preparation techniques, learning algorithms, and performance metrics. In this crash course, you will discover how you can get started and confidently work through an imbalanced classification project with Python in seven days. Let's get started.

Who Is This Crash-Course For?

Before we get started, let's make sure you are in the right place. This course is for developers that may know some applied machine learning. Maybe you know how to work through a predictive modeling problem end-to-end, or at least most of the main steps, with popular tools. The lessons in this course do assume a few things about you, such as:

You need to know:

- You know your way around basic Python for programming.
- You may know some basic NumPy for array manipulation.
- You want to learn probability to deepen your understanding and application of machine learning.

You do NOT need to know:

- You do not need to be a math wiz!
- You do not need to be a machine learning expert!

This crash course will take you from a developer who knows a little machine learning to a developer who can navigate an imbalanced classification project. This crash course assumes you have a working Python 3 SciPy environment with at least NumPy installed. If you need help with your environment, you can follow the step-by-step tutorial here:

- How to Setup a Python Environment for Machine Learning

Crash-Course Overview

This crash course is broken down into seven lessons. You could complete one lesson per day (recommended) or complete all of the lessons in one day (hardcore). It really depends on the time you have available and your level of enthusiasm. Below is a list of the seven lessons that will get you started and productive with probability for machine learning in Python:

- **Lesson 01**: Challenge of Imbalanced Classification.
- **Lesson 02**: Intuition for Imbalanced Data.
- **Lesson 03**: Evaluate Imbalanced Classification Models.
- **Lesson 04**: Undersampling the Majority Class.
- **Lesson 05**: Oversampling the Minority Class.
- **Lesson 06**: Combine Data Undersampling and Oversampling.
- **Lesson 07**: Cost-Sensitive Algorithms.

Each lesson could take you 60 seconds or up to 30 minutes. Take your time and complete the lessons at your own pace. Ask questions and even share your results online. The lessons expect you to go off and find out how to do things. I will give you hints, but part of the point of each lesson is to force you to learn where to go to look for help on and about the statistical methods and the NumPy API and the best-of-breed tools in Python. (Hint: I have all of the answers directly on this blog; use the search box.) Share your results online, I'll cheer you on!

Hang in there, don't give up!

Lesson 01: Challenge of Imbalanced Classification

In this lesson, you will discover the challenge of imbalanced classification problems. Imbalanced classification problems pose a challenge for predictive modeling as most of the machine learning algorithms used for classification were designed around the assumption of an equal number of examples for each class. This results in models that have poor predictive performance, specifically for the minority class. This is a problem because typically, the minority class is more important and therefore the problem is more sensitive to classification errors for the minority class than the majority class.

- **Majority Class**: More than half of the examples belong to this class, often the negative or normal case.
- **Minority Class**: Less than half of the examples belong to this class, often the positive or abnormal case.

A classification problem may be a little skewed, such as if there is a slight imbalance. Alternately, the classification problem may have a severe imbalance where there might be hundreds or thousands of examples in one class and tens of examples in another class for a given training dataset.

- **Slight Imbalance**. Where the distribution of examples is uneven by a small amount in the training dataset (e.g. 4:6).
- **Severe Imbalance**. Where the distribution of examples is uneven by a large amount in the training dataset (e.g. 1:100 or more).

Many of the classification predictive modeling problems that we are interested in solving in practice are imbalanced. As such, it is surprising that imbalanced classification does not get more attention than it does.

Your Task

For this lesson, you must list five general examples of problems that inherently have a class imbalance. One example might be fraud detection, another might be intrusion detection.

Next

In the next lesson, you will discover how to develop an intuition for skewed class distributions.

Lesson 02: Intuition for Imbalanced Data

In this lesson, you will discover how to develop a practical intuition for imbalanced classification datasets. A challenge for beginners working with imbalanced classification problems is what a specific skewed class distribution means. For example, what is the difference and implication for a 1:10 vs. a 1:100 class ratio? The `make_classification()` scikit-learn function can be used to define a synthetic dataset with a desired class imbalance. The `weights` argument specifies the ratio of examples in the negative class, e.g. [0.99, 0.01] means that 99 percent of the examples will belong to the majority class, and the remaining 1 percent will belong to the minority class.

```
...
# define dataset
X, y = make_classification(n_samples=1000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0)
```

Listing 1: Example of creating a synthetic dataset.

