O'REILLY®

Designing Machine Learning Systems

An Iterative Process for Production-Ready Applications

Early Release
RAW & UNEDITED

Chip Huyen

Designing Machine Learning Systems

An Iterative Process for Production-Ready Applications

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Chip Huyen

Designing Machine Learning Systems

by Chip Huyen

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Rebecca Novack

Development Editor: Jill Leonard

Production Editor: Kristen Brown

Copyeditor:

Proofreader:

Indexer:

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

- March 2022: First Edition

Revision History for the Early Release

- 2021-08-13: First Release
- 2021-10-13: Second Release

See http://oreilly.com/catalog/errata.csp?isbn=9781098107963 for release details.

978-1-098-10796-3

[LSI]

Chapter 1. Machine Learning Systems in Production

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or

examples in this book, or if you notice missing material within this chapter, please reach out to the author at *chip@huyenchip.com*.

In November 2016, Google announced that it had incorporated its multilingual neural machine translation system into Google Translate, marking one of the first success stories of deep neural artificial neural networks in production at scale[1]. According to Google, with this update, Google Translate's quality of translation improved more in a single leap than they had seen in the previous ten years combined.

Since then, more and more companies have turned towards machine learning (ML) for solutions to their most challenging problems. In just five years, ML has found its way into almost every aspect of our lives, from how we access information, how we communicate, how we work, to how we find love. The spread of ML has been so rapid that

it’s already hard to imagine life without it. Yet, there are still many more use cases for ML waiting to be explored: in healthcare, in transportation, in farming, even in helping us understand the universe[2].

Many people, when they hear “machine learning”, think of ML algorithms such as logistic regression or different types of neural networks. However, the algorithm is only a small part of an ML system in production. The system also includes the interface where users and developers interact with your system, the data stack to manage your data, the infrastructure to execute the required workloads, and the hardware backend your ML algorithm runs on. Figure 1.1 shows you the different components of an ML system.

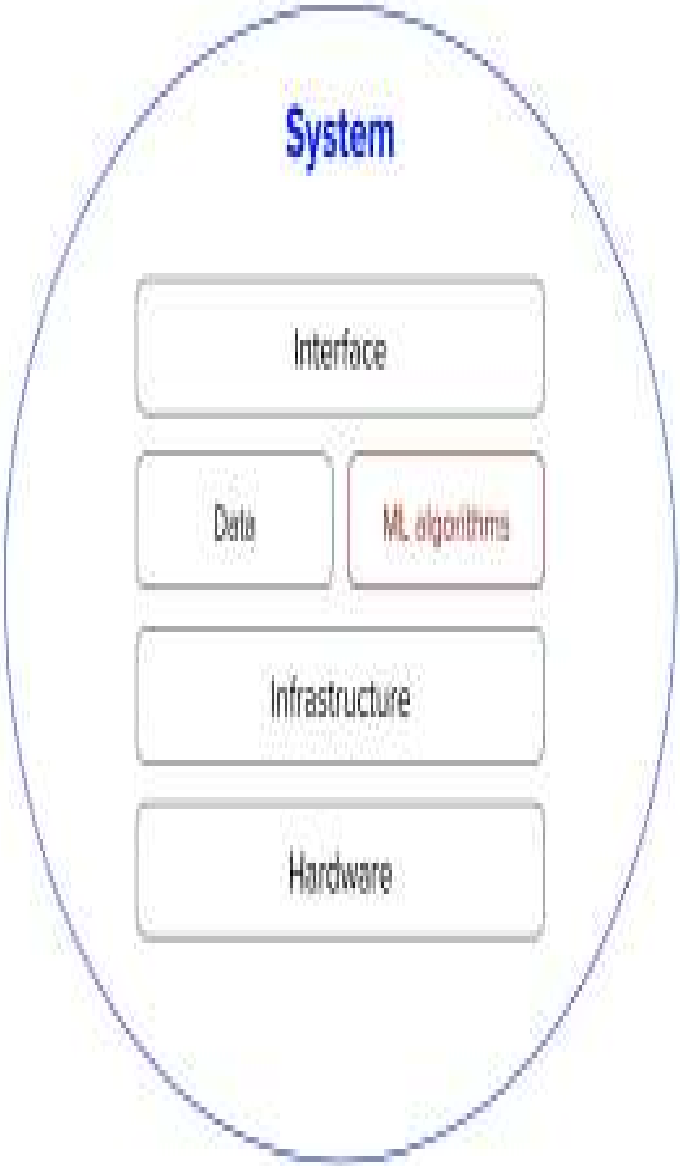

System
Interface
Data
ML algorithms
Infrastructure
Hardware

Figure 1-1. Different components of an ML system. "ML algorithms" is usually what people think of when they say machine learning, but it's only a small part of the entire system.

There are many excellent books that can give readers a deep understanding of various ML algorithms. This book doesn't aim to explain any specific algorithms in detail but to help readers understand the entire ML system as a whole. New algorithms are constantly being developed. This book hopes to provide you with a process to develop a solution that best works for your problem, regardless of which algorithm you might end up using. Chapter [TODO] includes a section that helps you evaluate which algorithm is best for your problem.

Because of the scale of many ML systems—they consume a massive amount of data, require heavy computational power, and have the potential to affect the lives of so many people—deploying them in production has many engineering and societal challenges. However, because of the speed at which these

applications are being deployed, these challenges are not always properly understood, let alone addressed. In the best case, the failure to address these challenges can lead to a few unhappy users. In the worst case, it can ruin people's lives and bankrupt companies.

This chapter aims to give you a high-level view of the challenges and requirements for deploying ML systems in production. However, before talking about how to develop ML systems, it's important to take a step back and ask a fundamental question: when and when not to use machine learning. We'll cover some of the popular use cases of ML to illustrate this point.

We will then move onto the challenges of deploying ML systems, and it'll do so by comparing ML in production to ML in research as well as to traditional software. It continues with an overview of ML systems design as well as the iterative process for designing an ML system that is deployable,

reliable, scalable, and adaptable.

If you've been in the trenches, you might already be familiar with what's written in this chapter. However, if you have only had experience with ML in an academic setting, this chapter will give an honest view of what it takes to deploy ML in the real world, and, hopefully, set your first application up for success.

When and When not to Use Machine Learning

As its adoption in the industry quickly grows, ML has proven to be a powerful tool for a wide range of problems. Despite an incredible amount of excitement and hype generated by people both inside and outside the field, machine learning (ML) is not a magic tool that can solve all problems. Even for problems that ML can solve, ML solutions might not be the optimal solutions.

Before starting an ML project, you might want to ask whether ML is necessary[3] or cost-effective.

When To Use Machine Learning

We expect that most readers are familiar with the basics of ML. However, to understand what ML can do, let's take a step back and understand what ML is:

Machine learning is an approach to (1) learn (2) complex (3) patterns from (4) existing data and use these patterns to make (5) predictions on (6) unseen data.

We'll look at each of the underlined keyphrases in the definition to understand its implications to the problems ML can solve.

Learn: the system has the capacity to learn

> A relational database isn't an ML system because it doesn't have the capacity to learn. You can explicitly state the relationship between two columns in a

relational database, but it's unlikely to have the capacity to figure out the relationship between these two columns by itself.

For an ML system to learn, there must be something for it to learn from. In most cases, ML systems learn from data. In supervised learning, based on examples of what inputs and outputs should look like, ML systems learn how to generate outputs for arbitrary inputs. For example, if you want to build an ML system to learn to predict the rental price for Airbnb listings, you need to provide a dataset where each input is a listing with all its characteristics (square footage, number of rooms, neighborhood, amenities, rating of that listing, etc.) and the associated output is the rental price of that listing. Once learned, this ML system can predict the price of a new listing given its characteristics.

Complex: the patterns are complex

Consider a website like Airbnb with a lot of house listings, each listing comes with a zip code. If you want to sort listings into the states they are located in, you wouldn't need an ML system. Since the pattern is simple—each zip code corresponds to a known state—you can just use a lookup table.

The relationship between a rental price and all its characteristics follows a much more complex pattern which would be very challenging to explicitly state by hand. ML is a good solution for this. Instead of telling your system how to calculate the price from a list of characteristics, you can provide prices and characteristics, and let your ML system figure out the pattern.

ML has been very successful with tasks with complex patterns such as object detection and speech recognition.

Algorithmic complexity is different from complexity in human perception. Many tasks that are hard for humans to do can be easy to express in algorithms, for example, raising a number of the power of 10. Vice versa, many tasks that are easy for humans can be hard to express in algorithms, e.g. deciding whether there's a cat in a picture.

Patterns: there are patterns to learn

ML solutions are only useful when there are patterns to learn. Sane people don't invest money into building an ML system to predict the next outcome of a fair die because there's no pattern in how these outcomes are generated[4].

However, there are patterns in how stocks are priced, and therefore companies have invested billions of dollars in building ML systems to learn those patterns.

Whether a pattern exists might not be obvious, or if patterns exist, your dataset

might not be sufficient to capture them. For example, there might be a pattern in how Elon Musk's tweets affect Bitcoin prices. However, you wouldn't know until you've rigorously trained and evaluated your ML models on his tweets. Even if all your models fail to make reasonable predictions of Bitcoin prices, it doesn't mean there's no pattern.

Existing data: data is available, or it's possible to collect data

Because ML learns from data, there must be data for it to learn from. It's amusing to think about building a model to predict how much tax a person should pay a year, but it's not possible unless you have access to tax and income data of a large population.

In the zero-shot learning (sometimes known as zero-data learning) context, it's possible for an ML system to make correct predictions for a task without

having been trained on data for that task. However, this ML system was previously trained on data for a related task. So even though the system doesn't require data for the task at hand to learn from, it still requires data to learn.

It's also possible to launch a ML system without data. For example, in the context of online learning, ML models can be deployed without having been trained on any data, but they will learn from data in production[5].

Without data and without online learning, many companies follow a 'fake-it-til-you make it' approach: launching a product that serves predictions made by humans, instead of ML algorithms, with the hope of using the generated data to train ML algorithms.

Predictions: it's a predictive problem

ML algorithms make predictions, so they can only solve problems that require

predictions. ML can be especially appealing when you can benefit from a large quantity of cheap but approximate predictions. In English, “predict” means “estimate a value in the future.” For example, what would the weather be like tomorrow? What would win the Super Bowl this year? What movie would a user want to watch next?

As predictive machines (e.g. ML models) are becoming more effective, more and more problems are being reframed as predictive problems. Whatever question you might have, you can always frame it as: “What would the answer to this question be?”, regardless of whether this question is about something in the future, the present, or even the past.

Compute-intensive problems are one class of problems that have been very successfully reframed as predictive. Instead of computing the exact outcome of a process, which might be even more

computationally costly and time-consuming than ML, you can frame the problem as: “What would the outcome of this process look like?” and approximate it using an ML algorithm. The output will be an approximation of the exact output, but often, it’s good enough. You can see a lot of it in graphic renderings, such as image denoising[6], screen-space shading[7].

Unseen data: Unseen data shares patterns with the training data

The patterns your model learns from existing data are only useful if unseen data also share these patterns. A model to predict whether an app will get downloaded on Christmas 2020 won’t perform very well if it’s trained on data from 2008 when the most popular app on the App Store was Koi Pond. What’s Koi Pond? Exactly.

In technical terms, it means your unseen data and training data should come from

similar distributions. You might ask: “If the data is unseen, how do we know what distribution it comes from?” We don’t, but we can make assumptions—such as we can assume that users’ behaviors tomorrow won’t be too different from users’ behaviors today—and hope that our assumptions hold. If they don’t, we’ll find out soon enough.

Due to the way most ML algorithms today learn, ML solutions will especially shine if your problem has these additional following characteristics.

It’s repetitive

Humans are great at few-shot learning: you can show kids a few pictures of cats and most of them will recognize a cat the next time they see one. Despite exciting progress in few-shots learning research, most ML algorithms still require many examples to learn a pattern. When a task is repetitive, each pattern is repeated

multiple times, which makes it easier for machines to learn it.

It’s at scale

ML solutions often require non-trivial upfront investment on data, compute, infrastructure, and talent, so it’d make sense if we can use these solutions a lot.

“At scale” means different things for different tasks, but it might mean making a lot of predictions. Examples include sorting through millions of mails a year or predicting which departments thousands of support tickets should be sent to a day.

A problem might appear to be a singular prediction but it’s actually a series of predictions. For example, a model that predicts who will win a US presidential election seems like it only makes one prediction every four years, but it might actually be making a prediction every hour or even less because that prediction has to be updated to new information over

time.

Having a problem at scale also means that there's a lot of data for you to collect, which is useful for training ML models.

The patterns are constantly changing

Cultures change. Tastes change. Technologies change. What's trendy today might be old news tomorrow. Consider the task of email spam classification. Today, an indication of a spam email is a Nigerian prince but tomorrow it might be a distraught Vietnamese writer.

If your problem involves one or more constantly changing patterns, solutions that don't allow you to learn from changing data might get you stuck in the past.

When not to Use Machine Learning

The list of use cases can go on and on, and it'll grow even longer as ML adoption matures in the industry. Even though ML can solve a subset of problems very well, it can't solve and/or shouldn't be used for a lot of problems. Most today's ML algorithms shouldn't be used under any of the following conditions.

1. It's unethical.
2. Simpler solutions do the trick. In chapter [TODO], we'll cover how to start with simple solutions first before trying out ML solutions.
3. One single prediction error can cause devastating consequences.
4. It's not cost-effective.

However, even if ML can't solve your problem, it might be possible to break your problem into smaller components and ML can solve some of them. For example, if you can't build a chatbot to answer all your

customers' queries, it might be possible to build an ML model to predict whether a query matches one of the frequently asked questions. If yes, automatically direct the customer to the answer. If not, direct them to customer service.

I'd also want to caution against dismissing a new technology because it's not as cost-effective as older technologies at the moment. Most technological advances are incremental. A type of technology might not be efficient now, but it might be in the future. If you wait for the technology to prove its worth to the rest of the industry before jumping in, you might be years or decades behind your competitors.

Machine Learning Use Cases

ML has found increasing usage in both enterprise and consumer applications. Since the mid-2010s, there has been an explosion of applications that leverage ML to deliver superior or previously impossible services to

the consumers.

With the explosion of information and services, it'd have been very challenging for us to find what we want without the help of ML, manifested in either a **search engine** or a **recommendation system**. When you visit a website like Amazon or Netflix, you're recommended items that are predicted to best match your taste. If you don't like any of your recommendations, you might want to search for specific items, and your search results are likely to be powered by ML.

If you have a smartphone, ML is likely already assisting you in many of your daily activities. Typing on your phone is made easier with **predictive typing**, an ML system that gives you suggestions on what you might want to say next. An ML system might run in your **photo editing** app to suggest how best to enhance your photos. You might **authenticate** your phone using your fingerprint or your face, which requires an ML system to predict whether a fingerprint or

a face matches yours.

The ML use case that drew me into the field was **machine translation**, automatically translating from one language to another. It has the potential to allow people from different cultures to communicate with each other, erasing the language barrier. My parents don't speak English, but thanks to Google Translate, now they can read my writing and talk to my friends who don't speak Vietnamese.

ML is increasingly present in our homes with smart **personal assistants** such as Alexa and Google Assistant. **Smart security cameras** can let you know when your pets leave home or if you have an uninvited guest. A friend of mine was worried about his aging mother living by herself -- if she falls, no one is there to help her get up -- so he relied on an **at-home health monitoring system** that predicts whether someone has fallen in the house.

Even though the market for consumer ML applications is booming, the majority of ML use cases are still in the enterprise world. Enterprise ML applications tend to have vastly different requirements and considerations from consumer applications. There are many exceptions, but for most cases, enterprise applications might have stricter accuracy requirements but be more forgiving with latency requirements. For example, improving a speech recognition system's accuracy from 95% to 95.5% might not be noticeable to most consumers, but improving a resource allocation system's efficiency by just 0.1% can help a corporation like Google or General Motors save millions of dollars. At the same time, latency of a second might get a consumer distracted and open something else, but enterprise users might be more tolerant of that. For people interested in building companies out of ML applications, consumer apps might be easier to distribute but much harder to make money out of. However, most enterprise use cases

aren’t obvious unless you’ve encountered them yourself.

According to Algorithmia’s 2020 state of enterprise machine learning survey, ML applications in enterprises are diverse, serving both internal use cases (reducing costs, generating customer insights and intelligence, internal processing automation) and external use cases (improving customer experience, retaining customers, interacting with customers).[8]

Figure 1-2. 2020 state of enterprise machine learning by Algorithmia.

Fraud detection is among the oldest applications of ML in the industry. If your

product or service involves transactions of any value, it'll be susceptible to fraud. By leveraging ML solutions for anomaly detection, you can have systems that learn from historical fraud transactions and predict whether a future transaction is fraudulent.

Deciding how much to charge for your product or service is probably one of the hardest business decisions, why not let ML do it for you? **Price optimization** is the process of estimating a price at a certain time period to maximize a defined objective function, such as the company's margin, revenue, or growth rate. ML-based pricing optimization is most suitable for cases with a large number of transactions where demand fluctuates and consumers are willing to pay a dynamic price e.g. Internet ads, flight tickets, accommodation bookings, ride-sharing, events.

To run a business, it's important to be able to **forecast customer demand** so that you can prepare a budget, stock inventory, allocate

resources, and update pricing strategy. For example, if you run a grocery store, you want to stock enough so that customers find what they're looking for, but you don't want to overstock, because if you do, your groceries might go bad and you lose money.

Acquiring a new user is expensive. As of 2019, the average cost for an app to acquire a user who'll make an in-app purchase is $86.61[9]. The acquisition cost for Lyft is estimated at $158/rider[10]. This cost is so much higher for enterprise customers. Customer acquisition cost is hailed by investors as a startup killer[11]. **Reducing customer acquisition costs** by a small amount can result in a large increase in profit. This can be done through better identifying potential customers, showing better-targeted ads, giving discounts at the right time, etc.—all of which are suitable tasks for ML.

After you've spent so much money acquiring a customer, it'd be a shame if they leave. **Churn prediction** is predicting when a

specific customer is about to stop using your products or services so that you can take appropriate actions to win them back. Churn prediction can be used not only for customers but also for employees.

To prevent customers from leaving, it's important to keep them happy by addressing their concerns as soon as they arise. **Automated support ticket classification** can help with that. Previously, when a customer opens a support ticket or sends an email, it needs to first be processed then passed around to different departments until it arrives at the inbox of someone who can address it. An ML system can analyze the ticket content and predict where it should go, which can shorten the response time and improve customer satisfaction. It can also be used to classify internal IT tickets.

Another popular use case of ML in enterprise is **brand monitoring**. The brand is a valuable asset of a business[12]. It's important to monitor how the public and how your

customers perceive your brand. You might want to know when/where/how it's mentioned, both explicitly (e.g. when someone mentions "Google") or implicitly (e.g. when someone says "the search giant") as well as the sentiment associated with it. If there's suddenly a surge of negative sentiment in your brand mentions, you might want to do something about it as soon as possible. Sentiment analysis is a typical ML task.

A set of ML use cases that has generated much excitement recently is in health care. There are ML systems that can **detect skin cancer** and **diagnose diabetes**. Healthcare prediction systems are technically geared towards consumers, but because of their strict requirements with accuracy and privacy, they might be provided through a healthcare provider such as a hospital or used to assist doctors in providing diagnosis.

Understanding Machine Learning Systems

Before learning how to design machine learning systems, we'll go over how ML systems are different from both ML in research (or as often taught in school) and traditional software, which motivates the need for this book.

Machine learning in research vs. in production

As ML usage in the industry is still fairly new, most people with ML expertise have gained it through academia: taking courses, doing research, reading academic papers. If that describes your background, it might be a steep learning curve for you to understand the challenges of deploying ML systems in the wild and navigate an overwhelming set of solutions to these challenges. ML in production is very different from ML in research. Table 1-1 shows five of the major

differences.

Table 1-1. Key differences between ML in research and ML in production.

	Research	Production
Objectives	Model performance	Different stakeholders have different objective
Computational priority	Fast training, high throughput	Fast inference, lo latency
Data	Static[a]	Constantly shiftin
Fairness	Good to have (sadly)	Important
Interpretability	Good to have	Important

a A subfield of research focuses on continual learning: developing models to work with changing data distributions. We'll cover continual learning in Chapte 7.

Stakeholders and their objectives

Research and leaderboard projects often have one single objective. The most common objective is model performance—develop a

model that achieves the state-of-the-art (SOTA) results on benchmark datasets. To edge out a small improvement in performance, researchers often resort to techniques that make models too complex to be useful.

There are many stakeholders involved in bringing an ML system into production. Each stakeholder has their own objective. Consider a project that recommends restaurants to users. The project involves ML engineers, salespeople, product managers, infrastructure engineers, and a manager.

- The **ML engineers** want a model that recommends restaurants that users will most likely order from, and they believe they can do so by using a more complex model with more data.
- The sales team wants a model that recommends restaurants that pay the highest advertising fee to be shown

in-app, since ads bring in more revenue than just service fees.

- The product team notices that every drop in latency leads to drop in orders through the service, so they want a model that can do inference faster than the model that the ML engineers are working on.
- As the traffic grows, the infrastructure team has been woken up in the middle of the night because of problems with scaling their existing system, so they want to hold off the production line so they could update the infrastructure.
- The manager wants to maximize the margin, and one way to achieve it is to let go of the ML team[13].

These objectives require different models, yet the stakeholders will have to collaborate to somehow create a model that will satisfy all

of them.

Production having different objectives from research is one of the reasons why successful research projects might not always be used in production. Ensembling is a technique popular among the winners of many ML competitions, including the famed $1M Netflix Prize. It combines "*multiple learning algorithms to obtain better predictive performance than could be obtained from any of the constituent learning algorithms alone*.[14]" While it can give you a small improvement, ensembled systems risk being too complex to be useful, e.g. more error-prone to deploy, slower to serve, or harder to interpret.

For many tasks, a small improvement in performance can result in a huge boost in revenue or cost save. For example, a 0.2% improvement in the click-through-rate for a product recommendation system can result in millions of dollars increase in revenue for an ecommerce site. However, for many tasks, a

small improvement might not be noticeable for users. From a user's point of view, a speech recognition app with a 95% accuracy is not that different from an app with a 95.2% accuracy. For the second type of tasks, if a simple model can do a reasonable job, complex models must perform significantly better to justify the complexity.

In recent years, there have been many critics of ML leaderboards, both research leaderboards such as GLUE and competitions such as Kaggle.

An obvious argument is that in these competitions, many hard steps needed for building ML systems are already done for you[15].

A less obvious argument is that due to the multiple-hypothesis testing scenario that happens when you have multiple teams testing on the same hold-out test set, a model can do better than the rest just by

chance[16].

The misalignment of interests between research and production has been noticed by researchers. In an EMNLP 2020 paper, Ethayarajh and Jurafsky argued that benchmarks have helped drive advances in NLP by incentivizing the creation of more accurate models at the expense of other qualities valued by practitioners such as compactness, fairness, and energy efficiency[17].

Computational priority

When designing an ML system, people who haven't deployed an ML system often make the mistake of focusing entirely on the model development part.

During the model development process, you train different iterations of your model multiple times. The trained model then runs inference on the test set once to report the score. This means training is the bottleneck.

Once the model has been deployed, however, its job is to do inference, so inference is the bottleneck. Most research prioritizes fast training whereas most production prioritizes fast inference**.**

Latency vs. throughput

One corollary of this is that research prioritizes high throughput whereas production prioritizes low latency. In case you need a refresh, latency refers to the time it takes from receiving a query to returning the result. Throughput refers to how many queries are processed within a specific period of time.

For example, the average latency of Google Translate is the average time it takes from when a user clicks Translate to when the translation is shown, and the throughput is how many queries it processes and serves a second.

If your system always processes one query at a time, higher latency means lower

throughput. If the average latency is 10ms, which means it takes 10ms to process a query, the throughput is 100 queries/second. If the average latency is 100ms, the throughput is 10 queries/second.

However, most modern distributed systems batch queries to process them together, often concurrently, higher latency might also mean higher throughput. If you process 10 queries at a time and it takes 10ms to run a batch, the average latency is still 10ms but the throughput is now 10 times higher—1000 queries/second. If you process 100 queries at a time and it takes 50ms to run a batch, the average latency now is 50ms and the throughput is 2000 queries/second. Both latency and throughput have increased!

This is further complicated if you want to batch online queries. Batching requires your system to wait for enough queries to arrive in a batch before processing them, which further increases latency.

In research, you care more about how many samples you can process in a second (throughput) and less about how long it takes for each sample to be processed (latency). You're willing to increase latency to increase throughput, e.g. with aggressive batching.

However, once you deploy your model into the real world, latency matters a lot. In 2009, Google's experiments demonstrated that increasing web search latency 100 to 400 ms reduces the daily number of searches per user by 0.2% to 0.6%[18]. In 2019, Booking.com found that an increase of about 30% in latency cost about 0.5% in conversion rates —"*a relevant cost for our business*."[19]

Reducing latency might reduce the number of queries you can process on the same hardware at a time. If your hardware is capable of processing much more than one sample at a time, using it to process only one sample means making processing one sample more expensive.

Data

During the research phase, the datasets you work with are often clean and well-formatted, freeing you to focus on developing and training models. They are static by nature so that the community can use them to benchmark new architectures and techniques. This means that many people might have used and discussed the same datasets, and quirks of the dataset are known. You might even find open-source scripts to process and feed the data directly into your models.

