# Designing Machine Learning Systems - Chip Huyen

## Chapter 4 – Training Data

* Chapter 3 = how to handle data from the systems perspective
* Chapter 4 = 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**, considered by many practitioners the “fun” part of the process
* Building a state-of-the-art model is interesting, while spending days wrangling with a massive amount of mal-formatted 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 MLE’s should learn how to **handle data well**, **saving us time and headache down the road**
* **Training data** = **encompasses all the data used in the developing phase of ML models, including the different splits used for training, validation, and testing**
* We start w/ different sampling techniques to select data for training + then address common challenges in creating training data, including the **label multiplicity problem**, the **lack of labels problem**, the **class imbalance** **problem**, + 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** while **data in production is neither finite nor stationary**
* 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
* **Data is full of potential biases** that 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 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

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

#### Non-Probability Sampling

* **Nonprobability sampling** = **selection of data isn’t based on any probability criteria**
* 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
* Ex: To scrape legitimate Twitter accounts without having access to Twitter databases, start with a small number of accounts, then scrape all the accounts they follow, + so on
* **Judgment sampling** = Experts decide what samples to include.
* **Quota sampling** = select samples based on quotas for certain slices of data without any randomization.
* Ex: When doing a survey, want 100 responses from each of the age groups: < 30 years, between 30 and 60 years, and > 60 years, *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**
* **Because of these biases, it’s a bad idea to select data to train ML models using this family of sampling methods**
* **Unfortunately, in many cases, the selection of data for ML models is still driven by convenience.**
* Ex1: *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)
* Ex2: Data for *sentiment analysis* of general text
* Much of this data is collected from sources with natural labels (ratings) such as IMDB reviews + Amazon reviews, + these datasets are then used for *other* sentiment analysis tasks
* **IMDB + 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**
* Ex3: Data for training self-driving cars
* Initially, data collected for self-driving cars came largely from 2 areas: Phoenix, Arizona (lax regulations), + the Bay Area, California (many self-driving car companies), + both areas have generally sunny weather
* In 2016, Waymo expanded its operations to Kirkland, Washington, 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.
* **Nonprobability sampling can be a quick + easy way to gather initial data to get a project off the ground**
* However, **for *reliable* models, you might want to use probability-based sampling**

#### Simple Random Sampling

* **Simplest form of random sampling** **= give** **all samples in the population equal probabilities of being selected**
* Ex: Randomly select 10% of the (statistical) population, giving all members an equal 10% chance of being selected
* Advantage = **easy to implement**
* Drawback = **rare categories of data might not appear in your selection**
* Ex: A class appears only in 0.01% of your data population
* If you randomly select 1% of this 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**
* Ex: To sample 1% of data that has 2 classes, A and B, sample 1% of class A and 1% of class B
* This way, no matter how rare class A or B is, you’ll ensure samples from them will be included in the selection
* **Each group is called a** **stratum**, and this **method is called** **stratified sampling**
* Drawback = it **isn’t always possible** (such as when it’s impossible to divide all samples into groups)
* This is **especially challenging when 1 sample might belong to multiple groups**, as in the case of **multilabel tasks** (*tasks where one example can have multiple labels*)
* 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**
* Ex: If you have 3 samples, A, B, + C, and want them to be selected w/ probabilities of 50%, 30%, and 20% respectively, give them the weights 0.5, 0.3, and 0.2
* **This method allows you to leverage domain expertise**
* Ex: **If you *know* that a certain subpopulation of data** (such as more recent data) **is more valuable to a model + want it to have a higher chance of being selected, 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**
* Ex: If you data has 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, give red samples weights 3X 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.*

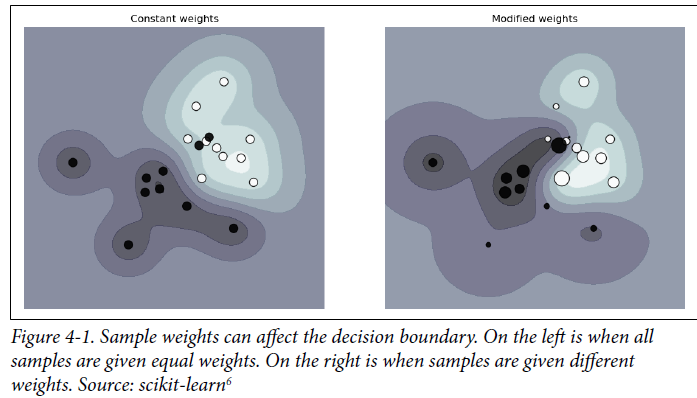
**import random**

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*

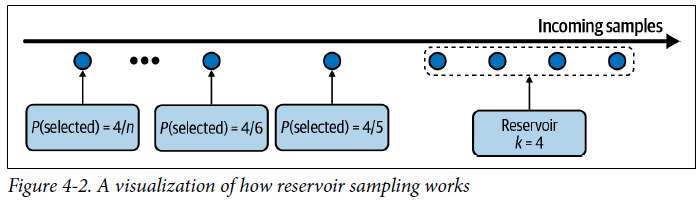
random.choices(population=[1, 1, 2, 2, 3, 3, 4, 4, 100, 1000], k=2)