Once defined, we can summarize the class distribution using a `Counter` object to get an idea of exactly how many examples belong to each class.

```
...
# summarize class distribution
counter = Counter(y)
print(counter)
```

Listing 2: Example of summarizing the class distribution.

We can also create a scatter plot of the dataset because there are only two input variables. The dots can then be colored by each class. This plot provides a visual intuition for what exactly a 99 percent vs. 1 percent majority/minority class imbalance looks like in practice. The complete example of creating and summarizing an imbalanced classification dataset is listed below.

```
# plot imbalanced classification problem
from collections import Counter
from sklearn.datasets import make_classification
from matplotlib import pyplot
from numpy import where
# define dataset
X, y = make_classification(n_samples=1000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99, 0.01], flip_y=0)
# summarize class distribution
counter = Counter(y)
print(counter)
```

```
# scatter plot of examples by class label
for label, _ in counter.items():
  row_ix = where(y == label)[0]
  pyplot.scatter(X[row_ix, 0], X[row_ix, 1], label=str(label))
pyplot.legend()
pyplot.show()
```

Listing 3: Example of creating and plotting an imbalanced classification dataset.

Your Task

For this lesson, you must run the example and review the plot. For bonus points, you can test different class ratios and review the results.

Next

In the next lesson, you will discover how to evaluate models for imbalanced classification.

Lesson 03: Evaluate Imbalanced Classification Models

In this lesson, you will discover how to evaluate models on imbalanced classification problems. Prediction accuracy is the most common metric for classification tasks, although it is inappropriate and potentially dangerously misleading when used on imbalanced classification tasks. The reason for this is because if 98 percent of the data belongs to the negative class, you can achieve 98 percent accuracy on average by simply predicting the negative class all the time, achieving a score that naively looks good, but in practice has no skill. Instead, alternate performance metrics must be adopted.

Popular alternatives are the precision and recall scores that allow the performance of the model to be considered by focusing on the minority class, called the positive class. Precision calculates the ratio of the number of correctly predicted positive examples divided by the total number of positive examples that were predicted. Maximizing the precision will minimize the false positives.

$$\text{Precision} = \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalsePositive}} \tag{1}$$

Recall predicts the ratio of the total number of correctly predicted positive examples divided by the total number of positive examples that could have been predicted. Maximizing recall will minimize false negatives.

$$\text{Recall} = \frac{\text{TruePositive}}{\text{TruePositive} + \text{FalseNegative}} \tag{2}$$

The performance of a model can be summarized by a single score that averages both the precision and the recall, called the F-measure. Maximizing the F-measure will maximize both the precision and recall at the same time.

$$\text{F-measure} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{3}$$

The example below fits a logistic regression model on an imbalanced classification problem and calculates the accuracy, which can then be compared to the precision, recall, and F-measure.

```
# evaluate imbalanced classification model with different metrics
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
```

```
from sklearn.metrics import f1_score
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0)
# split into train/test sets with same class ratio
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, stratify=y)
# define model
model = LogisticRegression(solver='liblinear')
# fit model
model.fit(trainX, trainy)
# predict on test set
yhat = model.predict(testX)
# evaluate predictions
print('Accuracy: %.3f' % accuracy_score(testy, yhat))
print('Precision: %.3f' % precision_score(testy, yhat))
print('Recall: %.3f' % recall_score(testy, yhat))
print('F-measure: %.3f' % f1_score(testy, yhat))
```

Listing 4: Example of evaluating a model using different metrics.

Your Task

For this lesson, you must run the example and compare the classification accuracy to the other metrics, such as precision, recall, and F-measure. For bonus points, try other metrics such as Fbeta-measure and ROC AUC scores.

Next

In the next lesson, you will discover how to undersample the majority class.

