# Chapter 4. Training Data

In [Chapter 3](), 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 toward modeling, which is considered by many practitioners 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 not handled properly, it can easily sink your entire ML operation. But this is precisely the reason why data scientists and 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 the section ["Data Distribution Shifts"](). 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 bi-

ases. 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 case is when you don't have access to all possible data in the real world, the data that you use to train your model is a subset of real-world data, created by one sampling method or another. Another case is when it's infeasible to process all the data that you have access to—because it requires too much time or resources—so you have to sample that data to create a subset that is feasible to 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 the new model is promising first before training this new model on all your data.[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: nonprobability sampling and random sampling. We'll start with nonprobability sampling methods, followed by several common random sampling methods.

## Nonprobability Sampling

Nonprobability sampling is when the selection of data isn't based on any probability criteria. Here are some of the criteria for nonprobability 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 they follow, 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, and above 60 years old, regardless of the actual age distribution.

The samples selected by nonprobability 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.

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, Common Crawl, 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 toward users who are willing to leave reviews online, and not necessarily representative of people who don't have access to the internet or people who aren't willing to put reviews online.

A third example is data for training self-driving cars. Initially, data collected for self-driving cars came largely from two areas: Phoenix, 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, Washington, specially for Kirkland's rainy weather,[3] but there's still a lot more self-driving car data for sunny weather than for rainy or snowy weather.

Nonprobability 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.[4] For example, you randomly select 10% of the population, giving all members of this population 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, 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 stratum, 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.[5] 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, and C, and want them to be selected with the probabilities of 50%, 30%, and 20% respectively, you can give them the weights 0.5, 0.3, and 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 weights three times higher than blue samples.

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

```python
# 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.
import random
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 4-1.
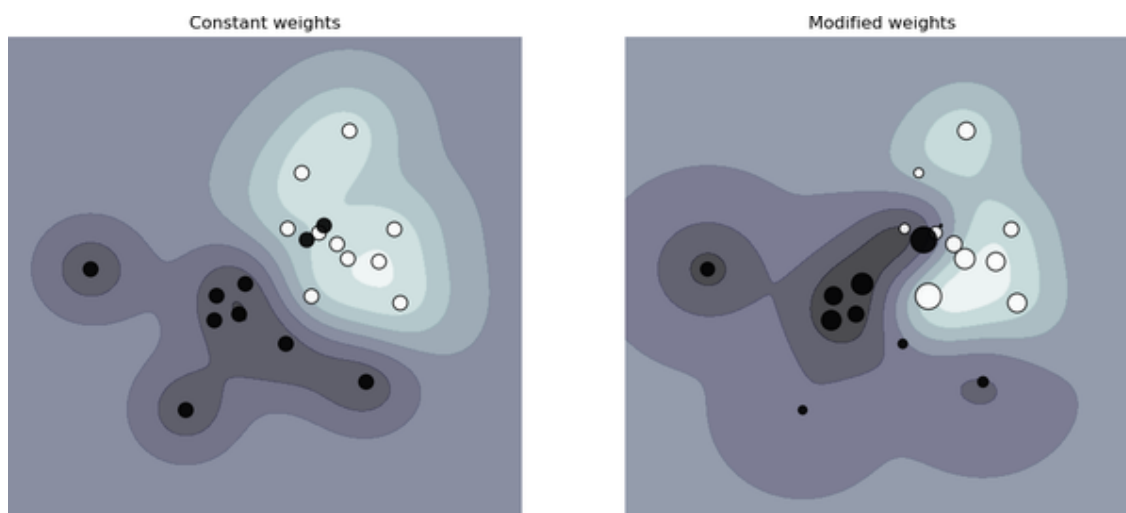


Figure 4-1. Sample weights can affect the decision boundary. On the left is when all samples are given equal weights. On the right is when samples are given different weights. Source: scikit-learn[6]

## Reservoir Sampling

Reservoir sampling is a fascinating algorithm that is especially useful when you have to deal with streaming 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 in advance the probability at which a tweet should be selected. You want to ensure that:

- Every tweet has an equal probability of being selected.
- 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 $n^{th}$ element, generate a random number $i$ such that $1 \leq i \leq n$.
3. If $1 \leq i \leq k$: replace the $i^{th}$ element in the reservoir with the $n^{th}$ element. Else, do nothing.

This means that each incoming $n^{th}$ 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 4-2 shows an illustrative example of how reservoir sampling works.
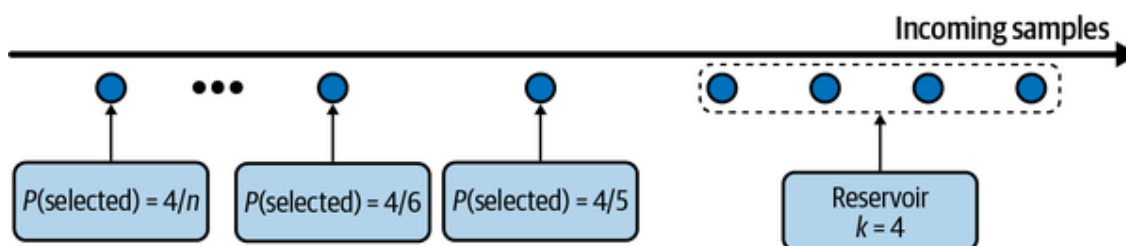


Figure 4-2. A visualization of how reservoir sampling works

# 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 following equation shows that in expectation, $x$ sampled from $P(x)$ is equal to $x$ sampled from $Q(x)$ weighted by $\frac{P(x)}{Q(x)}$:

$$E_{P(x)}[x] = \sum_x P(x)x = \sum_x Q(x)x\frac{P(x)}{Q(x)} = E_{Q(x)}\left[x\frac{P(x)}{Q(x)}\right]$$

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.

## Labeling

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

In a talk to my students, Andrej Karpathy, director of AI at Tesla, shared an anecdote about how 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 challenge of obtaining labels for your data. We'll first discuss the labeling method that usually comes first in data scientists' mind when talking about labeling: hand-labeling. We will then discuss tasks with natural labels, which are tasks where labels can be inferred from the system without requiring human annotations, followed by what to do when natural and hand labels are lacking.

## 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 20 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 patients' 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 premises.

Third, hand labeling is slow. For example, accurate transcription of speech utterance at the phonetic level can take 400 times longer than the utterance duration.[7] So if you want to annotate 1 hour of speech, it'll take 400 hours or almost 3 months for a person 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 conflicting 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 4-1. Three annotators have identified different entities. Which one should your model train on? A model trained on data labeled by annotator 1 will perform very differently from a model trained on data labeled by annotator 2.

Table 4-1. Entities identified by different annotators might be very different

| 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*]. |

Disagreements among annotators are extremely common. The higher the level of domain expertise required, the higher the potential for annotating disagreement.[8] 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 preceding entity recognition task, 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 the annotators' 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 crowd-sourced 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.

It's good practice to keep track of the origin of each of your data samples as well as its labels, a technique known as *data lineage*. Data lineage helps you both flag potential biases in your data and debug your models. For example, if your 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.

## Natural Labels

Hand-labeling isn't the only source for labels. You might be lucky enough to work on tasks with natural ground truth labels. Tasks with natural labels are tasks where the model's predictions can be automatically evaluated or partially evaluated by the system. An example is the model that estimates time of arrival for a certain route on Google Maps. If you take that route, by the end of your trip, Google Maps knows how long the trip actually took, and thus can evaluate the accuracy of the predicted time of arrival. Another example is stock price prediction. If your model predicts a stock's price in the next two minutes, then after two minutes, you can compare the predicted price with the actual price.

