Lecture 3. Training Data

Note: *This note is a work-in-progress, created for the course CS 329S: Machine Learning Systems Design (Stanford, 2022). For the fully developed text, see the book Designing Machine Learning Systems (Chip Huyen, O'Reilly 2022).*

Errata, questions, and feedback -- please send to chip@huyenchip.com. Thank you!

Table of contents

In chapter 2, we covered how to handle data from the systems perspective. In this chapter, we'll go over how to handle data from the data science perspective. Despite the importance of training data in developing and improving ML models, ML curricula are heavily skewed towards modeling, which is considered by many researchers and engineers as the "fun" part of the process. Building a state-of-the-art model is interesting. Spending days wrangling with a massive amount of malformatted data that doesn't even fit into your machine's memory is frustrating.

Data is messy, complex, unpredictable, and potentially treacherous. If in school, training data is a cute little puppy then, in production, it's a Kraken that, if not tamed, can easily sink your entire ML operation. But this is precisely the reason why ML engineers should learn how to handle data well, saving us time and headache down the road.

In this chapter, we will go over techniques to obtain or create good training data. Training data, in this chapter, encompasses all the data used in the developing phase of ML models, including the different splits used for training, validation, and testing (the train, validation, test splits). This chapter starts with different sampling techniques to select data for training. We'll then address common challenges in creating training data including the label multiplicity problem, the lack of labels problem, the class imbalance problem, and techniques in data augmentation to address the lack of data problem.

We use the term "training data" instead of "training dataset", because "dataset" denotes a set that is finite and stationary. Data in production is neither finite nor stationary, a phenomenon that we will cover in Chapter 7. Like other steps in building ML systems, creating training data is an iterative process. As your model evolves through a project lifecycle, your training data will likely also evolve.

Before we move forward, I just want to echo a word of caution that has been said many times yet is still not enough. Data is full of potential biases. These biases have many possible causes. There are biases caused during collecting, sampling, or labeling. Historical data might be embedded with human biases and ML models, trained on this data, can perpetuate them. Use data but don't trust it too much!

Sampling

Sampling is an integral part of the ML workflow that is, unfortunately, often overlooked in typical ML coursework. Sampling happens in many steps of an ML project lifecycle, such as sampling from all possible real-world data to create training data, sampling from a given dataset to create splits for training, validation, and testing, or sampling from all possible events that happen within your ML system for monitoring purposes. In this section, we'll focus on sampling methods for creating training data, but these sampling methods can also be used for other steps in an ML project lifecycle.

In many cases, sampling is necessary. One example is when you don't have access to all possible data in the real world, the data that you use to train a model are subsets of real-world data, created by one sampling method or another. Another example is when it's infeasible to process all the data that you have access to — because it requires either too much time or too much compute power or too much money — you have to sample that data to create a subset that you can process. In many other cases, sampling is helpful as it allows you to accomplish a task faster and cheaper. For example, when considering a new model, you might want to do a quick experiment with a small subset of your data to see if it's promising first before running this new model on all the data you have[1].

Understanding different sampling methods and how they are being used in our workflow can, first, help us avoid potential sampling biases, and second, help us choose the methods that improve the efficiency of the data we sample.

There are two families of sampling: non-probability sampling and random sampling. We will start with non-probability sampling methods, followed by several common random methods. We'll analyze the pros and cons of each method.

Non-Probability Sampling

Non-probability sampling is when the selection of data isn't based on any probability criteria. Here are some of the criteria for non-probability sampling.

- **Convenience sampling**: samples of data are selected based on their availability. This sampling method is popular because, well, it's convenient.
- **Snowball sampling**: future samples are selected based on existing samples. For example, to scrape legitimate Twitter accounts without having access to Twitter databases, you start with a small number of accounts then you scrape all the accounts in their following, and so on.
- **Judgment sampling**: experts decide what samples to include.
- **Quota sampling**: you select samples based on quotas for certain slices of data without any randomization. For example, when doing a survey, you might want 100 responses from each of the age groups: under 30 years old, between 30 and 60 years old, above 50 years old, regardless of the actual age distribution in the real world.

The samples selected by non-probability criteria are not representative of the real-world data, and therefore, are riddled with selection biases[2]. Because of these biases, you might think that it's a bad idea to select data to train ML models using this family of sampling methods. You're right. Unfortunately, in many cases, the selection of data for ML models is still driven by convenience.

[1] Some readers might argue that this approach might not work with large models, as certain large models don't work for small datasets but work well with a lot more data. In this case, it's still important to experiment with datasets of different sizes to figure out the effect of the dataset size on your model.

[2] Heckman, James J. "Sample Selection Bias as a Specification Error." Econometrica, vol. 47, no. 1, [Wiley, Econometric Society], 1979, pp. 153–61, https://doi.org/10.2307/1912352.

One example of these cases is language modeling. Language models are often trained not with data that is representative of all possible texts but with data that can be easily collected — Wikipedia, CommonCrawl, Reddit.

Another example is data for sentiment analysis of general text. Much of this data is collected from sources with natural labels (ratings) such as IMDB reviews and Amazon reviews. These datasets are then used for other sentiment analysis tasks. IMDB reviews and Amazon reviews are biased towards users who are willing to leave reviews online, and not necessarily representative of people who don't have access to the Internet and aren't willing to put reviews online.

The third example is data for training self-driving cars. Initially, data collected for self-driving cars came largely from two areas: Phoenix in Arizona (because of its lax regulations) and the Bay Area in California (because many companies that build self-driving cars are located here). Both areas have generally sunny weather. In 2016, Waymo expanded its operations to Kirkland, WA specially for Kirkland's rainy weather, but there's still a lot more self-driving car data for sunny weather than for rainy or snowy weather.

Non-probability sampling can be a quick and easy way to gather your initial data to get your project off the ground. However, for reliable models, you might want to use probability-based sampling, which we will cover next.

Simple Random Sampling

In the simplest form of random sampling, you give all samples in the population equal probabilities of being selected. For example, you randomly select 10% of all samples, giving all samples an equal 10% chance of being selected.

The advantage of this method is that it's easy to implement. The drawback is that rare categories of data might not appear in your selection. Consider the case where a class appears only in 0.01% of your data population. If you randomly select 1% of your data, samples of this rare class will unlikely be selected. Models trained on this selection might think that this rare class doesn't exist.

Stratified Sampling

To avoid the drawback of simple random sampling listed above, you can first divide your population into the groups that you care about and sample from each group separately. For example, to sample 1% of data that has two classes A and B, you can sample 1% of class A and 1% of class B. This way, no matter how rare class A or B is, you'll ensure that samples from it will be included in the selection. Each group is called a strata, and this method is called stratified sampling.

One drawback of this sampling method is that it isn't always possible, such as when it's impossible to divide all samples into groups. This is especially challenging when one sample might belong to multiple groups as in the case of multilabel tasks[3]. For instance, a sample can be both class A and class B.

Weighted Sampling

In weighted sampling, each sample is given a weight, which determines the probability of it being selected. For example, if you have three samples A, B, C and want them to be selected with the probabilities of 50%, 30%, 20% respectively, you can give them the weights 0.5, 0.3, 0.2.

This method allows you to leverage domain expertise. For example, if you know that a certain subpopulation of data, such as more recent data, is more valuable to your model and want it to have a higher chance of being selected, you can give it a higher weight.

This also helps with the case when the data you have comes from a different distribution compared to the true data. For example, if in your data, red samples account for 25% and blue samples account for 75%, but you know that in the real world, red and blue have equal probability to happen, you can give red samples the weights three times higher than blue samples.