* **A common concept in ML that is closely related to weighted sampling is sample weights**
* **Weighted sampling** = **used to *select* samples to train your model with**
* **Sample weights** = **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 seen below**



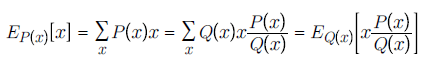
#### Reservoir Sampling

* **Reservoir sampling** = **algorithm** **especially useful when you have to deal with streaming data, which is usually what you have in production**.
* Ex: You have an incoming stream of tweets, and you want to sample a certain number, k, of tweets to do analysis/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 + the tweets are sampled w/ the correct probability
* One solution for this problem is **reservoir sampling**
* The algorithm involves a **reservoir** (which can be an array), and **consists of 3 steps**:
* **1. Put the first kelements into the reservoir**
* **2. For each incoming nth element, generate a random number isuch that 1 ≤ i ≤ n**
* **3. If 1 ≤ i ≤ k: replace the ith element in the reservoir with the nth element. Else, do nothing**
* This means that **each incoming nth element has k/n probability of being in the reservoir**
* You **can also prove that each element in the reservoir has 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**
* See an illustrative example of how reservoir sampling works:



#### Importance Sampling

* **Importance sampling** **= *1 of the most important sampling methods, not just in ML* 🡪** 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 DO have a distribution Q(x) that is a lot easier to sample from**
* So, **sample x from Q(x) instead and weigh this sample by 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) ≠ 0**
* The following equation shows that **in expectation, x sampled from P(x) == x sampled from Q(x) weighted by P(x)/Q(x)**



* One example where importance sampling is used in ML is **policy-based reinforcement learning**
* You want to **update your policy** 🡪 You want to **estimate the value functions of the *new* policy, but *calculating 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 + *reweight* them according to the new policy**
* The **rewards from the old policy make up the proposal distribution**

### Labeling

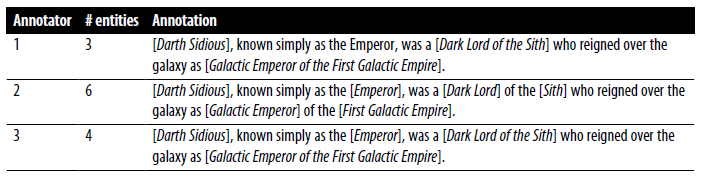
* Despite the promise of unsupervised ML, **most ML models in production today are supervised, which means that they need labeled data to learn from**, + ***performance of an ML model still depends heavily on the quality and quantity of the labeled data it’s trained o.***
* Data labeling has gone from being an auxiliary task to being a core function of many ML teams in production

#### Hand Labels

* **Acquiring hand labels** for your data is difficult for many, many reasons.
* 1) It 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.
* 2) It poses a **threat to data privacy**
* Hand labeling means someone has to look at your data, which isn’t always possible if your data has strict privacy requirements
* Ex: Can’t just ship patient 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.
* 3) It’s **slow**
* Ex: Accurate transcription of speech utterance at the phonetic level can take 400X longer than the utterance duration (want to annotate 1 hour of speech = take 400 hours/almost 3 months for a person to do so)
* **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**
* Ex: You have a sentiment analysis model to analyze the sentiment of every tweet that mentions your brand
* It has only 2 classes: NEGATIVE and POSITIVE
* However, after deployment, your PR team realizes the most damage comes from *angry* tweets + they want to attend to angry messages faster
* So, you have to update your sentiment analysis model to have 3 classes: NEGATIVE, POSITIVE, ANGRY
* To do so, you will need to look at your data *again* to see which existing training examples should be relabeled ANGRY
* Also, 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 + rely on multiple annotators who have different levels of expertise**
* These **different data sources and annotators also have different levels of accuracy** which leads to the problem of **label ambiguity** or **label multiplicity**: **what to do when there are multiple conflicting labels for a data instance.**
* Ex: Entity recognition: give 3 annotators the following sample + 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 3 different solutions where the 3 annotators have identified *different* entities.