The canonical example of tasks with natural labels is recommender systems. The goal of a recommender system is to recommend to users items

relevant to them. Whether a user clicks on the recommended item or not can be seen as the feedback for that recommendation. A recommendation that gets clicked on can be presumed to be good (i.e., the label is POSITIVE) and a recommendation that doesn't get clicked on after a period of time, say 10 minutes, can be presumed to be bad (i.e., the label is NEGATIVE).

Many tasks can be framed as recommendation tasks. For example, you can frame the task of predicting ads' click-through rates as recommending the most relevant ads to users based on their activity histories and profiles. Natural labels that are inferred from user behaviors like clicks and ratings are also known as behavioral labels.

Even if your task doesn't inherently have natural labels, it might be possible to set up your system in a way that allows you to collect some feedback on your model. For example, if you're building a machine translation system like Google Translate, you can have the option for the community to submit alternative translations for bad translations—these alternative translations can be used to train the next iteration of your models (though you might want to review these suggested translations first). Newsfeed ranking is not a task with inherent labels, but by adding the Like button and other reactions to each newsfeed item, Facebook is able to collect feedback on their ranking algorithm.

Tasks with natural labels are fairly common in the industry. In a survey of 86 companies in my network, I found that 63% of them work with tasks with natural labels, as shown in Figure 4-3. This doesn't mean that 63% of tasks that can benefit from ML solutions have natural labels. What is more likely is that companies find it easier and cheaper to first start on tasks that have natural labels.
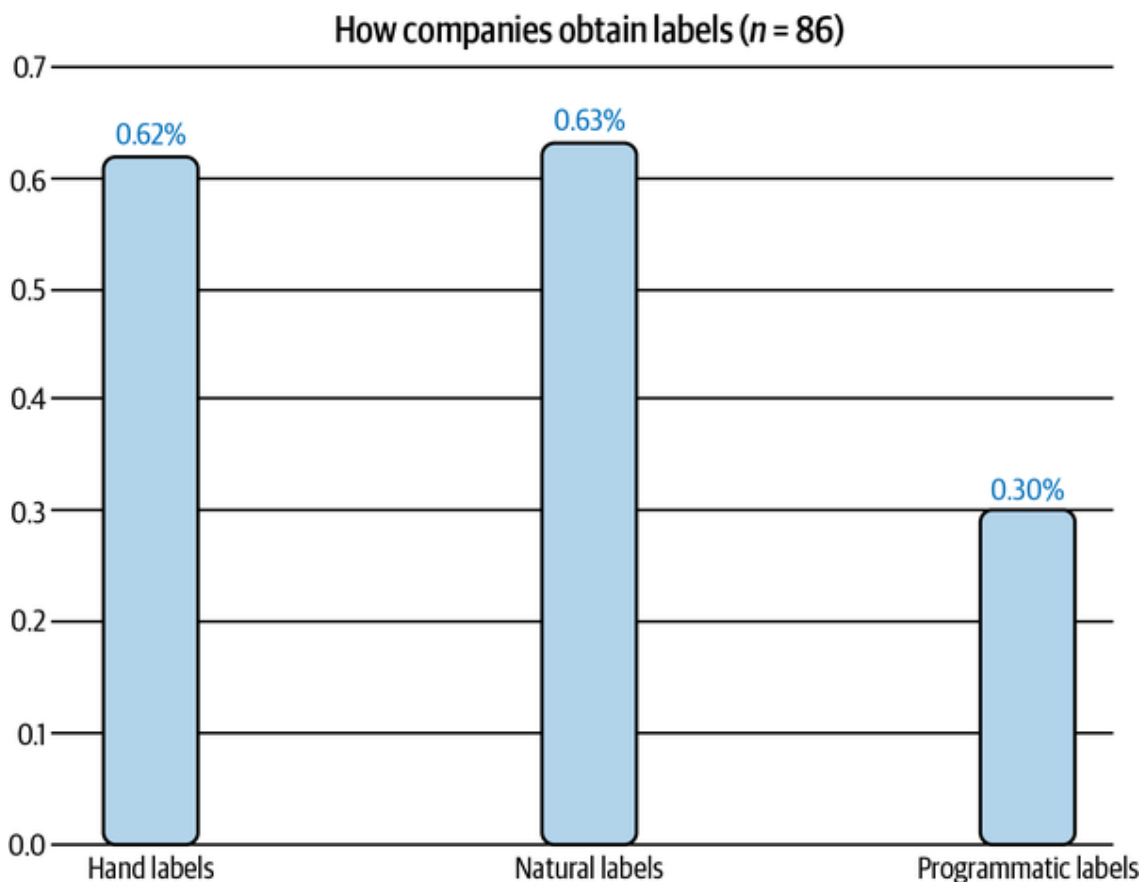
Figure 4-3. Sixty-three percent of companies in my network work on tasks with natural labels. The percentages don't sum to 1 because a company can work with tasks with different label sources.[9]

In the previous example, a recommendation that doesn't get clicked on after a period of time can be presumed to be bad. This is called an *implicit label*, as this negative label is presumed from the lack of a positive label. It's different from *explicit labels* where users explicitly demonstrate their feedback on a recommendation by giving it a low rating or downvoting it.

## Feedback loop length

For tasks with natural ground truth labels, the time it takes from when a prediction is served until when the feedback on it is provided is the feedback loop length. Tasks with short feedback loops are tasks where labels are generally available within minutes. Many recommender systems have short feedback loops. If the recommended items are related products on Amazon or people to follow on Twitter, the time between when the item is recommended until it's clicked on, if it's clicked on at all, is short.

However, not all recommender systems have minute-long feedback loops. If you work with longer content types like blog posts or articles or

YouTube videos, the feedback loop can be hours. If you build a system to recommend clothes for users like the one Stitch Fix has, you wouldn't get feedback until users have received the items and tried them on, which could be weeks later.

---

If you want to extract labels from user feedback, it's important to note that there are different types of user feedback. They can occur at different stages during a user journey on your app and differ by volume, strength of signal, and feedback loop length.

For example, consider an ecommerce application similar to what Amazon has. Types of feedback a user on this application can provide might include clicking on a product recommendation, adding a product to cart, buying a product, rating, leaving a review, and returning a previously bought product.

Clicking on a product happens much faster and more frequently (and therefore incurs a higher volume) than purchasing a product. However, buying a product is a much stronger signal on whether a user likes that product compared to just clicking on it.

When building a product recommender system, many companies focus on optimizing for clicks, which give them a higher volume of feedback to evaluate their models. However, some companies focus on purchases, which gives them a stronger signal that is also more correlated to their business metrics (e.g., revenue from product sales). Both approaches are valid. There's no definite answer to what type of feedback you should optimize for your use case, and it merits serious discussions between all stakeholders involved.

---

Choosing the right window length requires thorough consideration, as it involves the speed and accuracy trade-off. A short window length means that you can capture labels faster, which allows you to use these labels to detect issues with your model and address those issues as soon as possi-

ble. However, a short window length also means that you might prematurely label a recommendation as bad before it's clicked on.

No matter how long you set your window length to be, there might still be premature negative labels. In early 2021, a study by the Ads team at Twitter found that even though the majority of clicks on ads happen within the first five minutes, some clicks happen hours after when the ad is shown.[10] This means that this type of label tends to give an underestimate of the actual click-through rate. If you only record 1,000 POSITIVE labels, the actual number of clicks might be a bit over 1,000.

For tasks with long feedback loops, natural labels might not arrive for weeks or even months. Fraud detection is an example of a task with long feedback loops. For a certain period of time after a transaction, users can dispute whether that transaction is fraudulent or not. For example, when a customer read their credit card statement and saw a transaction they didn't recognize, they might dispute it with their bank, giving the bank the feedback to label that transaction as fraudulent. A typical dispute window is one to three months. After the dispute window has passed, if there's no dispute from the user, you might presume the transaction to be legitimate.