Lesson 04: Undersampling the Majority Class

In this lesson, you will discover how to undersample the majority class in the training dataset. A simple approach to using standard machine learning algorithms on an imbalanced dataset is to change the training dataset to have a more balanced class distribution. This can be achieved by deleting examples from the majority class, referred to as *undersampling*. A possible downside is that examples from the majority class that are helpful during modeling may be deleted. The imbalanced-learn library provides many examples of undersampling algorithms. This library can be installed easily using pip; for example:

```
pip install imbalanced-learn
```

Listing 5: Example of installing the imbalanced-learn library.

A fast and reliable approach is to randomly delete examples from the majority class to reduce the imbalance to a ratio that is less severe or even so that the classes are even. The example below creates a synthetic imbalanced classification data, then uses `RandomUnderSampler` class to change the class distribution from 1:100 minority to majority classes to the less severe 1:2.

```
# example of undersampling the majority class
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.under_sampling import RandomUnderSampler
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99, 0.01], flip_y=0)
# summarize class distribution
print(Counter(y))
# define undersample strategy
undersample = RandomUnderSampler(sampling_strategy=0.5)
# fit and apply the transform
X_over, y_over = undersample.fit_resample(X, y)
# summarize class distribution
print(Counter(y_over))
```

Listing 6: Example of random undersampling.

Your Task

For this lesson, you must run the example and note the change in the class distribution before and after undersampling the majority class. For bonus points, try other undersampling ratios or even try other undersampling techniques provided by the imbalanced-learn library.

Next

In the next lesson, you will discover how to oversample the minority class.

Lesson 05: Oversampling the Minority Class

In this lesson, you will discover how to oversample the minority class in the training dataset. An alternative to deleting examples from the majority class is to add new examples from the minority class. This can be achieved by simply duplicating examples in the minority class, but these examples do not add any new information. Instead, new examples from the minority can be synthesized using existing examples in the training dataset. These new examples will be *close* to existing examples in the feature space, but different in small but random ways.

The SMOTE algorithm is a popular approach for oversampling the minority class. This technique can be used to reduce the imbalance or to make the class distribution even. The example below demonstrates using the `SMOTE` class provided by the imbalanced-learn library on a synthetic dataset. The initial class distribution is 1:100 and the minority class is oversampled to a 1:2 distribution.

```
# example of oversampling the minority class
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.over_sampling import SMOTE
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99, 0.01], flip_y=0)
# summarize class distribution
print(Counter(y))
# define oversample strategy
oversample = SMOTE(sampling_strategy=0.5)
# fit and apply the transform
X_over, y_over = oversample.fit_resample(X, y)
# summarize class distribution
print(Counter(y_over))
```

Listing 7: Example of SMOTE oversampling.

Your Task

For this lesson, you must run the example and note the change in the class distribution before and after oversampling the minority class. For bonus points, try other oversampling ratios, or even try other oversampling techniques provided by the imbalanced-learn library.

Next

In the next lesson, you will discover how to combine undersampling and oversampling techniques.

Lesson 06: Combine Undersampling and Oversampling

In this lesson, you will discover how to combine data undersampling and oversampling on a training dataset. Data undersampling will delete examples from the majority class, whereas data oversampling will add examples to the majority class. These two approaches can be combined and used on a single training dataset. Given that there are so many different data sampling techniques to choose from, it can be confusing as to which methods to combine. Thankfully, there are common combinations that have been shown to work well in practice; some examples include:

- Random Undersampling with SMOTE oversampling.
- Tomek Links Undersampling with SMOTE oversampling.
- Edited Nearest Neighbors Undersampling with SMOTE oversampling.

These combinations can be applied manually to a given training dataset by first applying one sampling algorithm, then another. Thankfully, the imbalanced-learn library provides implementations of common combined data sampling techniques. The example below demonstrates how to use the SMOTEENN that combines both SMOTE oversampling of the minority class and Edited Nearest Neighbors undersampling of the majority class.

```
# example of both undersampling and oversampling
from collections import Counter
from sklearn.datasets import make_classification
from imblearn.combine import SMOTEENN
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99, 0.01], flip_y=0)
# summarize class distribution
print(Counter(y))
# define sampling strategy
sample = SMOTEENN(sampling_strategy=0.5)
# fit and apply the transform
X_over, y_over = sample.fit_resample(X, y)
# summarize class distribution
print(Counter(y_over))
```

Listing 8: Example of SMOTEENN combined oversampling and undersampling.