In Python, you can do weighted sampling with `random.choices` as follows:

```
# Choose two items from the list such that 1, 2, 3, 4 each has
# 20% chance of being selected, while 100 and 1000 each have only 10% chance.
random.choices(population=[1, 2, 3, 4, 100, 1000],
                  weights=[0.2, 0.2, 0.2, 0.2, 0.1, 0.1],
               k=2)
# This is equivalent to the following
random.choices(population=[1, 1, 2, 2, 3, 3, 4, 4, 100, 1000],
               k=2)
```

A common concept in ML that is closely related to weighted sampling is sample weights. Weighted sampling is used to select samples to train your model with, whereas sample weights are used to assign "weights" or "importance" to training samples. Samples with higher weights affect the loss function more. Changing sample weights can change your model's decision boundaries significantly, as shown in Figure 3-1.

[3] Multilabel tasks are tasks where one example can have multiple labels.

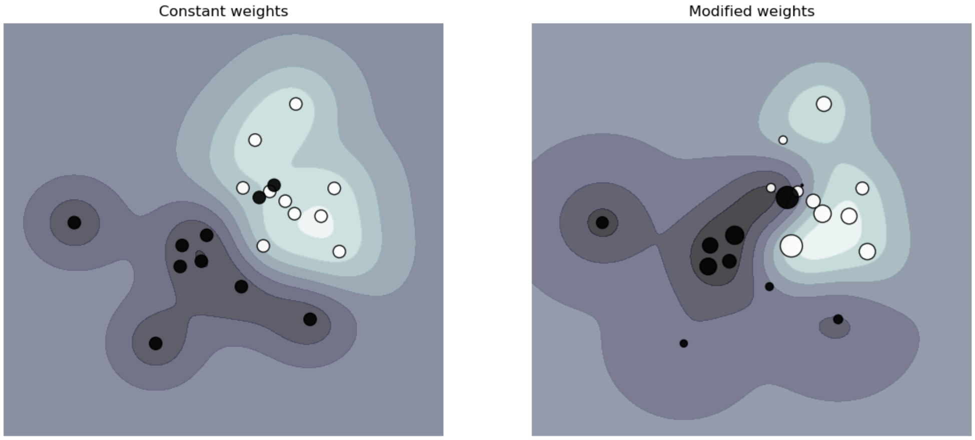


Figure 3-1: How sample weights can affect the decision boundary. On the left is when all samples are given equal weights. On the right is when samples are given different weights. Source: SVM: Weighted samples (sklearn).

Importance Sampling

Importance sampling is one of the most important sampling methods, not just in ML. It allows us to sample from a distribution when we only have access to another distribution.

Imagine you have to sample x from a distribution P(x), but P(x) is really expensive, slow, or infeasible to sample from. However, you have a distribution Q(x) that is a lot easier to sample from. So you sample x from Q(x) instead and weigh this sample by $\frac{P(x)}{Q(x)}$. Q(x) is called the proposal distribution or the importance distribution. Q(x) can be any distribution as long as $Q(x) > 0$ whenever $P(x) \neq 0$. The equation below shows that in expectation, x sampled from P(X) is equal to x sampled from Q(x) weighted by $\frac{P(x)}{Q(x)}$.

$$E_{P(x)}[x] = \sum_{x} P(x)\, x = \sum_{x} Q(x)\, x \frac{P(x)}{Q(x)} = E_{Q(x)}[x \frac{P(x)}{Q(x)}]$$

One example where importance sampling is used in ML is policy-based reinforcement learning. Consider the case when you want to update your policy. You want to estimate the value functions of the new policy, but calculating the total rewards of taking an action can be costly because it requires considering all possible outcomes until the end of the time horizon after that action. However, if the new policy is relatively close to the old policy, you can calculate the total rewards based on the old policy instead and reweight them according to the new policy. The rewards from the old policy make up the proposal distribution.

Reservoir Sampling

Reservoir sampling is a fascinating algorithm that is especially useful when you have to deal with continually incoming data, which is usually what you have in production.

Imagine you have an incoming stream of tweets and you want to sample a certain number, k, of tweets to do analysis or train a model on. You don't know how many tweets there are but you know you can't fit them all in memory, which means you don't know the probability at which a tweet should be selected. You want to ensure that:

1. Every tweet has an equal probability of being selected and,
2. You can stop the algorithm at any time and the tweets are sampled with the correct probability.

One solution for this problem is reservoir sampling. The algorithm involves a reservoir, which can be an array, and consists of three steps:

1. Put the first k elements into the reservoir.
2. For each incoming nth element, generate a random number i such that $1 \leq i \leq n$.
3. If $1 \leq i \leq k$: replace the ith element in the reservoir with the nth element. Else, do nothing.

This means that each incoming nth element has $\frac{k}{n}$ probability of being in the reservoir. You can also prove that each element in the reservoir has $\frac{k}{n}$ probability of being there. This means that all samples have an equal chance of being selected. If we stop the algorithm at any time, all samples in the reservoir have been sampled with the correct probability. Figure 3-2 shows an illustrative example of how reservoir sampling works.

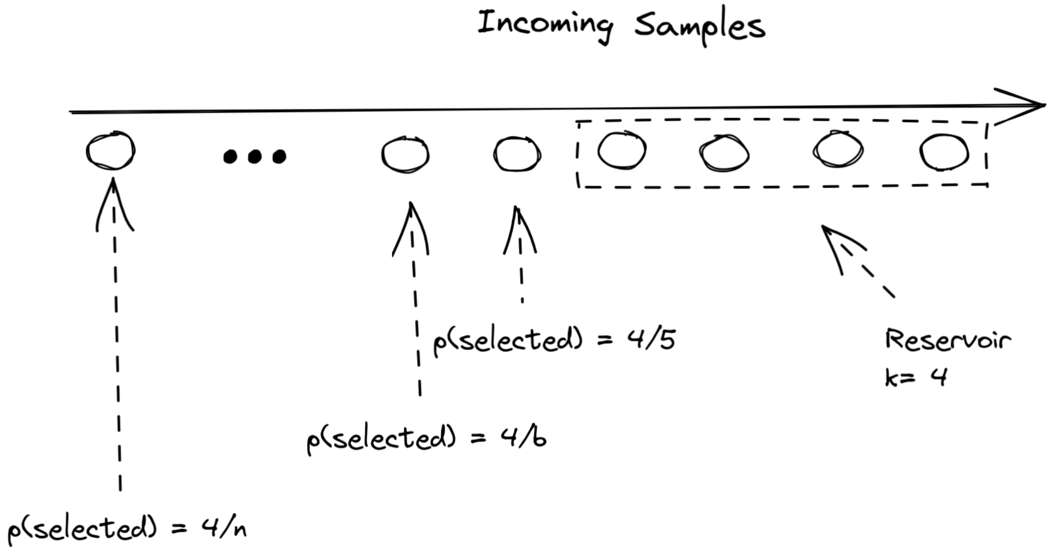


Figure 3-2: A visualization of how reservoir sampling works.

Labeling

Despite the promise of unsupervised ML, most ML models in production today are supervised, which means that they need labels to learn. The performance of an ML model still depends heavily on the quality and quantity of the labeled data it's trained on.

There are tasks where data has natural labels or it's possible to collect natural labels on the fly. For example, for predicting the click-through rate on an ad, labels are whether users click on an ad or not. Similarly, for recommendation systems, labels are whether users click on a recommended item or not. However, for most tasks, natural labels are not available or not accessible, and you will need to obtain labels by other means.

In a talk to my students, Andrej Karpathy, Director of AI at Tesla, shared an anecdote about when he decided to have an in-house labeling team, his recruiter asked how long he'd need this team for. He responded: "How long do we need an engineering team for?" Data labeling has gone from being an auxiliary task to being a core function of many ML teams in production.

In this section, we will discuss the challenges of obtaining labels for your data including the label multiplicity problem and what to do when you lack hand labeled data.