In production, data, if available, is a lot more messy. It's noisy, possibly unstructured, constantly shifting. It's likely biased, and you likely don't know how it's biased. Annotated labels, if there are any, are sparse, imbalanced, outdated, or incorrect. Changing project or business requirements might require adding another label class or merging two existing label classes. This can happen even after a model has been trained and deployed. If you work with users' data, you'll

also have to worry about privacy and regulatory concerns.

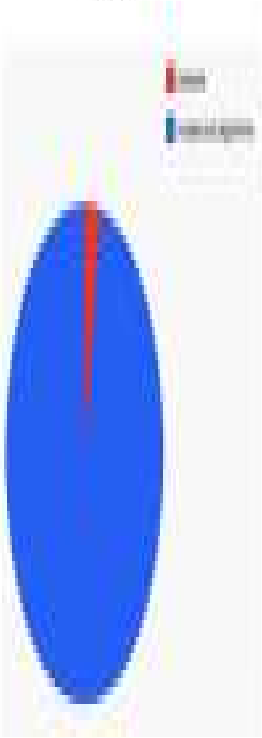
In research, since you don’t serve your models to users, you mostly work with historical data, e.g. data that already exists and is stored somewhere. In production, most likely you’ll also have to work with data that is being constantly generated by users, systems, and third-party data.

Figure 1-3 is a great graphic by Andrej Karpathy, head of AI at Tesla, that illustrates the data problems he encountered during his PhD compared to his time at Tesla.

Research	Production
• Clean • Static • Mostly historical data	• Messy • Constantly shifting • Historical + streaming data • Privacy + regulatory concerns

Amount of lost sleep over...

PhD



Tesla

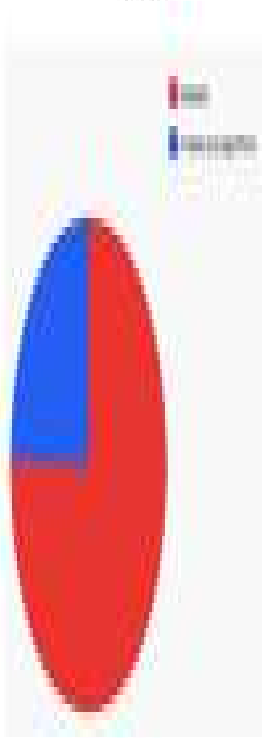


Figure 1-3. Data in research vs. data in production by Andrej Karpathy[20]

Fairness

During the research phase, a model is not yet used on people, so it's easy for researchers to put off fairness as an afterthought: "Let's try to get state-of-the-art first and worry about fairness when we get to production." When it gets to production, it's too late. On top of that, as of 2021, fairness isn't yet a metric for researchers to optimize on. If you optimize your models for better accuracy or lower latency, you can show that your models beat state-of-the-art. But there's no equivalent state-of-the-art for fairness metrics.

You or someone in your life might already be a victim of biased mathematical algorithms without knowing it. Your loan application might be rejected because the ML algorithm picks on your zip code, which embodies biases about one's socio-economic background. Your resume might be ranked

lower because the ranking system employers use picks on the spelling of your name. Your mortgage might get a higher interest rate because it relies partially on credit scores, which reward the rich and punish the poor. Other examples of ML biases in the real world are in predictive policing algorithms, personality tests administered by potential employers, college ranking. For even more galling examples, I recommend Cathy O'Neil's Weapon of Math Destruction[21].

ML algorithms don't predict the future, but encode the past, perpetuating the biases in the data and more. When ML algorithms are deployed at scale, they can discriminate against people at scale. If a human operator might only make sweeping judgments about a few individuals at a time, an ML algorithm can make sweeping judgments about millions in split seconds. This can especially hurt members of minority groups because misclassification on them has minor effects on models' overall performance metrics.

If an algorithm can already make correct predictions on 98% of the population, and improving the predictions on the other 2% would incur multiples of cost, some companies might, unfortunately, choose not to do it. During a McKinsey & Company research in 2019, only 13% of the large companies surveyed said they are taking steps to mitigate risks to equity and fairness, such as algorithmic bias and discrimination[22].

Interpretability

In early 2020, the Turing Award winner Professor Geoffrey Hinton proposed a heatedly debated question about the importance of interpretability in ML systems.

> *"Suppose you have cancer and you have to choose between a black box AI surgeon that cannot explain how it works but has a 90% cure rate and a human surgeon with an 80% cure rate. Do you want the AI surgeon to be illegal?"*[23]

A couple of weeks later, when I asked this

question to a group of 30 technology executives at public non-tech companies, only half of them would want the highly effective but unable to explain AI surgeon to operate on them. The other half wanted the human surgeon.

While most of us are comfortable with using a microwave without understanding how it works, many don't feel the same way about AI yet, especially if that AI makes important decisions about their lives.

Since most ML research is still evaluated on a single objective, model performance, researchers aren't incentivized to work on model interpretability. However, interpretability isn't just optional for most ML use cases in the industry, but a requirement.

First, interpretability is important for users, both business leaders and end-users, to understand why a decision is made so that they can trust a model and detect potential

biases mentioned above. Second, it's important for developers to debug and improve a model.

Just because interpretability is a requirement doesn't mean everyone is doing it. As of 2019, only 19% of large companies are working to improve the explainability of their algorithms[24].

Discussion

Some might argue that it's okay to know only the academic side of ML because there are plenty of jobs in research. The first part — it's okay to know only the academic side of ML — is true. The second part is false.

While it's important to pursue pure research, most companies can't afford it unless it leads to short-term business applications. This is especially true now that the research community took the "bigger, better" approach. Oftentimes, new models require a massive amount of data and tens of millions of dollars in compute alone.

As ML research and off-the-shelf models become more accessible, more people and organizations would want to find applications for them, which increases the demand for ML in production.

The vast majority of ML-related jobs will be, and already are, in productionizing ML.

Machine learning systems vs. traditional software

Since ML is part of software engineering (SWE), and software has been successfully used in production for more than half a century, some might wonder why we don't just take tried-and-true best practices in software engineering and apply them to ML.

That's an excellent idea. In fact, ML production would be a much better place if ML experts were better software engineers. Many traditional SWE tools can be used to develop and deploy ML applications.

However, many challenges are unique to ML

applications and require their own tools. In SWE, there's an underlying assumption that code and data are separated. In fact, in SWE, we want to keep things as modular and separate as possible (see Separation of concerns).

On the contrary, ML systems are part code, part data, and part artifacts created from the two. The trend in the last decade shows that applications developed with the most/best data win. Instead of focusing on improving ML algorithms, most companies will focus on improving their data. Because data can change quickly, ML applications need to be adaptive to the changing environment which might require faster development and deployment cycles.

In traditional SWE, you only need to focus on testing and versioning your code. With ML, we have to test and version our data too, and that's the hard part. How to version large datasets? How to know if a data sample is good or bad for your system? Not all data

samples are equal -- some are more valuable to your model than others. For example, if your model has already trained on 1M scans of normal lungs and only 1000 scans of cancerous lungs, a scan of a cancerous lung is much more valuable than a scan of a normal lung. Indiscriminately accepting all available data might hurt your model's performance and even make it susceptible to data poisoning attacks (see Figure 1-4).

Figure 1-4. An example of how a face recognition system can be poisoned, using malicious data, to allow unauthorized people to pose as someone else. Targeted Backdoor Attacks on Deep Learning Systems Using Data Poisoning (Chen et al., 2017)

The size of ML models gives another

challenge. As of 2020, it's common for ML models to have hundreds of millions, if not billions, of parameters, which requires GBs of RAM to load them into memory. A few years from now, a billion parameters might seem quaint—like *can you believe the computer that sent men to the moon only had 32MB of RAM*?

However, for now, getting these large models into production, especially on edge devices[25], is a massive engineering challenge. Then there is the question of how to get these models to run fast enough to be useful. An autocompletion model is useless if the time it takes to suggest the next character is longer than the time it takes for you to type.

Monitoring and debugging these models in production is also non-trivial. As ML models get more complex, coupled with the lack of visibility into their work, it's hard to figure out what went wrong or be alerted quickly enough when things go wrong.

The good news is that these engineering challenges are being tackled at a breakneck pace. Back in 2018, when the BERT (Bidirectional Encoder Representations from Transformers) paper first came out, people were talking about how BERT was too big, too complex, and too slow to be practical. The pretrained large BERT model has 340M parameters and is 1.35GB[26]. Fast forward two years later, BERT and its variants were already used in almost every English search on Google[27].

Designing ML Systems in Production

Now that we've discussed what it takes to develop and deploy an ML system, let's get to the fun part of actually designing one. This section aims to give you an overview of machine learning systems design. It starts by explaining what machine learning systems design is and covers the requirements for ML

systems. We will then go over the iterative process for designing systems to meet those requirements.

ML systems design is the process of defining all the components of an ML system, including **interface, algorithms, data**, **infrastructure**, and **hardware**, so that the system satisfies **specified requirements**.

Requirements for ML Systems

Before building a system, it’s essential to defy requirements for that system. Requirements vary from use case to use case. However, most systems should have these four characteristics: reliable, scalable, maintainable, and adaptable.

We’ll walk through each of these concepts in detail. Let’s take a closer look at reliability first.

Reliability

The system should continue to perform the

correct function at the **desired level of performance** even in the face of **adversity** (hardware or software faults, and even human error).

“Correctness” might be difficult to determine for ML systems. For example, your system might call the function “.predict()” correctly, but the predictions are wrong. How do we know if a prediction is wrong if we don’t have ground truth labels to compare it with?

With traditional software systems, you often get a warning, such as a system crash or runtime error or 404. However, ML systems fail silently. End users don’t even know that the system has failed and might have kept on using it as if it was working.

Scalability

As the system grows (in data volume, traffic volume, or complexity), there should be reasonable ways of dealing with that growth.

Scaling isn’t just up-scaling[28] — expanding

the resources to handle growth. In ML, it's also important to down-scale — reducing the resources when not needed. For example, at peak, your system might require 100 GPUs. However, most of the time, your system needs only 10 GPUs. Keeping 100 GPUs up all the time can be costly, so your system should be able to scale down to 10 GPUs.

An indispensable feature in many cloud services is autoscaling: automatically scaling up and down the number of machines depending on usage. This feature can be tricky to implement. Even Amazon fell victim to this when their autoscaling feature failed on Prime Day, causing their system to crash. An hour downtime was estimated to cost it between $72 million and $99 million[29].

Maintainability

There are many people who will work on an ML system. They are ML engineers, DevOps engineers, and subject matter experts (SMEs).

They might come from very different backgrounds, with very different languages and tools, and might own different parts of the process. It's important to structure your project and set up your infrastructure in a way such that different contributors can work using tools that they are comfortable with, instead of one group of contributors forcing their tools onto other groups. When a problem occurs, different contributors should be able to work together to identify the problem and implement a solution without finger-pointing. We'll go more into this in chapter 7.

Adaptability

To adapt to changing data distributions and business requirements, the system should have some capacity for both discovering aspects for performance improvement and allowing updates without service interruption.

Because ML systems are part code, part data, and data can change quickly, ML systems

need to be able to evolve quickly. This is tightly linked to maintainability. We'll go more into this in chapter 7.

Iterative Process

Developing an ML system is an iterative and, in most cases, never ending process[30]. You do reach the point where you have to put the system into production, but then that system will constantly need to be monitored and updated.

Before deploying my first ML system, I thought the process would be linear and straightforward. I thought all I had to do was to collect data, train a model, deploy that model, and be done. However, I soon realized that the process looks more like a cycle with a lot of back and forth between different steps.

For example, here is one workflow that you might encounter when building an ML model to predict whether an ad should be shown

when users enter a search query[31].

1. Choose a metric to optimize. For example, you might want to optimize for impressions -- the number of times an ad is shown.
2. Collect data and obtain labels.
3. Engineer features.
4. Train models.
5. During error analysis, you realize that errors are caused by wrong labels, so you relabel data.
6. Train model again.
7. During error analysis, you realize that your model always predicts that an ad shouldn’t be shown, and the reason is because 99.99% of the data you have is no-show (an ad shouldn’t be shown for most queries). So you have to collect more data of ads that should be shown.

8. Train model again.
9. Model performs well on your existing test data, which is by now two months ago. But it performs poorly on the test data from yesterday. Your model has degraded, so you need to collect more recent data.
10. Train model again.
11. Deploy model.
12. Model seems to be performing well but then the business people come knocking on your door asking why the revenue is decreasing. It turns out the ads are being shown but few people click on them. So you want to change your model to optimize for clickthrough rate instead.
13. Go to step 1.

Figure 1-5 is an oversimplified representation

of what the iterative process for developing ML systems in production looks like.

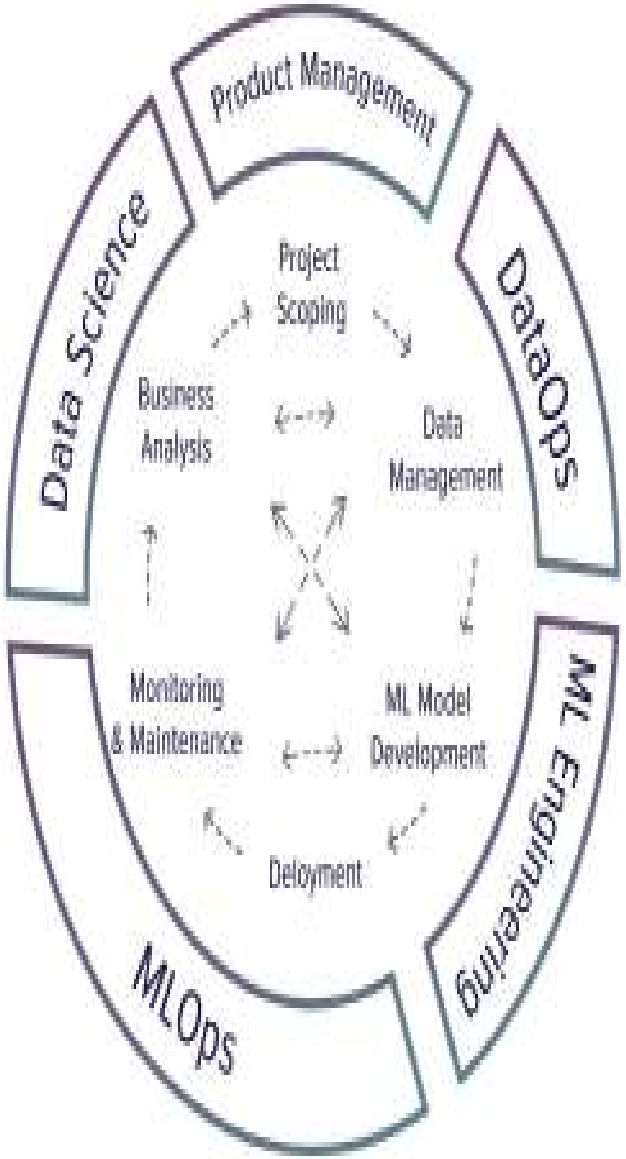

Product Management
DataOps
ML Engineering
MLOps
Data Science
Project Scoping
Data Management
ML Model Development
Deployment
Monitoring & Maintenance
Business Analysis

Figure 1-5. The process of developing an ML system looks more like a cycle with a lot of back and forth between steps.

While we'll take a deeper dive into what each of these steps mean in practice in later chapters, let's take a brief look at what happens during each of the steps.

Step 1. Project scoping

A project starts with scoping the project, laying out goals & objectives, constraints, and evaluation criteria. Stakeholders should be identified and involved. Resources should be estimated and allocated.

Step 2. Data engineering

Data used and generated by ML systems can be large and diverse, which requires scalable infrastructure to process and access it fast and reliably. Data engineering covers data sources, data formats, data processing, and data manipulation to create training data.

Step 3. ML model development

From raw data, you need to create training datasets and possibly label them, then generate features, train models, optimize models, and evaluate them. This is the stage that requires the most ML knowledge and is most often covered in ML courses.

Step 4. Deployment

After a model is developed, it needs to be made accessible to users. Developing an ML system is like writing—you will never reach the point when your system is done. But you do reach the point when you have to put your system out there.

Step 5. Monitoring and continual learning

Once in production, models need to be monitored for performance decay and maintained to be adaptive to changing environments and changing requirements.

Step 6. Business analysis

Model performance needs to be evaluated against business goals and analyzed to generate business insights. These insights can then be used to eliminate unproductive projects or scope out new projects.

Summary

I hope this chapter has given you a glimpse into what it takes to bring an ML system into production, how they differ from ML projects in a research setting, as well as how they differ from traditional software engineering systems.

It’s ambitious because, as we’ve covered in this chapter, ML systems are complex, consisting of many different components and involving many different stakeholders. They can be deployed to solve a wide range of tasks, both for consumers and enterprises.

Each task also comes with its own challenges and requirements. The effort is further complicated by the fact that as ML adoption matured, tools and best practices for ML systems will also evolve.

It's impossible to cover every aspect of ML systems in production, but I hope that this chapter has covered what I believe to be most applicable to ML systems in a wide range of tasks. I hope that this chapter can help mitigate surprises and help you to become better prepared when evaluating and planning the use of ML in your projects. If you believe that there's something I've missed, please let me know.

Fortunately, complex ML systems are made up of simpler building blocks. Now that we've covered the high-level overview of an ML system in production, we'll zoom into its building blocks in the following chapters, starting with the fundamentals of data engineering in the next chapter. If any of the challenges mentioned in this chapter seems

abstract to you, I hope that specific examples in the following chapters will make them more concrete.

---

1 Zero-Shot Translation with Google's Multilingual Neural Machine Translation System (Schuster et al., Google AI Blog 2016)

2 A method to image black holes (MIT News 2019)

3 I didn't ask whether ML is sufficient because the answer is always no.

4 Patterns are different from distributions. We know the distribution of the outcomes of a fair die, but there are no patterns in the way the outcomes are generated.

5 We'll go over online learning in Chapter 7.

6 Kernel-predicting convolutional networks for denoising Monte Carlo renderings (Bako et al., ACM Transactions on Graphics 2017)

7 Deep Shading: Convolutional Neural Networks for Screen-Space Shading (Nalbach et al., 2016)

8 2020 state of enterprise machine learning (Algorithmia, 2020)

9 Average mobile app user acquisition costs worldwide from September 2018 to August 2019, by user action and operating system (Statista, 2019)

10 Valuing Lyft Requires A Deep Look Into Unit

Economics (Forbes, 2019)

11 Startup Killer: the Cost of Customer Acquisition (David Skok, 2018)

12 Apple, Google, Microsoft, Amazon each has a brand estimated to be worth in the order of hundreds of millions dollars (Forbes, 2020)

13 It's common for the ML and data science teams to be among the first to go during a company's mass layoff. See IBM, Uber, Airbnb, and this analysis on How Data Scientists Are Also Susceptible To The Layoffs Amid Crisis (AIM, 2020).

14 Ensemble learning (Wikipedia)

15 Machine learning isn't Kaggle competitions (Julia Evans, 2014)

16 AI competitions don't produce useful models (Luke Oakden-Rayner, 2019)

17 Utility is in the Eye of the User: A Critique of NLP Leaderboards (Ethayarajh and Jurafsky, EMNLP 2020)

18 Speed Matters for Google Web Search (Jake Brutlag, Google 2009)

19 150 Successful Machine Learning Models: 6 Lessons Learned at Booking.com (Bernardi et al., KDD 2019)

20 Building the Software 2.0 Stack (Andrei Karpathy, Spark+AI Summit 2018)

21 Weapon of Math Destruction (Cathy O'Neil, Crown Books 2016)

22 AI Index 2019 (Stanford HAI, 2019)

23 https://twitter.com/geoffreyhinton/status/12305922384

24 AI Index 2019 (Stanford HAI, 2019)

25 We'll cover edge devices in Chapter 6. Deployment.

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

27 Google SearchOn 2020.

28 Up-scaling and down-scaling are two aspects of "scaling out", which is different from "scaling up". Scaling out is adding more equivalently functional components in parallel to spread out a load. Scaling up is making a component larger or faster to handle a greater load.

29 Wolfe, Sean. 2018. "Amazon's one hour of downtime on Prime Day may have cost it up to $100 million in lost sales." Business Insider. https://www.businessinsider.com/amazon-prime-day-website-issues-cost-it-millions-in-lost-sales-2018-7.

30 Which, as an early reviewer pointed out, is a property of traditional software.

31 Praying and crying not featured but present through the entire process.

Chapter 2. Data Engineering Fundamentals

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or

examples in this book, or if you notice missing material within this chapter, please reach out to the author at *chip@huyenchip.com*.

The rise of machine learning in recent years is tightly coupled with the rise of big data. Big data systems, even without machine learning, are complex. If you haven't spent years and years working with them, it's easy to get lost in acronyms. There are many challenges and possible solutions that these systems generate. Industry standards, if there are any, evolve quickly as new tools come out and the needs of the industry expand, creating a dynamic and ever-changing environment. If you look into the data stack for different tech companies, it might seem like each is doing its own thing.

In this chapter, we'll cover the basics of data engineering that will, hopefully, give you a steady piece of land to stand on as you explore the landscape for your own needs. It

will start with the question: how important data is for building intelligent systems? It will then cover the basics of data engineering. Knowing how to collect, handle, and process an increasingly growing amount of data is essential to people who want to build ML systems in production. If you're already familiar with data engineering fundamentals, you might want to move directly to Chapter 3 to learn more about how to sample and generate labels to create training data.

Mind vs. Data

Progress in the last decade shows that the success of an ML system depends largely on the data it was trained on. Instead of focusing on improving ML algorithms, most companies focus on managing and improving their data.

Despite the success of models using massive amounts of data, many are skeptical of the emphasis on data as the way forward. In the

last three years, at every academic conference I attended, there were always some debates among famous academics on the power of mind vs. data. *Mind* might be disguised as inductive biases or intelligent architectural designs. *Data* might be grouped together with computation since more data tends to require more computation.

In theory, you can both pursue intelligent design and leverage large data and computation, but spending time on one often takes time away from another.

On the mind over data camp, there's Dr. Judea Pearl, a Turing Award winner best known for his work on causal inference and Bayesian networks. The introduction to his book, "The book of why", is entitled "Mind over data," in which he emphasizes: "*Data is profoundly dumb*." In one of his more controversial posts on Twitter, he expressed his strong opinion against ML approaches that rely heavily on data and warned that data-centric ML people might be out of job in

3-5 years.

> *“ML will not be the same in 3-5 years, and ML folks who continue to follow the current data-centric paradigm will find themselves outdated, if not jobless. Take note.”*[1]

There’s also a milder opinion from Professor Christopher Manning, Director of the Stanford Artificial Intelligence Laboratory, who argued that huge computation and a massive amount of data with a simple learning algorithm create incredibly bad learners. The structure allows us to design systems that can learn more from fewer data[2].

Many people in ML today are on the data over mind camp. Professor Richard Sutton, a professor of computing science at the University of Alberta and a distinguished research scientist at DeepMind, wrote a great blog post in which he claimed that researchers who chose to pursue intelligent designs over methods that leverage

computation will eventually learn a bitter lesson.

> *"The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin. … Seeking an improvement that makes a difference in the shorter term, researchers seek to leverage their human knowledge of the domain, but the only thing that matters in the long run is the leveraging of computation."*[3]

When asked how Google search was doing so well, Peter Norvig, Google's Director of Search, emphasized the importance of having a large amount of data over intelligent algorithms in their success: "*We don't have better algorithms. We just have more data.*"[4]

Dr. Monica Rogati, Former VP of Data at Jawbone, argued that data lies at the foundation of data science. If you want to use data science, a discipline of which machine learning is a part of, to improve your products

or processes, you need to start with building out your data, both in terms of quality and quantity. Without data, there’s no data science.