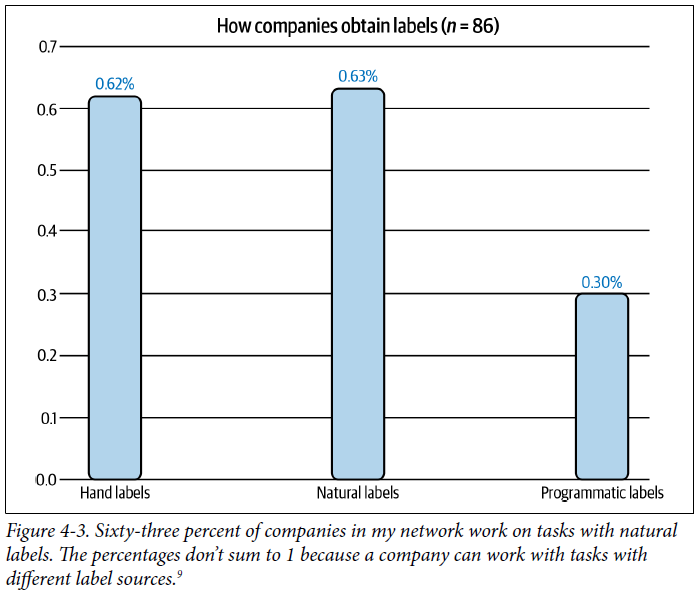
* *Which one should your model train on?* A model trained on data labeled by annotator 1 will perform very differently from a model trained on data labeled by annotator 2.
* **Disagreements among annotators are extremely common**, + the **higher the level of domain expertise required, the higher the potential for annotating disagreement** (*If something is so obvious to label, you wouldn’t need domain expertise*.)
* 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
* 1) First, **have a clear problem definition**
* Ex: Above, some disagreements could’ve been eliminated if we clarify that in case of multiple possible entities, pick the entity that comprises the longest substring
* i.e., “Galactic Emperor of the First Galactic Empire” instead of “Galactic Emperor” *and* “First Galactic Empire”
* 2) **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**
* Ex: You’ve trained a moderately good model with 100K data samples, + your MLE’s are confident that more data will improve model performance, so you spend a lot of money to hire annotators to label another million data samples
* However, model performance actually *decreases* after being trained on the new data because 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**
* **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**
* Ex: If your model fails mostly on recently acquired data samples, you might want to look into how the new data was acquired

#### Natural Labels

* Might be lucky enough to work on tasks with ***natural* ground truth labels** 🡪 **come from tasks where the model’s *predictions can be automatically or partially evaluated by the system***
* Ex: 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, Maps knows how long the trip actually took, + thus can evaluate the accuracy of the predicted time of arrival
* Ex: If your model predicts a stock’s price in the next 2 minutes, then after 2 minutes, you can compare the predicted price with the actual price
* **The canonical example of tasks with natural labels is recommender systems**, where the **goal** of the system is **to recommend to users items relevant to them**
* Whether a user clicks on a 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 (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 (label is NEGATIVE).
* **Many tasks can be framed as recommendation tasks**
* Ex: You can frame the task of predicting ads’ CTR’s as recommending the most relevant ads to users based on their activity histories and profiles
* ***Natural labels inferred from user behaviors* like clicks + 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**
* EX: If 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*)
* Ex: Newsfeed ranking is not a task with inherent labels, but by adding the “Like” button + other reactions to each newsfeed item, Facebook is able to collect feedback on their ranking algorithm
* **Tasks with natural labels are fairly common in industry**
* Doesn’t mean these tasks that can benefit from ML solutions actually have natural labels
* What is more likely is that **companies find it easier and cheaper to first start on tasks that have natural labels**



* Ex: A recommendation that *doesn’t* get clicked on after a period of time can be presumed to be “bad”, + this is 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

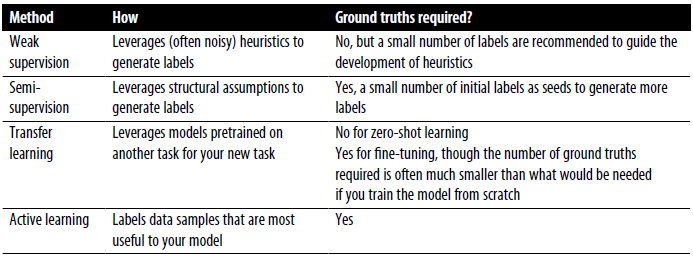
* **For tasks *with* natural ground truth labels**, the **time it takes from when a prediction is served until when the feedback on it is provided** is the **feedback loop length**
* **Tasks with *short* feedback loops are tasks where labels are generally available within minutes**
* Many recommender systems have short feedback loops
* If the recommended items are related products on Amazon/people to follow on Twitter, the time between when the item is recommended until it’s clicked on, if 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 Stitch Fix, you wouldn’t get feedback until users have received the items + tried them on, which could be weeks later
* **Choosing the right window length requires thorough consideration, as it involves the speed and accuracy trade-off**
* A short window length = 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 possible**
* 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**
* For tasks with long feedback loops, **natural labels might not arrive for weeks or even months**
* Ex: Fraud detection 🡪 For a certain period of time after a transaction, users can dispute whether that transaction is fraudulent or not
* A typical dispute window is 1 to 3 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 ASAP**
* *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*