Hand Labels

Anyone who has ever had to work with data in production has probably felt this at a visceral level: acquiring hand labels for your data is difficult for many, many reasons. First, **hand-labeling data can be expensive, especially if subject matter expertise is required**. To classify whether a comment is spam, you might be able to find 200 annotators on a crowdsourcing platform and train them in 15 minutes to label your data. However, if you want to label chest X-rays, you'd need to find board-certified radiologists, whose time is limited and expensive.

Second, **hand labeling poses a threat to data privacy**. Hand labeling means that someone has to look at your data, which isn't always possible if your data has strict privacy requirements. For example, you can't just ship your patient's medical records or your company's confidential financial information to a third-party service for labeling. In many cases, your data might not even be allowed to leave your organization, and you might have to hire or contract annotators to label your data on-premise.

Third, **hand labeling is slow**. For example, accurate transcription of speech utterance at phonetic level can take 400 times longer than the utterance duration. So if you want to annotate 1 hour of speech, it'll take 400 hours or almost 3 working months to do so. In a study to use ML to help classify lung cancers from X-rays, my colleagues had to wait almost a year to obtain sufficient labels.

Slow labeling leads to **slow iteration speed** and makes your model less adaptive to changing environments and requirements. If the task changes or data changes, you'll have to wait for your data to be relabeled before updating your model. Imagine the scenario when you have a sentiment analysis model to analyze the sentiment of every tweet that mentions your brand. It has only two classes: **NEGATIVE** and **POSITIVE**. However, after deployment, your PR team realizes that the most damage comes from angry tweets and they want to attend to angry messages faster. So you have to update your sentiment analysis model to have three classes: **NEGATIVE**, **POSITIVE**, and **ANGRY**. To do so, you will need to look at your data again to see which existing training examples should be relabeled **ANGRY**. If you don't have enough **ANGRY** examples, you will have to collect more data. The longer the process takes, the more your existing model performance will degrade.

Label Multiplicity

Often, to obtain enough labeled data, companies have to use data from multiple sources and rely on multiple annotators who have different levels of expertise. These different data sources and annotators also have different levels of accuracy. This leads to the problem of label ambiguity or label multiplicity: what to do when there are multiple possible labels for a data instance.

Consider this simple task of entity recognition. You give three annotators the following sample and ask them to annotate all entities they can find.

Darth Sidious, known simply as the Emperor, was a Dark Lord of the Sith who reigned over the galaxy as Galactic Emperor of the First Galactic Empire.

You receive back three different solutions, as shown in Table 3-1. Three annotators have identified different entities. Which one should your model train on? A model trained on data labeled mostly by annotator 1 will perform very differently from a model trained on data labeled mostly by annotator 2.

Annotator	# entities	Annotation
1	3	[**Darth Sidious**], known simply as the Emperor, was a [**Dark Lord of the Sith**] who reigned over the galaxy as [**Galactic Emperor of the First Galactic Empire**]
2	6	[**Darth Sidious**], known simply as the [**Emperor]**, was a **[Dark Lord**] of the [**Sith**] who reigned over the galaxy as [**Galactic Emperor**] of the [**First Galactic Empire**].
3	4	[**Darth Sidious**], known simply as the [**Emperor]**, was a **[Dark Lord of the Sith**] who reigned over the galaxy as [**Galactic Emperor of the First Galactic Empire**].

Table 3-1: Identities identified by different annotators might be very different.

Disagreements among annotators are extremely common. The higher level of domain expertise required, the higher the potential for annotating disagreement[4]. If one human-expert thinks the label should be A while another believes it should be B, how do we resolve this conflict to obtain one single ground truth? If human experts can't agree on a label, what does human-level performance even mean?

To minimize the disagreement among annotators, it's important to, first, have a clear problem definition. For example, in the entity recognition task above, some disagreements could have been eliminated if we clarify that in case of multiple possible entities, pick the entity that comprises the longest substring. This means **Galactic Emperor of the First Galactic Empire** instead of **Galactic Emperor** and **First Galactic Empire**. Second, you need to incorporate that definition into training to make sure that all annotators understand the rules.

Data Lineage

Indiscriminately using data from multiple sources, generated with different annotators, without examining their quality can cause your model to fail mysteriously. Consider a case when you've trained a moderately good model with 100K data samples. Your ML engineers are confident that more data will improve the model performance, so you spend a lot of money to hire annotators to label another million data samples.

However, the model performance actually decreases after being trained on the new data. The reason is that the new million samples were crowdsourced to annotators who labeled data with much less accuracy than the original data. It can be especially difficult to remedy this if you've already mixed your data and can't differentiate new data from old data.

On top of that, it's good practice to keep track of the origin of each of our data samples as well as its labels, a technique known as **data lineage**. Data lineage helps us both flag potential biases in our data as well as debug our models. For example, if our model fails mostly on the recently acquired data samples, you might want to look into how the new data was acquired. On more than one occasion, we've discovered that the problem wasn't with our model, but because of the unusually high number of wrong labels in the data that we'd acquired recently.

Handling the Lack of Hand Labels

Because of the challenges in acquiring sufficient high-quality labels, many techniques have been developed to address the problems that result. In this section, we will cover four of them: weak supervision, semi-supervision, transfer learning, and active learning.

[4] If something is so obvious to label, you wouldn't need domain expertise.

Method	How	Ground truths required?
Weak supervision	Leverages (often noisy) heuristics to generate labels	No, but a small number of labels are recommended to guide the development of heuristics
Semi-supervision	Leverages structural assumptions to generate labels	Yes. A small number of initial labels as seeds to generate more labels
Transfer learning	Leverages models pretrained on another task for your new task	No for zero-shot learning Yes for fine-tuning, though the number of ground truths required is often much smaller than what would be needed if you train the model from scratch.
Active learning	Labels data samples that are most useful to your model	Yes

Table 3-2: Summaries for four techniques for handling the lack of hand labeled data.

Weak supervision

If hand labeling is so problematic, what if we don't use hand labels altogether? One approach that has gained popularity is weak supervision. One of the most popular open-source tools for weak supervision is Snorkel, developed at the Stanford AI Lab[5]. The insight behind weak supervision is that people rely on heuristics, which can be developed with subject matter expertise, to label data. For example, a doctor might use the following heuristics to decide whether a patient's case should be prioritized as emergent.

If the nurse's note mentions a serious condition like pneumonia, the patient's case should be given priority consideration.

Libraries like Snorkel are built around the concept of a **labeling function (LF)**: a function that encodes heuristics. The above heuristics can be expressed by the following function.

```
def labeling_function(note):
   if "pneumonia" in note:
       return "EMERGENT"
```

LFs can encode many different types of heuristics. Here are some of the heuristics.

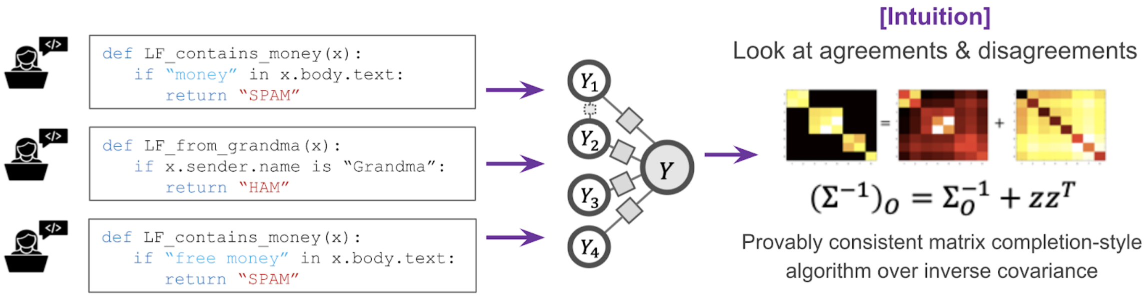
- Keyword heuristic, such as the example above.