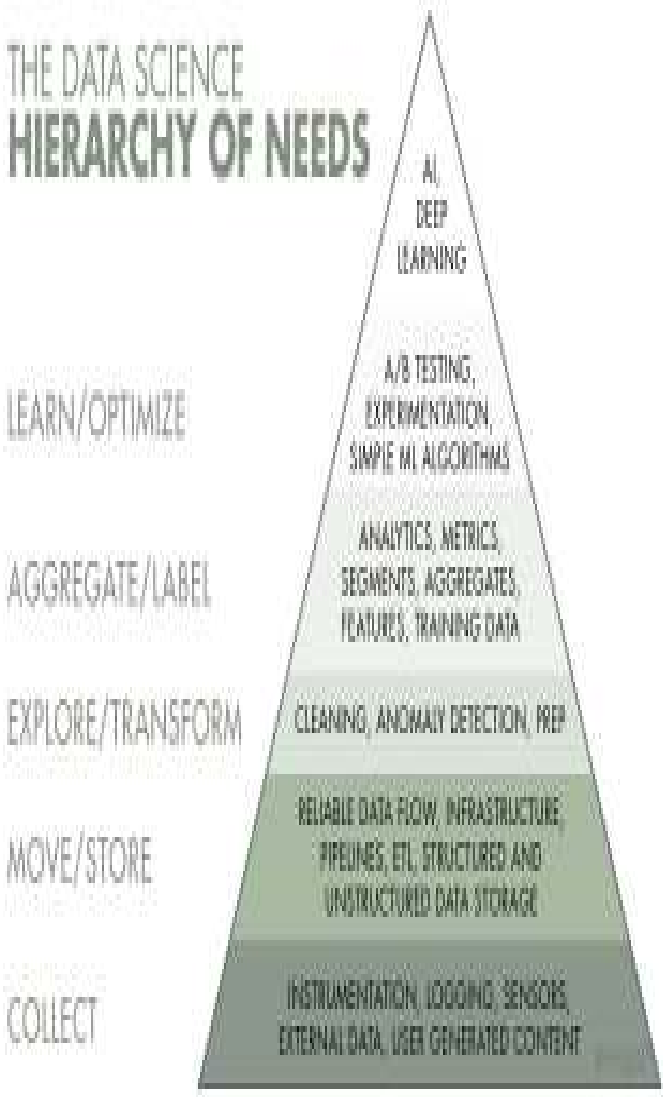

THE DATA SCIENCE
HIERARCHY OF NEEDS
AI,
DEEP
LEARNING
A/B TESTING,
EXPERIMENTATION,
SIMPLE ML ALGORITHMS
ANALYTICS, METRICS,
SEGMENTS, AGGREGATES,
FEATURES, TRAINING DATA
CLEANING, ANOMALY DETECTION, PREP
RELIABLE DATA FLOW, INFRASTRUCTURE,
PIPELINES, ETL, STRUCTURED AND
UNSTRUCTURED DATA STORAGE
INSTRUMENTATION, LOGGING, SENSORS,
EXTERNAL DATA, USER GENERATED CONTENT
LEARN/OPTIMIZE
AGGREGATE/LABEL
EXPLORE/TRANSFORM
MOVE/STORE
COLLECT

Figure 2-1. The data science hierarchy of needs (Monica Rogati, 2017[5])

The debate isn't about whether *finite* data is necessary, but whether it's sufficient. The term *finite* here is important, because if we had infinite data, we can just look up the answer. Having a lot of data is different from having infinite data.

Regardless of which camp will prove to be right eventually, no one can deny that data is essential, for now. Both the research and industry trends in the recent decades show the success of machine learning relies more and more on the quality and quantity of data. Models are getting bigger and using more data. Back in 2013, people were getting excited when the One Billion Words Benchmark for Language Modeling was released, which contains 0.8 billion tokens[6]. Six years later, OpenAI's GPT-2 used a dataset of 10 billion tokens. And another year later, GPT-3 used 500 billion tokens. The growth rate of the sizes of datasets is shown

in Figure 2-2.

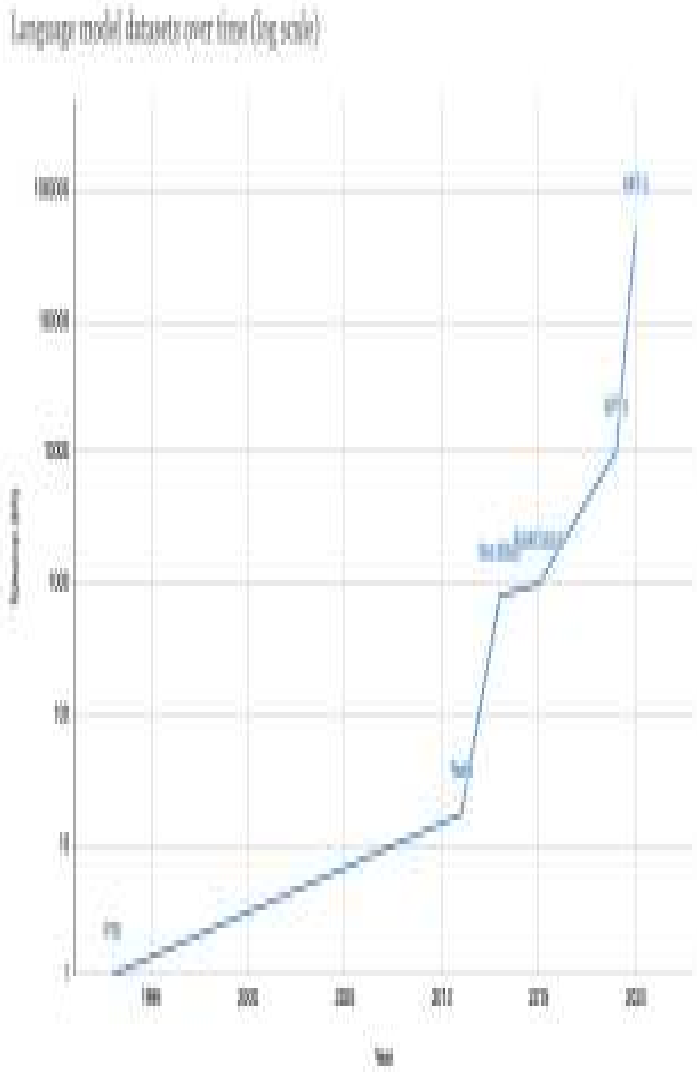


Figure 2-2. The size of the datasets used for language

models over time (log scale)

Even though much of the progress in deep learning in the last decade was fueled by an increasingly large amount of data, more data doesn't always lead to better performance for your model. More data at lower quality, such as data that is outdated or data with incorrect labels, might even hurt your model's performance.

Data Sources

An ML system works with data from many different sources. They have different characteristics with different access patterns, can be used for different purposes, and require different processing methods. Understanding the sources your data comes from can help you use your data more efficiently. This section aims to give a quick overview of different data sources to those unfamiliar with data in production. If you've already worked with ML in production for a

while, feel free to skip this section.

One source is **user input data**, data explicitly input by users, which is often the input on which ML models can make predictions. User input can be texts, images, videos, uploaded files, etc. If there is a wrong way for humans to input data, humans are going to do it, and as a result, user input data can be easily mal-formatted. If user input is supposed to be texts, they might be too long or too short. If it's supposed to be numerical values, users might accidentally enter texts. If you expect users to upload files, they might upload files in the wrong formats. User input data requires more heavy-duty checking and processing. Users also have little patience. In most cases, when we input data, we expect to get results back immediately. Therefore, user input data tends to require fast processing.

Another source is **system-generated data**. This is the data generated by different components of your systems, which include various types of logs and system outputs such

as model predictions.

Logs can record the state of the system and significant events in the system, such as memory usage, number of instances, services called, packages used, etc. It can record the results of different jobs, including large batch jobs for data processing and model training. These types of logs provide visibility into how the system is doing, and the main purpose of this visibility is for debugging and possibly improving the application. Most of the time, you don't have to look at this type of log, but they are essential when something is on fire.

Because logs are system generated, they are much less likely to be mal-formatted the way users input data is. Overall, logs don't need to be processed as soon as they arrive, the way you would want to process user input data. For many use cases, it's acceptable to process logs periodically, such as hourly or even daily. However, you might still want to process your logs fast to be able to detect and

be notified whenever something interesting happens[7].

Because debugging ML systems is hard, it's a common practice to log everything you can. This means that your volume of logs can grow very, very quickly. This leads to two problems. The first is that it can be hard to know where to look because signals are lost in the noise. There have been many services that process and analyze logs, such as Logstash, DataDog, Logz, etc. Many of them use ML models to help you process and make sense of your massive amount of logs.

The second problem is how to store a rapidly growing amount of logs. Luckily, in most cases, you only have to store logs for as long as they are useful, and can discard them when they are no longer relevant for you to debug your current system. If you don't have to access your logs frequently, they can also be stored in low-access storage that costs much less than higher-frequency-access storage.

System also generates data to record users' behaviors, such as clicking, choosing a suggestion, scrolling, zooming, ignoring a popup, or spending an unusual amount of time on certain pages. Even though this is system-generated data, it's still considered part of **user data**[8] and might be subject to privacy regulations. This kind of data can also be used for ML systems to make predictions and to train their future versions.

There are also **internal databases**, generated by various services and enterprise applications in a company. These databases manage their assets such as inventory, customer relationship, users, and more. This kind of data can be used by ML models directly or by various components of an ML system. For example, when users enter a search query on Amazon, one or more ML models will process that query to detect the intention of that query — what products users are actually looking for? — then Amazon will need to check their internal databases for

the availability of these products before ranking them and showing them to users.

Then there's the wonderfully weird word of **third-party data** that, to many, is riddled with privacy concerns. First-party data is the data that your company already collects about your users or customers. Second-party data is the data collected by another company on their own customers that they make available to you, though you'll probably have to pay for it. Third-party data companies collect data on the public who aren't their customers.

The rise of the Internet and smartphones has made it much easier for all types of data to be collected. It's especially easy with smartphones since each phone has a Mobile Advertiser ID, which acts as a unique ID to aggregate all activities on a phone. Data from apps, websites, check-in services, etc. are collected and (hopefully) anonymized to generate activity history for each person.

You can buy all types of data such as social

media activities, purchase history, web browsing habits, car rentals, political leaning for different demographic groups getting as granular as men, age 25-34, working in tech, living in the Bay Area. From this data, you can infer information such as people who like brand A also like brand B. This data can be especially helpful for systems such as recommendation systems to generate results relevant to users' interests. Third-party data is usually sold as structured data after being cleaned and processed by vendors.

Data Formats

Once you have data, you might want to store it (or "persist" it, in technical terms). Since your data comes from multiple sources with different access patterns, storing your data isn't always straightforward and can be costly. Some of the questions you might want to consider are: How do I store multimodal data? When each sample might contain both

images and texts? Where to store your data so that it's cheap and still fast to access? How to store complex models so that they can be loaded and run correctly on different hardware?

The process of converting a data structure or object state into a format that can be stored or transmitted and reconstructed later is **data serialization**. There are many, many data serialization formats. When considering a format to work with, you might want to consider different characteristics such as human readability, access patterns, and whether it's based on text or binary, which influences the size of its files. Table 2-1 consists of just a few of the common formats that you might encounter in your work. For a more comprehensive list, check out the wonderful Wikipedia page *Comparison of data-serialization formats*.

Table 2-1. Common data formats and where they are used.

Format	Binary/Text	Human-readable?
JSON	Text	Yes
CSV	Text	Yes
Parquet	Binary	No
Avro	Binary primary	No
Protobuf	Binary primary	No
Pickle	Binary	No

We’ll go over a few of these formats, starting with JSON.

JSON

JSON, JavaScript Object Notation, is everywhere. Even though it was derived from JavaScript, it’s language-independent — most modern programming languages can generate and parse JSON. It’s human-readable. Its key-value pair paradigm is simple but powerful, capable of handling data

of different levels of structuredness. For example, your data can be stored in a structured format like the following.

```
{
  "firstName": "Boatie",
  "lastName": "McBoatFace",
  "isVibing": true,
  "age": 12,
  "address": {
    "streetAddress": "12 Ocean Drive",
    "city": "Port Royal",
    "postalCode": "10021-3100"
  }
}
```

The same data can also be stored in an unstructured blob of text like the following.

```
{
  "text": "Boatie McBoatFace, aged 12, is
vibing, at 12 Ocean Drive,
           Port Royal, 10021-3100"
}
```

Row-major vs. Column-major Format

The two formats that are common and represent two distinct paradigms are CSV and

Parquet. CSV is row-major, which means consecutive elements in a row are stored next to each other in memory. Parquet is column-major, which means consecutive elements in a column are stored next to each other.

Because modern computers process sequential data more efficiently than non-sequential data, if a table is row-major, accessing its rows will be faster than accessing its columns in expectation. This means that for row-major formats, accessing data by rows is expected to be faster than accessing data by columns.

Imagine we have a dataset of 1000 examples, each example has 10 features. If we consider each example as a row and each feature as a column, then the row-major formats like CSV are better for accessing examples, e.g. accessing all the examples collected today. Column-major formats like Parquet are better for accessing features, e.g. accessing the timestamps of all your examples.

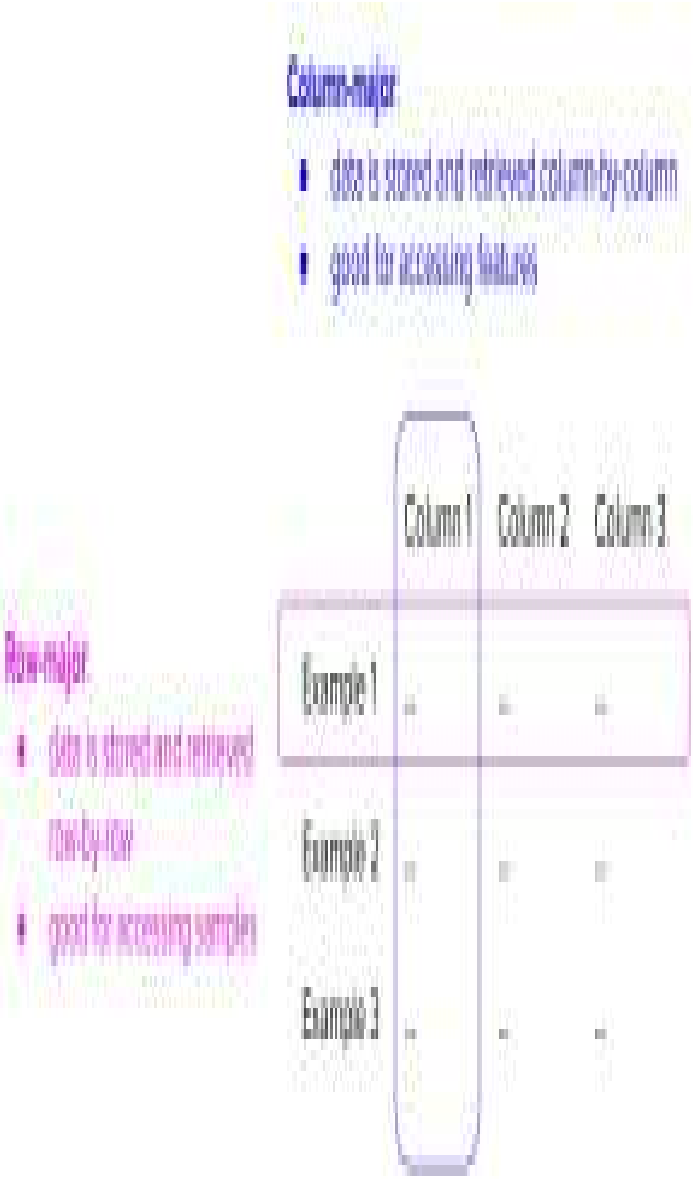

Column-major
data is stored and retrieved column-by-column
good for accessing features
Row-major
data is stored and retrieved row-by-row
good for accessing samples
Column 1
Column 2
Column 3
Example 1
Example 2
Example 3

Figure 2-3. Row-major vs. column-major formats

> I use CSV as an example of the row-major format because it's popular and generally recognizable by everyone I've talked to in tech. However, some of the early reviewers of this book got upset by the mention of CSV because they believe CSV is a horrible data format. It serializes non-text characters poorly. For example, when you write float values to a CSV file, some precision might be lost — 0.12345678901232323 could be arbitrarily rounded up as "0.12345678901" — as complained about here and here. People on Hacker News have passionately argued against using CSV.

Column-major formats allow flexible column-based reads, especially if your data is large with thousands, if not millions, of features. Consider if you have data about

ride-sharing transactions that has 1000 features but you only want 4 features: time, location, distance, price. With column-major formats, you can read the 4 columns corresponding to these 4 features directly. However, with row-major formats, if you don't know the sizes of the rows, you will have to read in all columns then filtering down to these 4 columns. Even if you know the sizes of the rows, it can still be slow as you'll have to jump around the memory, unable to take advantage of caching.

Row-major formats allow faster data writes. Consider the situation when you have to keep adding new individual examples to your data. For each individual example, it'd be much faster to write it to a file that your data is already in a row-major format.

Overall, row-major formats are better when you have to do a lot of writes, whereas column-major ones are better when you have to do a lot of column-based reads.

NUMPY VS. PANDAS

One subtle point that a lot of people don't pay attention to, which leads to misuses of Pandas, is that this library is built around the columnar format.

Pandas is built around DataFrame, a concept inspired by R's Data Frame, which is column-major. A DataFrame is a two-dimensional table with rows and columns.

In NumPy, the major order can be specified. When an ndarray is created, it's row-major by default if you don't specify the order. People coming to pandas from NumPy tend to treat DataFrame the way they would ndarray, e.g. trying to access data by rows, and find DataFrame slow.

In Figure 2-4, you can see that accessing a DataFrame by is so much slower than accessing the same DataFrame by column. If you convert this same DataFrame to a NumPy ndarray,

accessing a row becomes much faster, as you can see in Figure 2-5.[9]

```
# Iterating pandas DataFrame by column
start = time.time()
for col in df.columns:
    for item in df[col]:
        pass
print(time.time() - start, "seconds")
```

```
0.06656503677368164 seconds
```

```
# Iterating pandas DataFrame by row
n_rows = len(df)
start = time.time()
for i in range(n_rows):
    for item in df.iloc[i]:
        pass
print(time.time() - start, "seconds")
```

```
2.4123919010162354 seconds
```

Figure 2-4. Iterating a pandas DataFrame by column

takes 0.07 seconds but iterating the same DataFrame by row takes 2.41 seconds.

```
df_np = df.to_numpy()
n_rows, n_cols = df_np.shape
```

```
# Iterating NumPy ndarray by column
start = time.time()
for j in range(n_cols):
    for item in df_np[:, j]:
        pass
print(time.time() - start, "seconds")
```

```
0.005830049514770508 seconds
```



```
# Iterating NumPy ndarray by row
start = time.time()
for i in range(n_rows):
    for item in df_np[i]:
        pass
print(time.time() - start, "seconds")
```

```
0.019572019577026367 seconds
```

Figure 2-5. When you convert the same DataFrame into a NumPy ndarray, accessing its rows becomes much faster.

Text vs. Binary Format

CSV and JSON are text files whereas Parquet files are binary files. Text files are files that are in plain texts, which usually mean they are human-readable. Binary files, as the name suggests, are files that contain 0's and 1's, and meant to be read or used by programs that know how to interpret the raw bytes. A program has to know exactly how the data inside the binary file is laid out to make use of the file. If you open text files in your text editors (e.g. VSCode, Notepad), you'll be able to read the texts in them. If you open a binary file in your text editors, you'll see blocks of numbers, likely in hexadecimal values, for corresponding bytes of the file.

Binary files are more compact. Here's a simple example to show how binary files can

save space compared to text files. Consider you want to store the number 1000000. If you store it in a text file, it'll require 7 characters, and if each character is 1 byte, it'll require 7 bytes. If you store it in a binary file as int32, it'll take only 32 bits or 4 bytes.

As an illustration, I use interviews.csv, which is a CSV file (text format) of 17,654 rows and 10 columns. When I converted it to a binary format (Parquet), the file size went from 14MB to 6MB, as shown in Figure 2-6.

```
In [2]: df = pd.read_csv("data/interviews.csv")
        df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 17654 entries, 0 to 17653
Data columns (total 10 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   Company     17654 non-null  object 
 1   Title       17654 non-null  object 
 2   Job         17654 non-null  object 
 3   Level       17654 non-null  object 
 4   Date        17652 non-null  object 
 5   Upvotes     17654 non-null  int64  
 6   Offer       17654 non-null  object 
 7   Experience  16365 non-null  float64
 8   Difficulty  16376 non-null  object 
 9   Review      17654 non-null  object 
dtypes: float64(1), int64(1), object(8)
memory usage: 1.3+ MB
```

```
In [3]: Path("data/interviews.csv").stat().st_size
```

```
Out[3]: 14200063
```

```
In [4]: df.to_parquet("data/interviews.parquet")
        Path("data/interviews.parquet").stat().st_size
```

```
Out[4]: 6211862
```

Figure 2-6. When stored in CSV format, my interview file is 14MB. But when stored in Parquet, the same file is 6MB.

AWS recommends using the Parquet format because "the Parquet format is up to 2x faster to unload and consumes up to 6x less storage in Amazon S3, compared to text formats."[10]

Data Processing and Storage

In this section, we will cover the basics of data processing, starting with two major types of processing: transaction processing and analytical processing, and their uses. We will then cover the basics of the ETL (Extract, Transform, Load) process that you will inevitably encounter when building an ML system in production. When dealing with a large amount of data, a question that often comes up is whether you want to store your data as structured or unstructured. In the last part of this section, we will discuss the pros and cons of both formats.

Readers tuned into data engineering trends might wonder why batch processing versus stream processing is missing from this chapter. We'll cover this topic in Chapter 6: Deployment since I believe it's more related to other deployment concepts.

Transactional and Analytical Databases

Systems in production generate data. To access and process data generated in production, there are two types of queries: transactional queries and analytical queries. We'll go over an example to illustrate their differences.

Imagine you're running a consumer application that generates many short transactions within a short amount of time, such as food ordering, online shopping, ridesharing, or money transferring. You want to process and store these transactions as they are generated. They need to be processed fast,

in the order of milliseconds. The processing method needs to have extremely high availability, because, without a way to record transactions, you won't be able to serve your users. On top of that, the processing needs to satisfy the ACID (Atomicity, Consistency, Isolation, Durability) requirements:

- **A**tomicity: to guarantee that all the steps in a transaction are completed successfully as a group. If any steps between the transaction fail, all other steps must fail also. For example, if a user's payment fails, you don't want to still assign a driver to that user.
- Consistency: to guarantee that all the transactions coming through must follow predefined rules. For example, a transaction must be made by a valid user.
- Isolation: to guarantee that two transactions happen at the same time

as if they were isolated. Two users accessing the same data won't change it at the same time. For example, you don't want two users to book the same driver at the same time.

- Durability: to guarantee that once a transaction has been committed, it will remain committed even in the case of a system failure. For example, after you've ordered a ride and your phone dies, you still want your ride to come.

Transactional databases are designed to process online transactions and satisfy all those requirements. Most of the operations they do will be inserting, deleting, and updating an existing transaction. This means that most transactional databases are more row-oriented.

Because transactional databases are more row-oriented, they might not be efficient for

questions such as “What’s the average price for all the rides in September in San Francisco?”. This kind of analytical question requires aggregating data in columns across multiple rows of data. Analytical databases are designed for this purpose. They are efficient with queries that allow you to look at data from different viewpoints.

Traditionally, transactional databases are called **OnLine Transaction Processing (OLTP)** and analytical databases are called **OnLine Analytical Processing (OLAP)**. However, both the terms OLTP and OLAP have become outdated, as shown in Figure 2-7, for three reasons. First, the separation of transactional and analytical databases was due to limitations of technology — it was hard to have databases that could handle both transactional and analytical queries efficiently. However, this separation is being closed. Today, we have transactional databases that can handle analytical queries, such as CockroachDB. We also have

analytical databases that can handle transactional queries, such as Apache Iceberg.

Figure 2-7. OLAP and OLTP are outdated terms, as of 2021, according to Google Trends

Second, in the traditional OLTP or OLAP paradigms, storage and processing are tightly coupled — how data is stored is also how data is processed. This may result in the same data being stored in multiple databases and use different processing engines to solve different types of queries. An interesting paradigm in the last decade has been to decouple storage from processing (also known as compute), as adopted by many data vendors including Google's BigQuery, Snowflake, IBM, and Teradata. In this paradigm, the data can be stored in the same place, with a processing layer on top that can be optimized for different types of queries.

Third, "online" has become an overloaded term that can mean many different things. Online used to just mean "connected to the Internet". Then, it grew to also mean "in production" — we say a feature is online after that feature has been deployed in

production.

In the data world today, “online” might refer to the speed at which your data is processed and made available: online, nearline, or offline. According to Wikipedia, online processing means data is immediately available for input/output. Nearline, which is short for near-online, means data is not immediately available, but can be made online quickly without human intervention. Offline means data is not immediately available, and requires some human intervention to become online.

As the speed at which applications respond to users queries has become a competitive advantage, it’s become more and more important to make data available for use as fast as possible. In many use cases, companies want online processing not just for transactional queries but also for analytical queries. Online, in this case, is synonymous to “real-time”. Both online processing and nearline processing are covered by stream

processing that we'll cover in Chapter 6.

ETL: Extract, Transform, Load

Even before ML, ETL (extract, transform, load) was all the rage in the data world, and it's still relevant today for ML applications. ETL refers to the general purpose processing and aggregating data into the shape and the format that you want.

Extract is extracting the data you want from data source(s). Your data will likely come from multiple sources in different formats. Some of them will be corrupted or malformatted. In the extracting phase, you need to validate your data and reject the data that doesn't meet your requirements. For rejected data, you might have to notify the sources. Since this is the first step of the process, doing it correctly can save you a lot of time downstream.

Transform is the meaty part of the process, where most of the data processing is done.

You might want to join data from multiple sources and clean it. You might want to standardize the value ranges (e.g. one data source might use “Male” and “Female” for genders, but another uses “M” and “F” or “1” and “2”). You can apply operations such as transposing, deduplicating, sorting, aggregating, deriving new features, more data validating, etc..

Load is deciding how and how often to load your transformed data into the target destination, which can be a file, a database, or a data warehouse.

The idea of ETL sounds simple but powerful, and it’s the underlying structure of the data layer at many organizations. An overview of the ETL process is shown in Figure 2-8.