###### Different Types of User Feedback

* If you want to extract labels from user feedback, it’s important to note that **there are different types of user feedback that can occur at different stages during a user journey** on your app and **differ by volume, strength of signal, and feedback loop length**
* Ex: An ecommerce application similar to Amazon
* 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,+ returning a previously bought product.
* **Clicking on a product happens much faster + 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***

#### 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



##### Weak supervision

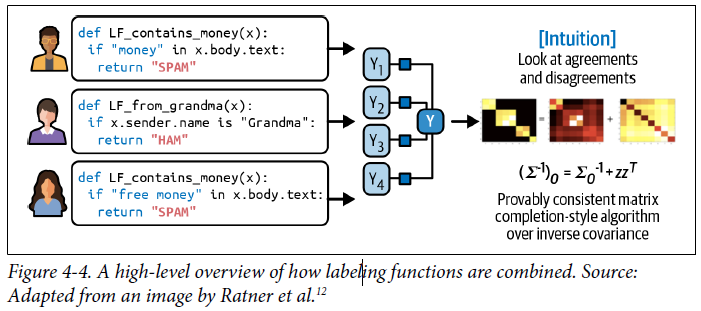
* If hand labeling is so problematic, what if we don’t use hand labels altogether?
* 1 approach that has gained popularity is **weak supervision** (like the open source tool Snorkel, developed at the Stanford AI Lab)
* The **insight** behind weak supervision is that **people rely on heuristics**, which **can be developed with subject matter expertise, to label data**
* Ex: 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 heuristic can be expressed by the following function:

**def** labeling\_function(note):

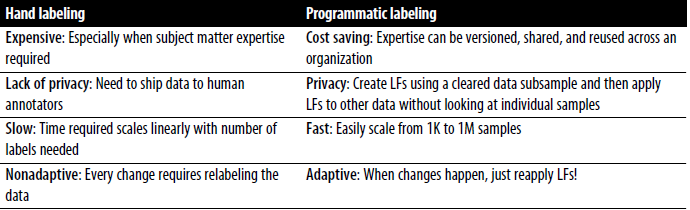
**if** "pneumonia" **in** note:

**return** "EMERGENT"

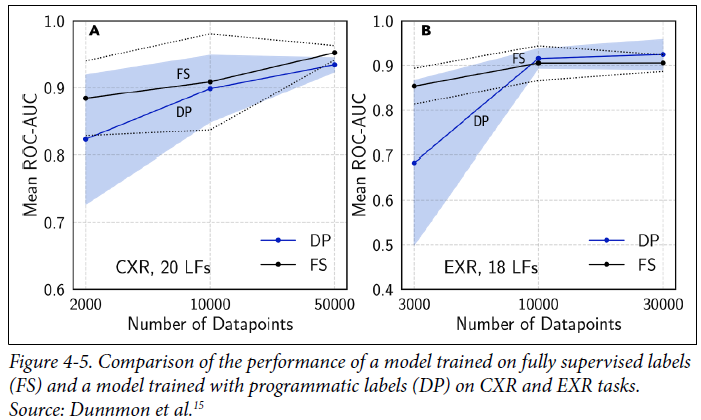
* **LFs can encode many different types of heuristics**
* **Keyword heuristic** (like above)
* **Regular expressions** (if the note matches or fails to match a certain REGEX)
* **Database lookup** (if the note contains the disease listed in the dangerous disease list)
* **The outputs of other models** (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**
* See at a high level how LFs work:



* 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** to **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 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**
* See some advantages of programmatic labeling over hand labeling:



* Ex: Case study to show how well weak supervision works in practice from a Stanford Medicine study
* Models trained with weakly supervised labels obtained by a single radiologist after 8 hours of writing LFs had comparable performance with models trained on data obtained through almost a year of hand labeling



* 2 interesting facts about the results of the experiment
* **1) The models continued improving with more unlabeled data even without more LFs**
* **2) LFs were being reused across tasks**
* The researchers were able to reuse 6 LFs between the CXR (chest X-rays) task and EXR (extremity X-rays) task
* *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