[5] Snorkel: Rapid Training Data Creation with Weak Supervision (Ratner et al., 2017, Proceedings of the VLDB Endowment, Vol. 11, No. 3)

- Regular expressions, such as if the note matches or fails to match a certain regular expression.
- Database lookup, such as if the note contains the disease listed in the dangerous disease list.
- The outputs of other models, such as if an existing system classifies this as EMERGENT.

After you've written LFs, you can apply them to the samples you want to label.

Because LFs encode heuristics, and heuristics are noisy, LFs are noisy. Multiple labeling functions might apply to the same data examples, and they might give conflicting labels. One function might think a note is EMERGENT but another function might think it's not. One heuristic might be much more accurate than another heuristic, which you might not know because you don't have ground truth labels to compare them to. It's important to combine, denoise, and reweight all LFs to get a set of most likely-to-be-correct labels.



```
def LF_contains_money(x):
    if "money" in x.body.text:
        return "SPAM"
```

```
def LF_from_grandma(x):
    if x.sender.name is "Grandma":
        return "HAM"
```

```
def LF_contains_money(x):
    if "free money" in x.body.text:
        return "SPAM"
```

Y_1
Y_2
Y_3
Y_4
Y
[Intuition]
Look at agreements & disagreements
=
+
$(\Sigma^{-1})_O = \Sigma_O^{-1} + zz^T$
Provably consistent matrix completion-style algorithm over inverse covariance

Figure 3-3: A high level overview of how labeling functions are combined.

Image by Ratner et al.

In theory, you don't need any hand labels for weak supervision. However, to get a sense of how accurate your LFs are, a small amount of hand labels is recommended. These hand labels can help you discover patterns in your data to write better LFs.

Weak supervision can be especially useful when your data has strict privacy requirements. You only need to see a small, cleared subset of data to write LFs, which can be applied to the rest of your data without anyone looking at it.

With LFs, subject matter expertise can be versioned, reused, and shared. Expertise owned by one team can be encoded and used by another team. If your data changes or your requirements change, you can just reapply LFs to your data samples. The approach of using labeling functions to generate labels for your data is also known as programmatic labeling. Table 3-3 shows some of the advantages of programmatic labeling over hand labeling.

Hand labeling	**Programmatic labeling**
Expensive: Especially when subject matter expertise required	**Cost saving**: Expertise can be versioned, shared, and reused across an organization
Non-private: Need to ship data to human annotators	**Privacy**: Create LFs using a cleared data subsample then apply LFs to other data without looking at individual samples.
Slow: Time required scales linearly with # labels needed	**Fast**: Easily scale from 1K to 1M samples
Non-adaptive: Every change requires re-labeling the data	**Adaptive**: When changes happen, just reapply LFs!

Table 3-3: The advantages of programmatic labeling over hand labeling.

Here is a case study to show how well weak supervision works in practice. In a study with Stanford Medicine[6], models trained with weakly-supervised labels obtained by a single radiologist after 8 hours of writing labeling functions had comparable performance with models trained on data obtained through almost a year of hand labeling. There are two interesting facts about the results of the experiment. First, the models continued improving with more unlabeled data even without more labeling functions. Second, labeling functions were being reused across tasks. The researchers were able to reuse 6 labeling functions between the CXR (Chest X-Rays) task and EXR (Extremity X-Rays) task.

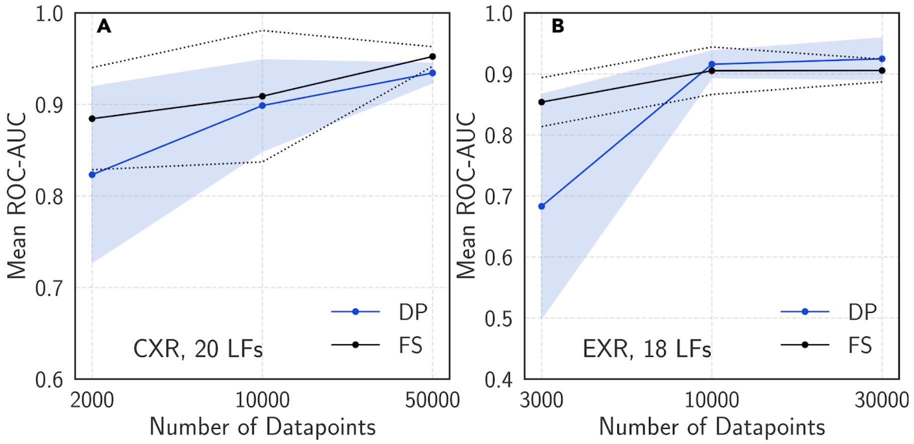


Figure 3-4: Comparison of the performance of a model trained on fully supervised labels (FS) and a model trained with programmatic labels (DP) on CXR and EXR tasks.

Image by Dunnmon et al.

[6] Cross-Modal Data Programming Enables Rapid Medical Machine Learning (Dunnmon et al., 2020)

My students often ask that if heuristics work so well to label data, why do we need machine learning models? One reason is that your labeling functions might not cover all your data samples, so you need to train ML models to generalize to samples that aren't covered by any labeling function.

Weak supervision is a simple but powerful paradigm. However, it's not perfect. In some cases, the labels obtained by weak supervision might be too noisy to be useful. But it's often a good method to get you started when you want to explore the effectiveness of ML without wanting to invest too much in hand labeling upfront.

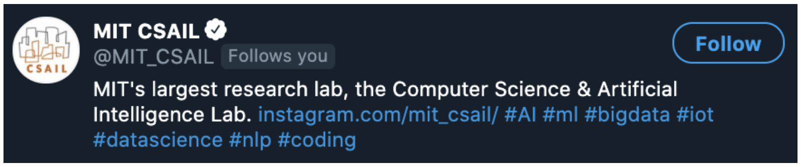
Semi-supervision

If weak supervision leverages heuristics to obtain noisy labels, semi-supervision leverages structural assumptions to generate new labels based on a small set of initial labels. Unlike weak supervision, semi-supervision requires an initial set of labels.

Semi-supervised learning is a technique that was used back in the 90s[7], and since then, many semi-supervision methods have been developed. A comprehensive review of semi-supervised learning is out of the scope of this book. We'll go over a small subset of these methods to give readers a sense of how they are used. For a comprehensive review, I recommend *Semi-Supervised Learning Literature Survey* (Xiaojin Zhu, 2008) and *A survey on semi-supervised learning* (Engelen and Hoos, 2018).

A classic semi-supervision method is **self-training**. You start by training a model on your existing set of labeled data, and use this model to make predictions for unlabeled samples. Assuming that predictions with high raw probability scores are correct, you add the labels predicted with high probability to your training set, and train a new model on this expanded training set. This goes on until you're happy with your model performance.

Another semi-supervision method assumes that data samples that share similar characteristics share the same labels. The similarity might be obvious, such as in the task of classifying the topic of Twitter hashtags as follows. You can start by labeling the hashtag "#AI" as **Computer Science**. Assuming that hashtags that appear in the same tweet or profile are likely about the same topic, given the profile of MIT CSAIL below, you can also label the hashtags "#ML" and "#BigData" as **Computer Science**.



MIT CSAIL

@MIT_CSAIL

MIT's largest research lab, the Computer Science & Artificial Intelligence Lab. instagram.com/mit_csail/ #AI #ml #bigdata #iot #datascience #nlp #coding

Figure 3-5: Because #ML and #BigData appears in the same Twitter profile as #AI,

[7] Combining Labeled and Unlabeled Data with Co-Training (Blum and Mitchell, 1998)

we can assume that they belong to the same topic.

In most cases, the similarity can only be discovered by more complex methods. For example, you might need to use a clustering method or a K-nearest neighbor method to discover samples that belong to the same cluster.