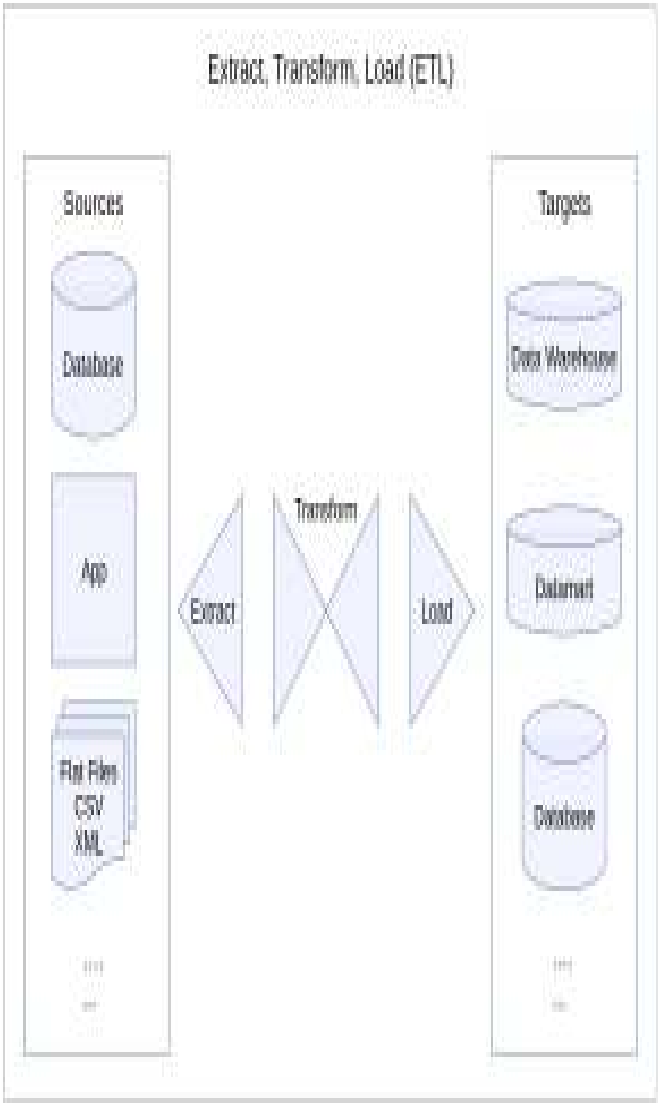

Extract, Transform, Load (ETL)
Sources
Database
App
Flat Files
CSV
XML
Extract
Transform
Load
Targets
Data Warehouse
Datamart
Database

Figure 2-8. An overview of the ETL process

Structured vs. unstructured data

Structured data is data that follows a predefined data model, also known as a data schema. For example, the data model might specify that each data item consists of two values: the first value, “name”, is a string at most 50 characters, and the second value, “age”, is an 8-bit integer in the range between 0 and 200. The predefined structure makes your data easier to analyze. If you want to know the average age of people in the database, all you have to do is to extract all the age values and get their mean.

The disadvantage of structured data is that you have to commit your data to a predefined schema. If your schema changes, you’ll have to retrospectively update all your data and/or the changes will cause mysterious bugs. For example, you’ve never kept your users’ email addresses before but now you do, so you have to retrospectively update email information to

all previous users. One of the strangest bugs one of my colleagues encountered was when they could no longer use users' age with their transactions, and their data schema replaced all the null age with 0, and their ML model thought the transactions were made by people of 0 years old.

Because business requirements change over time, committing to a predefined data schema can become too restricting. Or you might have data from multiple data sources, many of the sources are beyond your control, and it's impossible to make them follow the same schema. This is where unstructured data becomes appealing. Unstructured data is data that doesn't adhere to a predefined data schema. It's usually text but can also be numbers, dates, etc. For example, a text file of logs generated by your ML model is unstructured data.

Even though unstructured data doesn't adhere to a schema, it might still contain intrinsic patterns that help you extract structures. For

example, the following text is unstructured, but you can notice the pattern that each line contains two values separated by a comma, the first value is textual and the second value is numerical. However, there is no guarantee that all lines must follow this format. You can add a new line to that text even if that line doesn't follow this format.

```
 “Lisa, 43
Jack, 23
Nguyen, 59”
```

Unstructured data also allows for more flexible storage options. For example, if your storage follows a schema, you can only store data following that schema. But if your storage doesn't follow a schema, you can store any type of data. You can convert all your data, regardless of types and formats into bytestrings and store them together.

A repository for storing structured data is called a data warehouse. A repository for storing unstructured data is called a data lake. Data lakes are usually used to store raw data

before processing. Data warehouses are used to store data that have been processed into formats ready to be used.

ETL to ELT

When the Internet first became ubiquitous and hardware had just become so much more powerful, collecting data suddenly became so much easier. The amount of data grew rapidly. Not only that, but the nature of data also changed. The number of data sources expanded, and data schemas evolved.

Finding it difficult to keep data structured, some companies had this idea: “Why not just store all data in a data lake so we don’t have to deal with schema changes? Whichever application needs data can just pull out raw data from there and process it.” This process of loading data into storage first then processing it later is sometimes called ELT (extract, load, transform). This paradigm allows for the fast arrival of data since there’s little processing needed before data is stored.

However, as data keeps on growing, this idea becomes less attractive. It's expensive to store everything, and it's inefficient to search through a massive amount of raw data for the piece of data that you want. At the same time, as companies switch to running applications on the cloud and infrastructures become standardized, data structures also become standardized. Committing data to a predefined schema becomes more feasible.

Table 2-2 shows a summary of the key differences between structured and unstructured data.

Table 2-2. The key differences between structured and unstructured data

Structured data	Unstructured data
Schema clearly defined	Data doesn't have to follow a schema
Easy to search and analyze	Fast arrival
Can only handle data with a specific schema	Can handle data from any source

Schema changes will cause a lot of troubles	No need to worry about schema changes (yet) as the worry is shifted to the downstream applications that use this data
Stored in data warehouses	Stored in data lakes

Summary

In this chapter, we started with the question about the role of data in building intelligent systems. There are still many people who believe that having intelligent algorithms will eventually trump having a large amount of data. However, the success of systems including AlexNet, BERT, GPT showed that the progress of ML in the last decade relies on having access to a large amount of data.

Therefore, it's important for ML practitioners to know how to manage and process a large amount of data. This chapter covered the fundamentals of data engineering that I wish I knew when I started my ML career, from handling data from different data sources,

choosing the right data format, to processing structured and unstructured data. These fundamentals will hopefully help readers become better prepared when facing seemingly overwhelming data in production.

---

1 Tweet by Dr. Judea Pearl (2020)

2 Deep Learning and Innate Priors (Chris Manning vs. Yann LeCun debate).

3 The Bitter Lesson (Richard Sutton, 2019)

4 The Unreasonable Effectiveness of Data (Alon Halevy, Peter Norvig, and Fernando Pereira, Google 2009)

5 The AI Hierarchy of Needs (Monica Rogati, 2017)

6 1 Billion Word Language Model Benchmark (Chelba et al., 2013)

7 "Interesting" in production usually means catastrophic, such as a crash or when your cloud bill hits an astronomical amount.

8 An ML engineer once mentioned to me that his team only used users' historical product browsing and purchases to make recommendations on what they might like to see next. I responded: "So you don't use personal data at all?" He looked at me, confused. "If you meant demographic data like users' age, location

then no, we don’t. But I’d say that a person’s browsing and purchasing activities are extremely personal.”

9 For more Pandas quirks, check out just-pandas-things (Chip Huyen, GitHub 2020).

10 Announcing Amazon Redshift data lake export: share data in Apache Parquet format (Amazon AWS 2019).

Chapter 3. Training Data

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter,

please reach out to the author at *chip@huyenchip.com*.

Despite the importance of training data in developing and improving ML models, ML curricula are heavily skewed towards modeling, which is considered by many researchers and engineers as the "fun" part of the process. Building a state-of-the-art model is interesting. Spending days wrangling with a massive amount of malformatted data that doesn't even fit into your machine's memory is frustrating.

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

In this chapter, we will go over techniques to obtain or create good training data. Training

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

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

Before we move forward, I just want to echo

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

Sampling

Sampling is an integral part of the ML workflow that is, unfortunately, often overlooked in typical ML coursework. Sampling happens in many steps of an ML project lifecycle, such as sampling from all possible real-world data to create training data, sampling from a given dataset to create splits for training, validation, and testing, or sampling from all possible events that happen within your ML system for monitoring purposes.

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

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

data we sample.

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

Non-Probability Sampling

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

- **Convenience sampling**: samples of data are selected based on their availability. This sampling method is popular because, well, it's convenient.
- **Snowball sampling**: future samples are selected based on existing samples. For example, to scrape

legitimate Twitter accounts without having access to Twitter databases, you start with a small number of accounts then you scrape all the accounts in their following, and so on.

- **Judgment sampling**: experts decide what samples to include.
- **Quota sampling**: you select samples based on quotas for certain slices of data without any randomization.

The samples selected by non-probability criteria are not representative of the real-world data, and therefore, are riddled with selection biases. 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, CommonCrawl, Reddit.

Another example is data for sentiment analysis of general text. Much of this data is collected from sources with natural labels (ratings) — IMDB reviews, Amazon reviews — even for the tasks where you want to predict sentiments of texts that aren't IMDB or Amazon reviews. These sources don't include people who don't have access to the Internet and aren't willing to put reviews online.

The third example is data for training self-driving cars. Initially, data collected for self-driving cars came largely from two areas: Phoenix in Arizona (because of its lax regulations) and the Bay Area in California (because many companies that build self-driving cars are located here). Both areas have generally sunny weather. In 2016,

Waymo expanded its operations to Kirkland, WA specially for Kirkland's rainy weather, but there's still a lot more self-driving car data for sunny weather than for rainy or snowy weather.

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

Simple Random Sampling

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

The advantage of this method is that it's easy to implement. The drawback is that rare categories of data might not appear in your

selection. Consider the case where a class appears only in 0.01% of your data population. If you randomly select 1% of your data, samples of this rare class will unlikely be selected. Models trained on this selection might think that this rare class doesn't exist.

Stratified Sampling

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

One drawback of this sampling method is that it isn't always possible, such as when it's impossible to divide all samples into groups.

This is especially challenging when one sample might belong to multiple groups as in the case of multilabel tasks[2]. For instance, a sample can be both class A and class B.

Weighted Sampling

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

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

This also helps with the case when the data you have comes from a different distribution compared to the true data. For example, if in

your data, red samples account for 25% and blue samples account for 75%, but you know that in the real world, red and blue have equal probability to happen, you can give red samples the weights three times higher than blue samples.

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

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

A common concept in ML that is closely related to weighted sampling is sample weights. Weighted sampling is used to select samples to train your model with, whereas sample weights are used to assign “weights”

or “importance” to training samples. Samples with higher weights affect the loss function more. Changing sample weights can change your model’s decision boundaries significantly, as shown in Figure 3-1.

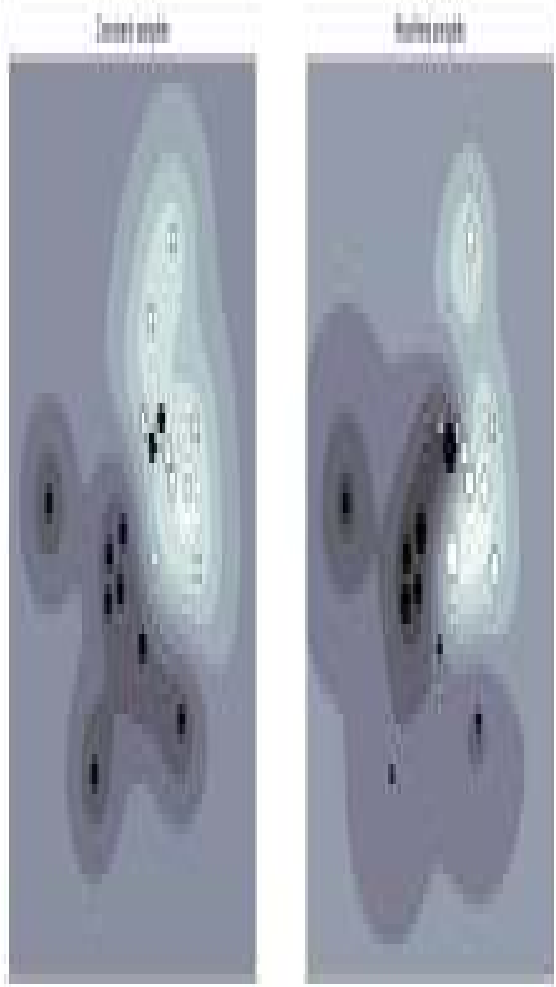


Figure 3-1. How sample weights can affect the decision boundary. On the left is when all samples are given equal weights. On the right is when samples are given different weights. Source: SVM: Weighted samples (sklearn), BSD License.

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 weight this sample by $\frac{P(x)}{Q(x)}$ $Q(x)$ is called the proposal distribution or the importance distribution. $Q(x)$ can be any distribution as long as $Q(x) > 0$ whenever $P(x) \neq 0$. The equation below shows that in expectation, x sampled

from $P(X)$ is equal to x sampled from $Q(x)$ weighted by $\frac{P(x)}{Q(x)}$.

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

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

Reservoir Sampling

Reservoir sampling is a fascinating algorithm

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

Imagine you have an incoming stream of tweets and you want to sample a certain number, k, of tweets to do analysis or training a model on. You don't know how many tweets there are but you know you can't fit them all in memory, which means you don't know the probability at which a tweet should be selected. You want to ensure that one, every tweet has an equal probability of being selected, and two, you can stop the algorithm at any time and the tweets are sampled with the correct probability.

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

1. Put the first k elements into the reservoir.
2. For each incoming nth element,

generate a random number i such that 1 ≤ 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 i^{th} element has $\frac{k}{i}$ probability of being in the reservoir. You can also prove that each element in the reservoir has $\frac{k}{i}$ probability of being there. This means that all samples have an equal chance of being selected. If we stop the algorithm at any time, all samples in the reservoir have been sampled with the correct probability. Figure 3-2 shows an illustrative example of how reservoir sampling works.

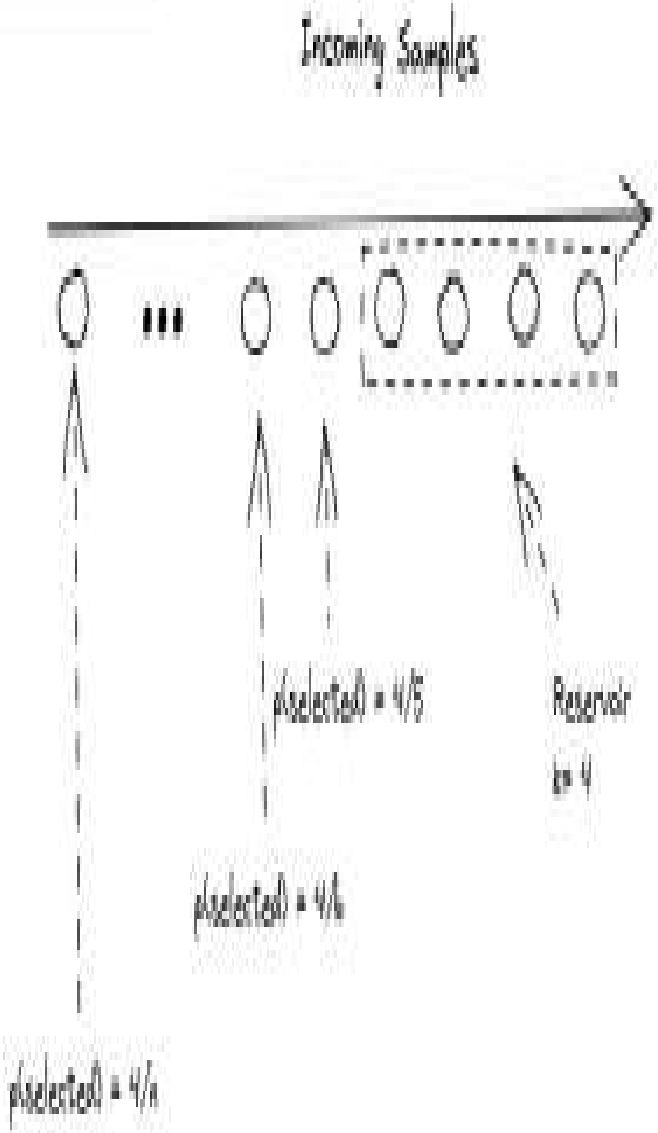

Incoming Samples
p(select(x)) = 4/n
p(select(x)) = 4/5
p(select(x)) = 4/6
Reservoir
of 4

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

Labeling

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

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

In a talk to my students, Andrej Karpathy, Director of AI at Tesla, shared an anecdote about when he decided to have an in-house

labeling team, his recruiter asked how long he'd need this team for. He responded: "How long do we need an engineering team for?" Data labeling has gone from being an auxiliary task to being a core function of many ML teams in production.

In this section, we will discuss the challenges of obtaining labels for your data and their proposed solutions.

Hand Labels

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

radiologists, whose time is limited and expensive.

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

Third, **hand labeling is slow**. While the more data a person labels, the faster their annotation speed will become, the improvement isn't in order of magnitude. Labeling 1000 samples takes approximately 10 times longer than labeling 100 samples. For example, accurate transcription of speech utterance at phonetic level can take 400 times longer than the utterance duration. So if you

want to annotate 1 hour of speech, it'll take 400 hours or almost 3 working months to do so. In a study to use ML to help classify lung cancers from X-rays, my colleagues had to wait almost a year to obtain sufficient labels.

Slow labeling leads to **slow iteration speed** and makes your model less adaptive to changing environments and requirements. If the task changes or data changes, you'll have to wait for your data to be relabeled before updating your model. Imagine the scenario when you have a sentiment analysis model to analyze the sentiment of every tweet that mentions your brand. It has only two classes: **NEGATIVE** and **POSITIVE**. However, after deployment, your PR team realizes that the most damage comes from angry tweets and they want to attend to angry messages faster. So you have to update your sentiment analysis model to have three classes: **NEGATIVE**, **POSITIVE**, and **ANGRY**. To do so, you will need to look at your data again to see which existing training examples

should be relabeled **ANGRY**. If you don't have enough **ANGRY** examples, you will have to collect more data. The longer the process takes, the more your existing model performance will degrade.

Label Multiplicity

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

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

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

Galactic Empire.

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

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

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

		reigned over the galaxy as [**Galact Emperor**] of the [**First Galactic Empire**].
3	4	[Darth Sidious], known simply as the [Emperor], wa a [Dark Lord of th Sith] who reigned over the galaxy as [Galactic Empero of the First Galact Empire].

Disagreements among annotators are extremely common. The higher level of domain expertise required, the higher the potential for annotating disagreement[3]. If one human-expert thinks the label should be A while another believes it should be B, how do we resolve this conflict to obtain one single ground truth? If human experts can't agree on a label, what does human-level performance even mean?

To minimize the disagreement among

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

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

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

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

Handling the Lack of Hand Labels

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

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

Method	How	Ground truths required?
Semi supervision	Leverages structural assumptions to generate labels	A small number o initial labels as seeds to generate more labels
Weak supervision	Leverages (often noisy) heuristics to generate labels	A small number o labels are recommended to guide the development of heuristics
Transfer learning	Leverages models	No for zero-shot

	pretrained on another task for your new task	learning Yes for fine-tunin though the numbe of ground truths required is often much smaller thar what would be needed if you trai 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 Stanford AI Lab[4]. 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 **labeling function (LF)**: a function that encodes heuristics. The above heuristics can be expressed by the following function.

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

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

- Keyword heuristic, such as the example above.
- Regular expressions, such as if the note matches or not matches a certain regular expression.
- Database lookup, such as if the note contains the disease listed in the dangerous disease list.

- The outputs of other models, such as if an existing system classifies this as EMERGENT

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

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

Figure 3-3. A high level overview of how labeling functions are combined.[5]

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

Weak supervision can be especially useful when your data has strict privacy requirements. You only need to see a small, cleared subset of data to write LFs, which can be applied to the rest of your data without 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. Table 3-3 shows some of the advantages of programmatic labeling over hand labeling.

Table 3-3. 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, reused across organization
Non-private: Need to ship data to human annotators	**Privacy**: Create LFs using a cleared data subsample then apply LFs to other data without looking at individual samples.
Slow: Time required scales linearly with # labels needed	**Fast**: Easily scale from 1K to 1M samples
Non-adaptive: Every change requires re-labeling	**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[6], models trained with weakly-supervised labels obtained by a single radiologist after 8 hours of writing labeling functions had comparable performance with models trained on data obtained through almost a year of hand labeling. There are two interesting facts about the results of the experiment. First, the models continued improving with more unlabeled data even without more labeling functions. Second, labeling functions were being reused across tasks. The researchers were able to reuse 6 labeling functions between the CXR (Chest X-Rays) task and EXR (Extremity X-Rays) task.

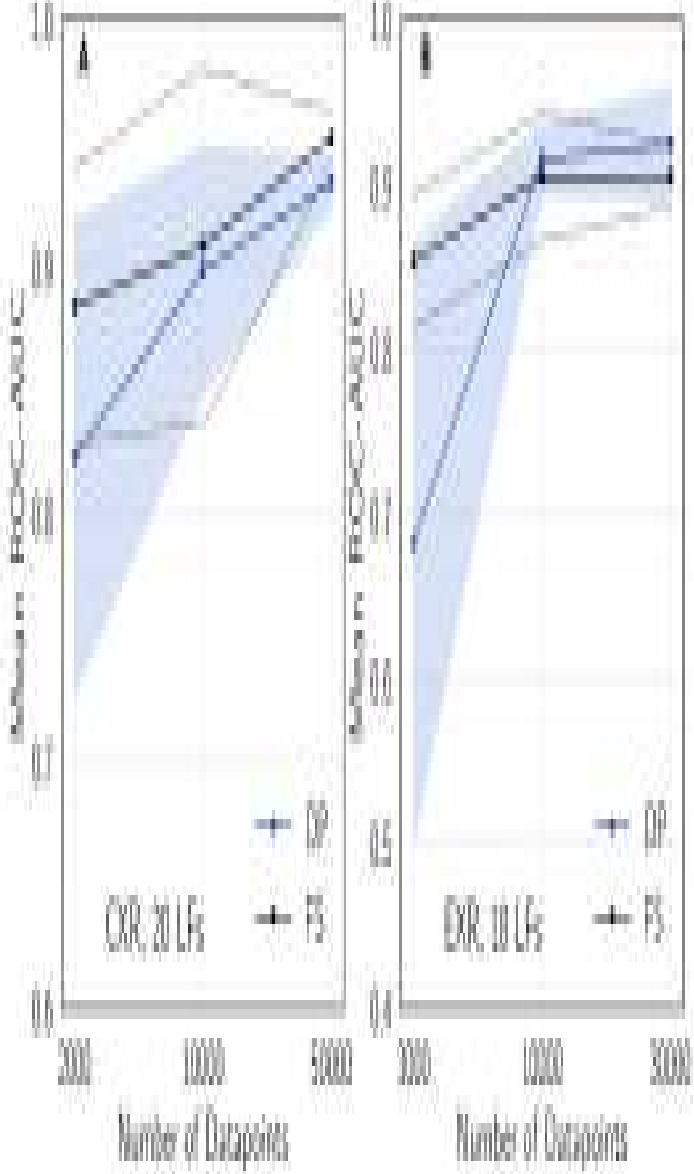

A
B
1.0
0.9
0.8
0.7
0.6
0.5
0.4
Mean ROC-AUC
Mean ROC-AUC
DBP
PS
DBP
PS
EXR: 20 LFs
EXR: 10 LFs
2000
10000
50000
2000
10000
50000
Number of Datapoints
Number of Datapoints

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

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

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

Semi supervision

If weak supervision leverages heuristics to obtain noisy labels, semi supervision leverages structural assumptions to generate

new labels based on a small set of initial labels. Unlike weak supervision, semi supervision requires an initial set of labels.

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

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

this expanded training set. This goes on until you're happy with your model performance.

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

Figure 3-5. Because #ML and #BigData appears 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 neighbor method to discover samples that belong to the same cluster.

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

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

Transfer learning

Transfer learning refers to the family of methods where a model developed for a task is reused as the starting point for a model on a

second task. First, the base model is trained for a base task such as language modeling. The base task is usually a task that has cheap and abundant training data. Language modeling is a great candidate because it doesn't require labeled data. You can collect any body of text — books, Wikipedia articles, chat histories — and the task is: given a sequence of tokens[9], predict the next token.

You then fine-tune this pretrained base model on the task that you're interested in, such as sentiment analysis, intent detection, question answering, etc. This task is called a downstream task, and we say that this model is fine-tuned for this downstream task.

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

Transfer learning has gained a lot of interest in recent years for the right reasons. It has enabled many applications that were previously impossible due to the lack of training samples. A non-trivial portion of ML models in production today are the results of transfer learning, including object detection models that leverage models pretrained on ImageNet and text classification models that leverage pretrained language models such as BERT[10] or GPT-3[11]. It also lowers the entry barriers into ML, as it helps reduce the upfront cost needed for labeling data to build ML applications.

A trend that has emerged in the last five years is that 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 million USD. Many have hypothesized that in the future, only a handful of companies can afford to train large

pretrained models. The rest of the industry will use these pretrained models directly or finetune them for their specific needs.

Active learning

Active learning is a method for improving the efficiency of data labels. The hope here is that ML models can achieve greater accuracy with fewer training labels if they can choose which data samples to learn from. Active learning is sometimes called query learning, though this term is getting increasingly unpopular, because a model (active learner) sends back queries in the form of unlabeled samples to be labeled by annotators (usually humans).