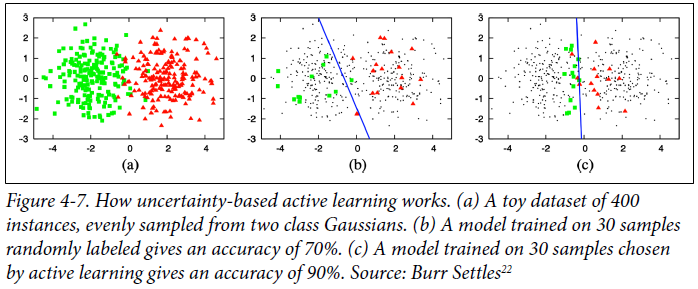
* If **weak supervision leverages heuristics** to obtain **noisy labels**, **semi-supervision** leverages **structural assumptions** to generate new labels **based on a small set of initial labels**
* Unlike weak supervision, **semi-supervision requires an initial set of labels**.
* Semi-supervised learning = technique used back in the 90s, + since then many semi-supervision methods have been developed
* 1) A classic semi-supervision method is **self-training**: start by **training a model on your existing set of labeled data and use this model to make predictions for unlabeled samples**
* **Assuming predictions with high raw probability scores are correct, add the labels predicted with high probability to your training set and train a *new* model on this *expanded* training set**
* **Continue until you’re happy with your model performance**
* 2) 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
* Ex: 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, you can also label the hashtags ”#ML” and “#BigData” as Computer Science.
* ***In most cases, the similarity can only be discovered by more complex methods***
* Ex: Might need to use a clustering method or a KNN algorithm to discover samples that belong to the same cluster
* 3) A semi-supervision method that has gained popularity in recent years is the **perturbation-based method** **based on the assumption that small perturbations to a sample *shouldn’t* change its label**
* So, **apply small perturbations to training instances to obtain *new* training instances**
* Perturbations might be applied *directly* to the samples (adding white noise to images) or to their *representations* (adding small random values to embeddings of words).
* **The perturbed samples have the *same* labels as the unperturbed samples**
* 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
* **Semi-supervision is the most useful when the number of training labels is limited**
* **1 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**
* 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 best model, then continuing training the champion model on the evaluation set**

##### Transfer learning

* **Transfer learning** = **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** (**usually a task that has cheap + abundant training data**)
* Language modeling is a great candidate because it doesn’t require labeled data
* They can be trained on any body of text (books, Wikipedia articles, chat histories), and the task is: “given a sequence of tokens, 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** (**downstream task**) such as sentiment analysis, intent detection, or question answering
* **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** 🡪 **making small changes to the base model, such as continuing to train it or a part of it on data from a given downstream task**
* **Sometimes**, you **might need to modify the inputs using a template to prompt the base**
* **model to generate the outputs you want**
* Ex: 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 *DO* 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, as 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, + text classification models that leverage pretrained language models such as BERT or GPT-3
* **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 5 years = (***usually***) **the larger the pretrained base model, the better its performance on downstream tasks**
* But **large models are expensive to train**
* Based on the configuration of GPT-3, it’s estimated cost of is in the 10’s 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** (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 a model is the least certain about, hoping they will help the model learn the decision boundary better**
* Ex: 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



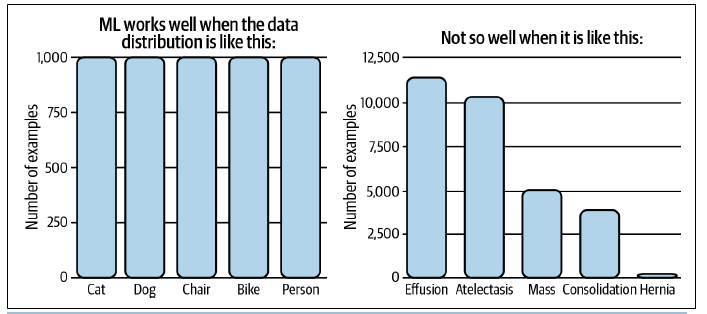
* **Another common heuristic** is based on **disagreement among *multiple* candidate models** = **query-by-committee**, + is an example of **an ensemble method**
* You **need a committee of several candidate models**, **usually the same model trained with different sets of hyperparameters or the same 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**
* The **samples to be labeled can come from different data regimes**, or they **can be synthesized (your model generates samples in the region of the input space that it’s most uncertain about**
* 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**
* Data changes all the time
* *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** = a problem in **classification tasks** where there is **a substantial difference in the number of samples in each class of the training data**
* Ex: Detecting lung cancer from X-ray images, 99.99% of training data 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***
* Ex: Estimating health-care bills, which are highly skewed (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 the Class Imbalance

* **ML, especially DL, works well in situations when the data distribution is more balanced**, and **usually not so well when the classes are heavily imbalanced**, as illustrated below:



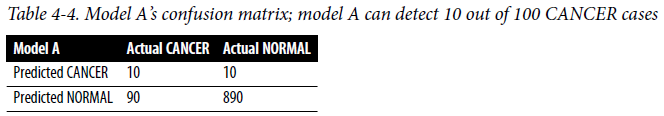
* 3 reasons class imbalance can make learning difficult
* **1)** It **often means there’s insufficient signal for a model to learn to detect the minority classes**
* In the case where there is a small number of instances in the minority class, **the problem becomes a few-shot learning problem** where 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**.
* 2) It **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**
* Ex: Lung cancer detection example 🡪 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**
* 2) It **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**
* Ex: 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***
* **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**
* Ex: Fraud detection 🡪 Most CC transactions are *not* fraudulent
* As of 2018, 6.8￠ for every $100 in cardholder spending is fraudulent
* Ex: Churn prediction 🡪 The majority of 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 = 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 = **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**
* Ex: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
* 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.**
* Annotators might have read the instructions wrong or followed the wrong instructions (thinking there’re only 2 classes, POSITIVE + NEGATIVE, while there are actually 3), 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**