Labels with long feedback loops are helpful for reporting a model's performance on quarterly or yearly business reports. However, they are not very helpful if you want to detect issues with your models as soon as possible. If there's a problem with your fraud detection model and it takes you months to catch, by the time the problem is fixed, all the fraudulent transactions your faulty model let through might have caused a small business to go bankrupt.

## Handling the Lack of 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. A summary of these methods is shown in Table 4-2.

Table 4-2. Summaries of four techniques for handling the lack of hand-labeled data

| 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 |

## 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.[11] 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 preceding heuristics can be expressed by the following function:

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

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

*Keyword heuristic*

> Such as the preceding example

*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, labels produced by LFs are noisy. Multiple LFs might apply to the same data examples,

and they might give conflicting labels. One function might think a nurse's 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. Figure 4-4 shows at a high level how LFs work.
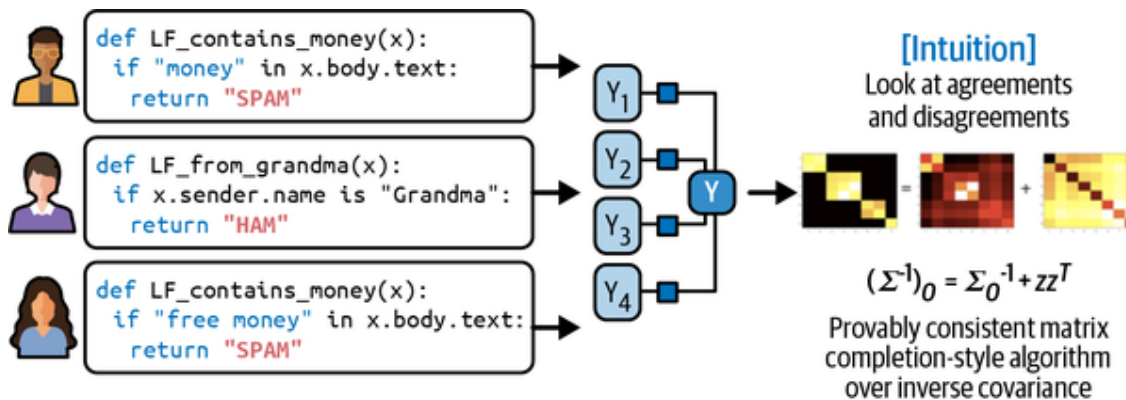


Figure 4-4. A high-level overview of how labeling functions are combined. Source: Adapted from an image by Ratner et al.[12]

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 number 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 LFs to generate labels for your data is also known as programmatic labeling. Table 4-3 shows some of the advantages of programmatic labeling over hand labeling.

Table 4-3. 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 |
| **Lack of privacy**: Need to ship data to human annotators | **Privacy**: Create LFs using a cleared data subsample and then apply LFs to other data without looking at individual samples |
| **Slow**: Time required scales linearly with number of labels needed | **Fast**: Easily scale from 1K to 1M samples |
| **Nonadaptive**: Every change requires relabeling the data | **Adaptive**: When changes happen, just reapply LFs! |

Here is a case study to show how well weak supervision works in practice. In a study with Stanford Medicine,[13] models trained with weakly supervised labels obtained by a single radiologist after eight hours of writing LFs had comparable performance with models trained on data obtained through almost a year of hand labeling, as shown in Figure 4-5. There are two interesting facts about the results of the experiment. First, the models continued improving with more unlabeled data even without more LFs. Second, LFs were being reused across tasks. The researchers were able to reuse six LFs between the CXR (chest X-rays) task and EXR (extremity X-rays) task.[14]
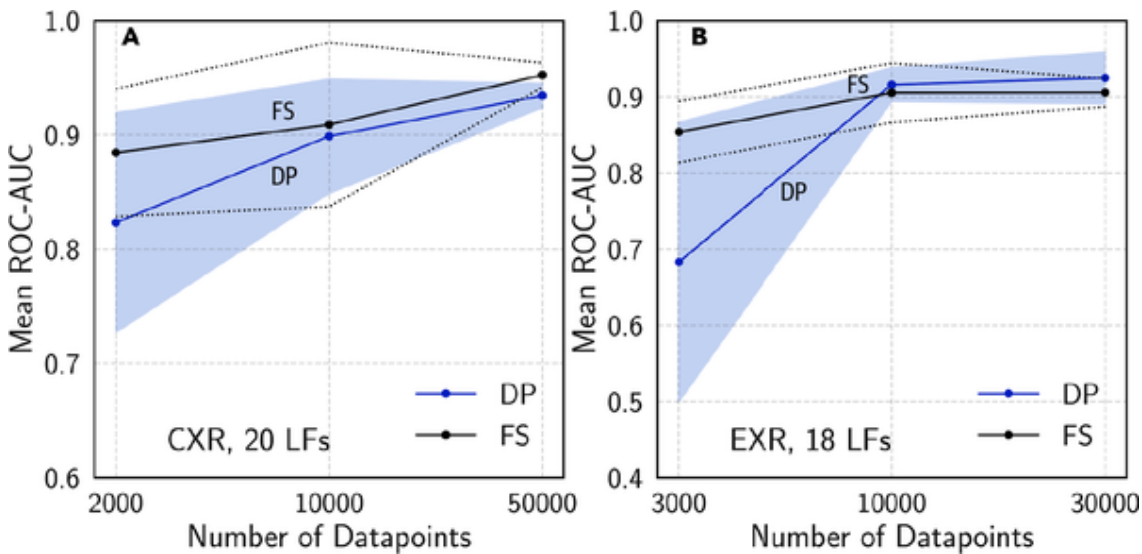
Figure 4-5. Comparison of the performance of a model trained on fully supervised labels (FS) and a model trained with programmatic labels (DP) on CXR and EXR tasks. Source: Dunnmon et al.[15]

My students often ask that if heuristics work so well to label data, why do we need ML models? One reason is that LFs might not cover all data samples, so we can train ML models on data programmatically labeled with LFs and use this trained model to generate predictions for samples that aren't covered by any LF.

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 even in these cases, weak supervision can be a good way to get you started when you want to explore the effectiveness of ML without wanting to invest too much in hand labeling up front.

## 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,[16] 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. 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 in Figure 4-6, you can also label the hashtags "#ML" and "#BigData" as Computer Science.



Figure 4-6. Because #ML and #BigData appear in the same Twitter profile as #AI, 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 neighbors algorithm 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 instances to obtain new training instances. The perturbations might be applied directly to the samples (e.g., adding white noise to images) or to their representations (e.g., adding small ran-

dom 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".

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.[17]

Semi-supervision is the most useful when the number of training labels is limited. One thing to consider when doing semi-supervision with limited data is how much of this limited data should be used to evaluate multiple candidate models and select the best one. If you use a small amount, the best performing model on this small evaluation set might be the one that overfits the most to this set. On the other hand, if you use a large amount of data for evaluation, the performance boost gained by selecting the best model based on this evaluation set might be less than the boost gained by adding the evaluation set to the limited training set. Many companies overcome this trade-off by using a reasonably large evaluation set to select the best model, then continuing training the champion model on the evaluation 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. 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. Language models can be trained on any body of text—books, Wikipedia articles, chat histories—and the task is: given a sequence of tokens,[18] predict the next token. When given the sequence "I bought NVIDIA shares because I believe in the importance of," a language model might output "hardware" or "GPU" as the next token.

The trained model can then be used for the task that you're interested in —a downstream task—such as sentiment analysis, intent detection, or question answering. In some cases, such as in zero-shot learning scenar-

ios, 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, such as continuing to train the base model or a part of the base model on data from a given downstream task.[19]

Sometimes, you might need to modify the inputs using a template to prompt the base model to generate the outputs you want.[20] For example, to use a language model as the base model for a question answering task, you might want to use this 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 nontrivial 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 or GPT-3.[21] Transfer learning also lowers the entry barriers into

ML, as it helps reduce the up-front 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 will be able to afford to train large pretrained models. The rest of the industry will use these pretrained models directly or fine-tune 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 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 metrics or heuristics. The most straightforward metric 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 samples with the lowest probabilities for the predicted class. Figure 4-7 illustrates how well this method works on a toy example.