Instead of randomly labeling data samples, you label the samples that are most helpful to your models according to some heuristics. The most straightforward heuristic is uncertainty measurement — label the examples that your model is the least certain about hoping that they will help your model

learn the decision boundary better. For example, in the case of classification problems where your model outputs raw probabilities for different classes, it might choose the data examples with the lowest probabilities for the predicted class. Figure 3-6 illustrates how well this method works on a toy example.

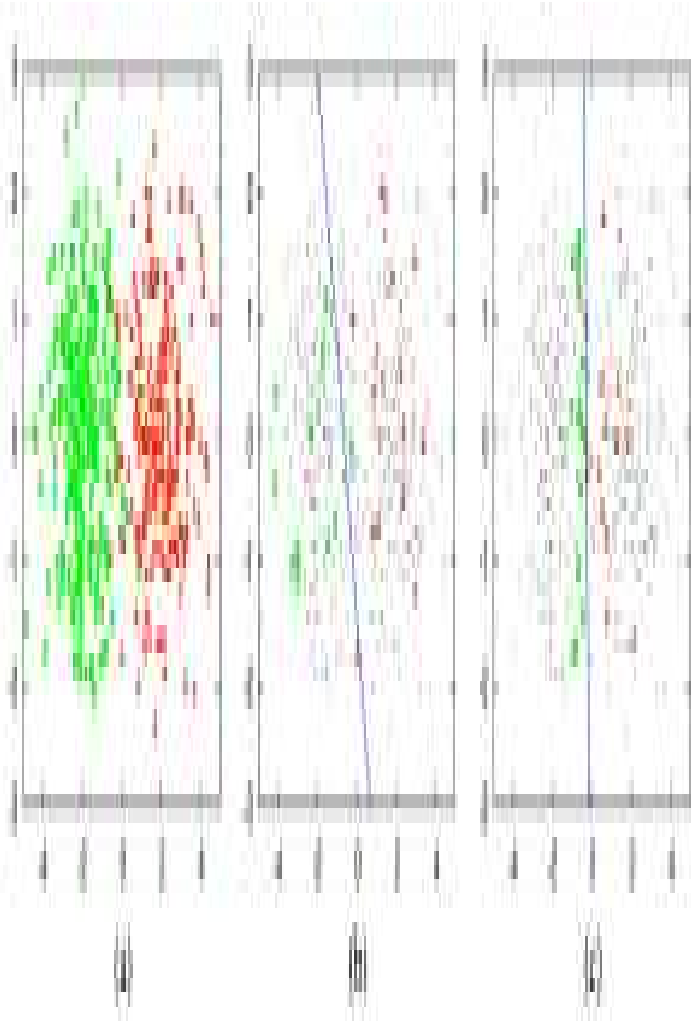


Figure 3-6. How uncertainty-based active learning works. (a) A toy data set of 400 instances, evenly sampled from two class Gaussians. (b) A model trained on 30 examples randomly labeled gives an accuracy of 70%. (c) A model

trained on 30 examples chosen by active learning gives an accuracy of 90%. Image by Burr Settles.[12]

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

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

The examples to be labeled can come from different data regimes. They can be

synthesized where your model generates examples in the region of the input space that it's most uncertain about to be labeled[13]. They can come from a stationary distribution where you've already collected a lot of unlabeled data and your model chooses examples from this pool to label. They can come from the real-world distribution where you have a stream of data coming in, as in production, and your model chooses examples from this stream of data to label.

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

Class Imbalance

Class imbalance typically refers to a problem

in classification tasks where there is a substantial difference in the number of samples in each class of the training data. For example, in a training dataset for the task of detecting lung cancer from X-Ray images, 99.99% of the X-Rays might be of normal lungs, and only 0.01% might contain cancerous cells. Class imbalance can also happen with regression tasks where the labels are continuous. For example, you build a model to predict house prices but in your data, 99% of the houses are priced between $100,000 and $500,000 and 0.1% of the houses are priced above $5,000,000.

Challenges of Class Imbalance

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

Small data and rare occurrences

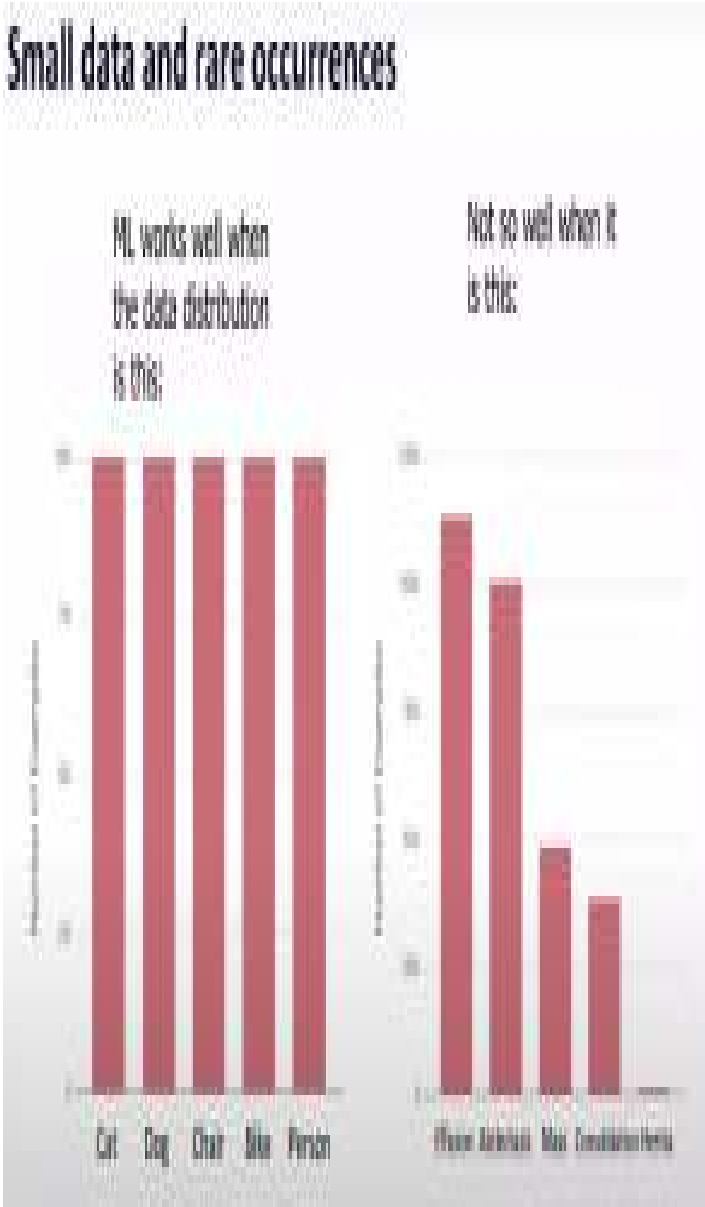


Figure 3-7. ML works well in situations where the classes are balanced. Image by Andrew Ng[14].

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

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

gradient-descent algorithms to beat because a small amount of randomness added to this heuristic might lead to worse accuracy.

The third reason is that class imbalance leads to **asymmetric costs of error** — the cost of a wrong prediction on an example of the rare class might be much higher than a wrong prediction on an example of the majority class. For example, misclassification on an X-Ray with cancerous cells is much more dangerous than misclassification on an X-Ray of a normal lung. If your loss function isn't configured to address this asymmetry, your model will treat all examples the same way. As a result, you might obtain a model that performs equally well on both majority and minority classes, while you much prefer a model that performs less well on the majority class but much better on the minority one.

When I was in school, most datasets I was given had more or less balanced classes, because I imagined that it would be easier for me to learn about neural networks if I was

free from the huge roadblock caused by class imbalance. It was a shock for me to start working and realize that class imbalance is the norm. In real-world settings, rare events are often more interesting (or more dangerous) than regular events, and many tasks focus on detecting those rare events.

The classical example of tasks with class imbalance is **fraud detection**. Most credit card transactions are not fraudulent. As of 2018, 6.8¢ for every $100 in cardholder spending is fraudulent. Another is **churn prediction**. The majority of your customers are not planning on cancelling their subscription. If they are, your business has more to worry about than churn prediction algorithms. Other examples include **disease screening** — most people, fortunately, don't have terminal illness, and **resume screening** — 98% of job seekers are eliminated at the initial resume screening. A less obvious example of a task with class imbalance is **object detection**. Object detection algorithms

currently work by generating a large number of bounding boxes over an image then predicting which boxes are most likely to have objects in them. Most bounding boxes do not contain a relevant object.

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

Another cause for class imbalance, though less common, is due to labeling errors. Your annotators might have read the instructions wrong or followed the wrong instructions (thinking there are only two classes

POSITIVE and NEGATIVE while there are actually three), or simply made errors. Whenever faced with the problem of class imbalance, it's important to examine your data to understand the causes of it.

Handling Class Imbalance

Because of its prevalence in real-world applications, class imbalance has been thoroughly studied over the last two decades[15]. 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, and that non-complex, linearly separable problems are unaffected by all levels of class imbalance[16]. 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[17].

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 like in the real world. A good model should learn to model that class imbalance. However, developing a model good enough for that can be challenging, so we still have to rely on special training techniques.

In this section, we will cover three aspects: 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, Journal of Big Data 2019).

In practice, ensembles have shown to help with the class imbalance problem[18], but this isn't usually why ensembles are used in production. Ensemble techniques will be covered in Chapter 5: Model Development and Evaluation.

Using the right evaluation metrics

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

The overall accuracy and error rate are the most frequently used metrics to report the performance of ML models. However, they are insufficient metrics for tasks with class

imbalance because they treat all classes equally, which means the performance of your model on the majority class will dominate the accuracy. This is especially bad when the majority class isn't what you care about.

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

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

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

Model A can detect 10 out of 100 CANCER cases.

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

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

Model B can detect 90 out of 100 CANCER cases.

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

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

F1 and recall are metrics that measure your model’s performance with respect to the

positive class in binary classification problems, as they rely on true positive — an outcome where the model correctly predicts the positive class[19]. F1 and recall are asymmetric metrics, which means that their values change depending on which class is considered the positive class. In our case, if we consider CANCER the positive class, model A's F1 is 0.17. However, if we consider NORMAL the positive class, model A's F1 is 0.95.

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

Table 3-6. Model A and model B have the same accuracy even though one model is clearly superior to another.

	CANCER (1)	NORMAL (0)
Model A	10/100	890/900
Model B	90/100	810/900

Many classification problems can be modeled

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

The area under the curve (AUC) measures the area under the ROC curve. Since the closer to

the perfect line the better, the larger this area the better.

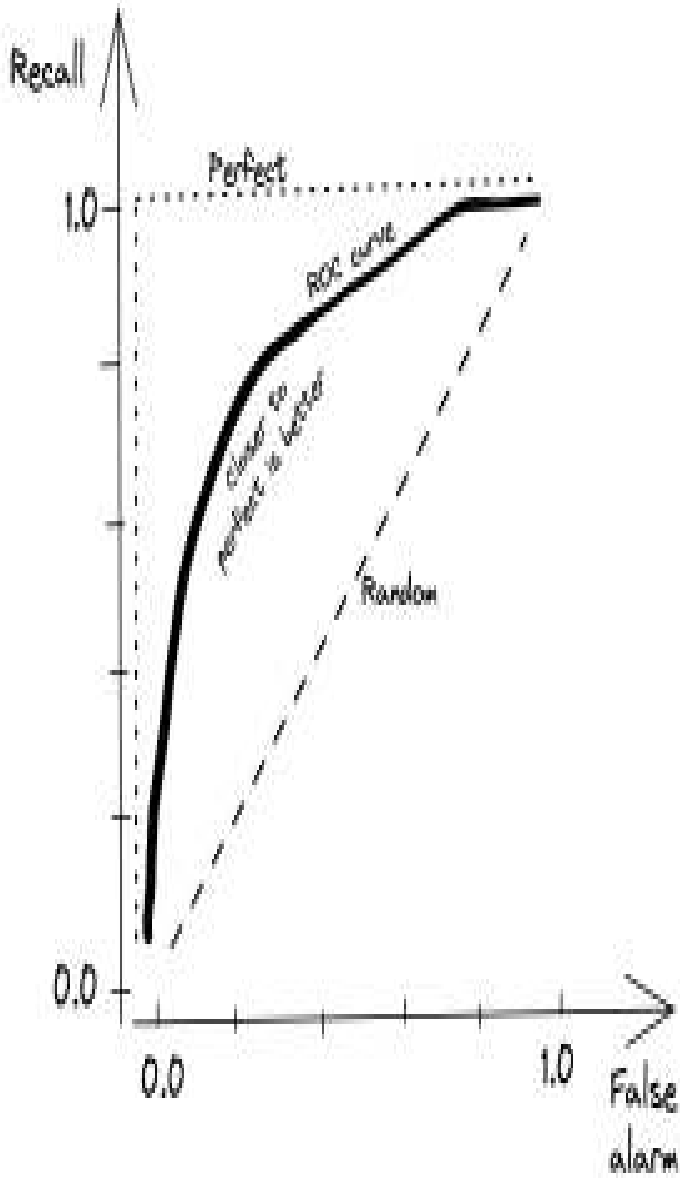

Recall
Perfect
1.0
ROC curve
closer to perfect is better
Random
0.0
0.0
1.0
False alarm

Figure 3-8. ROC curve

Like F1 and recall, the ROC curve focuses only on the positive class and doesn't show how well your model does on the negative class. Davis and Goadrich suggested that we should plot precision against recall instead, in what they termed the Precision-Recall Curve. They argued that this curve s give a more informative picture of an algorithm's performance on tasks with heavy class imbalance[20].

Data-level methods: Resampling

Data-level methods modify the distribution of the training data to reduce the level of imbalance to make it easier for the model to learn. A common family of techniques is resampling. Resampling includes oversampling, adding more examples from the minority classes and undersampling, removing examples of the majority classes. The simplest way to undersample is to randomly remove instances from the majority

class, while the simplest way to oversample is to randomly make copies of the minority class until you have a ratio that you're happy with.

Figure 3-9. Illustrations of how undersampling and oversampling works. Image by Rafael Alencar [21]

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

While this makes the decision boundary more clear and arguably helps models learn the

boundary better, it may make the model less robust by removing some of the subtleties of the true decision boundary.

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

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

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

resampled distribution.

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

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

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

Algorithm-level methods

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

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

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

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

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

Cost-sensitive learning

Back in 2001, based on the insight that misclassification of different classes incur different cost, Elkan proposed cost-sensitive learning where the individual loss function is modified to take into account this varying cost[28]. The method started by using a cost matrix to specify C_{ij}: the cost if class i is classified as class j. If $i = j$, it's a correct classification, and the cost is usually 0. If not, it's a misclassification. If classifying POSITIVE examples as NEGATIVE is twice as costly as the other way around, you can

make C_{10} twice as high as C_{01}.

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

Table 3-7. 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.

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

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

Class-balanced loss

What might happen with a model trained on an imbalance 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.

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

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.

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

A more sophisticated version of this loss can take in account the overlapping among

existing samples, such as Class-Balanced Loss Based on Effective Number of Samples (Cui et al., CVPR 2019).

Focal loss

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

Figure 3-10. The model trained with focal loss (FL) shows reduced loss values compared to the model trained with

cross entropy loss (CE). Image by Lin et al.[30]

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 projects. However, in the last few years, they have shown to be useful even when we have a lot of data because 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 and Keras both 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.[31]"

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. 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 3-8. Three sentences generated from an original sentence by replacing a word with another word with similar meaning.

Original sentences	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, even 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 by 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 Kaggle CIFAR-10 test dataset and 16.04% of the ImageNet test images can be misclassified by changing just one pixel[32] (See Figure 3-11).

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

Adding noisy samples to our training data can

help our models recognize the weak spots in their learned decision boundary and improve their performance[3334]. Noisy samples can be created by either adding random noise or by an efficient 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[35]. This type of augmentation is called adversarial augmentation[36].

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 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

replaces “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[37].

In chapter 5, we’ll go over how to use perturbation not just as a way to improve your model’s performance, but also a way to evaluate its performance.

AllConv	NiN	VGG
		
SHIP CAR(99.7%)	HORSE FROG(99.9%)	DEER AIRPLANE(85.3%)
		
HORSE DOG(70.7%)	DOG CAT(75.5%)	BIRD FROG(86.5%)
		
CAR AIRPLANE(82.4%)	DEER DOG(86.4%)	CAT BIRD(66.2%)
		
DEER AIRPLANE(49.8%)	BIRD FROG(88.8%)	SHIP AIRPLANE(88.2%)
		
HORSE DOG(88.0%)	SHIP AIRPLANE(62.7%)	CAT DOG(78.2%)

Figure 3-11. Changing one pixel can cause a neural network to make wrong predictions. Three models used are AllConv, NiN, and VGG. The original labels made by those models are in black, and the labels made after one pixel was changed are below. Image by Su et al.

Data Synthesis

Since collecting data is expensive, 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 of the teams 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]." With lists of all possible cuisines, reasonable numbers (you would probably never want to

search for restaurants beyond 1000 miles), and locations (home, office, landmarks, exact addresses) for each city, you can generate thousands of training queries from a template.

Table 3-9. 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. Find me a Thai restaurant within 5 miles of my home. 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: $\gamma * 0 + (1 — \gamma) * 1$. 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.[38]

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 a CycleGAN to their original training data, they were able to improve their model's performance significantly on CT segmentation tasks.[39]

If you're interested in learning more about data augmentation for computer vision, A survey on Image Data Augmentation for Deep Learning (Connor Shorten & Taghi M.

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.

Once you've had 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 Multilabel tasks are tasks where one example can have multiple labels.

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

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

5 Snorkel: Rapid Training Data Creation with Weak Supervision (Ratner et al., 2017)

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

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

8 Realistic Evaluation of Deep Semi-Supervised Learning Algorithms (Oliver et al., NeurIPS 2018)

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

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

11 Language Models are Few-Shot Learners (OpenAI 2020)

12 Active Learning Literature Survey (Burr Settles, 2010)

13 Queries and Concept Learning (Dana Angluin, 1988)

14 Bridging AI's Proof-of-Concept to Production Gap (Andrew Ng, Stanford HAI 2020)

15 The Class Imbalance Problem: A Systematic Study (Nathalie Japkowciz and Shaju Stephen, 2002)

16 The Class Imbalance Problem: Significance and Strategies (Nathalie Japkowicz, 2000)

17 Facial action recognition using very deep networks for highly imbalanced class distribution (Ding et al., 2017)

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

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

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

21 Resampling strategies for imbalanced datasets (Rafael Alencar, Kaggle 2018)

22 An Experiment with the Edited Nearest-Neighbor Rule (Ivan Tomek, IEEE 1876)

23 “Convex” here approximately means “linear”.

24 KNN Approach to Unbalanced Data Distributions: A Case Study Involving Information Extraction (Zhang and Mani, 2003)

25 Addressing the curse of imbalanced training sets: one-sided selection (Kubat and Matwin, 2000)

26 Plankton classification on imbalanced large scale

database via convolutional neural networks with transfer learning (Lee et al., 2016)

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

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

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

30 Focal Loss for Dense Object Detection (Lin et al., 2017)

31 ImageNet Classification with Deep Convolutional Neural Networks (Krizhevsky et al., 2012)

32 One pixel attack for fooling deep neural networks (Su et al., 2017)

33 Explaining and Harnessing Adversarial Examples (Goodfellow et al., 2015)

34 Maxout Networks (Goodfellow et al., 2013)

35 DeepFool: a simple and accurate method to fool deep neural networks (Moosavi-Dezfooli et al., 2016)

36 Virtual Adversarial Training: A Regularization Method for Supervised and Semi-Supervised Learning (Miyato et al., 2017)

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

38 mixup: BEYOND EMPIRICAL RISK MINIMIZATION (Zhang et al., 2017)

39 Data augmentation using generative adversarial networks (CycleGAN) to improve generalizability in CT segmentation tasks | Scientific Reports (Sandfort et al., Nature 2019)

Chapter 4. Feature Engineering

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter,

please reach out to the author at *chip@huyenchip.com*.

In 2014, the paper Practical Lessons from Predicting Clicks on Ads at Facebook claimed that having the right features is the most important thing in developing their ML models. Since then, many of the companies that I've worked with have discovered time and time again that once they have a workable model, having the right features tends to give them the biggest performance boost compared to clever algorithmic techniques such as hyperparameter tuning. State-of-the-art model architectures can still perform poorly if they don't use a good set of features.

Due to its importance, a large part of many ML engineering and data science jobs is to come up with new useful features. In this chapter, we will go over common techniques and important considerations with respect to feature engineering. We will dedicate a

section to go into detail about a subtle yet disastrous problem that has derailed many ML systems in production: data leakage and how to detect and avoid it.

We will end the chapter discussing how to engineer good features, taking into account both the feature importance and feature generalization.

Learned Features vs. Engineered Features

When I cover this topic in class, my students frequently ask: “Why do we have to worry about feature engineering? Doesn’t deep learning promise us that we no longer have to engineer features?”

They are right. The promise of deep learning is that we won’t have to handcraft features. For this reason, deep learning is sometimes called feature learning[12]. Many features can be automatically learned and extracted by

algorithms. However, we're still far from the point where all features can be automated. Let's go over an example to understand what features can be automatically extracted and what features still need to be handcrafted.

Imagine that you want to build a sentiment analysis classifier to classify whether a comment is spam or not. Before deep learning, when given a piece of text, you would have to manually apply classical text processing techniques such as lemmatization, expanding contraction, removing punctuation, and lowercasing everything. After that, you might want to split your text into n-grams with n values of your choice.

As a refresher, an n-gram is a contiguous sequence of n items from a given sample of text. The items can be phonemes, syllables, letters, or words. For example, given the post "I like food", its word-level 1-grams are ["I", "like", "food"] and its word-level 2-grams are ["I like", "like food"]. This sentence's set of n-gram features, if we want n to be 1 and 2,

is: [“I”, “like”, “food”, “I like”, “like food”].

Figure 4-1 shows an example of classical text processing techniques you can use to handcraft n-gram features for your text.

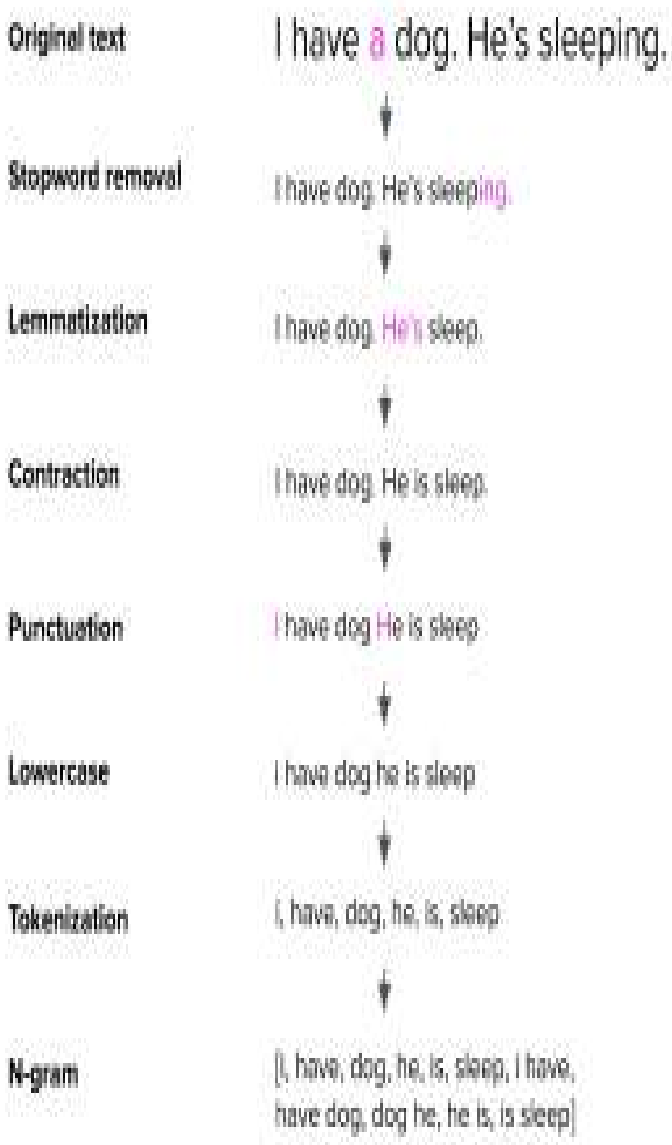

Original text
I have a dog. He's sleeping.
Stopword removal
I have dog. He's sleeping.
Lemmatization
I have dog. He's sleep.
Contraction
I have dog. He is sleep.
Punctuation
I have dog He is sleep
Lowercase
I have dog he is sleep
Tokenization
[I, have, dog, he, is, sleep]
N-gram
[I, have, dog, he, is, sleep, I have, have dog, dog he, he is, is sleep]

Figure 4-1. An example of techniques that you can use to handcraft n-gram features for your text

Once you've generated n-grams for your training data, you can create a vocabulary that matches each n-gram to an index. Then you can convert each post into a vector based on its n-grams' indices. For example, if we have a vocabulary of 7 n-grams as shown in Table 4-1, each post can be a vector of 7 elements. Each element corresponds to the number of times the n-gram at that index appears in the post. "I like food" will be encoded as the vector [1, 1, 0, 1, 1, 0, 1]. This vector can then be inputted into an ML model such as logistic regression.