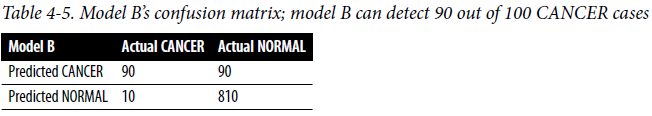
#### Handling Class Imbalance

* Because of its prevalence in real-world applications**, class imbalance has been thoroughly**
* **studied over the last two decades**
* **Class imbalance affects tasks differently based on the level of imbalance**
* **Some tasks are more sensitive to class imbalance than others**
* **Sensitivity to imbalance increases with the complexity of the problem, and noncomplex/linearly separable problems are unaffected by all levels of class imbalanc**e
* **Class imbalance in *binary* classification problems is a much easier problem than class imbalance in multiclass classification problems**
* Very deep NN’s (> 10 layers back in 2017) performed much better on imbalanced data than shallower NN’s
* There have been **many techniques suggested to mitigate the effect of class imbalance**.
* However**, as NN’s have grown to be much larger + much deeper, with more learning capacity, some argue you shouldn’t try to “fix” class imbalance if that’s how the data looks in the real world, and that 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**
* 3 approaches to handling class imbalance: **choosing the right metrics** for your problem, **data-level methods** (changing the data distribution to make it less imbalanced), and then **algorithm-level methods** (changing your learning method to make it more robust to class imbalance)

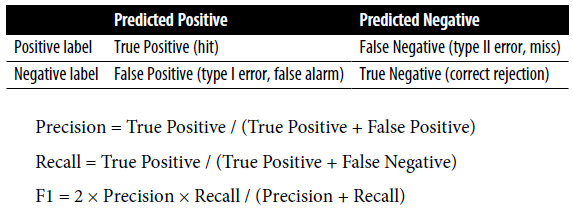
##### Using the right evaluation metrics

* 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.***
* Ex: Consider a task with 2 labels: CANCER (positive class) and NORMAL (negative class), where 90% of the labeled data is NORMAL
* Consider 2 models, A and B, with the confusion matrices shown in Tables 4-4 and 4-5

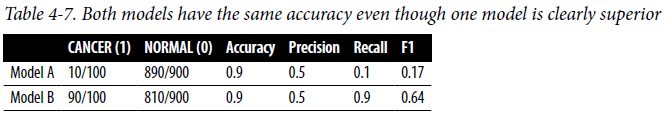




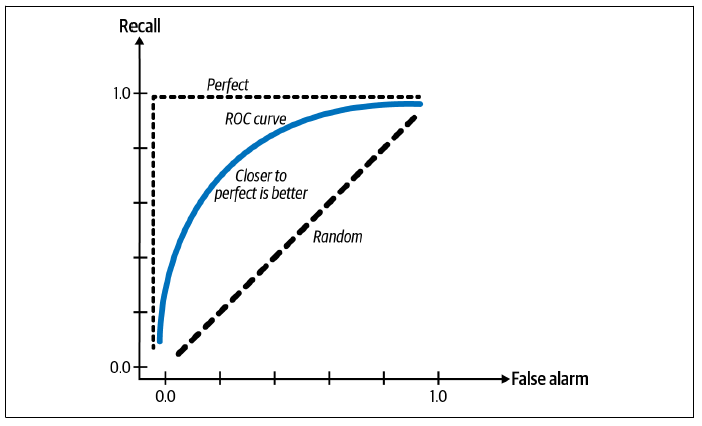
* If you’re like most people, you’d probably prefer model B to make predictions for you since it has a better chance of telling you if you actually have cancer
* *However, they both have the same accuracy of 0.9*
* **Metrics that help you understand your model’s performance *with respect to specific classes* would be better choices**
* **Accuracy *can* still be a good metric if you use it for each class *individually***
* Accuracy of model A on the CANCER class is 10%, 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 positives** (when t**he model correctly predicts the positive class**)
* Precision, recall, and F1 scores, for binary tasks, are **calculated using the count of true positives, true negatives, false positives, and false negatives**



* F1, precision, and recall are **asymmetric metrics** = ***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 below