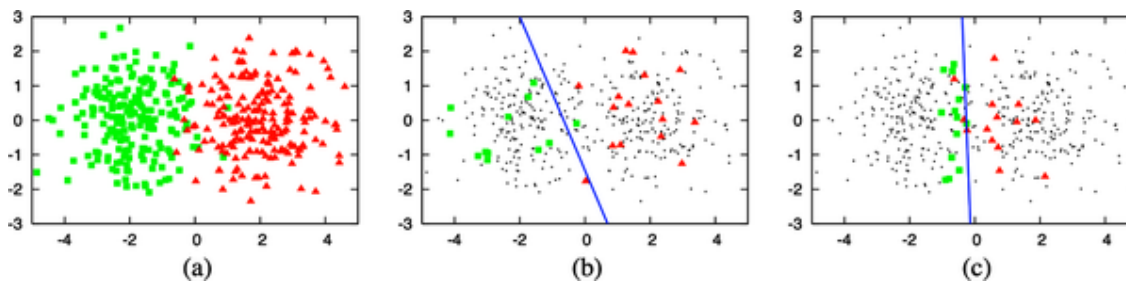
Figure 4-7. How uncertainty-based active learning works. (a) A toy dataset of 400 instances, evenly sampled from two class Gaussians. (b) A model trained on 30 samples randomly labeled gives an accuracy of 70%. (c) A model trained on 30 samples chosen by active learning gives an accuracy of 90%. Source: Burr Settles[22]

Another common heuristic is based on disagreement among multiple candidate models. This method is called query-by-committee, an example of an ensemble method.[23] You need a committee of several candidate models, which are usually the same model trained with different sets of hyperparameters or the same model trained on different slices of data. Each model can make one vote for which samples to label next, and it might vote based on how uncertain it is about the prediction. You then label the samples that the committee disagrees on the most.

There are other heuristics such as choosing samples 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" (Settles 2010).

The samples to be labeled can come from different data regimes. They can be synthesized where your model generates samples in the region of the input space that it's most uncertain about.[24] They can come from a stationary distribution where you've already collected a lot of unlabeled data and your model chooses samples 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 samples from this stream of data to label.

I'm 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 further detail in Chapter 8. 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 health-care bills.[25] Health-care bills are highly 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, especially deep learning, works well in situations when the data distribution is more balanced, and usually not so well when the classes are heavily imbalanced, as illustrated in Figure 4-8. Class imbalance can make learning difficult for the following three reasons.
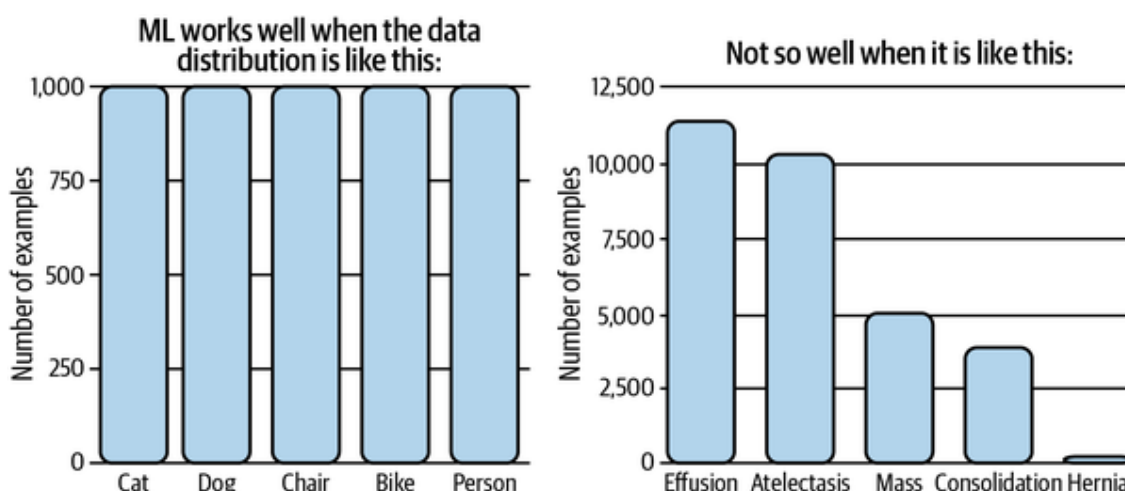


Figure 4-8. ML works well in situations where the classes are balanced. Source: Adapted from an image by Andrew Ng[26]

The first reason is that class imbalance often means 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 these rare classes don't exist.

The second reason is that class imbalance makes it easier for your model to get stuck in a nonoptimal solution by exploiting a simple heuristic instead of learning anything useful about the underlying pattern of the data. Consider the preceding lung cancer detection example. If your model learns to always output the majority class, its accuracy is already 99.99%.[27] 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 a sample of the rare class might be much higher than a wrong prediction on a sample 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 samples 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.[28] 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.[29] Another is churn predic-

tion. The majority of your customers are probably 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[30]).

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.

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.[31] 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. 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.[32] 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 noncomplex, linearly separable problems are unaffected by all

levels of class imbalance.[33] 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.[34]

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 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 handling 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.

These techniques might be necessary but not sufficient. For a comprehensive survey, I recommend "Survey on Deep Learning with Class Imbalance" (Johnson and Khoshgoftaar 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, these 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 these metrics. This is especially bad when the majority class isn't what you care about.

Consider a task with two labels: CANCER (the positive class) and NORMAL (the negative class), where 90% of the labeled data is NORMAL. Consider two models, A and B, with the confusion matrices shown in Tables 4-4 and 4-5.

Table 4-4. Model A's confusion matrix; model A can detect 10 out of 100 CANCER cases

| Model A | Actual CANCER | Actual NORMAL |
| --- | --- | --- |
| Predicted CANCER | 10 | 10 |
| Predicted NORMAL | 90 | 890 |

Table 4-5. Model B's confusion matrix; model B can detect 90 out of 100 CANCER cases

| Model B | Actual CANCER | Actual NORMAL |
| --- | --- | --- |
| Predicted CANCER | 90 | 90 |
| Predicted NORMAL | 10 | 810 |

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 class is 10% and the accuracy of model B on the CANCER class is 90%.

F1, precision, 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.[35]

For readers needing a refresh, precision, recall, and F1 scores, for binary tasks, are calculated using the count of true positives, true negatives, false positives, and false negatives. Definitions for these terms are shown in Table 4-6.

Table 4-6. Definitions of True Positive, False Positive, False Negative, and True Negative in a binary classification task

|  | **Predicted Positive** | **Predicted Negative** |
|---|---|---|
| Positive label | True Positive (hit) | False Negative (type II error, miss) |
| Negative label | False Positive (type I error, false alarm) | True Negative (correct rejection) |

Precision = True Positive / (True Positive + False Positive)

Recall = True Positive / (True Positive + False Negative)

F1 = 2 × Precision × Recall / (Precision + Recall)

F1, precision, 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. Accuracy, precision, recall, and F1 scores of model A and model B when CANCER is the positive class are shown in Table 4-7.

Table 4-7. Both models have the same accuracy even though one model is clearly superior

|  | 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 |

Many classification problems can be modeled as regression problems. Your model can output a probability, and based on that probability, you classify the sample. 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, as shown in Figure 4-9.