Table 4-1. Example of an 1-gram and 2-gram vocabulary

I	like	good
0	1	2

Feature engineering requires knowledge of domain-specific techniques—in this case, the

domain in natural language processing (NLP)—and tends to be repetitive and error-prone. When I followed this method for one of my early NLP projects, I kept having to restart my process either because I had forgotten to apply one technique or because one technique I used turned out to be working poorly and I had to undo it.

However, much of this pain has been alleviated since the rise of deep learning. Instead of having to worry about lemmatization, punctuation, or stopword removal, you can just split your raw text into words, create a vocabulary out of those words, and convert each of your words into one-hot vectors using this vocabulary. Your model will hopefully learn to extract useful features from this. In this new method, much of feature engineering for text has been automated. Similar progress has been made for images too. Instead of having to manually extract features from raw images and input those features into your ML models, you can

just input raw images directly into your deep learning models.

However, an ML system will likely need data beyond just text and images. For example, when detecting whether a comment is spam or not, on top of the text in the comment itself, you might want to use other information about:

- the comment: such as who posted this comment, how many upvotes/downvotes it has.
- the user who posted this comment: such as when this account was created, how often they post, how many upvotes/downvotes they have.
- the thread in which the comment was posted: such as how many views it has, because popular threads tend to attract more spam.

Comment ID	Time	User	Text	# ▲	# ▼	Link	# img	Thread ID	Reply to	# replies	...
1000000000	2020-05-20 T 19:08:15.0Z	gjones	Yes it is on a mountain	1	1	0	0	2000000	1000000	1	...

User ID	Created	User	Bubo	# ▲	# ▼	# replies	Karma	# threads	Verified email	Awards	...
4400000	2009-01-07T 1:08:19ZT	gjones	Java, streams, monadic	15	10	28	564	179	No		...

Thread ID	Time	User	Text	# ▲	# ▼	Link	# img	# replies	# views	Awards	...
1000001000	2020-05-20 T 2:16:19ZT	ebegin	Hamlet is temporary. Kill is forever.	120	10	1	0	12	3400	1	...

Figure 4-2. Some of the possible features about a comment, a thread, or a user to be included in your model

There are so many possible features to use in your model, some of them are shown in Figure 4-2. The process of choosing what to use and extracting the information you want to use is feature engineering. For important tasks such as recommending videos for users to watch next on Tiktok, the number of features used can go up to millions. For domain-specific tasks such as predicting whether a transaction is fraudulent, you might need subject matter expertise with banking and frauds to be able to extract useful features.

Common Feature Engineering Operations

Because of the importance and the ubiquity of feature engineering in ML projects, there have been many techniques developed to streamline the process. In this section, we

will discuss several of the most important operations that you should consider, if you haven't already, while engineering features from your data. They include handling missing values, scaling, discretization, encoding categorical features, generating the old-school but still very effective cross features as well as the newer and exciting positional features. This list is nowhere near being comprehensive, but it does comprise some of the most common and useful operations to give you a good starting point. Let's dive in!

Handling Missing Values

One of the first things you might notice when dealing with data in production is that some values are missing. However, one thing that many ML engineers I've interviewed don't know is that not all types of missing values are equal[3]. To illustrate this point, consider the task of predicting whether someone is going to buy a house in the next 12 months.

A portion of the data we have is in Table 4-2.

Table 4-2. A portion of the data we have for the task of predicting whether someone will buy a house in the next 12 months

ID	Age	Gender
1		A
2	27	B
3		A
4	40	B
5	35	B
6		A
7	33	B
8	20	B

There are three types of missing values. The official names for these types are a little bit confusing so we'll go into detailed examples to mitigate the confusion.

1. Missing not at random (MNAR): when the reason a value is missing is because of the value itself. In this

example, we might notice that male respondents with higher income tend not to disclose their income. The income values are missing for reasons related to the values themselves.

2. Missing at random (MAR): when the reason a value is missing is not due to the value itself, but due to another observed variable. In this example, we might notice that age values are often missing for respondents of the gender “A”, which might be because the people of gender A in this survey don’t like disclosing their age.

3. Missing completely at random (MCAR): when there’s no pattern in when the value is missing. In this example, we might think that the missing values for the column “Job” might be completely random, not because of the job itself and not because of another variable. People

just forget to fill in that value sometimes for no particular reason. However, this type of missing is very rare. There are usually reasons why certain values are missing, and you should investigate.

When encountering missing values, you can either fill in the missing values with certain values, (imputation), or remove the missing values (deletion). We'll go over both.

Deletion

The candidates who I've talked to tend to prefer deletion, not because it's a better method, but because it's easier to do.

One way to delete is **column deletion**: if a variable has too many missing values, just remove that variable. For example, in the example above, over 50% of the values for the variable "Marital status" are missing, so you might be tempted to remove this variable from your model. The drawback of this

approach is that you might remove important information and reduce the accuracy of your model. Marital status might be highly correlated to buying houses, as married couples are much more likely to be homeowners than single people[4].

Another way to delete is **row deletion**: if an example has missing value(s), just remove that example from the data. This method can work when the missing values are completely at random (MCAR) and the number of examples with missing values is small, such as less than 0.1%. You don't want to do row deletion if that means 10% of your data examples are removed.

However, removing rows of data can also remove important information that your model needs to make predictions, especially if the missing values are not at random (MNAR). For example, you don't want to remove examples of male respondents with missing income because whether income is missing is information itself (missing income

might mean higher income, and thus, more correlated to buying a house) and can be used to make predictions.

On top of that, removing rows of data can create biases in your model, especially if the missing values are at random (MAR). For example, if you remove all examples missing age values in the data in Table 4-2, you will remove all respondents with gender A from your data, and your model won't be able to make predictions for respondents with gender A.

Imputation

Even though deletion is tempting because it's easy to do, deleting data can lead to losing important information or cause your model to be biased. If you don't want to delete missing values, you will have to impute them, which means "fill them with certain values." Deciding which "certain values" to use is the hard part.

One common practice is to fill in missing

values with their defaults. For example, if the number of children is missing, you might fill it with 0. If the job is missing, you might fill it with an empty string “”. This approach works well in many cases, but sometimes, it can cause hair-splitting bugs. One time, in one of the projects I was helping with, we discovered that the model was spitting out garbage because the app’s front-end no longer asked users to enter their age, so age values were missing, and the model filled them with 0. But the model never saw the age value of 0 during, so it couldn’t make reasonable predictions.

Another common practice is to fill in missing values with the mean, median, or most common value. For example, if the temperature value is missing for a data example whose month value is July, it’s not a bad idea to fill it with the median temperature of July.

But what if the value isn’t numerical and you can’t calculate its mean or median? If your

values are categorical, you can give each of the categories a numerical value, then calculate the mean or median of those numerical values.

Multiple techniques might be used at the same time or in sequence to handle missing values for a particular set of data. Regardless of what techniques you use, one thing is certain: there is no perfect way to handle missing values. With deletion, you risk losing important information or accentuating biases. With imputation, you risk adding noise to your data, or worse, data leakage. If you don't know what data leakage is, don't panic, we'll cover it in the **Data Leakage** section of this chapter.

Scaling

Consider the task of predicting whether someone will buy a house in the next 12 months, and the data is shown in Table 4-2. The values of the variable Age in our data go between 20 and 40, whereas the values of the

variable Annual Income go between 10,000 and 150,000. When we input these two variables into an ML model, it won't understand that 150,000 and 40 represent different things. It will just see them both as numbers, and because the number 150,000 is much bigger than 40, it might give it more importance, regardless of which variable is actually more useful for the predicting task.

During data processing, it's important to scale your features so that they're in similar ranges. This process is called feature scaling. This is one of the simplest things you can do to give your model a performance boost. Neglecting to do so can cause your model to make gibberish predictions, especially with classical algorithms like gradient-boosted trees and logistic regression[5].

An intuitive way to scale your features is to get each feature to be in the range [0, 1]. Given a variable x, its values can be rescaled to be in this range using the following formula.

$$x\hat{a} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

You can validate that if x is the maximum value, the scaled value x’ will be 1. If x is the minimum value, the scaled value x’ will be 0.

If you want your feature to be in an arbitrary range [a, b] — empirically, I find the range [-1, 1] to work better than the range [0, 1] — you can use the following formula.

$$x\hat{a} = a + \frac{(x - \min(x))(b - a)}{\max(x) - \min(x)}$$

Scaling to an arbitrary range works well when you don’t want to make any assumptions about your variables. If you think that your variables might follow a normal distribution, it might be helpful to normalize them so that they have zero-mean and unit variance. This process is called standardization.

–

$$x\hat{a} = \frac{x - \bar{x}}{\sigma}$$

with $\bar{x}$ being the mean of variable x, and σ being its standard deviation.

There are two important things to note about scaling. One is that it's a common source of data leakage, (this will be covered in greater detail in the Data Leakage section). Another is that it often requires global statistics — you have to look at the entire or a subset of training data to calculate its min, max, or mean. During inference, you reuse the statistics you had obtained during training to scale new data. If the new data has changed significantly compared to the training, these statistics won't be very useful. Therefore, it's important to retrain your model often to account for these changes. We'll discuss more on how to handle changing data in production in the section on continual learning in Chapter 7.

Discretization

Imagine that we've built a model with the data in Table 4-2. During training, our model has seen the annual income values of 150000, 50000, 100000, 50000, 60000, and 10000. During inference, our model encounters an example with an annual income of 9000.50.

Intuitively, we know that $9000.50 a year isn't much different from $10,000/year, and we want our model to treat both of them the same way. But the model doesn't know that. Our model only knows that 9000.50 is different from 10000, and will treat them differently.

Discretization is the process of turning a continuous feature into a discrete feature. This process is also known as quantization. This is done by creating buckets for the given values. For annual income, you might want to group them into three buckets as follows.

- Lower income: less than $35,000/year

- Middle income: between $35,000 and $100,000/year
- Upper income: more than $100,000/year

Now, instead of having to learn an infinite number of possible incomes, our model can focus on learning only three categories, which is a much easier task to learn.

Even though by definition, it’s used for continuous features, the same technique can be used for discrete features too. The age variable is discrete, it might still be useful to group them into buckets such as follows.

- Less than 18
- Between 18 and 22
- Between 22 and 30
- Between 30 - 40
- Between 40 - 65
- Over 65

A question with this technique is how to best choose the boundaries of categories. You can try to plot the histograms of the values and choose the boundaries that make sense. In general, common sense, basic quantiles, and sometimes subject matter expertise can get you a long way.

Encoding Categorical Features

We've talked about how to turn continuous features into categorical features. In this section, we'll discuss how to best handle categorical features.

People who haven't worked with data in production tend to assume that categories are *static*, which means the categories don't change over time. This is true for many categories. For example, age brackets and income brackets are unlikely to change and you know exactly how many categories there are in advance. Handling these categories is straightforward. You can just give each category a number and you're done.

However, in production, categories change. Imagine you're building a recommendation system to predict what products users might want to buy for Amazon. One of the features you want to use is the product brand. When looking at Amazon's historical data, you realize that there are a lot of brands. Even back in 2019, there were already over 2 million brands on Amazon[6]!

The number of brands is overwhelming but you think: "I can still handle this." You encode each brand a number, so now you have 2 million numbers from 0 to 1,999,999 corresponding to 2 million brands. Your model does spectacularly on the historical test set, and you get approval to test it on 1% of today's traffic.

In production, your model crashes because it encounters a brand it hasn't seen before and therefore can't encode. New brands join Amazon all the time. You create a category "UNKNOWN" with the value of 2,000,000 to

catch all the brands your model hasn’t seen during training.

Your model doesn’t crash anymore but your sellers complain that their new brands are not getting any traffic. It’s because your model didn’t see the category UNKNOWN in the train set, so it just doesn’t recommend any product of the UNKNOWN brand. Then you fix this by encoding only the top 99% most popular brands and encode the bottom 1% brand as UNKNOWN. This way, at least your model knows how to deal with UNKNOWN brands.

Your model seems to work fine for about 1 hour, then the click rate on recommended products plummets. Over the last hour, 20 new brands joined your site, some of them are new luxury brands, some of them are sketchy knockoff brands, some of them are established brands. However, your model treats them all the same way it treats unpopular brands in the training set.

This isn’t an extreme example that only happens if you work at Amazon on this task. This problem happens quite a lot. For example, if you want to predict whether a comment is spam, you might want to use the account that posted this comment as a feature, and new accounts are being created all the time. The same goes for new product types, new website domains, new restaurants, new companies, new IP addresses, and so on. If you work with any of them, you’ll have to deal with this problem.

Finding a way to solve this problem turns out to be surprisingly difficult. You don’t want to put them into a set of buckets because it can be really hard—how would you even go about putting new user accounts into different groups?

One solution to this problem is **the hashing trick**, popularized by the package Vowpal Wabbit developed at Microsoft[7]. The gist of this trick is that you use a hash function to generate a hashed value of each category. The

hashed value will become the index of that category. Because you can specify the hash space, you can fix the number of encoded values for a feature in advance, without having to know how many categories there will be. For example, if you choose a hash space of 18 bits, which corresponds to 2^{18} = 262,144 possible hashed values, all the categories, even the ones that your model has never seen before, will be encoded by an index between 0 and 262, 143.

One problem with hashed functions is collision: two categories being assigned the same index. However, with many hash functions, the collisions are random, new brands can share index with any of the old brands instead of always sharing index with unpopular brands, which is what happens when we use the UNKNOWN category above. The impact of colliding hashed features is, fortunately, not that bad. In a research done by Booking.com, even for 50% colliding features, the performance lost is less

than half percent, as shown in Figure 4-3.

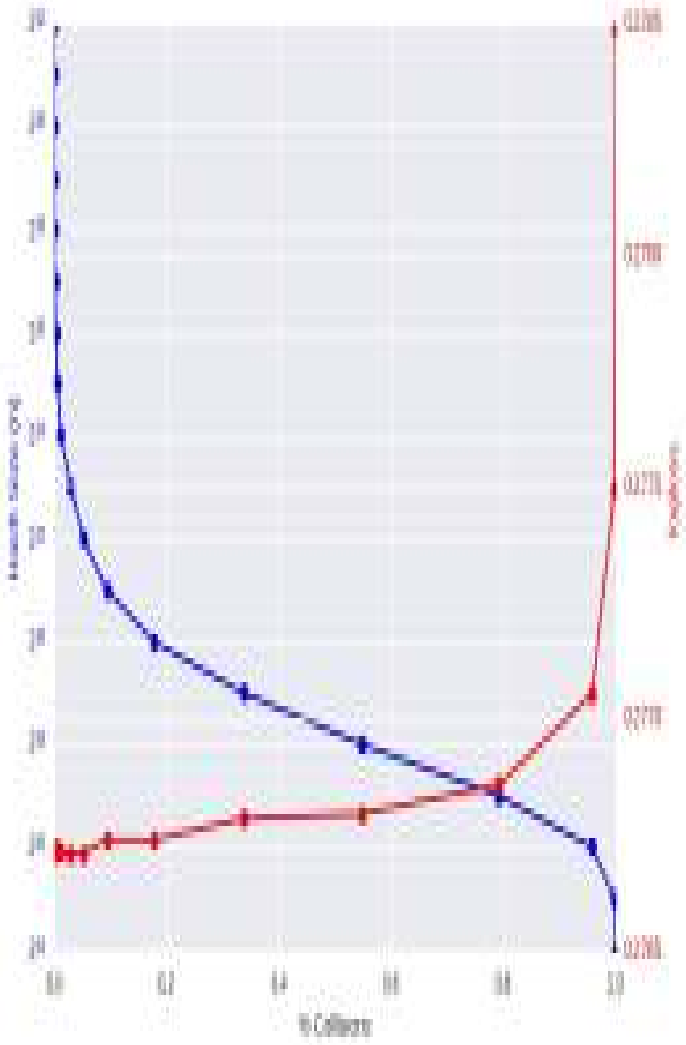


Figure 4-3. 50% collision rate only causes the log loss to increase less than half a percent. Image by Lucas Bernardi.

You can choose a hash space large enough to reduce the collision. You can also choose a hash function with properties that you want, such as a locality-sensitive hashing function where similar categories (such as websites with similar names) are hashed into values close to each other.

Because it's a trick, it's often considered hacky by academics and excluded from ML curricula. But its wide adoption in the industry is a testimonial to how effective the trick is. It's essential to Vowpal Wabbit and it's part of the frameworks scikit-learn, TensorFlow, and gensim. It can be especially useful in continual learning settings where your model learns from incoming examples in production. We'll cover continual learning in Chapter [TODO].

Feature Crossing

Feature crossing is the technique to combine two or more features to generate new features. This technique is useful to model the non-linear relationships between features. For example, for the task of predicting whether someone will want to buy a house in the next 12 months, you suspect that there might be a non-linear relationship between marital status and number of children, so you combine them to create a new feature “marriage and children” as in Table 4-3.

Table 4-3. Example of how two features can be combined to create a new feature

Marriage	Single	Married
Children	0	2
Marriage & children	Single, 0	Married, 2

Because feature crossing helps model non-linear relationships between variables, it’s essential for models that can’t learn or are bad at learning non-linear relationships, such

as linear regression, logistic regression, and tree-based models. It's less important in neural networks, but can still be useful because explicit feature crossing occasionally helps neural networks learn non-linear relationships faster.

A caveat of feature crossing is that it can make your feature space blow up. Imagine feature A has 100 possible values and feature B has 100 possible features, crossing these two features will result in a feature with 100 x 100 = 10,000 possible values. You will need a lot more data for models to learn all these possible values. Another caveat is that because feature crossing increases the number of features models use, it can make models overfit to the training data.

Discrete and Continuous Positional Embeddings

First introduced to the deep learning community in the paper Attention Is All You

Need (Vaswani et al., 2017), positional embedding has become a standard data engineering technique for many applications in both computer vision and natural language processing. We'll walk through an example to show why positional embedding is necessary and how to do it.

Consider the task of language modeling where you want to predict the next token based on the previous sequence of tokens. In practice, a token can be a word, a character, or a subword, and a sequence length can be up to 512 if not larger. However, for simplicity, let's use words as our tokens and use the sequence length of 8. Given an arbitrary sequence of 8 words, such as, "*Sometimes all I really want to do is*", we want to predict the next word.

If we use a recurrent neural network, it will process words in sequential order, which means the order of words is implicitly inputted. However, if we use a model like a transformer, words are processed in parallel,

so words' positions need to be explicitly inputted so that our model knows which word follows which word ("a dog bites a child" is very different from "a child bites a dog"). We don't want to input the absolute positions: 0, 1, 2, …, 7 into our model because empirically, neural networks don't work well with inputs that aren't unit-variance (that's why we scale our features, as discussed previously in the section Scaling).

If we rescale the positions to between 0 and 1, so 0, 1, 2, …, 7 become 0, 0.143, 0.286, …, 1, the differences between the two positions will be too small for neural networks to learn to differentiate.

A way to handle position embeddings is to treat it the way we'd treat word embedding. With word embedding, we use an embedding matrix with the vocabulary size as its number of columns, and each column is the embedding for the word at the index of that column. With position embedding, the number of columns is the number of

positions. In our case, since we only work with the previous sequence size of 8, the positions go from 0 to 7 (see Figure 4-4).

The embedding size for positions is usually the same as the embedding size for words so that they can be summed. For example, the embedding for the word “food” at position 0 is the sum of the embedding vector for the word “food” and the embedding vector for position 0. This is the way position embeddings are implemented in HuggingFace’s BERT as of August 2021. Because the embeddings change as the model weights get updated, we say that the position embeddings are learned.

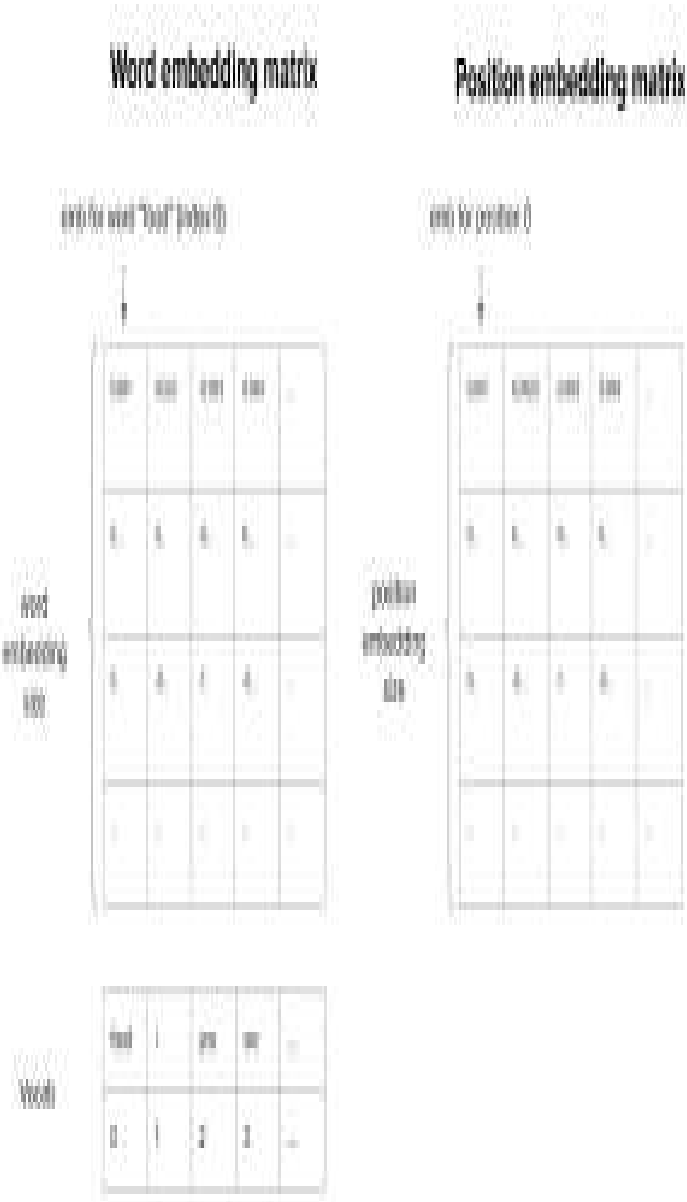

Word embedding matrix
Position embedding matrix
word embedding size
position embedding size
Words

Figure 4-4. One way to embed positions is to treat them the way you'd treat word embeddings

Position embeddings can also be fixed. The embedding for each position is still a vector with S elements (S is the position embedding size), but each element is predefined using a function, usually sine and cosine. In the original Transformer paper, if the element is at an even index, use sine. Else, use cosine. See Figure 4-5.

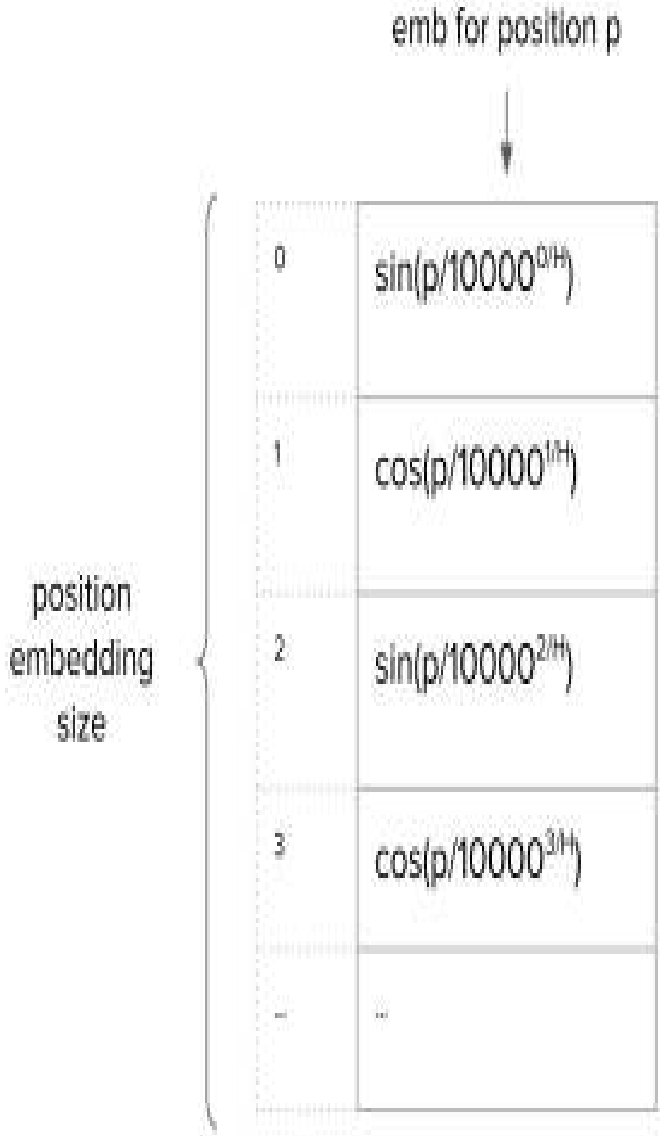

emb for position p
position embedding size
0
sin(p/10000^0/H)
1
cos(p/10000^1/H)
2
sin(p/10000^2/H)
3
cos(p/10000^3/H)
..
..