Your Task

For this lesson, you must run the example and note the change in the class distribution before and after the data sampling. For bonus points, try other combined data sampling techniques or even try manually applying oversampling followed by undersampling on the dataset.

Next

In the next lesson, you will discover how to use cost-sensitive algorithms for imbalanced classification.

Lesson 07: Cost-Sensitive Algorithms

In this lesson, you will discover how to use cost-sensitive algorithms for imbalanced classification. Most machine learning algorithms assume that all misclassification errors made by a model are equal. This is often not the case for imbalanced classification problems, where missing a positive or minority class case is worse than incorrectly classifying an example from the negative or majority class.

Cost-sensitive learning is a subfield of machine learning that takes the costs of prediction errors (and potentially other costs) into account when training a machine learning model. Many machine learning algorithms can be updated to be cost-sensitive, where the model is penalized for misclassification errors from one class more than the other, such as the minority class. The scikit-learn library provides this capability for a range of algorithms via the `class_weight` attribute specified when defining the model. A weighting can be specified that is inversely proportional to the class distribution.

If the class distribution was 0.99 to 0.01 for the majority and minority classes, then the `class_weight` argument could be defined as a dictionary that defines a penalty of 0.01 for errors made for the majority class and a penalty of 0.99 for errors made with the minority class, e.g. 0:0.01, 1:0.99. This is a useful heuristic and can be configured automatically by setting the `class_weight` argument to the string `'balanced'`. The example below demonstrates how to define and fit a cost-sensitive logistic regression model on an imbalanced classification dataset.

```
# example of cost-sensitive logistic regression for imbalanced classification
from sklearn.datasets import make_classification
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import f1_score
# generate dataset
X, y = make_classification(n_samples=10000, n_features=2, n_redundant=0,
    n_clusters_per_class=1, weights=[0.99], flip_y=0)
# split into train/test sets with same class ratio
trainX, testX, trainy, testy = train_test_split(X, y, test_size=0.5, stratify=y)
# define model
model = LogisticRegression(solver='liblinear', class_weight='balanced')
# fit model
model.fit(trainX, trainy)
# predict on test set
yhat = model.predict(testX)
# evaluate predictions
print('F-measure: %.3f' % f1_score(testy, yhat))
```

Listing 9: Example of cost-sensitive logistic regression.

Your Task

For this lesson, you must run the example and review the performance of the cost-sensitive model. For bonus points, compare the performance to the cost-insensitive version of logistic regression. This was the final lesson in the mini-course.

Final Word Before You Go...

You made it. Well done! Take a moment and look back at how far you have come. You discovered:

- The challenge of imbalanced classification is the lack of examples for the minority class and the difference in importance of classification errors across the classes.
- How to develop a spatial intuition for imbalanced classification datasets that might inform data preparation and algorithm selection.
- The failure of classification accuracy and how alternate metrics like precision, recall, and the F-measure can better summarize model performance on imbalanced datasets.
- How to delete examples from the majority class in the training dataset, referred to as data undersampling.
- How to synthesize new examples in the minority class in the training dataset, referred to as data oversampling.
- How to combine data oversampling and undersampling techniques on the training dataset, and common combinations that result in good performance.
- How to use cost-sensitive modified versions of machine learning algorithms to improve performance on imbalanced classification datasets.

This is just the beginning of your journey with imbalanced classification. Keep practicing and developing your skills. Take the next step and check out my book on *Imbalanced Classification with Python*.

How Did You Go With The Crash-Course?

Did you enjoy this crash-course?
Do you have any questions or sticking points?

Let me know, send me an email at: **jason@MachineLearningMastery.com**

Take the Next Step

Looking for more help with Imbalanced Classification?

Grab my new book:
Imbalanced Classification with Python
https://machinelearningmastery.com/imbalanced-classification-with-python/