**Class Imbalance in Machine Learning Problems: A Practical Guide**

Five lessons from the trenches of applied data science

Class imbalance, where one class is much more abundant than the other, is one of the most ubiquitous topics in data science literature. Searching for ‘class imbalance’ on Medium alone reveals numerous articles with titles such as:

* *“Dealing With Class Imbalanced Datasets For Classification”*
* *“How to Effortlessly Handle Class Imbalance with Python and SMOTE”*
* *“Stop Using SMOTE to Treat Class Imbalance”*
* *“A Loss Function Suitable for Class Imbalanced Data: Focal Loss”*
* *“Class Imbalance: a classification headache”*

This is a headache indeed. Should you rebalance? Use SMOTE? *Not* use SMOTE? How about using focal loss? It feels like you could spend months on experimentation, and there’ll still be things left that you haven’t tried. It’s the data scientist’s equivalent of Sisyphus’ hill.

My goal with this article is to cut through the noise and explain five concepts that I believe to be most important to understand about class imbalance, written for the data scientist in the business trenches with a limited time budget for experimentation. I’ll conclude with a simple set of ‘rules of thumb’ that you can use as a practical guide in your next imbalanced ML problem.

Let’s get started.

**1. Class imbalance is the norm, not the exception**

Class imbalance is normal and expected in typical ML applications. For example:

* in credit card fraud detection, most transactions are legitimate, and only a small fraction are fraudulent.
* in spam detection, it’s the other way around: [most](https://dataprot.net/statistics/spam-statistics/) Emails sent around the globe today are spam.
* in ads conversion prediction, most users will ignore the ad, and only a small fraction will click on it.
* in user churn prediction, most users will stay on the platform, and only a small fraction will ‘churn’ (i.e., leave the platform).

Imbalance is simply part of the reality that we live in. In fact, the opposite is rare: you’ll rarely encounter a classification problem in the real world where all classes occur equally often. Many real-world ML problems, in other words, are about finding a ‘needle in a haystack’.

**2. Class imbalance itself is not the problem**

Despite what is being communicated in some articles, the imbalance itself is not actually the problem. Instead, when working on an imbalanced ML problem, there are 3 things can go wrong:

**Choosing the wrong metric.** Accuracy is a bad metric to quantify the performance of an ML model on an imbalanced problem: if the positivity rate is just 1%, then a naive classifier labeling everything as negative has 99% accuracy by definition. This sounds good on paper, but is bad in practice. This problem can of course be avoided by choosing a more suitable metric such as precision at fixed recall, recall at fixed precision, PR-AUC, or ROC-AUC.

**Training/serving skew.** This refers to the problem when the data used for training the model is not the same as the data used at inference time, for example because the training data has been manually rebalanced. The problem with training/serving skew is that the performance on the training data is not a good proxy for the performance at serving time, and the model may end up being worse than expected. For example, if at training time we down-sampled negatives by a factor of 10X, then, in the worst case, the precision in production may be 10X worse than expected.

**Data scarcity.** In imbalanced problems, it may be hard to gather a sufficiently large number of labeled positive samples to train a ML model with reasonable performance. For example, if you only have 10 to 100 positive samples, the model may easily memorize these samples, leading to an overfit model that generalized poorly. The more imbalanced the problem, the fewer positive samples you may have available for training the model.

Note that in none of these 3 cases the imbalance itself is the issue. For example, it is perfectly ok to train a model on 1M negatives and 10K positives, as long as you avoid using accuracy as a metric.

**3. Upsampling the minority class may not be a good idea**

Upsampling the minority class, i.e. adding copies of negative samples to the training data, is often recommended as one of the first things to try on an imbalanced problem (for example [here](https://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/)). The motivation is to achieve a more balanced positive/negative ratio, and therefore ‘fix’ the class imbalance before training the model. However, there are two reasons why upsampling can actually hurt model performance.

First, upsampling introduces training/serving skew, i.e. the training data is no longer representative of the serving data. When you then pick an [operating point](https://medium.com/towards-data-science/deploying-your-machine-learning-model-is-just-the-beginning-b4851e665b11) on the the training data, that operating point may be sub-optimal in the real world. [Samuele Mazzanti](https://towardsdatascience.com/your-dataset-is-imbalanced-do-nothing-abf6a0049813) demonstrated this effect on synthetic data: after upsampling the minority class in the training data, the log-loss on unseen data increased from 1.28 to 2.3, a notable degradation. His conclusion:

*Your Dataset Is Imbalanced? Do Nothing!*

Second, another potential problem with upsampling is **data leakage**: if you first upsample the data and then split the data into training and validation folds, your model can simply memorize the positives from the training data and achieve artificially strong performance on the validation data, causing you to think that the model is much better than it actually is. If you have to upsample, always do it after splitting the data into training and validation folds, not before.