Figure 4-5. Example of fixed position embedding. H is the dimension of the outputs produced by the model.

Fixed positional embedding is a special case of what is known as Fourier features. If positions in positional embeddings are discrete, Fourier features can also be continuous. Consider the task involving representations of 3D objects, such as a teapot. Each position on the surface of the teapot is represented by a 3-dimensional coordinate, which is continuous. When positions are continuous, it'd be very hard to build an embedding matrix with continuous column indices, but fixed position embeddings using sine and cosine functions still work.

This is the generalized format for the embedding vector at coordinate v, also called the Fourier features of coordinate v. Fourier features have been shown to improve models' performance for tasks that take in coordinates (or positions) as inputs.[8]

$$\gamma\left(v\right)=\left[a_1\cos\left(2\pi{b_1}^Tv\right),a_1\sin\left(2\pi{b_1}^Tv\right)\right.$$

Data Leakage

One pattern of failures that I've encountered often in production is when models perform beautifully on the test set, but fail mysteriously in production. The cause, after a long and painful investigation, is data leakage.

Data leakage refers to the phenomenon when a form of the label "leaks" into the set of features used for making predictions, and this same information is not available during inference.

Data leakage is subtle because often, the leakage is non-obvious. It's dangerous because it can cause your models to fail in an unexpected and spectacular way, even after extensive evaluation and testing. Let's go over an example to demonstrate what data leakage is.

Suppose you want to build an ML model to predict whether a CT scan of a lung shows signs of cancer. You obtained the data from hospital A, removed the doctors' diagnosis from the data, and trained your model. It did really well on the test data from hospital A, but poorly on the data from hospital B.

After extensive investigation, you learned that at hospital A, when doctors think that a patient has lung cancer, they send that patient to a more advanced scan machine, which outputs slightly different CT scan images. Your model learned to rely on the information on the scan machine used to make predictions on whether a scan image shows signs of lung cancer. Hospital B sends the patients to different CT scan machines at random, so your model has no information to rely on. We say that labels are leaked into the features during training.

Data leakage can happen not only with newcomers to the field, but has also happened to several experienced researchers whose

work I admire, and in one of my own projects. Despite its prevalence, data leakage is rarely covered in ML curricula.

CAUTIONARY TALE: DATA LEAKAGE WITH KAGGLE COMPETITION

In 2020, University of Liverpool launched an Ion Switching competition on Kaggle. The task was to identify the number of ion channels open at each time point. They synthesized test data from train data, and some people were able to reverse engineer and obtain test labels from the leak[9]. The two winning teams in this competition are the two teams that were able to exploit the leak, though they might have still been able to win without exploiting the leak[10].

Common Causes for Data Leakage

In this section, we'll go over some common causes for data leakage and how to avoid them.

1. Splitting time-correlated data randomly instead of by time

   When I learned ML in college, I was taught to randomly split my data into train, validation, and test splits. This is also how data is often split in many ML research papers. However, this is also one common cause for data leakage.

   In many cases, data is time-correlated, which means that the time the data is generated affects how it should be labeled. Sometimes, the correlation is obvious, as in the case of stock prices. To oversimplify it, the prices of many stocks tend to go up and down together. If 90% of the stocks go down today, it's very likely the

other 10% of the stocks go down too. When building models to predict the future stock prices, you want to split your training data by time, such as training your model on data from the first 6 days and evaluating it on data from the 7th day. If you randomly split your data, prices from the 7th day will be included in your train split and leak into your model the condition of the market on that day. We say that the information from the future is leaked into the training process.

However, in many cases, the correlation is non-obvious but it's still there. Consider the task of predicting whether someone will click on a song commendation. Whether someone will listen to a song depends not only on their taste in music but also on the general music trend that day. If an artist

passes away one day, people will be much more likely to listen to that artist. By including examples from that day into the train split, information about the music trend that day will be passed into your model, making it easier for it to make predictions on other examples on that day.

To prevent future information from leaking into the training process and allowing models to cheat during evaluation on the test split, split your training data by time when you can, instead of random splitting. For example, if you have data from 5 weeks, use the first 4 weeks for the train split, then randomly split week 5 into validation and test splits as shown in Figure 4-6.

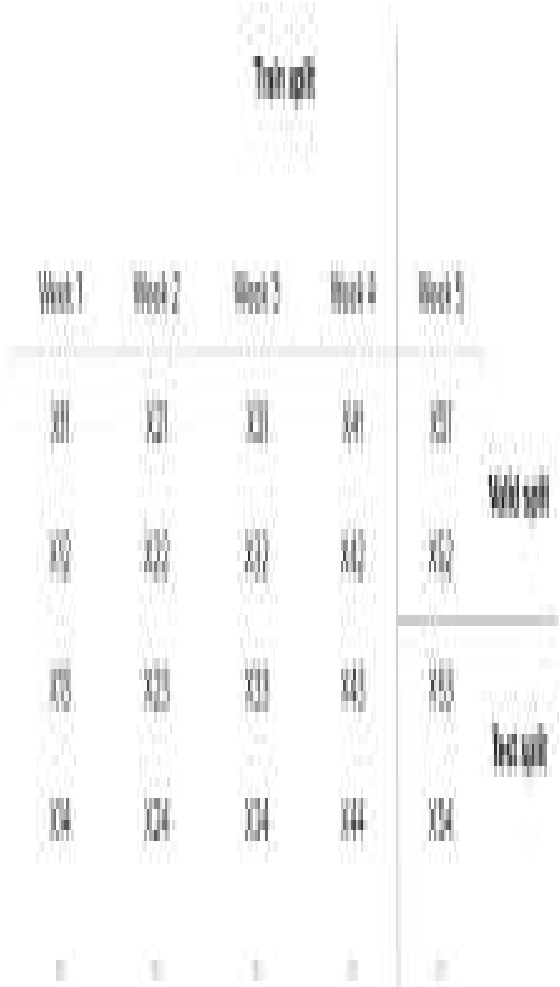

Month eight
Week 1
Week 2
Week 3
Week 4
Week 5
X11
X21
X31
X41
X51
X12
X22
X32
X42
X52
X13
X23
X33
X43
X53
X14
X24
X34
X44
X54
Middle month
Next month

Figure 4-6. Split data by time to prevent future information from leaking into the training process

2. Scaling before splitting

   As discussed in the Scaling section in this chapter, it's important to scale your features. Scaling requires global statistics about your data, such as the mean of your data. One common mistake is to use the entire training data to generate global statistics before splitting it into different splits, leaking the mean of the test examples into the training process, allowing a model to adjust to the mean of the future examples. This information isn't available in production so the model's performance will likely degrade.

   To avoid this type of leakage, always split your data first before scaling, then use the statistics from the train split to scale all the splits.

3. Filling in missing data with statistics from the test split

   One common way to handle the missing values of a feature is to fill them with the mean or median of all values present. Leakage might occur if the mean or median is calculated using entire data instead of just the train split. This type of leakage is similar to the type of leakage above, and can be prevented by using only statistics from the train split to fill in missing values in all the splits.

4. Poor handling of data duplication before splitting

   Data duplication is quite common. It can happen because the data handed to you has duplication in it[11], which likely results from data collection or merging of different data sources. It can also happen because of data processing — for example,

oversampling might result in duplicating certain examples. If you fail to remove duplicates before splitting your data into different splits, the same examples might appear in both the train and the validation/test splits. To avoid this, first, always check for duplicates before splitting and also after splitting just to make sure. If you oversample your data, do it only after splitting.

5. Group leakage

   A group of examples have strongly correlated labels but are divided into different splits. For example, a patient might have two lung CT scans that are a week apart, which likely have the same labels on whether they contain signs of lung cancer, but one of them is in the train split and the third is in the test split. To avoid this type of data leakage,

an understanding of your data is essential.

6. Leakage from data collection process

   The example above about how information on whether a CT scan shows signs of lung cancer is leaked via the scan machine is an example of this type of leakage. Detecting this type of data leakage requires a deep understanding of the way data is collected and occasionally, subject matter expertise in the task. For example, it would be very hard to figure out that the model's poor performance in hospital B is due to its different scan machine procedure if you don't know about different scan machines or that the procedures at the two hospitals are different.

   There's no foolproof way to avoid this type of leakage, but you can mitigate the risk by keeping track of

the sources of your data, and understanding how it is collected and processed. Normalize your data so that data from different sources can have the same means and variances. If different CT scan machines output images with different resolutions, normalizing all the images to have the same resolution would make it harder for models to know which image is from which scan machine.

Detecting Data Leakage

Data leakage can happen during many steps, from collecting, sampling, splitting, processing data to feature engineering. It's important to monitor for data leakage during the entire lifecycle of an ML project.

Measure how each feature or a set of features are correlated to the target variable. If a feature has unusually high correlation, investigate how this feature is generated and whether the correlation makes sense. It's

possible that two features independently don’t contain leakage, but two features together can contain leakage. For example, the starting date and the end date separately doesn’t tell us anything about how long someone is at a company, but both together can give us that information.

Do ablation studies to measure how important a feature or a set of features is to your model. If removing a feature causes the model’s performance to deteriorate significantly, investigate why that feature is so important. If you have a massive amount of features, say a million features, it might be infeasible to do ablation studies on every possible combination of them, but it can still be useful to occasionally do ablation studies with a subset of features that you suspect the most. This is another example why subject matter expertise can come in handy in feature engineering. Ablation studies can be run offline at your own schedule, so you can leverage your machines during down time for

this purpose.

Keep an eye out for new features added to your model. If adding a new feature significantly improves your model's performance, either that feature is really good or that feature just contains leaked information about labels.

Be very careful every time you look at the test split. If you use the test split in any way other than to report a model's final performance, whether to come up with ideas for new features or to tune hyperparameters, you risk leaking information from the future into your training process.

Engineering Good Features

Generally, adding more features leads to better model performance. In my experience, the list of features used for a model in production only grows over time. However, more features doesn't always mean better

model performance. Having too many features can be bad for your model for the following reasons.

1. The more features you have, the more opportunities there are for data leakage.
2. Too many features can cause overfitting.
3. Useless features become technical debts. Whenever your data pipeline changes, all the affected features need to be adjusted accordingly. For example, if one day your application decides to no longer take in information about users' age, all features that use users' age need to be updated.

In theory, if a feature doesn't contribute to help a model make good predictions, regularization techniques like L1 regularization should reduce that feature's

weight to 0. However, in practice, it might help models learn faster if the features that are no longer useful (and even possibly harmful) are removed, prioritizing good features.

You can store removed features to add them back later. You can also just store general features to reuse and share across teams in an organization. A service that helps you manage, store, and automate your feature engineer pipeline is called a Feature Store.

There are two factors you might want to consider when evaluating whether a feature is good for a model: importance to the model and generalization to unseen data.

Feature Importance

There are many different methods for measuring a feature's importance. Two of the popular ones are Boosted Feature Importances for boosted gradient trees and SHAP (SHapley Additive exPlanations) for

general models[12]. Their exact algorithms are complex, but intuitively, a feature's importance to a model is measured by how much that model's performance deteriorates if that feature or a set of features containing that feature is removed from the model. SHAP is great because not only it measures a feature's importance to an entire model, it also measures each feature's contribution to a model's specific prediction. Figure 4-7 and Figure 4-8 show how SHAP can help you understand the contribution of each feature to a model's predictions.


f(x) = 24.019
4.98 = LSTAT
+5.70
6.575 = RM
−2.37
0.538 = NOX
−0.73
1 = RAD
−0.49
296 = TAX
−0.47
0.006 = CRIM
−0.43
4.09 = DIS
−0.41
15.3 = PTRATIO
+0.26
65.2 = AGE
+0.19
4 other features
−0.04
19
20
21
22
23
24
E[f(X)] = 22.533

Figure 4-7. How much each feature contributes to a model's single prediction, measured by SHAP. The value LSTAT=4 contributes the most to this specific prediction. Image by Scott Lundberg from the GitHub repository github.com/slundberg/shap.

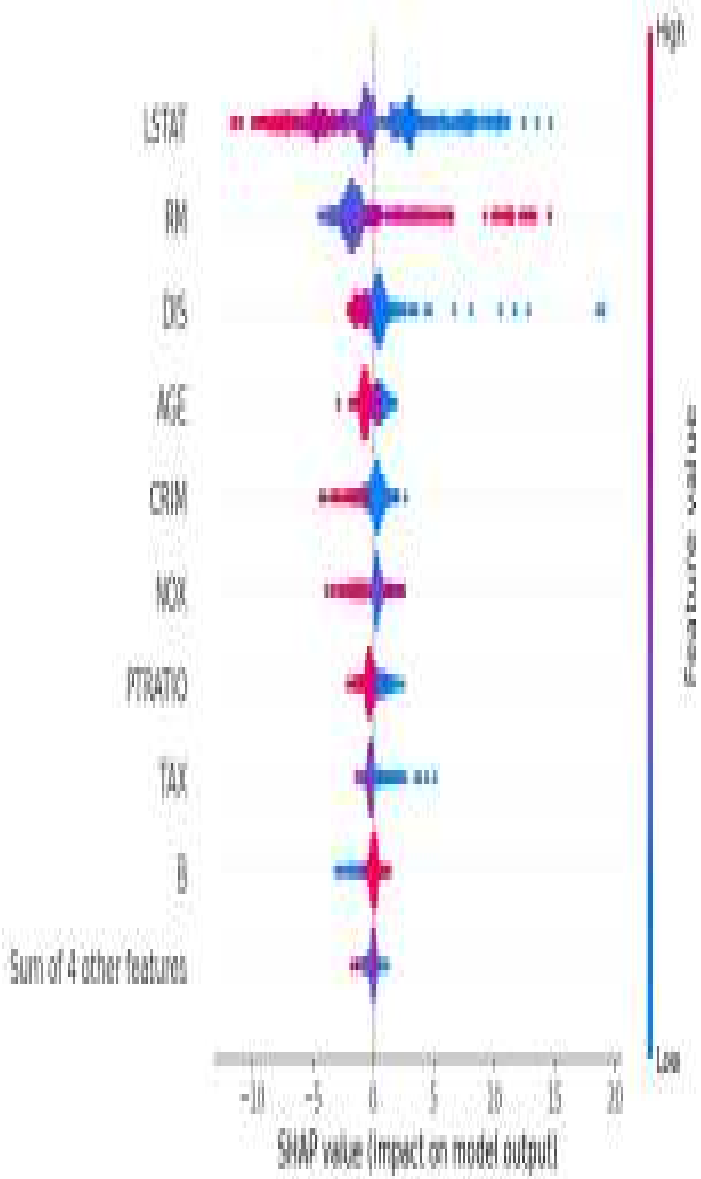

LSTAT
RM
DIS
AGE
CRIM
NOX
PTRATIO
TAX
B
Sum of 4 other features
−10
−5
0
5
10
15
20
SHAP value (impact on model output)
High
Low
Feature value

Figure 4-8. How much each feature contributes to a model, measured by SHAP. The feature LSTAT has the highest importance. Image by Scott Lundberg from the GitHub repository github.com/slundberg/shap.

Often, a small number of features accounts for a large portion of your model's feature importance. When using Boosted Feature Importance for a click-through rate prediction model, the ads team at Facebook found out that the top 10 features are responsible for about half of the model's total feature importance, while the last 300 features contribute less than 1% feature

Importance, as shown in Figure 4-9[13].

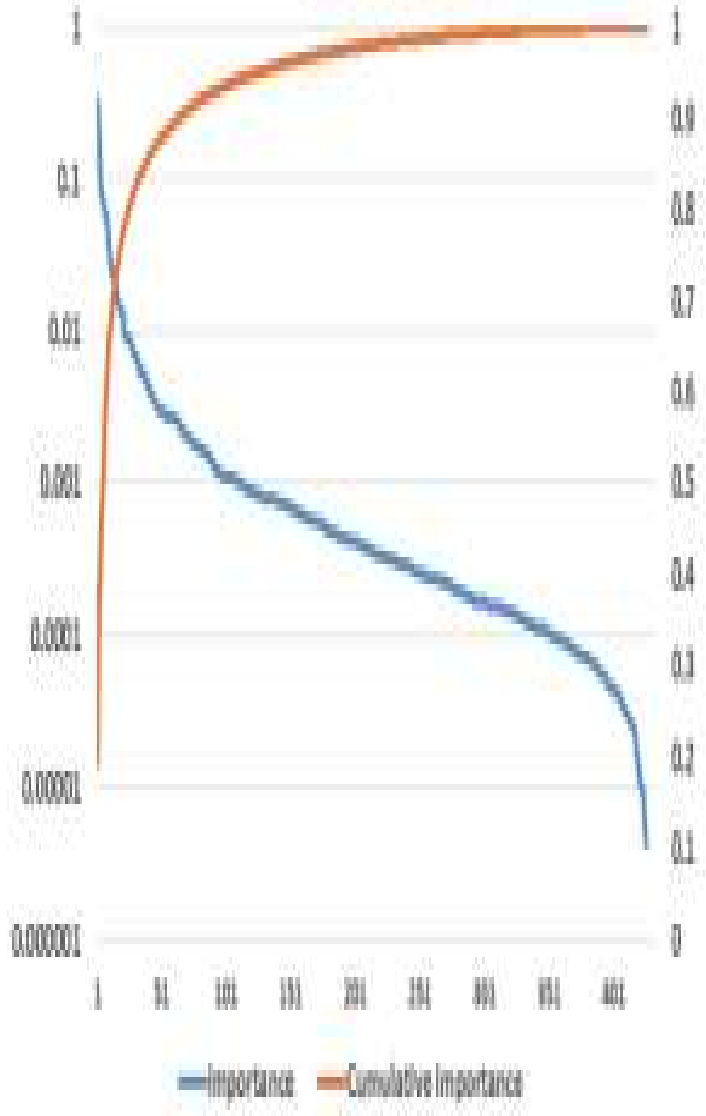

1
0.1
0.01
0.001
0.0001
0.00001
0.000001
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
1
91
181
271
361
451
541
631
721
Importance
Cumulative Importance

Figure 4-9. Boosting Feature Importance. X-axis corresponds to the number of features. Feature importance is in log scale. Image by He et al.

Not only good for choosing the right features, feature importance techniques are also great for interpretability as they help you understand how your models work under the hood. We will discuss more about Interpretability in Chapter [TODO].

Feature Generalization

Since the goal of an ML model is to make correct predictions on unseen data, features used for the model should be generalizable to unseen data. Not all features generalize equally. For example, for the task of predicting whether a comment is spam, the identifier of each comment is not generalizable at all and shouldn't be used as a feature for the model. However, the identifier of the user who posts the comment, such as username, might still be useful for a model to make predictions.

Measuring feature generalization is a lot less scientific than measuring feature importance, and requires more intuition and subject matter expertise than statistical knowledge. Overall, there are two aspects you might want to consider with regards to generalization: feature coverage and distribution of feature values.

Coverage is the percentage of the examples that has values for this feature in the data. A rough rule of thumb is that if this feature appears in a very small percentage of your data, it's not going to be very useful. For example, if you want to build a model to predict whether someone will buy a house in the next 12 months and you think that the number of children someone has will be a good feature, but you can only get this information for 1% of your data, this feature might not be very useful.

This rule of thumb is rough because some features can still be useful even if they are missing in most of your data. This is

especially useful if the missing values are not at random, which means having the feature or not might be a strong indication of its value). For example, if a feature appears only in 1% of your data, but 99% of the examples with this feature have POSITIVE labels, this feature is very useful and you should use it.

Coverage of a feature can differ wildly between different slices of data and in the same slice of data over time. If the coverage of a feature differs a lot between the train and test split (such as it appears in 90% of the examples in the train split but only in 20% of the examples in the test split), this is an indication that your train and test splits don't come from the same distribution. You might want to investigate whether the way you split your data makes sense and whether this feature is a cause for data leakage.

For the feature values that are present, you might want to look into their distribution. If the set of values that appears in the seen data (such as the train split) has no overlap with

the set of values that appears in the unseen data (such as the test split), this feature might even hurt your model's performance.

As a concrete example, imagine you want to build a model to estimate the time it will take for a given taxi ride. You retrain this model every week, and you want to use the data from the last 6 days to predict the ETAs[14] for today. One of the features is DAY_OF_THE_WEEK, which you think is useful because the traffic on weekdays is usually worse than on the weekend. This feature coverage is 100%, because it's present in every feature. However, in the train split, the values for this feature are Monday to Saturday, while in the test split, the value for this feature is Sunday. If you include this feature in your model, it won't generalize to the test split, and might harm your model's performance.

On the other hand, HOUR_OF_THE_DAY is a great feature, because the time in the day affects the traffic too, and the range of values

for this feature in the train split overlaps with the test split 100%.

When considering a feature's generalization, there's a tradeoff between generalization and specificity. You might realize that the traffic during an hour only changes depending on whether that hour is the rush hour. So you generate the feature IS_RUSH_HOUR and set it to 1 if the hour is between 7am and 9am or between 4pm and 6pm. IS_RUSH_HOUR is more generalizable but less specific than HOUR_OF_THE_DAY. Using IS_RUSH_HOUR without HOUR_OF_THE_DAY might cause models to lose important information about the hour.

Summary

Because the success of today's ML systems still depend on their features, it's important for organizations interested in using ML in production to invest time and effort into feature engineering.

How to engineer good features is a complex question with no fool-proof answers. The best way to learn is through experience: trying out different features and observing how they affect your models' performance. It's also possible to learn from experts. I find it extremely useful to read about how the winning teams of Kaggle competitions engineer their features to learn more about their techniques and the considerations they went through.

Feature engineering often involves subject matter expertise, and subject matter experts might not always be engineers, so it's important to design your workflow in a way that allows non-engineers to contribute to the process.

Here is a summary of best practices for feature engineering.

- Split data by time into train/valid/test spits instead of doing it randomly.
- If you oversample your data, do it

after splitting.

- Scale and normalize your data to avoid data leakage.
- Use statistics from only the train split, instead of the entire data, to scale your features and handle missing values.
- Understand how your data is collected and processed, and keep track of its lineage.
- Understand feature importance to your model.
- Use features that generalize well.
- Remove no longer useful features from your models.

With a set of good features, we'll move to the next part of the workflow: training ML models. Before we move on, I just want to reiterate that moving to modeling doesn't mean we're done with handling data or

feature engineering. We are never done with data and features. In most real-world ML projects, the process of collecting data and feature engineering goes on as long as your models are in production. We need to use new, incoming data to continually improve models. We'll cover continual learning in Chapter [TODO].

---

1 Handcrafted vs. non-handcrafted features for computer vision classification (Nanni et al., 2017)

2 Feature learning (Wikipedia)

3 In my experience, how well a person handles missing values for a given dataset during interviews strongly correlates with how well they will do in their day to day job.

4 3 FACTS ABOUT MARRIAGE AND HOMEOWNERSHIP (Rachel Bogardus Drew, Joint Center for Housing Studies at Harvard University 2014)

5 Feature scaling once boosted my model's performance by almost 10%.

6 Jun 11, 2019 Two Million Brands on Amazon (Marketplace Pulse)

7 Feature hashing (Wikipedia)

8 Fourier features let networks learn high frequency functions in low dimensional domains (Tancik et al., 2020)

9 The leak explained! (Zidmie, Kaggle).

10 Competition Recap - Congratulations to our Winners! (Kaggle)

11 Many commonly used datasets have duplicates in them. For example, 3.3% and 10% of the images from the test sets of the CIFAR-10 and CIFAR-100 datasets have duplicates in the training set.

12 A great open-source Python package for calculating SHAP can be found at https://github.com/slundberg/shap

13 Practical Lessons from Predicting Clicks on Ads at Facebook (He et al., 2014)

14 Estimated time of arrival.

Chapter 5. Model Development

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter,

please reach out to the author at *chip@huyenchip.com*.

Now that we've walked through training data and an initial set of features, we can move to the ML part of machine learning systems. Many ML engineers consider the fun part. This is when you can see your data being transformed into intelligent systems and play around with their predictions.

I expect that most readers already have an understanding of common ML algorithms such as decision trees, K-nearest neighbors, and different types of neural networks. This chapter will discuss techniques surrounding algorithms but won't explain these algorithms. Because this chapter deals with ML algorithms, it requires a lot more ML knowledge than other chapters. If you're not familiar with them, I recommend taking an online course or reading a book on ML algorithms before reading this chapter.

In this chapter, we'll first discuss how to

select the best model for your problem. We discuss how to select a model, out of many possible ML models, for your problem. Once we have our model, we'll discuss how to train it, covering distributed training and experiment tracking. We'll end the chapter with how to evaluate the trained models.

Model Selection

There are many possible solutions to any given problem, both ML solutions and non-ML solutions. You might wonder what solutions are best for your problem. Should you start with the good old logistic regression? You've heard of a fancy new model that is supposed to be the new state-of-the-art for your problem, should you spend two weeks learning that process then three months implementing it? Should you try an ensemble of various decision trees?

If you had unlimited time and compute power, the rational thing to do would be to try

all possible solutions and see what is best for you. However, time and compute power are limited resources, and you have to be strategic about what models we select.

In this section, we'll start with a basic ML review then cover three topics: choosing a model for your problem, creating ensembles of multiple models for your problem, and then using ML to automatically choose a model best for your problem.

Before we get into the details of model selection, let's take a step back and review what a model consists of, and what to select when we say "model selection". If you're already familiar with different types of ML algorithms and model selection, you might want to go to the next section **Choosing ML Models**.