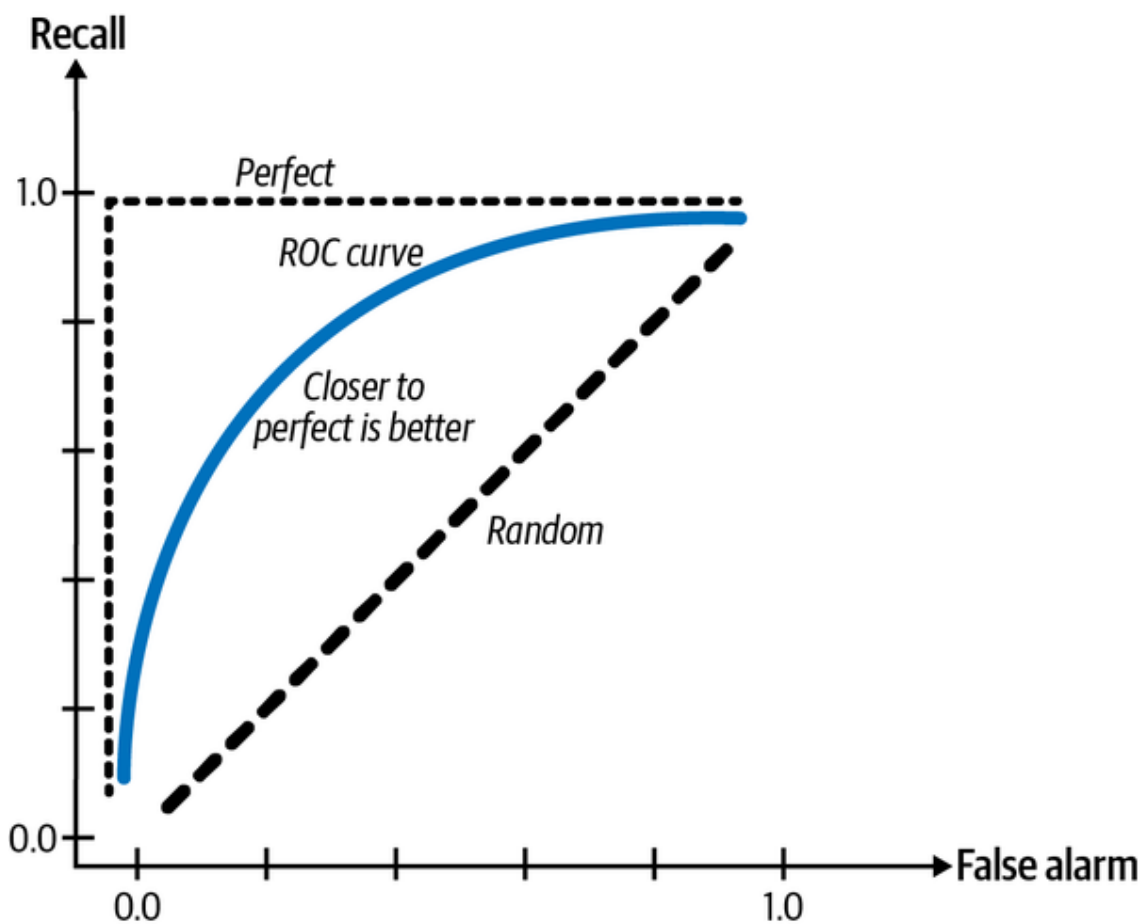
Figure 4-9. ROC curve

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 gives a more informative picture of an algorithm's performance on tasks with heavy class imbalance.[36]

## 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 instances from the minority classes, and undersampling, removing instances of the majority classes. The simplest way to undersample is to randomly remove instances from the majority class, whereas the simplest way to oversample is to randomly make copies of the minority class until you have a ratio that you're happy with. Figure 4-10 shows a visualization of oversampling and undersampling.
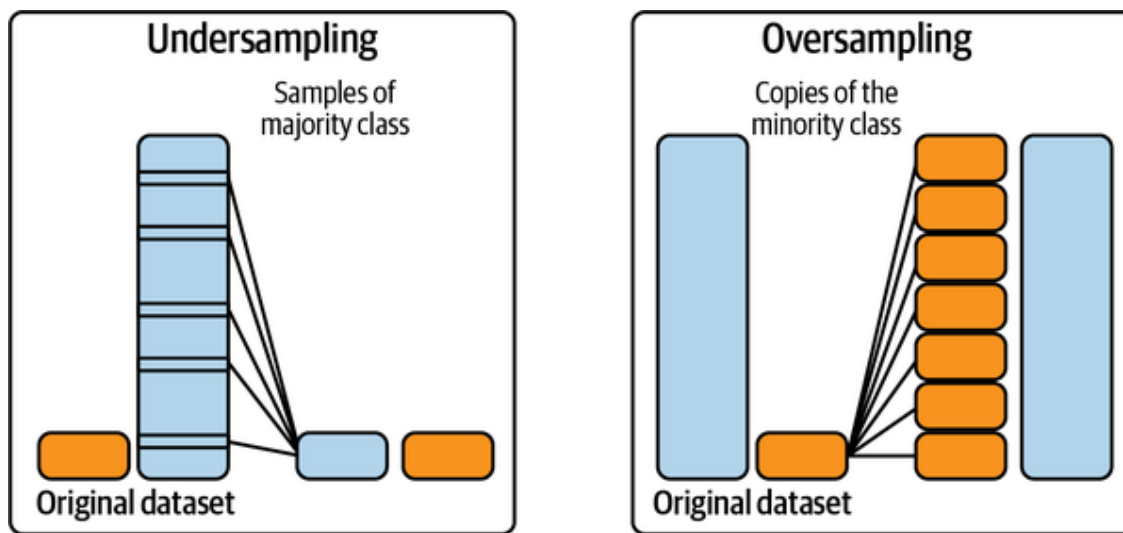
Figure 4-10. Illustrations of how undersampling and oversampling work. Source: Adapted from an image by Rafael Alencar[37]

A popular method of undersampling low-dimensional data that was developed back in 1976 is Tomek links.[38] 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 because the model doesn't get to learn from the subtleties of the true decision boundary.

A popular method of oversampling low-dimensional data is SMOTE (synthetic minority oversampling technique).[39] It synthesizes novel samples of the minority class through sampling convex combinations of existing data points within the minority class.[40]

Both SMOTE and Tomek links have only been proven effective in low-dimensional data. Many of the sophisticated resampling techniques, such as Near-Miss and one-sided selection,[41] 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 will 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.

One such technique is two-phase learning.[42] 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 fine-tune 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.,[43] 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 $\theta$. The model's loss is often defined as the average loss caused by all instances. $N$ denotes the total number of training samples.

$$L(X; \theta) = \sum_x \frac{1}{N} L(x; \theta)$$

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 incurs different costs, Elkan proposed cost-sensitive learning in which the individual loss function is modified to take into account this varying cost.[44] The method started by using a cost matrix to specify $C_{ij}$: the cost if 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 that in Table 4-8.

Table 4-8. Example of a cost matrix

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

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

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

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:

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

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.

$$L(x; \theta) = W_i \sum_j P(j|x; \theta) \text{Loss}(x, j)$$

A more sophisticated version of this loss can take into account the overlap among existing samples, such as class-balanced loss based on effective number of samples.[45]

**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 it still has 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.[46] The equation for focal loss and its performance compared to cross entropy loss is shown in Figure 4-11.

In practice, ensembles have shown to help with the class imbalance problem.[47] 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 6.