A semi-supervision method that has gained popularity in recent years is the perturbation-based method. It's based on the assumption that small perturbations to a sample shouldn't change its label. So you apply small perturbations to your training samples to obtain new training samples. The perturbations might be applied directly to the samples (e.g. adding white noise to images) or to their representations (e.g. adding small values to embeddings of words). The perturbed samples have the same labels as the unperturbed samples. We'll discuss more about this in the section Perturbation later in this chapter under Augmentation.

In some cases, semi-supervision approaches have reached the performance of purely supervised learning, even when a substantial portion of the labels in a given dataset has been discarded[8]. Semi-supervision is the most useful when the number of training labels is limited. One thing to consider when doing semi-supervision is how much of this limited amount should be used for evaluation. If you evaluate multiple model candidates on the same test set and choose the one that performs best on the test set, you might have chosen a model that overfits the most on the test set. On the other hand, if you choose models based on a validation set, the value gained by having a validation set might be less than the value gained by adding the validation set to the limited training set.

Transfer learning

Transfer learning refers to the family of methods where a model developed for a task is reused as the starting point for a model on a second task. First, the base model is trained for a base task such as language modeling. The base task is usually a task that has cheap and abundant training data. Language modeling is a great candidate because it doesn't require labeled data. You can collect any body of text — books, Wikipedia articles, chat histories — and the task is: given a sequence of tokens[9], predict the next token. When given a sequence "I bought NVIDIA shares because I believe in the importance of", a language model might output "hardware" as the next token.

You then use this pretrained base model on the task that you're interested in, such as sentiment analysis, intent detection, question answering, etc. This task is called a downstream task. In some cases, such as in zero-shot learning scenarios, you might be able to use the base model on a downstream task directly. In many cases, you might need to *fine-tune* the base model. Fine-tuning means making small changes to the base model, which can be continuing training the entire base model or a subset of the base model on data from a given downstream task[10].

[8] Realistic Evaluation of Deep Semi-Supervised Learning Algorithms (Oliver et al., NeurIPS 2018)
[9] A token can be a word, a character, or part of a word.
[10] Universal Language Model Fine-tuning for Text Classification (Howard and Ruder, 2018)

Sometimes, you might need to modify the inputs using a template that can prompt the base model to generate the outputs that you want[11]. For example, to use a language model as the base model for a question answering task, you might want to use the following prompt.

Q: When was the United States founded?

A: July 4, 1776.

Q: Who wrote the Declaration of Independence?

A: Thomas Jefferson.

Q: What year was Alexander Hamilton born?

A:

When you input this prompt into a language model such as GPT-3, it might output the year Alexander Hamilton was born.

Transfer learning is especially appealing for tasks that don't have a lot of labeled data. Even for tasks that have a lot of labeled data, using a pretrained model as the starting point can often boost the performance significantly compared to training from scratch.

Transfer learning has gained a lot of interest in recent years for the right reasons. It has enabled many applications that were previously impossible due to the lack of training samples. A non-trivial portion of ML models in production today are the results of transfer learning, including object detection models that leverage models pretrained on ImageNet and text classification models that leverage pretrained language models such as BERT[12] or GPT-3[13]. It also lowers the entry barriers into ML, as it helps reduce the upfront cost needed for labeling data to build ML applications.

A trend that has emerged in the last five years is that usually, the larger the pretrained base model, the better its performance on downstream tasks. Large models are expensive to train. Based on the configuration of GPT-3, it's estimated that the cost of training this model is in the tens of millions USD. Many have hypothesized that in the future, only a handful of companies can afford to train large pretrained models. The rest of the industry will use these pretrained models directly or finetune them for their specific needs.

Active learning

Active learning is a method for improving the efficiency of data labels. The hope here is that ML models can achieve greater accuracy with fewer training labels if they can choose which data

[11] [2107.13586] Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing

[12] BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (Devlin et al., 2018)

[13] Language Models are Few-Shot Learners (OpenAI 2020)

samples to learn from. Active learning is sometimes called query learning — though this term is getting increasingly unpopular — because a model (active learner) sends back queries in the form of unlabeled samples to be labeled by annotators (usually humans).

Instead of randomly labeling data samples, you label the samples that are most helpful to your models according to some heuristics. The most straightforward heuristic is uncertainty measurement — label the examples that your model is the least certain about hoping that they will help your model learn the decision boundary better. For example, in the case of classification problems where your model outputs raw probabilities for different classes, it might choose the data examples with the lowest probabilities for the predicted class. Figure 3-6 illustrates how well this method works on a toy example.

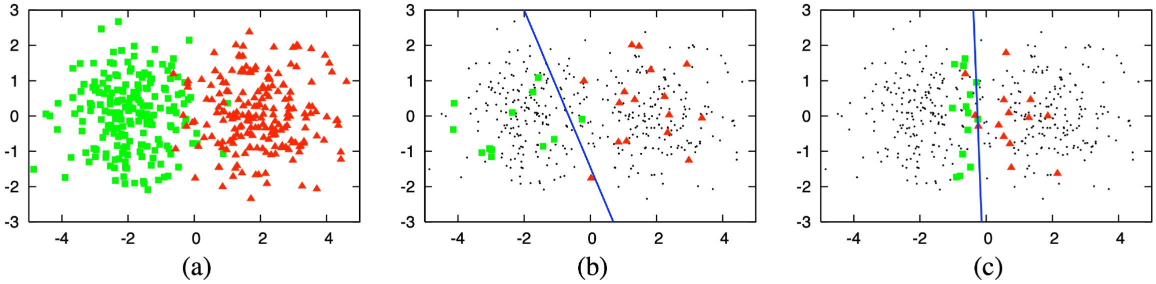


Figure 3-6: How uncertainty-based active learning works. (a) A toy data set of 400 instances, evenly sampled from two class Gaussians. (b) A model trained on 30 examples randomly labeled gives an accuracy of 70%. (c) A model trained on 30 examples chosen by active learning gives an accuracy of 90%. Image by Burr Settles.

Another common heuristic is based on disagreement among multiple candidate models. This method is called query-by-committee. You need a committee of several candidate models, which are usually the same model trained with different sets of hyperparameters. Each model can make one vote for which examples to label next, which it might vote based on how uncertain it is about the prediction. You then label the examples that the committee disagrees on the most.

There are other heuristics such as choosing examples that, if trained on them, will give the highest gradient updates, or will reduce the loss the most. For a comprehensive review of active learning methods, check out *Active Learning Literature Survey* (Burr Settles, 2010).

The examples to be labeled can come from different data regimes. They can be synthesized where your model generates examples in the region of the input space that it's most uncertain about[14]. They can come from a stationary distribution where you've already collected a lot of unlabeled data and your model chooses examples from this pool to label. They can come from the real-world distribution where you have a stream of data coming in, as in production, and your model chooses examples from this stream of data to label.

[14] Queries and Concept Learning (Dana Angluin, 1988)

I'm the most excited about active learning when a system works with real-time data. Data changes all the time, a phenomenon we briefly touched on in Chapter 1 and will go more in detail in Chapter 7. Active learning in this data regime will allow your model to learn more effectively in real-time and adapt faster to changing environments.

Class Imbalance

Class imbalance typically refers to a problem in classification tasks where there is a substantial difference in the number of samples in each class of the training data. For example, in a training dataset for the task of detecting lung cancer from X-Ray images, 99.99% of the X-Rays might be of normal lungs, and only 0.01% might contain cancerous cells.

Class imbalance can also happen with regression tasks where the labels are continuous. Consider the task of estimating healthcare bills[15]. Healthcare cost is very high skewed — the median bill is low, but the 95th percentile bill is astronomical. When predicting hospital bills, it might be more important to predict accurately the bills at the 95th percentile than the median bills. A 100% difference in a $250 bill is acceptable (actual $500, predicted $250), but a 100% difference on a $10k bill is not (actual $20k, predicted $10k). Therefore, we might have to train the model to be better at predicting 95th percentile bills, even if it reduces the overall metrics.