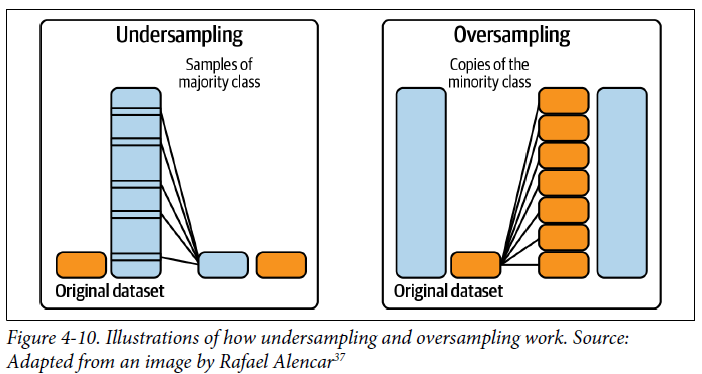
* **Many classification problems can be modeled as regression problems**
* Your **model can output a probability**, and **based on that probability,** **classify the sample**
* Ex: If the value is > 0.5, it’s a positive label, and if it’s <= 0.5, it’s a negative label
* This means you **can tune the threshold to increase the true positive rate(or recall) while decreasing the false positive rate(or the probability of false alarm), and vice versa**
* **Plot TPR against FPR for different thresholds** = **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**
* **ROC curve shows 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***
* 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** = argued this curve gives a more informative picture of an algorithm’s performance on tasks with heavy class imbalance

##### 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** = **oversampling** (**adding more instances from the minority classes** and **undersampling** (**removing instances of the majority classes**)
* The **simplest way to undersample = randomly remove instances from the majority class**
* The **simplest way to oversample = randomly make copies of the minority class until you have a ratio you’re happy with**
* Visualization of oversampling and undersampling:



* A **popular method of undersampling low-dimensional data** is **Tomek links**
* 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)**
* Synthesizes novel samples of the minority class through sampling convex combinations of existing data points within the minority class (“Convex” here approximately means “linear”)
* **Both SMOTE and Tomek links have only been proven effective in low-dimensional data**
* **Many sophisticated resampling techniques**, such as Near-Miss and one-sided selection, 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 NN’s**
* **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.
* **1) Two-phase learning**
* First **train your model on the resampled data** (which can be achieved by randomly undersampling large classes until each class has only Ninstances)
* Then **fine-tune your model on the *original* data**
* **2) Dynamic sampling**
* **Oversample the low-performing classes and undersample the high-performing classes during the training process**
* **Aims to show the model less of what it has already learned and more of what it has not**

##### Algorithm-level methods

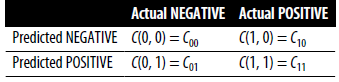
* If **data-level methods** mitigate the challenge of class imbalance by **altering the distribution of your training data**, **algorithm-level methods** **keep the training data distribution intact but alter the algorithm to make it more robust to class imbalance**
* Because the **loss/cost function guides the learning process**, **many algorithm-level methods involve adjustment to the loss function**
* **Key idea** = if there are **2 instances, x1 and x2,** and the **loss resulting from making the wrong prediction on x1 is higher than x2**, the **model will prioritize making the correct prediction on x1** over making the correct prediction on x2
* By **giving the training instances we care about higher** **weight**, we can **make the model focus more on learning these instances**
* **Let L(x, θ) = the loss caused by the instance xfor the model with the parameter set θ**
* The **model’s loss is often defined as the average loss caused by all instances** (N= the total number of training samples):



* **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**, 3 of them are **cost-sensitive learning**, **class-balances loss**, and **focal loss**
* *In practice, ensembles have shown to help with the class imbalance problem*
* However, we don’t include ensembling because **class imbalance isn’t usually why ensembles are used**

###### Cost-sensitive learning

* Based on the insight that **misclassification of different classes incurs different costs**, **cost-sensitive learning proposes the individual loss function be modified to take into account this varying cost**
* The method **started by using a cost matrix to specify Cij: the cost if class iis 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 2X as costly as the other way around, you can make C10 twice as high as C01*
* Ex: If you have 2 classes, POSITIVE and NEGATIVE, the cost matrix can look like this:



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



* 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

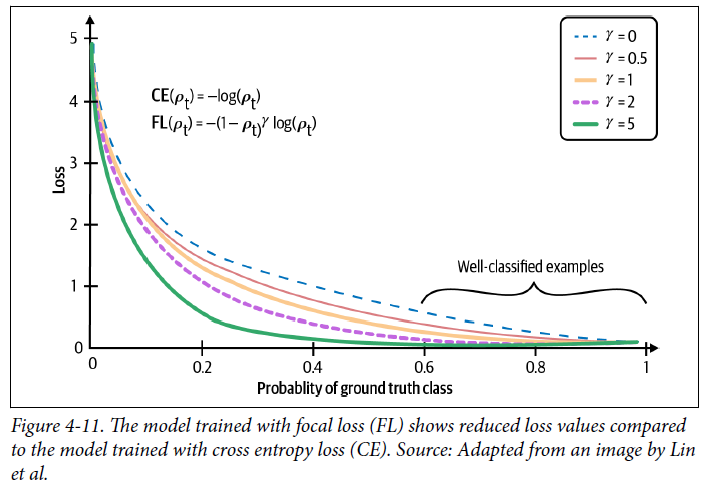
* *What might happen with a model trained on an imbalanced dataset is that it’ll bias toward majority classes and make wrong predictions on minority classes*
* What if we **punish a model for making wrong predictions on minority classes to correct this bias?**
* **Vanilla form** = **make the weight of each class inversely proportional to the number of samples in that class, so that the rarer classes have higher weights 🡪** Wi = (N / number of samples of class i)
* The **loss caused by instance x of class iwill 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)