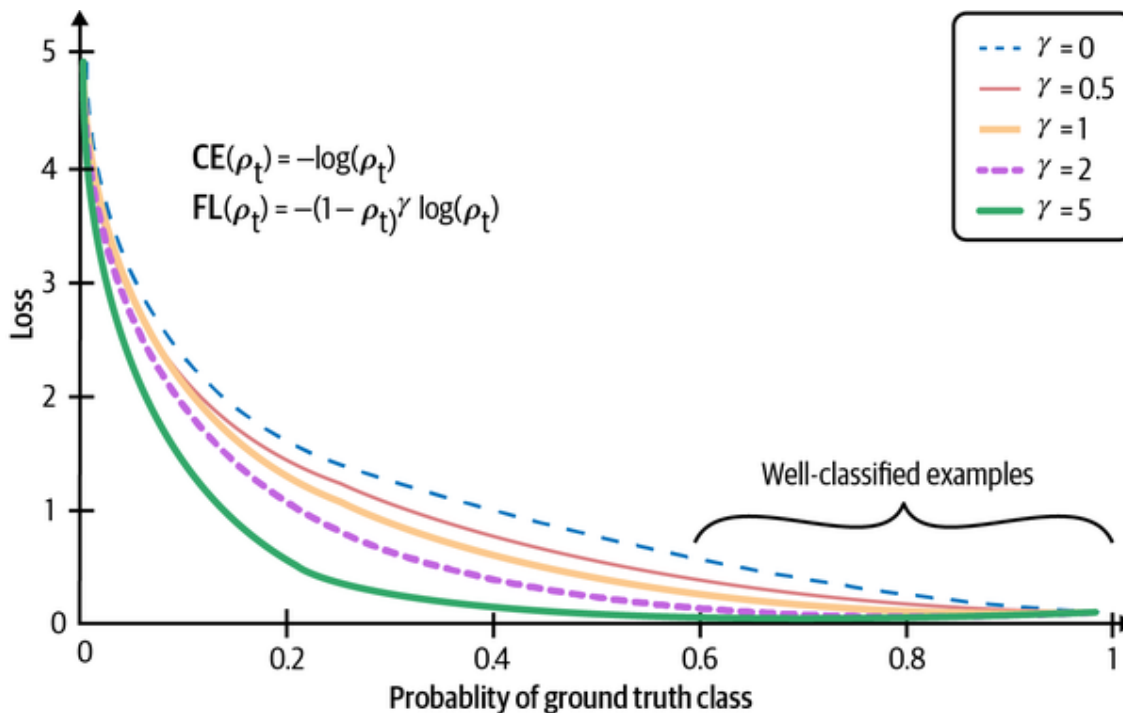
Figure 4-11. The model trained with focal loss (FL) shows reduced loss values compared to the model trained with cross entropy loss (CE). Source: Adapted from an image by Lin et al.

# Data Augmentation

Data augmentation is a family of techniques that are used to increase the amount of training data. Traditionally, these techniques are used for tasks that have limited training data, such as in medical imaging. However, in the last few years, they have shown to be useful even when we have a lot of data—augmented data can make our models more robust to noise and even adversarial attacks.

Data augmentation has become a standard step in many computer vision tasks and is finding its way into natural language processing (NLP) tasks. The techniques depend heavily on the data format, as image manipulation is different from text manipulation. In this section, we will cover three main types of data augmentation: simple label-preserving transformations; perturbation, which is a term for "adding noises"; and data synthesis. In each type, we'll go over examples for both computer vision and NLP.

## Simple Label-Preserving Transformations

In computer vision, the simplest data augmentation technique is to randomly modify an image while preserving its label. You can modify the image by cropping, flipping, rotating, inverting (horizontally or vertically), erasing part of the image, and more. This makes sense because a rotated image of a dog is still a dog. Common ML frameworks like PyTorch, TensorFlow, and Keras all have support for image augmentation. According to Krizhevsky et al. in their legendary AlexNet paper, "The transformed images are generated in Python code on the CPU while the GPU is training on the previous batch of images. So these data augmentation schemes are, in effect, computationally free."[48]

In NLP, you can randomly replace a word with a similar word, assuming that this replacement wouldn't change the meaning or the sentiment of the sentence, as shown in Table 4-9. Similar words can be found either with a dictionary of synonymous words or by finding words whose embeddings are close to each other in a word embedding space.

Table 4-9. Three sentences generated from an original sentence

| Original sentence | I'm so happy to see you. |
|---|---|
| Generated sentences | I'm so *glad* to see you. |
| | I'm so happy to see *y'all*. |
| | I'm *very* happy to see you. |

This type of data augmentation is a quick way to double or triple your training data.

## Perturbation

Perturbation is also a label-preserving operation, but because sometimes it's used to trick models into making wrong predictions, I thought it deserves its own section.

Neural networks, in general, are sensitive to noise. In the case of computer vision, this means that adding a small amount of noise to an image can cause a neural network to misclassify it. Su et al. showed that 67.97%

of the natural images in the Kaggle CIFAR-10 test dataset and 16.04% of the ImageNet test images can be misclassified by changing just one pixel (see Figure 4-12).[49]
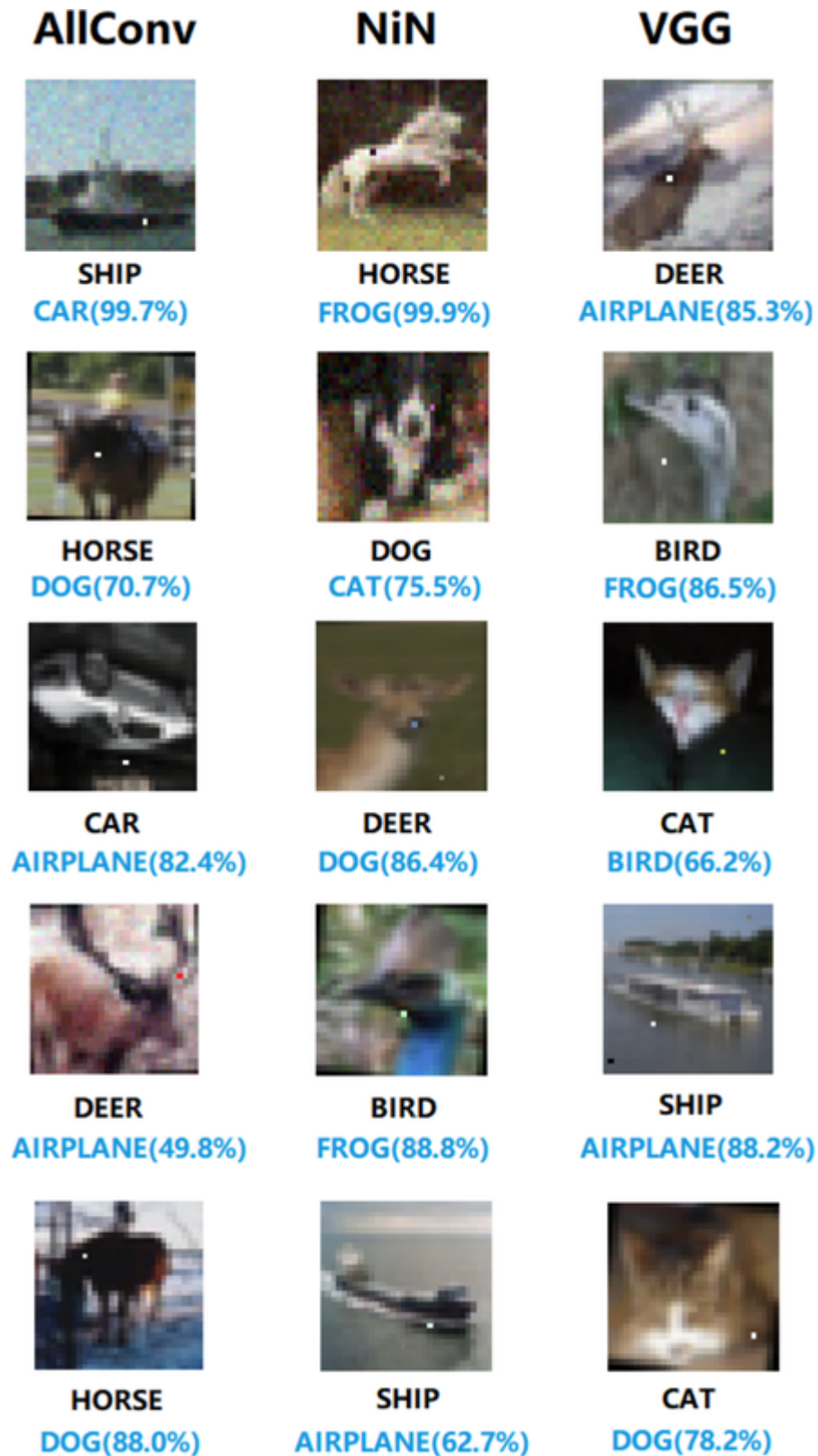


Figure 4-12. Changing one pixel can cause a neural network to make wrong predictions. The three models used are AllConv, NiN, and VGG. The original labels made by those models are above the labels made after one pixel was changed. Source: Su et al.[50]

Using deceptive data to trick a neural network into making wrong predictions is called adversarial attacks. Adding noise to samples is a common technique to create adversarial samples. The success of adversarial attacks is especially exaggerated as the resolution of images increases.