Challenges of Class Imbalance

ML works well in situations when the data distribution is more balanced, and not so well when the classes are heavily imbalanced, as illustrated in Figure 3-7. Class imbalance can make learning difficult for the three following reasons.

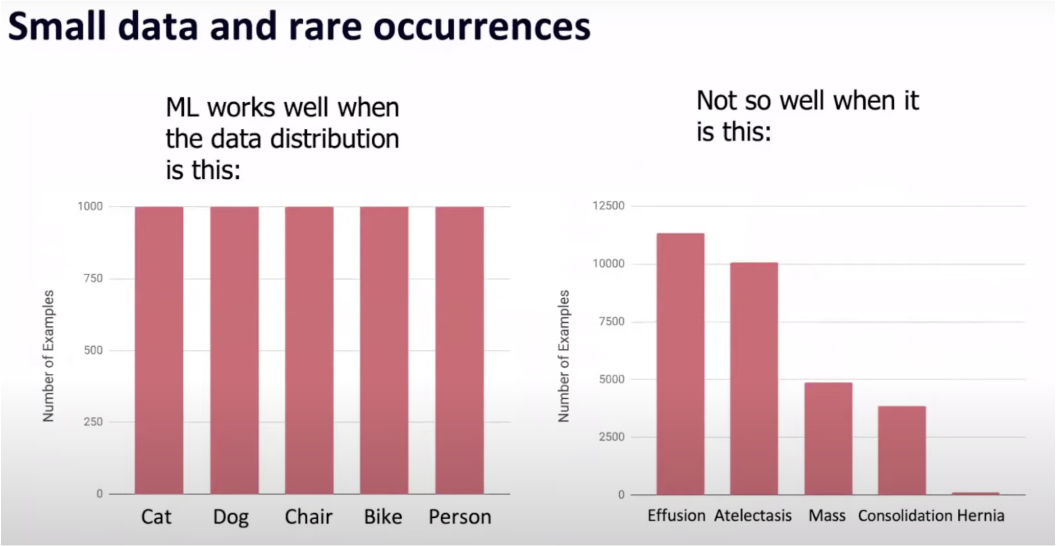


Figure 3-7: ML works well in situations where the classes are balanced.

[15] Thanks Eugene Yan for this wonderful example!

Image by Andrew Ng.

The first reason is that class imbalance often means that there's **insufficient signal** for your model to learn to detect the minority classes. In the case where there is a small number of instances in the minority class, the problem becomes a few-shot learning problem where your model only gets to see the minority class a few times before having to make a decision on it. In the case where there is no instance of the rare classes in your training set, your model might assume that these rare classes don't exist.

The second reason is that class imbalance makes it easier for your model to **get stuck in a non-optimal solution** by learning a simple heuristic instead of learning anything useful about the underlying structure of the data. Consider the lung cancer detection example above. If your model learns to always output the majority class, its accuracy is already 99.99%. This heuristic can be very hard for gradient-descent algorithms to beat because a small amount of randomness added to this heuristic might lead to worse accuracy.

The third reason is that class imbalance leads to **asymmetric costs of error** — the cost of a wrong prediction on an example of the rare class might be much higher than a wrong prediction on an example of the majority class.

For example, misclassification on an X-Ray with cancerous cells is much more dangerous than misclassification on an X-Ray of a normal lung. If your loss function isn't configured to address this asymmetry, your model will treat all examples the same way. As a result, you might obtain a model that performs equally well on both majority and minority classes, while you much prefer a model that performs less well on the majority class but much better on the minority one.

When I was in school, most datasets I was given had more or less balanced classes[16]. It was a shock for me to start working and realize that class imbalance is the norm. In real-world settings, rare events are often more interesting (or more dangerous) than regular events, and many tasks focus on detecting those rare events.

The classical example of tasks with class imbalance is **fraud detection**. Most credit card transactions are not fraudulent. As of 2018, 6.8¢ for every $100 in cardholder spending is fraudulent. Another is **churn prediction**. The majority of your customers are not planning on canceling their subscription. If they are, your business has more to worry about than churn prediction algorithms. Other examples include **disease screening** — most people, fortunately, don't have terminal illness, and **resume screening** — 98% of job seekers are eliminated at the initial resume screening. A less obvious example of a task with class imbalance is **object detection**. Object detection algorithms currently work by generating a large number of bounding boxes over an image then predicting which boxes are most likely to have objects in them. Most bounding boxes do not contain a relevant object.

[16] I imagined that it'd be easier to learn machine learning theory if I don't have to figure out how to deal with class imbalance.

Outside the cases where class imbalance is inherent in the problem, class imbalance can also be caused by biases during the sampling process. Consider the case when you want to create training data to detect whether an email is spam or not. You decide to use all the anonymized emails from your company's email database. According to Talos Intelligence, as of May 2021, nearly 85% of all emails are spam. But most spam emails were filtered out before they reached your company's database, so in your dataset, only a small percentage is spam.

Another cause for class imbalance, though less common, is due to labeling errors. Your annotators might have read the instructions wrong or followed the wrong instructions (thinking there are only two classes POSITIVE and NEGATIVE while there are actually three), or simply made errors. Whenever faced with the problem of class imbalance, it's important to examine your data to understand the causes of it.

Handling Class Imbalance

Because of its prevalence in real-world applications, class imbalance has been thoroughly studied over the last two decades[17]. Class imbalance affects tasks differently based on the level of imbalance. Some tasks are more sensitive to class imbalance than others. Japkowicz showed that sensitivity to imbalance increases with the complexity of the problem, and that non-complex, linearly separable problems are unaffected by all levels of class imbalance[18]. Class imbalance in binary classification problems is a much easier problem than class imbalance in multiclass classification problems. Ding et al. showed that very-deep neural networks — with "very deep" meaning over 10 layers back in 2017 — performed much better on imbalanced data than shallower neural networks[19].

There have been many techniques suggested to mitigate the effect of class imbalance. However, as neural networks have grown to be much larger and much deeper, with more learning capacity, some might argue that you shouldn't try to "fix" class imbalance if that's how the data looks in the real world. A good model should learn to model that class imbalance. However, developing a model good enough for that can be challenging, so we still have to rely on special training techniques.

In this section, we will cover three approaches to handle class imbalance: choosing the right metrics for your problem, data-level methods, which means changing the data distribution to make it less imbalanced, and algorithm-level methods, which means changing your learning method to make it more robust to class imbalance.

[17] The Class Imbalance Problem: A Systematic Study (Nathalie Japkowciz and Shaju Stephen, 2002)
[18] The Class Imbalance Problem: Significance and Strategies (Nathalie Japkowicz, 2000)
[19] Facial action recognition using very deep networks for highly imbalanced class distribution (Ding et al., 2017)

These techniques might be necessary but not sufficient. For a comprehensive survey, I recommend *Survey on deep learning with class imbalance* (Johnson and Khoshgoftaar, Journal of Big Data 2019).

Using the right evaluation metrics

The most important thing to do when facing a task with class imbalance is to choose the appropriate evaluation metrics. Wrong metrics will give you the wrong ideas of how your models are doing, and subsequently, won't be able to help you develop or choose models good enough for your task.

The overall accuracy and error rate are the most frequently used metrics to report the performance of ML models. However, they are insufficient metrics for tasks with class imbalance because they treat all classes equally, which means the performance of your model on the majority class will dominate the accuracy. This is especially bad when the majority class isn't what you care about.

Consider a task with two labels: CANCER (positive) and NORMAL, where 90% of the labeled data is NORMAL. Consider two models A and B with the following confusion matrices.

Model A	**Actual CANCER**	**Actual NORMAL**
Predicted CANCER	10	10
Predicted NORMAL	90	890

Table 3-4: Model A's confusion matrix.