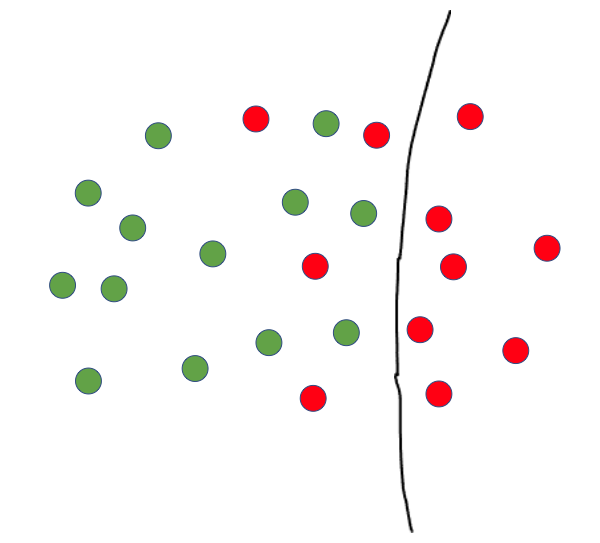
**4. Downsample the majority class with caution**

Downsampling the majority class refers to the practice of randomly deleting a certain fraction of the majority class in the training data. For example, you may decide to keep only 10%, 1%, or a smaller ratio of the original majority class. There are two scenarios when you’ll want to consider doing this:

* when the training data doesn’t fit into memory (and your ML training pipeline requires it to be in memory), or
* when model training takes unreasonably long (days to weeks), causing too long iteration cycles, and preventing you from iterating quickly.

In these cases, it’s reasonable to downsample the majority class. However, it is important to keep in mind that doing so is guaranteed to hurt model performance. Why?

Random downsampling assumes that all samples are equally informative, and therefore we can simply keep a random subset of them. But clearly, this assumption is false: some samples are much more informative than others, namely those that are closer to the decision boundary of the classification problem. When you downsample randomly, you may drop some of the most informative samples from the data, leading to worse model performance than if you hadn’t dropped anything.



Cartoon of a classification problem with 2 classes, red and green. Points closer to the decision boundary (black line) are more informative than points further away from it. (Image by author)

Instead of random downsampling, a better idea may therefore be a **domain filter**: a simple heuristic rule that cuts down most of the majority class, while keeping nearly all of the minority class. For example, if a rule can retain 99% of positives but only 1% of the negatives, this would make a great domain filter. Then, apply that rule both at training time and at inference time prior to your ML model. Here are some examples of good population filters:

* in credit card fraud prediction, filter for new credit cards, i.e. those without a purchase history.
* in spam detection, filter for Emails from addresses that haven’t been seen before.
* in e-commerce product classification, filter for products that contain a certain keyword, or combination of keywords.
* in ads conversion prediction, filter for a certain demographic segment of the user population.

Coming up with a good domain filter requires domain knowledge, so a good tip is to source ideas from the program stakeholders, who know the problem domain best, and then validate these ideas on the data.

**5. Hyperparameters should be the last thing to experiment with**

Certain ML model APIs expose hyperparameters that are claimed to make the model better on imbalanced data. For example, both XGBoost and LightGBM have a parameter called scale\_pos\_weight, which up-weighs positive samples when computing the gradient at each boosting iteration.

However, in practice the impact of tuning such hyperparameters may be not that high. For example, [Jason Brownlee](https://machinelearningmastery.com/xgboost-for-imbalanced-classification/) shows an improvement in ROC-AUC from 0.9572 to 0.9599, a boost of merely 0.0027, by tuning the scale\_pos\_weight parameter with a grid search.

My recommendation is therefore that hyperparameters, in particular those specific to class imbalance, should be the last thing to experiment with. Instead, if the model performance is acceptable, deploy the model into production as soon as possible, so that you can confirm that the modeling pipeline is actually working, and that the [performance is as expected](https://medium.com/towards-data-science/is-my-model-really-better-560e729f81d2). If the model performance is not acceptable, instead of tuning hyperparameters, a better use of your time may be to invest in data quality: collect more training data, construct better features, and make sure the labels are correct.

**Conclusion: ‘rules of thumb’ for imbalanced problems**

Let me conclude with a few simple ‘rules of thumb’ for imbalanced classification problems:

* if you have at least 1K-10K positives, the training set fits in memory, and the model can be trained in reasonable time (hours), then you’re good. This is a good dataset for training. Just make sure to not use accuracy as a performance metric.
* if you have at least 1K-10K positives, but the training set does not fit into memory, or the model takes too long to train (days-weeks), consider downsampling negatives in order to be able to iterate faster. Even better, try to design a good population filter instead, so that you can gain iteration speed without sacrificing model performance.
* if you have <1K positives, your training set may be too small, especially so if the model has a large number of degrees of freedom (such as neural nets or tree ensembles). The model may overfit to such a small number of positives. Try to collect more training data, in particular positives.

Lastly, always remember the principle of [leverage](https://medium.com/towards-data-science/the-most-effective-creatives-maximize-leverage-not-hours-worked-20ed0070fdd7): as a data scientist, there are always an infinite number of ‘shiny new things’ that you can try, but you should be constantly asking yourself which things are expected to yield the biggest return on your time investment. There are numerous techniques for class imbalance that I did not cover here, but in practice oftentimes the highest-leverage thing you can do is stick to the simplest methods and deploy your model as soon as possible.