Adding noisy samples to training data can help models recognize the weak spots in their learned decision boundary and improve their performance.[51] Noisy samples can be created by either adding random noise or by a search strategy. Moosavi-Dezfooli et al. proposed an algorithm, called DeepFool, that finds the minimum possible noise injection needed to cause a misclassification with high confidence.[52] This type of augmentation is called adversarial augmentation.[53]

Adversarial augmentation is less common in NLP (an image of a bear with randomly added pixels still looks like a bear, but adding random characters to a random sentence will likely render it gibberish), but perturbation has been used to make models more robust. One of the most notable examples is BERT, where the model chooses 15% of all tokens in each sequence at random and chooses to replace 10% of the chosen tokens with random words. For example, given the sentence "My dog is hairy," and the model randomly replacing "hairy" with "apple," the sentence becomes "My dog is apple." So 1.5% of all tokens might result in nonsensical meaning. Their ablation studies show that a small fraction of random replacement gives their model a small performance boost.[54]

In Chapter 6, we'll go over how to use perturbation not just as a way to improve your model's performance, but also as a way to evaluate its performance.

## Data Synthesis

Since collecting data is expensive and slow, with many potential privacy concerns, it'd be a dream if we could sidestep it altogether and train our models with synthesized data. Even though we're still far from being able to synthesize all training data, it's possible to synthesize some training data to boost a model's performance.

In NLP, templates can be a cheap way to bootstrap your model. One team I worked with used templates to bootstrap training data for their conversational AI (chatbot). A template might look like: "Find me a [CUISINE] restaurant within [NUMBER] miles of [LOCATION]" (see Table 4-10). With lists of all possible cuisines, reasonable numbers (you would probably never want to search for restaurants beyond 1,000 miles), and locations (home, office, landmarks, exact addresses) for each city, you can generate thousands of training queries from a template.

Table 4-10. Three sentences generated from a template

| | |
|---|---|
| Template | Find me a [CUISINE] restaurant within [NUMBER] miles of [LOCATION]. |
| Generated queries | Find me a *Vietnamese* restaurant within *2* miles of *my office*.<br>Find me a *Thai* restaurant within *5* miles of *my home*.<br>Find me a *Mexican* restaurant within *3* miles of *Google headquarters*. |

In computer vision, a straightforward way to synthesize new data is to combine existing examples with discrete labels to generate continuous labels. Consider a task of classifying images with two possible labels: DOG (encoded as 0) and CAT (encoded as 1). From example $x_1$ of label DOG and example $x_2$ of label CAT, you can generate $x'$ such as:

$$x' = \gamma x_1 + (1 - \gamma)x_2$$

The label of $x'$ is a combination of the labels of $x_1$ and $x_2$: $\gamma \times 0 + (1 - \gamma) \times 1$. This method is called mixup. The authors showed that mixup improves models' generalization, reduces their memorization of corrupt labels, increases their robustness to adversarial examples, and stabilizes the training of generative adversarial networks.[55]

Using neural networks to synthesize training data is an exciting approach that is actively being researched but not yet popular in production. Sandfort et al. showed that by adding images generated using CycleGAN to their original training data, they were able to improve their model's

performance significantly on computed tomography (CT) segmentation tasks.[56]

If you're interested in learning more about data augmentation for computer vision, ["A Survey on Image Data Augmentation for Deep Learning"](#) (Shorten and Khoshgoftaar 2019) is a comprehensive review.

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

In this chapter, we've discussed the multiple steps to create training data. We first covered different sampling methods, both nonprobability sampling and random sampling, that can help us sample the right data for our problem.

Most ML algorithms in use today are supervised ML algorithms, so obtaining labels is an integral part of creating training data. Many tasks, such as delivery time estimation or recommender systems, have natural labels. Natural labels are usually delayed, and the time it takes from when a prediction is served until when the feedback on it is provided is the feedback loop length. Tasks with natural labels are fairly common in the industry, which might mean that companies prefer to start on tasks that have natural labels over tasks without natural labels.

For tasks that don't have natural labels, companies tend to rely on human annotators to annotate their data. However, hand labeling comes with many drawbacks. For example, hand labels can be expensive and slow. To combat the lack of hand labels, we discussed alternatives including weak supervision, semi-supervision, transfer learning, and active learning.

ML algorithms work well in situations when the data distribution is more balanced, and not so well when the classes are heavily imbalanced.

Unfortunately, problems with class imbalance are the norm in the real world. In the following section, we discussed why class imbalance made it hard for ML algorithms to learn. We also discussed different techniques to handle class imbalance, from choosing the right metrics to resampling data to modifying the loss function to encourage the model to pay attention to certain samples.

We ended the chapter with a discussion on data augmentation techniques that can be used to improve a model's performance and generalization for both computer vision and NLP tasks.

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.

---

**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** James J. Heckman, "Sample Selection Bias as a Specification Error," *Econometrica* 47, no. 1 (January 1979): 153–61, *https://oreil.ly/I5AhM*.

**3** Rachel Lerman, "Google Is Testing Its Self-Driving Car in Kirkland," *Seattle Times*, February 3, 2016, *https://oreil.ly/3IA1V*.

**4** Population here refers to a "statistical population", a (potentially infinite) set of all possible samples that can be sampled.

**5** Multilabel tasks are tasks where one example can have multiple labels.

**6** "SVM: Weighted Samples," scikit-learn, *https://oreil.ly/BDqbk*.

**7** Xiaojin Zhu, "Semi-Supervised Learning with Graphs" (doctoral diss., Carnegie Mellon University, 2005), *https://oreil.ly/VYy4C*.

**8** If something is so obvious to label, you wouldn't need domain expertise.

**9** We'll cover programmatic labels in the section "Weak supervision".

**10** Sofia Ira Ktena, Alykhan Tejani, Lucas Theis, Pranay Kumar Myana, Deepak Dilipkumar, Ferenc Huszar, Steven Yoo, and Wenzhe Shi, "Addressing Delayed Feedback for Continuous Training with Neural Networks in CTR Prediction," *arXiv*, July 15, 2019, *https://oreil.ly/5y2WA*.

**11** Alexander Ratner, Stephen H. Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré, "Snorkel: Rapid Training Data Creation with Weak Supervision," *Proceedings of the VLDB Endowment* 11, no. 3 (2017): 269–82, *https://oreil.ly/vFPjk*.

**12** Ratner et al., "Snorkel: Rapid Training Data Creation with Weak Supervision."

**13** Jared A. Dunnmon, Alexander J. Ratner, Khaled Saab, Matthew P. Lungren, Daniel L. Rubin, and Christopher Ré, "Cross-Modal Data Programming Enables Rapid Medical Machine Learning," *Patterns* 1, no. 2 (2020): 100019, *https://oreil.ly/nKt8E*.