Model A can detect 10 out of 100 CANCER cases.

Model B	**Actual CANCER**	**Actual NORMAL**
Predicted CANCER	90	90
Predicted NORMAL	10	810

Table 3-5: Model B's confusion matrix.

Model B can detect 90 out of 100 CANCER cases.

If you're like most people, you'd probably prefer model B to make predictions for you since it has a better chance of telling you if you actually have cancer. However, they both have the same accuracy of 0.9.

Metrics that help you understand your model's performance with respect to specific classes would be better choices. Accuracy can still be a good metric if you use it for each class individually. The accuracy of Model A on the CANCER is 10% and the accuracy of model B on the CANCER class is 90%.

F1 and recall are metrics that measure your model's performance with respect to the positive class in binary classification problems, as they rely on true positive — an outcome where the model correctly predicts the positive class[20]. F1 and recall are asymmetric metrics, which means that their values change depending on which class is considered the positive class. In our case, if we consider CANCER the positive class, model A's F1 is 0.17. However, if we consider NORMAL the positive class, model A's F1 is 0.95.

In multiclass classification problems, you can calculate F1 for each individual class.

	CANCER (1)	NORMAL (0)	Accuracy	Precision	Recall	F1
Model A	10/100	890/900	0.9	0.5	0.1	0.17
Model B	90/100	810/900	0.9	0.5	0.9	0.64

Table 4-6: Model A and model B have the same accuracy even though one model is clearly superior to another.

Many classification problems can be modeled as regression problems. Your model can output a value, and based on that value, you classify the example. For example, if the value is greater than 0.5, it's a positive label, and if it's less than or equal to 0.5, it's a negative label. This means that you can tune the threshold to increase the **true positive rate** (also known as **recall**) while decreasing the **false positive rate** (also known as the **probability of false alarm),** and vice versa. We can plot the true positive rate against the false positive rate for different thresholds. This plot is known as the **ROC curve** (Receiver Operating Characteristics). When your model is perfect, the recall is 1.0, and the curve is just a line at the top. This curve shows you how your model's performance changes depending on the threshold, and helps you choose the threshold that works best for you. The closer to the perfect line the better your model's performance.

The area under the curve (AUC) measures the area under the ROC curve. Since the closer to the perfect line the better, the larger this area the better.

[20] As of July 2021, when you use scikit-learn.metrics.f1_score, pos_label is set to 1 by default, but you can change to 0 if you want 0 to be your positive label.

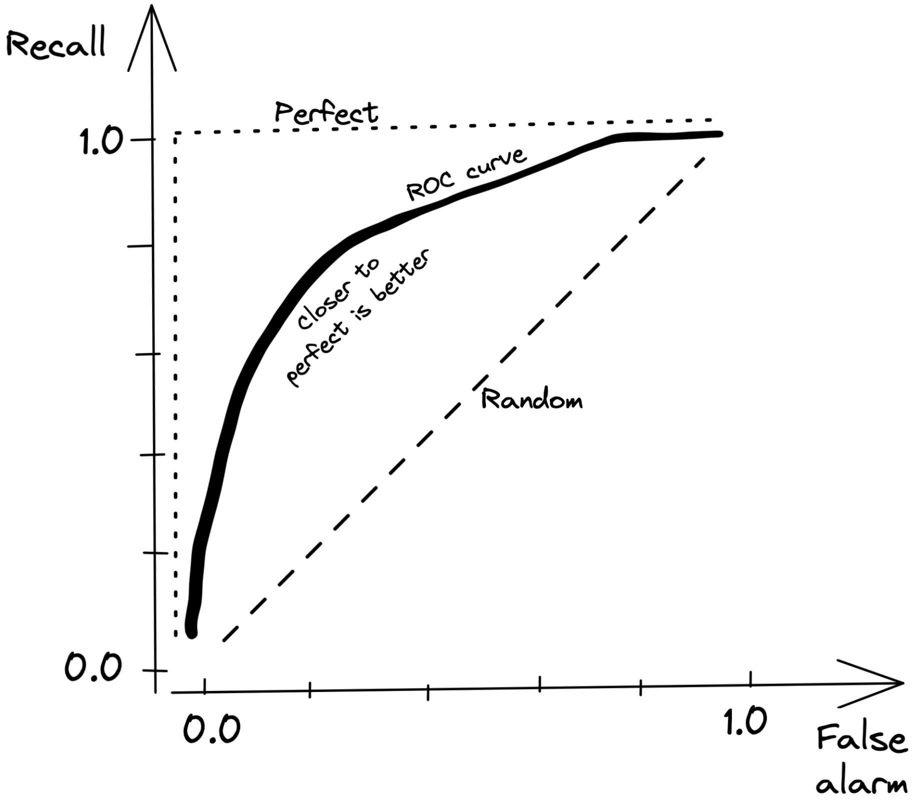


Figure 3-8: ROC curve

Like F1 and recall, the ROC curve focuses only on the positive class and doesn't show how well your model does on the negative class. Davis and Goadrich suggested that we should plot precision against recall instead, in what they termed the Precision-Recall Curve. They argued that this curve s give a more informative picture of an algorithm's performance on tasks with heavy class imbalance[21].

Data-level methods: Resampling

Data-level methods modify the distribution of the training data to reduce the level of imbalance to make it easier for the model to learn. A common family of techniques is resampling. Resampling includes oversampling, adding more examples from the minority classes and undersampling, removing examples of the majority classes. The simplest way to undersample is to randomly remove instances from the majority class, while the simplest way to oversample is to randomly make copies of the minority class until you have a ratio that you're happy with.

[21] The Relationship Between Precision-Recall and ROC Curves (Davis and Goadrich, 2006).

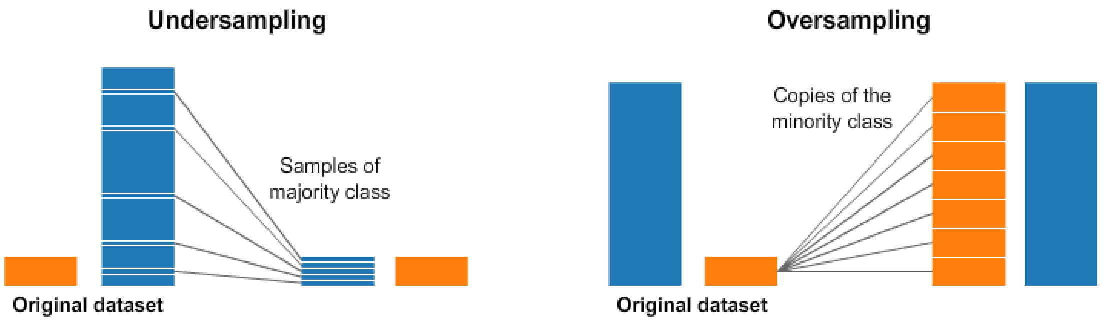


Figure 3-9: Illustrations of how undersampling and oversampling works.
Image by Rafael Alencar

A popular method of undersampling low-dimensional data that was developed back in 1976 is Tomek links[22]. With this technique, you find pairs of samples from opposite classes that are close in proximity, and remove the sample of the majority class in each pair.

While this makes the decision boundary more clear and arguably helps models learn the boundary better, it may make the model less robust by removing some of the subtleties of the true decision boundary.

A popular method of oversampling low-dimensional data is SMOTE. It synthesizes novel samples of the minority class through sampling convex[23] combinations of existing data points within the minority class.

Both SMOTE and Tomek Links have only been proven effective in low-dimensional data. Many of the sophisticated resampling techniques, such as Near-Miss[24] and one-sided selection[25], require calculating the distance between instances or between instances and the decision boundaries, which can be expensive or infeasible for high-dimensional data or in high-dimensional feature space, such as the case with large neural networks.

When you resample your training data, never evaluate your model on resampled data, since it'll cause your model to overfit to that resampled distribution.