* 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**

###### 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?** 🡪 s exactly what **focal loss** does
* The equation for focal loss and its performance compared to cross entropy loss is shown below

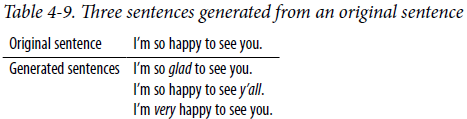


### Data Augmentation

* **Data augmentation** **is a family of techniques that are used to increase the amount of training data**
* **Traditionally**, they are **used for tasks that have limited training data**, such as in medical imaging.
* However, they have **recently shown to be useful even when we have a lot of data**
* **Augmented data can make models more robust to noise and even adversarial attacks**
* Data augmentation has become a **standard step in many CV tasks** and is **finding its way into NLP tasks**
* The **techniques depend heavily on the data format** (image manipulation is different from text manipulation)

#### Simple Label-Preserving Transformations

* **This type of data augmentation is a quick way to double or triple your training data**
* **In CV, the simplest data augmentation technique is to randomly modify an image while preserving its label** 🡪 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 (PyTorch, TensorFlow, Keras) all have support for image augmentation
* Krizhevsky et al.’s 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”
* **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 below



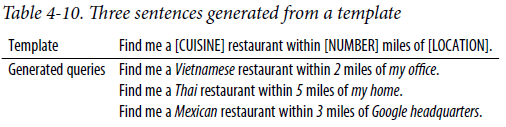
* Similar words can be found either with a dictionary of synonymous words or by finding words whose embeddings are close to each other in a word embedding space

#### Perturbation

* **Perturbation** is ***also* a label-preserving operation, but** because **sometimes it’s used to trick models into making wrong predictions**, it deserves its own section.
* **NN’s, in general, are sensitive to noise**
* In the case of CV, adding a small amount of noise to an image can cause a NN to misclassify it
* 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
* **Using deceptive data to trick a NN into making wrong predictions** is called **adversarial attacks**
* **Adding noise to samples is a common technique to create adversarial samples**
* *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**
* **Noisy samples can be created by either adding random noise or by a search strategy**
* Ex: DeepFool algorithm finds the minimum possible noise injection needed to cause a misclassification with high confidence
* **This type of augmentation is called adversarial augmentation**
* **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 make gibberish), but **perturbation has been used to make models more robust**
* Ex: BERT model chooses 15% of all tokens in each sequence at random and chooses to replace 10% of the chosen tokens with random words
* Ex: 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
* BERT ablation studies show that a small fraction of random replacement gives their model a small performance boost

#### 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**
* Ex: Templates to bootstrap training data for a conversational AI (chatbot) that might look like: ”Find me a [CUISINE] restaurant within [NUMBER] miles of [LOCATION]”



* *With lists of all possible cuisines, reasonable numbers (you’d probably never want to search for restaurants > 1,000 miles), and locations (home, office, landmarks, exact addresses) for each city, you can generate thousands of training queries from a template*
* **In CV, a straightforward way to synthesize new data is to combine existing examples with discrete labels to generate continuous labels**
* Ex: A task of classifying images with 2 possible labels: DOG (encoded 0) and CAT (encoded 1)
* From example x1 of label DOG + example x2 of label CAT, you can **generate x’ = γx1 + 1 − γx2**
* The **label x’ is a combination of the labels of x1 and x2: γ × 0 + 1 − γ × 1**
* This method is called **mix-up**, + it’s been shown that mix-up improves models’ generalization, reduces their memorization of corrupt labels, increases their robustness to adversarial examples, and stabilizes the training of generative adversarial networks
* **Using NN’s to synthesize training data is an exciting approach that is actively being researched *but not yet popular in production***

### 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, so **it’s worth it to invest time and effort to curate and create training data that will enable your algorithms to learn something meaningful**
* Different **sampling methods**, both **nonprobability sampling and random sampling**, 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**, which 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 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 data**
* Hand labeling comes with **many drawbacks** (expensive and slow)
* **To combat the lack of hand labels**, there are **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**
* **Class imbalance makes it hard for ML algorithms to learn**
* There’re **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
* **Data augmentation** techniques can be used to improve a model’s performance and generalization for both CV and NLP tasks
* **Once you have your training data, you will want to extract features from it to train ML models,**