**14** The two tasks in this study use only 18 and 20 LFs respectively. In practice, I've seen teams using hundreds of LFs for each task.

**15** Dummon et al., "Cross-Modal Data Programming."

**16** Avrim Blum and Tom Mitchell, "Combining Labeled and Unlabeled Data with Co-Training," in *Proceedings of the Eleventh Annual Conference on Computational Learning Theory* (July 1998): 92–100, *https://oreil.ly/T79AE*.

**17** Avital Oliver, Augustus Odena, Colin Raffel, Ekin D. Cubuk, and Ian J. Goodfellow, "Realistic Evaluation of Deep Semi-Supervised Learning Algorithms," *NeurIPS 2018 Proceedings*, *https://oreil.ly/dRmPV*.

**18** A token can be a word, a character, or part of a word.

**19** Jeremy Howard and Sebastian Ruder, "Universal Language Model Fine-tuning for Text Classification," *arXiv*, January 18, 2018, *https://oreil.ly/DBEbw*.

**20** Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig, "Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing," *arXiv*, July 28, 2021, *https://oreil.ly/0lBgn*.

**21** Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," *arXiv*, October 11, 2018, *https://oreil.ly/RdIGU*; Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, et

al., "Language Models Are Few-Shot Learners," OpenAI, 2020, https://oreil.ly/YVmrr.

22  Burr Settles, *Active Learning* (Williston, VT: Morgan & Claypool, 2012).

23  We'll cover ensembles in Chapter 6.

24  Dana Angluin, "Queries and Concept Learning," *Machine Learning* 2 (1988): 319–42, https://oreil.ly/0uKs4.

25  Thanks to Eugene Yan for this wonderful example!

26  Andrew Ng, "Bridging AI's Proof-of-Concept to Production Gap" (HAI Seminar, September 22, 2020), video, 1:02:07, https://oreil.ly/FSFWS.

27  And this is why accuracy is a bad metric for tasks with class imbalance, as we'll explore more in the section "Handling Class Imbalance".

28  I imagined that it'd be easier to learn ML theory if I didn't have to figure out how to deal with class imbalance.

29  The Nilson Report, "Payment Card Fraud Losses Reach $27.85 Billion," PR Newswire, November 21, 2019, https://oreil.ly/NM5zo.

30  "Job Market Expert Explains Why Only 2% of Job Seekers Get Interviewed," WebWire, January 7, 2014, https://oreil.ly/UpL8S.

31  "Email and Spam Data," Talos Intelligence, last accessed May 2021, https://oreil.ly/lI5Jr.

32  Nathalie Japkowciz and Shaju Stephen, "The Class Imbalance Problem: A Systematic Study," 2002, https://oreil.ly/d7lVu.

33  Nathalie Japkowicz, "The Class Imbalance Problem: Significance and Strategies," 2000, https://oreil.ly/Ma50Z.

34  Wan Ding, Dong-Yan Huang, Zhuo Chen, Xinguo Yu, and Weisi Lin, "Facial Action Recognition Using Very Deep Networks for Highly Imbalanced Class Distribution," *2017 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC)*, 2017, https://oreil.ly/WeW6J.

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

36  Jesse Davis and Mark Goadrich, "The Relationship Between Precision-Recall and ROC Curves," *Proceedings of the 23rd International Conference on Machine Learning*, 2006, *https://oreil.ly/s40F3*.

37  Rafael Alencar, "Resampling Strategies for Imbalanced Datasets," Kaggle, *https://oreil.ly/p8Whs*.

38  Ivan Tomek, "An Experiment with the Edited Nearest-Neighbor Rule," *IEEE Transactions on Systems, Man, and Cybernetics* (June 1976): 448–52, *https://oreil.ly/JCxHZ*.

39  N.V. Chawla, K.W. Bowyer, L.O. Hall, and W.P. Kegelmeyer, "SMOTE: Synthetic Minority Over-sampling Technique, *Journal of Artificial Intelligence Research* 16 (2002): 341–78, *https://oreil.ly/f6y46*.

40  "Convex" here approximately means "linear."

41  Jianping Zhang and Inderjeet Mani, "kNN Approach to Unbalanced Data Distributions: A Case Study involving Information Extraction" (Workshop on Learning from Imbalanced Datasets II, ICML, Washington, DC, 2003), *https://oreil.ly/qnpra*; Miroslav Kubat and Stan Matwin, "Addressing the Curse of Imbalanced Training Sets: One-Sided Selection," 2000, *https://oreil.ly/8pheJ*.

42  Hansang Lee, Minseok Park, and Junmo Kim, "Plankton Classification on Imbalanced Large Scale Database via Convolutional Neural Networks with Transfer Learning," *2016 IEEE International Conference on Image Processing (ICIP)*, 2016, *https://oreil.ly/YiA8p*.

43  Samira Pouyanfar, Yudong Tao, Anup Mohan, Haiman Tian, Ahmed S. Kaseb, Kent Gauen, Ryan Dailey, et al., "Dynamic Sampling in Convolutional Neural Networks for Imbalanced Data Classification," *2018 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*, 2018, *https://oreil.ly/D3Ak5*.

44  Charles Elkan, "The Foundations of Cost-Sensitive Learning," *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence* (IJCAI'01), 2001, *https://oreil.ly/WGq5M*.

**45**  Yin Cui, Menglin Jia, Tsung-Yi Lin, Yang Song, and Serge Belongie, "Class-Balanced Loss Based on Effective Number of Samples," *Proceedings of the Conference on Computer Vision and Pattern*, 2019, *https://oreil.ly/jCzGH*.

**46**  Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár, "Focal Loss for Dense Object Detection," *arXiv*, August 7, 2017, *https://oreil.ly/Km2dF*.

**47**  Mikel Galar, Alberto Fernandez, Edurne Barrenechea, Humberto Bustince, and Francisco Herrera, "A Review on Ensembles for the Class Imbalance Problem: Bagging-, Boosting-, and Hybrid-Based Approaches," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42, no. 4 (July 2012): 463–84, *https://oreil.ly/1ND4g*.

**48**  Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks, 2012, *https://oreil.ly/aphzA*.

**49**  Jiawei Su, Danilo Vasconcellos Vargas, and Sakurai Kouichi, "One Pixel Attack for Fooling Deep Neural Networks," *IEEE Transactions on Evolutionary Computation* 23, no. 5 (2019): 828–41, *https://oreil.ly/LzN9D*.

**50**  Su et al., "One Pixel Attack."

**51**  Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy, "Explaining and Harnessing Adversarial Examples," *arXiv*, March 20, 2015, *https://oreil.ly/9v2No*; Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio, "Maxout Networks, *arXiv*, February 18, 2013, *https://oreil.ly/L8mch*.

**52**  Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard, "DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks," in *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, *https://oreil.ly/dYVL8*.

**53**  Takeru Miyato, Shin-ichi Maeda, Masanori Koyama, and Shin Ishii, "Virtual Adversarial Training: A Regularization Method for Supervised and Semi-Supervised Learning," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2017, *https://oreil.ly/MBQeu*.

**54**  Devlin et al., "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding."

**55** Hongyi Zhang, Moustapha Cisse, Yann N. Dauphin, and David Lopez-Paz, *"mixup*: Beyond Empirical Risk Minimization," *ICLR 2018, https://oreil.ly/lIM5E*.

**56** Veit Sandfort, Ke Yan, Perry J. Pickhardt, and Ronald M. Summers, "Data Augmentation Using Generative Adversarial Networks (CycleGAN) to Improve Generalizability in CT Segmentation Tasks," *Scientific Reports* 9, no. 1 (2019): 16884, *https://oreil.ly/TDUwm*.

Support      Sign Out