Undersampling runs the risk of losing important data from removing data. Oversampling runs the risk of overfitting on training data, especially if the added copies of the minority class are replicas of existing data. Many sophisticated sampling techniques have been developed to mitigate these risks.

[22] An Experiment with the Edited Nearest-Neighbor Rule (Ivan Tomek, IEEE 1876)
[23] "Convex" here approximately means "linear".
[24] KNN Approach to Unbalanced Data Distributions: A Case Study Involving Information Extraction (Zhang and Mani, 2003)
[25] Addressing the curse of imbalanced training sets: one-sided selection (Kubat and Matwin, 2000)

One such technique is two-phase learning[26]. You first train your model on the resampled data. This resampled data can be achieved by randomly undersampling large classes until each class has only N instances. You then finetune your model on the original data.

Another technique is dynamic sampling: oversample the low performing classes and undersample the high performing classes during the training process. Introduced by Pouyanfar et al.[27], the method aims to show the model less of what it has already learned and more of what it has not.

Algorithm-level methods

If data-level methods mitigate the challenge of class imbalance by altering the distribution of your training data, algorithm-level methods keep the training data distribution intact but alter the algorithm to make it more robust to class imbalance.

Because the loss function (or the cost function) guides the learning process, many algorithm-level methods involve adjustment to the loss function. The key idea is that if there are two instances x_1 and x_2 and the loss resulting from making the wrong prediction on x_1 is higher than x_2, the model will prioritize making the correct prediction on x_1 over making the correct prediction on x_2. By giving the training instances we care about higher weight, we can make the model focus more on learning these instances.

Let $L(x\ ; \theta)$ be the loss caused by the instance x for the model with the parameter set θ. The model's loss is often defined as the average loss caused by all instances.

$$L(X;\ \theta)\ =\ \sum_x L(x\ ; \theta)$$

This loss function values the loss caused by all instances equally, even though wrong predictions on some instances might be much costlier than wrong predictions on other instances. There are many ways to modify this cost function. In this section, we will focus on three of them, starting with cost-sensitive learning.

Cost-sensitive learning

Back in 2001, based on the insight that misclassification of different classes incur different cost, Elkan proposed cost-sensitive learning where the individual loss function is modified to take into account this varying cost[28]. The method started by using a cost matrix to specify C_{ij}: the cost if

[26] Plankton classification on imbalanced large scale database via convolutional neural networks with transfer learning (Lee et al., 2016)

[27] Dynamic sampling in convolutional neural networks for imbalanced data classification (Pouyanfar et al., 2018)

[28] The foundations of cost-sensitive learning (Elkan, IJCAI 2001)

class i is classified as class j. If i = j, it's a correct classification, and the cost is usually 0. If not, it's a misclassification. If classifying POSITIVE examples as NEGATIVE is twice as costly as the other way around, you can make C_{10} twice as high as C_{01}.

For example, if you have two classes: POSITIVE and NEGATIVE, the cost matrix can look like this.

	Actual NEGATIVE	Actual POSITIVE
Predicted NEGATIVE	$C(0,\ 0)\ =\ C_{00}$	$C(1,\ 0)\ =\ C_{10}$
Predicted POSITIVE	$C(0,\ 1)\ =\ C_{01}$	$C(1,\ 1)\ =\ C_{11}$

Table 3-7: Example of a cost matrix.

The loss caused by instance x of class i will become the weighted average of all possible classifications of instance x.

$$L(x\ ;\theta)\ =\ \sum_j C_{ij}P(j\mid x;\ \theta)$$

The problem with this loss function is that you have to manually define the cost matrix, which is different for different tasks at different scales.

Class-balanced loss

What might happen with a model trained on an imbalanced dataset is that it'll bias toward majority classes and make wrong predictions on minority classes. What if we punish the model for making wrong predictions on minority classes to correct this bias?

In its vanilla form, we can make the weight of each class inversely proportional to the number of samples in that class, so that the rarer classes have higher weights. In the following equation, N denotes the total number of training samples.

$$W_i = \frac{N}{number\ of\ samples\ of\ class\ i}$$

The loss caused by instance x of class i will become as follows, with $Loss(x,\ j)$ being the loss when x is classified as class j. It can be cross entropy or any other loss function.

$$L(x\ ;\theta)\ =\ W_i\sum_j P(j\mid x;\ \theta)Loss(x,\ j)$$

A more sophisticated version of this loss can take in account the overlap among existing samples, such as *Class-Balanced Loss Based on Effective Number of Samples* (Cui et al., CVPR 2019).

Focal loss

In our data, some examples are easier to classify than others, and our model might learn to classify them quickly. We want to incentivize our model to focus on learning the samples they still have difficulty classifying. What if we adjust the loss so that if a sample has a lower probability of being right, it'll have a higher weight? This is exactly what Focal Loss does[29].

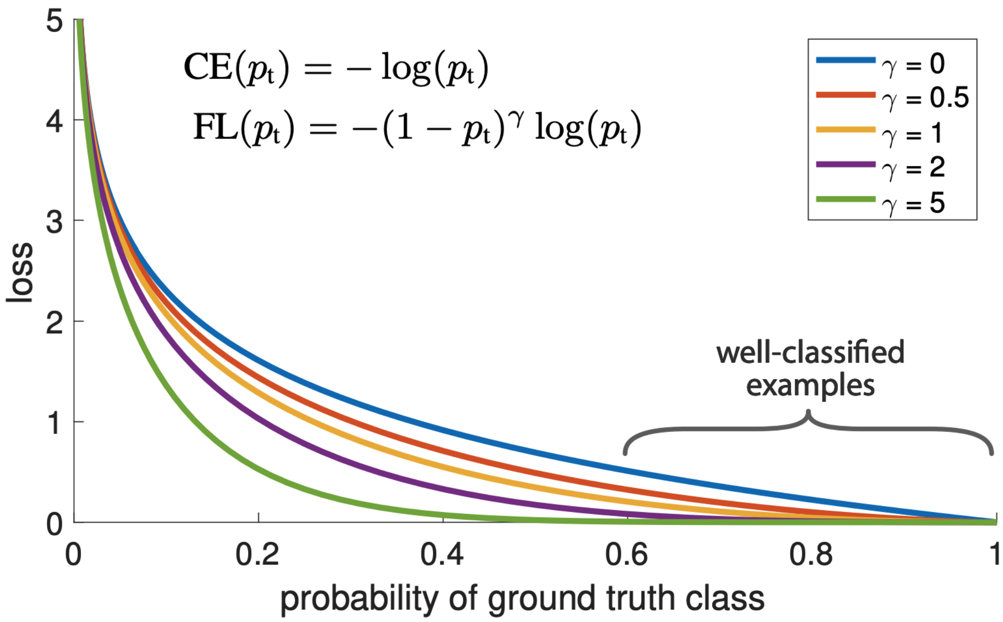


Figure 3-10: The model trained with focal loss (FL) shows reduced loss values compared to the model trained with cross entropy loss (CE). Image by Lin et al.

In practice, ensembles have shown to help with the class imbalance problem[30]. However, we don't include ensembling in this section because class imbalance isn't usually why ensembles are used. Ensemble techniques will be covered in Chapter 5: Model Development and Evaluation.

Summary

Training data still forms the foundation of modern ML algorithms. No matter how clever your algorithms might be, if your training data is bad, your algorithms won't be able to perform well.

[29] Focal Loss for Dense Object Detection (Lin et al., 2017)

[30] A Review on Ensembles for the Class Imbalance Problem: Bagging-, Boosting-, and Hybrid-Based Approaches (Galar et al., 2011)

It’s worth it to invest time and effort to curate and create training data that will enable your algorithms to learn something meaningful.

Once you have your training data, you will want to extract features from it to train your ML models, which we will cover in the next chapter.