Basic ML Review

A model is a function that transforms inputs into outputs, which can then be used to make

predictions. For example, a binary text classification model might take sentences as inputs and output values between 0 and 1. You can use these output values to make predictions, such as if the value is less than 0.5, output the NEGATIVE class, and if the value is greater than or equal to 0.5, output the POSITIVE class.

In traditional programming, functions are given and outputs are calculated from given inputs. For example, your function f(x) might be given as: f(x) = 2x + 3. Given x = 1, the output will be f(1) = 2 * 1 + 3 = 5. Given x = 3, the output will be f(3) = 2 * 3 + 3 = 9.

In supervised ML, the inputs and outputs are given, which are called data, and the function is derived from data. Given x as input and y as output, you want to learn a function f such that applying f on x will produce y. However, ML isn't powerful enough to derive arbitrary functions from data yet, so you still need to specify the form that you think the function should take[1]. It can be a linear function, a

decision tree, a feedforward neural network with two hidden layers, each with 768 neurons[2].

For example, given a dataset with only two examples (x = 1, y = 5) and (x = 3, y = 9), you might specify that the function is a linear function, which means that it takes the form f(x) = wx + b. Then you learn the values of w and b to fit this dataset. Because w and b are learned during the training process, they are called parameters.

For each type of model, there are many possible values for the parameters. You need an objective function to evaluate how good a given set of parameters is for a dataset, and a procedure to derive the set of parameters best suited for the given data according to that objective, known as a learning procedure.

> Some readers might wonder if the above paragraph about parameters still applies to non-parametric models such as K-

means clustering and decision trees. Being non-parametric doesn't mean that models don't use parameters. In a parametric model, the number of parameters is fixed with respect to the sample size. In a nonparametric model, the effective number of parameters can grow with the sample size. So the complexity of the function underlying a neural network remains the same even if the amount of data grows. But the complexity of the function underlying a decision tree grows as its number of nodes grows.

When talking about model selection, most people think about selecting a function form. However, choosing the right objective function and a learning procedure is extremely important in finding a good set of parameters for your model.

Objective Function

The **objective function**, also known as the loss function, is highly dependent on the model type and whether the labels are available. If the labels aren't available, as in the case of unsupervised learning, the objective functions depend on the data points themselves. For example, for k-means clustering, the objective function is the variance within data points in the same cluster (so the objective is to put data points into clusters so that the within-cluster variance is minimized). But unsupervised learning is much less commonly used in production.

Most algorithms you'll encounter in production are supervised or some form of weakly or semi-supervised, as mentioned in Chapter 3. Given a set of parameter values, you calculate the outputs from the given inputs, and compare the given function's predicted outputs (y') to the actual outputs (y). Objective functions evaluate how good a set of parameter values is by measuring the

distance between the set of y' and the set of y.

To make this concrete, let's go back to the example above where we have only 2 data points (x = 1, y = 5) and (x = 3, y = 9). We want to find w and b such that f(x) = wx + b best suited this data. Given the set of parameter values w = 3 and b = 4, we get the predicted outputs of 7 and 13 as shown in Table 5-1. The objective function measures the distance between the predicted outputs (7, 13) and the actual outputs (5, 9).

Table 5-1. Predicted outputs when w = 4 and b = 4

Input	Predicted output made by f(x | w=3, b=4) = 3x + 4	Actual output
x = 1	3 * 1 + 4 = 7	5
x = 3	3 * 3 + 4 = 13	9

There are many types of distance metrics you can use to derive your objective functions.

When the outputs are scalars (numbers), two common metrics are Root Mean Squared Error and Mean Absolute Error as shown in Table 5.2.

Table 5-2. Two common objective functions for scalar outputs

Objective function	How to calculate	Distance metrics
Root Mean Squared Error (RMSE)	$\sqrt{\sum_{i=1}^{n} \frac{(y_i\hat{a} - y_i)^2}{n}}$	Euclidean
Mean Absolute Error (MAE)	$\sum_{i=1}^{n} \frac{\lvert y_i\hat{a} - y_i \rvert}{n}$	Manhattan

However, many types of models don't output just one number given an input, but output a distribution. For example, if your task has three classes: [cat, dog, chicken], your model might output an array of how likely it is that your input belongs to each class. So the predicted output might look like [0.1, 0.5, 0.4], which means the input has 10% chance of being cat, 50% chance of being dog, 40% chance of being chicken. The actual label for

this example is chicken, so the output is [0, 0, 1]. We want to measure the distance between the predicted outputs that take the form [0.1, 0.5, 0.4] and the actual outputs that take the form [0, 0, 1]. In this case, the common objective function is cross entropy and its variation.

You can modify the objective function to enforce your model to learn a set of parameters with certain properties. As discussed in the section Class Imbalance in Chapter 3, you can modify the objective function to encourage your model to focus on examples of rare classes or examples that are difficult to learn. You can also add regularizers such as L1 and L2 to your loss function to encourage your model to choose parameters of smaller values.

Each objective function gives you a set of possible values your parameters can take. This set of possible values for parameters is known as the loss surface of a given objective function. A small change to your objective

function can give you a very different loss surface, which, in turn, gives you a very different function for your model.

Understanding the possible parameters given by different objective functions can help you choose the objective function that is best suited for your needs. However, this understanding tends to require advanced linear algebra, so it's common for ML engineers to use popular objective functions that are known to give decent performance for their problems without giving them much thought.

While developing your model, if time permits, you should experiment with different objective functions to see how your model's behaviors change, both globally on all your data or with respect to different slices of your data. You might be surprised.

Learning Procedure

Learning procedures the procedures that help your model find the set of parameters that

minimize a given objective function for a given set of data, are diverse[3]. In some cases, the parameters might be calculated exactly. For example, in the case of linear functions, the values of w and b can be calculated from the averages and variances of x and y. In most cases, however, the values of parameters can't be calculated exactly and have to be approximated, usually via an iterative procedure. For example, K-means clustering uses an iterative procedure called expectation–maximization algorithm.

The most popular family of iterative procedures today is undoubtedly **gradient descent**. The loss of a model at a given train step is given by the objective function. The gradient of an objective function with respect to a parameter tells us how much that parameter contributes to the loss. In other words, the gradient is the direction that lowers the loss from a current value the most. The idea is to subject that gradient value from that parameter, hoping that this would make

the parameter contribute less to the loss, and eventually drive the loss down to 0.

Subtracting the raw gradient values from parameters doesn't work extremely well. Transforming the gradient values first (such as multiplying the gradient value with 0.003) then subtracting that transformed values from parameters helps models converge much faster. The function that determines how to update a parameter given a gradient value is called an update algorithm, or an **optimizer**. Common optimizers include Momentum, Adam, and RMSProp.

Good optimizers can both speed up your model training process and help your model converge to a better set of parameters. Even though optimizers help your model find the set of parameters that minimize a given objective function for a given set of data, the set of parameters that minimize the loss for your training data isn't always the best optimizer for you, as you might want the parameters that will perform well on the data

your model will encounter in production too[4]. While developing ML models, especially with gradient descent-based models, it's often helpful to explore with different types of optimizers. In section AutoML, we'll discuss how to use ML to find the best optimizers for your model.

Choosing ML Models

In this section, we'll first discuss different types of ML problems, then we'll discuss different types of ML algorithms used to solve these problems.

Framing ML Problems

To decide which ML model to use to solve your problem, you first need to frame your problem into a problem that ML can solve.

The most general types of ML problems are classification and regression. Classification models classify inputs into different categories. For example, you want to classify

each email to be either spam or not spam. Regression models output a continuous value. An example is a house prediction model that outputs the price of a given house.

A regression model can easily be framed as a classification model and vice versa. For example, house prediction can become a classification task if we quantize the house prices into buckets such as under $100,000, $100,000 - 200,000, $200,000 - 500,000, … and predict the bucket the house should be in. The email classification model can become a regression model if we make it outputs values between 0 and 1, and use a threshold to determine which values should be SPAM (for example, if the value is above 0.5, the email is spam).

Within classification problems, the fewer classes there are to classify, the simpler the problem is. The simplest is **binary classification** where there are only two possible classes. Examples of binary classification include classifying whether a

comment is toxic or not toxic, whether a lung scan shows signs of cancer or not, whether a transaction is fraudulent or not. It's unclear whether this type of problem is common because they are common in nature or simply because ML practitioners are most comfortable handling them. Dealing with binary classification problems, including upsampling and downsampling as well as visualizing ROC curves and confusion matrices, is much easier than dealing with multiclass classifiers.

When there are more than two classes, the problem becomes **multiclass classification**. When the number of classes is high, such as disease diagnosis where the number of diseases can go up to thousands or product classifications where the number of products can go up to tens of thousands, the problem can be very challenging. The first challenge is in data collection. In my experience, ML models typically need at least 100 examples for each class to learn to classify that class.

So if you have 1000 classes, you already need at least 100,000 examples. The data collection can be especially challenging when some of the classes are rare, and if you have thousands of classes, there's a high likelihood that some of them are rare.

When the number of classes is large, hierarchical classification might be useful. In hierarchical classification, you have a classifier to first classify each example into one of the large groups. Then you have another classifier to classify this example into one of the subgroups. For example, for product classification, you can first classify each product into one of the four main categories: electronics, home & kitchen, fashion, and pet supplies. After a product has been classified into a category, say fashion, you can use another classifier to put this product into one of the subgroups: shoes, shirt, jeans, accessories.

In both binary and multiclass classification, each example belongs to exactly one class.

When an example can belong to multiple classes, we have a **multilabel classification** problem. For example, when building a model to classify articles into three topics: [tech, entertainment, finance], an article can be in both tech and finance.

There are two major approaches to multilabel classification problems. The first is to treat it as you would a multiclass classification. In multiclass classification, if there are three possible classes [tech, entertainment, finance] and the label for an example is entertainment, you represent this label with the vector [0, 1, 0]. In multilabel classification, if an example has both labels entertainment and finance, its label will be represented as [0, 1, 1].

The second approach is to turn it into multiple binary classification problems. For the article classification problem above, you can have three models corresponding to three topics, where each model outputs whether an article is in that topic or not.

Changing the way you frame your problem might make your problem significantly harder or easier. Consider the task of predicting what app a phone user wants to use next. A naive setup would be to use user's and context's features (user demographic information, time, location, previous apps used) as input and output a probability distribution for every single app on the user's phone. This means that the last layer of your model will have the shape |number of hidden layer| x |number of apps|. This is a bad approach because whenever a new app is added, you have to retrain your model, or at least the last layer of your model. A better approach is to have the user profile, the environment, and the app profile as input, and output a value between 0 and 1, the higher the value, the more likely the user will open the app given the context.

Decoupling Objectives

When solving a problem, you might have

multiple objectives in mind. Imagine you're building a system to rank items on users' newsfeed. Your original goal is to maximize users' engagement. You want to achieve this goal through the following three objectives.

1. Filter out spam
2. Filter out NSFW content
3. Rank posts by engagement: how likely users will click on it

However, you quickly learned that optimizing for users' engagement alone can lead to questionable ethical concerns. Because extreme posts tend to get more engagements, your algorithm learned to prioritize extreme content[56]. You want to create a more wholesome newsfeed. So you have a new goal: **maximize users' engagement** while **minimizing the spread of extreme views and misinformation**. To obtain this goal, you add two new objectives to your original plan.

1. Filter out spam
2. Filter out NSFW content
3. Filter out misinformation
4. Rank posts by quality
5. Rank posts by engagement: how likely users will click on it

Now, objectives 4 and 5 are in conflict with each other. If a post is very engaging but it's of questionable quality, should that post rank high or low?

Let's take a step back to see what exactly each objective does. To rank posts by quality, you first need to predict posts' quality and you want posts' predicted quality to be as close to their actual quality as possible. Essentially, you want to minimize **quality_loss**: the difference between each post's predicted quality and its true quality[7].

Similarly, to rank posts by engagement, you first need to predict the number of clicks each

post will get. You want to minimize **engagement_loss**: the difference between each post's predicted clicks and its actual number of clicks.

One approach is to combine these two losses into one loss and train one model to minimize that loss.

loss = α quality_loss + β engagement_loss

You can randomly test out different values of α and β to find the values that work best. If you want to be more systematic about tuning these values, you can check out Pareto optimization, "*an area of multiple criteria decision making that is concerned with mathematical optimization problems involving more than one objective function to be optimized simultaneously*[8]".

A problem with this approach is that each time you tune α and β—for example, if your users' newsfeed's quality goes up but users' engagement goes down, you might want to decrease α and increase β—you'll have to

retrain your model.

Another approach is to train two different models, each optimizing one loss. So you have two models:

- **quality_model** minimizes **quality_loss** and outputs the predicted quality of each post.
- **engagement_model** minimizes **engagement_loss** and outputs the predicted number of clicks of each post.

You can combine the outputs of these two models and rank posts by their combined scores:

α quality_score + β engagement_score

Now you can tweak α and β without retraining your models!

In general, when there are multiple objectives, it’s a good idea to decouple them first because it makes model development

and maintenance easier. First, it's easier to tweak your system without retraining models, as explained above. Second, it's easier for maintenance since different objectives might need different maintenance schedules. Spamming techniques evolve much faster than the way post quality is perceived, so spam filtering systems need updates at a much higher frequency than quality ranking systems.

Evaluating ML Algorithms

When thinking about ML algorithms, many people think of classical ML algorithms versus neural networks. There are a lot of interests in neural networks, especially in deep learning, which is understandable given that most of the AI progress in the last decade is due to neural networks getting bigger and deeper.

Many newcomers to the field that I've talked to think that deep learning is replacing classical ML algorithms. However, even

though deep learning is finding more use cases in production, classical ML algorithms are not going away. Many recommendation systems still rely on collaborative filtering and matrix factorization. Tree-based algorithms, including gradient-boosted trees, still power many classification tasks with strict latency requirements.

Even in applications where neural networks are deployed, classic ML algorithms are still being used in tandem, either in an ensemble or to help extract features to feed into neural networks.

When selecting a model for your problem, you don’t choose from every possible model out there, but usually focus on a set of models suitable for your problem. For example, if your boss tells you to build a system to detect toxic tweets, you know that this is a text classification problem — given a piece of text, classify whether it’s toxic or not — and common models for text classification include Naive Bayes, Logistic Regression,

recurrent neural networks, Transformer-based models such as BERT, GPT, and their variants.

If your client wants you to build a system to detect fraudulent transactions, you know that this is the classic abnormality detection problem — fraudulent transactions are abnormalities that you want to detect — and common algorithms for this problem are many, including k-nearest neighbors, isolation forest, clustering, and neural networks.

Knowledge of common ML tasks and the typical approaches to solve them is essential in this process.

Different types of algorithms require different amounts of labels as well as different amounts of compute power. Some take longer to train than others, while some take longer to make predictions. Non-neural network algorithms tend to be much easier to explain (for example, what features in the email that

made the model classify it as spam) than neural networks.

When considering what model to use, it's important to consider not only the model's performance, measured by metrics such as accuracy, F1 score, log loss, but also its other properties such as how much data it needs to run, how much compute and time it needs to both train and do inference, and interpretability. For example, a simple logistic regression model might lower accuracy than a complex neural network, but it requires less labeled data to start, it's much faster to train, it's much easier to deploy, and it's also much easier to explain why it's making certain predictions.

Comparing ML algorithms is out of the scope for this book. No matter how good a comparison is, it will be outdated as soon as new algorithms come out. Back in 2016, LSTM-RNNs were all the rage and the backbone of the architecture seq2seq (Sequence-to-Sequence) that powered many

NLP tasks from machine translation to text summarization to text classification. However, just two years later, recurrent architectures were largely replaced by Transformer architectures for NLP tasks.

To understand different algorithms, the best way is to equip yourself with basic ML knowledge and run experiments with the algorithms you're interested in. To keep up-to-date with so many new ML techniques and models, I find it helpful to monitor trends at major ML conferences such as NeurIPS, ICLR, and ICML as well as following researchers whose work has a high signal-to-noise ratio on Twitter.

Without getting into specifics of different algorithms, here are six tips that might help you decide what ML algorithms to work on next.

1. Avoid the state-of-the-art trap

   While helping companies as well as recent graduates get started in ML, I

usually have to spend a non-trivial amount of time steering them away from jumping straight into state-of-the-art (SOTA) models. I can see why people want SOTA models. Many believe that these models would be the best solutions for their problems — why try an old solution if you believe that a newer and superior solution exists? Many business leaders also want to use SOTA models because they want to use them to make their businesses appear cutting-edge. Developers might also be more excited getting their hands on new models than getting stuck into the same old things over and over again.

Researchers often only evaluate models in academic settings, which means that a model being SOTA often only means that it performs better than existing models on some

static datasets. It doesn't mean that this model will be fast enough or cheap enough for you to implement in your case. It doesn't even mean that this model will perform better than other models on **your** data.

While it's essential to stay up-to-date to new technologies and beneficial to evaluate them for your businesses, the most important thing to do when solving a problem is finding solutions that can solve that problem. If there's a solution that can solve your problem that is much cheaper and simpler than SOTA models, use the simpler solution.

2. Start with the simplest models

   Zen of Python states that "simple is better than complex", and this principle is applicable to ML as well. Simplicity serves three purposes. First, simpler models are easier to

deploy, and deploying your model early allows you to validate that your prediction pipeline is consistent with your training pile. Second, starting with something simple and adding more complex components step-by-step makes it easier to understand your model and debug it. Third, the simplest model serves as a baseline to which you can compare your more complex models.

Simplest models are not always the same as models with the least effort. For example, pretrained BERT models are complex, but they require little effort to get started with, especially if you use a ready-made implementation like the one in HuggingFace's Transformer. In this case, it's not a bad idea to use the complex solution, given that the community around this solution is well-developed enough to help you

get through any problems you might encounter. However, you might still want to experiment with simpler solutions, if you haven't already, to make sure that pretrained BERT is indeed better than those simpler solutions for your problem.

3. Avoid human biases in selecting models

   Imagine an engineer on your team is assigned the task of evaluating which model is better for your problem: a gradient boosted tree or a pretrained BERT model. After two weeks, this engineer announced that the best BERT model outperforms the best gradient boosted tree by 5%. Your team decides to go with the pretrained BERT model.

   A few months later, however, a seasoned engineer joins your team. She decides to look into gradient

boosted trees again and finds out that this time, the best gradient boosted tree outperforms the pretrained BERT model you currently have in production. What happened?

There are a lot of human biases in evaluating models. Part of the process of evaluating an ML architecture is to experiment with different features and different sets of hyperparameters to find the best model of that architecture. If an engineer is more excited about an architecture, she will likely spend a lot more time experimenting with it, which might result in better performing models for that architecture.

When comparing different architectures, it's important to compare them under comparable setups. If you run 100 experiments for an architecture, it's not fair to

only run a couple of experiments for the architecture you're evaluating it against. You also want to run 100 experiments for the other architectures too.

4. Evaluate good performance now vs. good performance later
9 10

   While evaluating models, you might want to take into account their potential for improvements in the near future, and how easy/difficult it is to achieve those improvements.

5. Trade-offs

   There are many tradeoffs you have to make when selecting models. Understanding what's more important in the performance of your ML system will help you choose the most suitable model.

   One classic example of tradeoff is

the false positives and false negatives tradeoff. Reducing the number of false positives might increase the number of false negatives, and vice versa. In a task where false positives are more dangerous than false negatives, such as fingerprint unlocking (unauthorized people shouldn't be classified as authorized and given access), you might prefer a model that makes less false positives. Similarly, in a task where false negatives are more dangerous than false positives, such as covid screening (patients with covid shouldn't be classified as no covid), you might prefer a model that makes less false negatives.

Another example of tradeoff is latency and accuracy — a more complex model might deliver higher accuracy but might give higher

latency. Many people care about the interpretability and performance tradeoff. A more complex model can give a better performance but its results are less interpretable.

6. Understand your model's assumptions

   The statistician George Box said in 1976 that "all models are wrong, but some are useful". The real world is intractably complex, and models can only approximate using assumptions. Every single model comes with its own assumptions. Understanding what assumptions a model makes and whether our data satisfies those assumptions can help you evaluate which model works best for your use case.

Below are some of the common assumptions. It's not meant to be an exhaustive list, but just a demonstration.

- Prediction assumption: every model that aims to predict an output Y from an input X makes the assumption that it's possible to predict Y based on X.
- IID: Neural networks assume that the examples are independent and identically distributed, which means that all the examples are independently drawn from the same joint distribution.
- Smoothness: Every supervised machine learning method assumes that there's a set of functions that can transform inputs into outputs such that similar inputs are transformed into similar outputs. If an input X produces an output Y, then an input close to X would produce an output proportionally close to Y.
- Tractability: Let X be the input and

Z be the latent representation of X. Every generative model makes the assumption that it's tractable to compute the probability P(Z|X).

- Boundaries: A linear classifier assumes that decision boundaries are linear.
- Conditional independence: A Naive Bayes classifier assumes that the attribute values are independent of each other given the class.
- Normally distributed: many statistical methods assume that data is normally distributed.

Ensembles

When considering an ML solution to your problem, you might want to start with a system that contains just one model, and the process of selecting one model for your problem is discussed above. After you've

deployed your system, you might think about how to continue improving its performance. One method that has consistently given your system a performance boost is to use an ensemble of multiple models instead of just an individual model to make predictions. Each model in the ensemble is called a base learner. For example, for the task of predicting whether an email is SPAM or NOT SPAM, you might have 3 different models. The final prediction for each email is the majority vote of all these three models. So if at least two base learners output SPAM, the email will be classified as SPAM.

20 out of 22 winning solutions on Kaggle competitions in 2021, as of August 2021, use ensembles[11]. An example of an ensemble used for a Kaggle competition is shown in Figure 5-1. Ensembling methods are less favored in production because ensembles are more complex to deploy and harder to maintain. However, they are still common for tasks where a small performance boost can

lead to a huge financial gain such as predicting click-through rate (CTR) for ads.

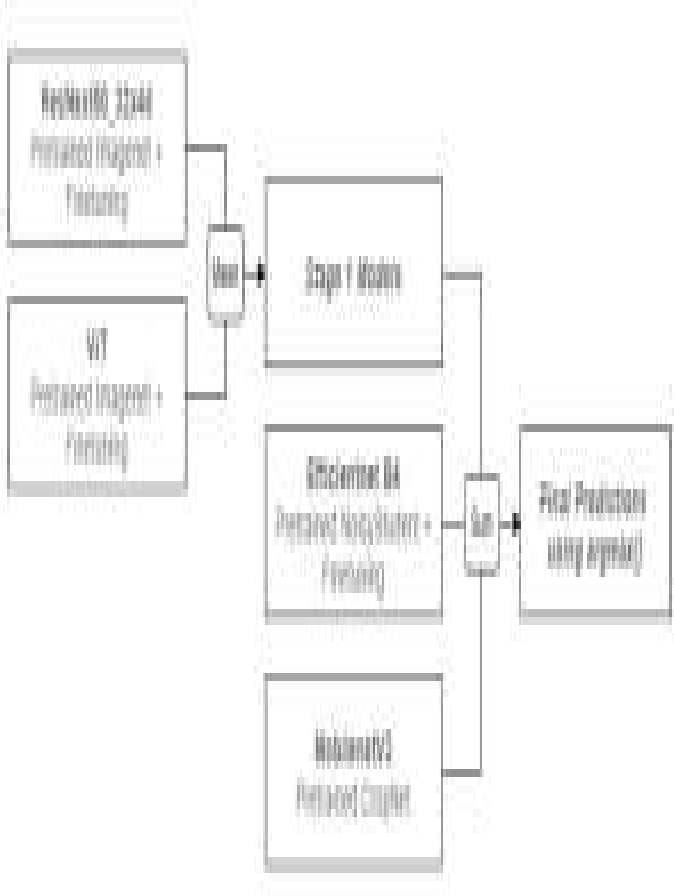


Figure 5-1. The ensemble used in the top solution for the Cassava Leaf Disease Classification competition on Kaggle. Image by Jannis Hanke.

We'll go over an example to give you the intuition of why ensembling works. Imagine you have three email spam classifiers, each with an accuracy of 70%. Assuming that each classifier has an equal probability of making a correct prediction for each email, and that these three classifiers are not correlated, we'll show that by taking the majority vote of these three classifiers, we can get an accuracy of 78.4%.

For each email, each classifier has a 70% chance of being correct. The ensemble will be correct if at least 2 classifiers are correct. Table 5-1 shows the probabilities of different possible outcomes of the ensemble given an email. This ensemble will have an accuracy of 0.343 + 0.441 = 0.784, or 78.4%.

Table 5-3. Possible outcomes of the ensemble that takes the majority vote from three classifiers

Outputs of 3 models	Probability	Ensemble's outpu

All 3 are correct	0.7 * 0.7 * 0.7 = 0.343	Correct
Only 2 are correct	(0.7 * 0.7 * 0.3) * 3 = 0.441	Correct
Only 1 is correct	(0.3 * 0.3 * 0.7) * 3 = 0.189	Wrong
None is correct	0.3 * 0.3 * 0.3 = 0.027	Wrong

This calculation only holds if the classifiers in an ensemble are uncorrelated. If all classifiers are perfectly correlated — all three of them make the same prediction for every email — the ensemble will have the same accuracy as each individual classifier. When creating an ensemble, the less correlation there is among base learners, the better the ensemble will be. Therefore, it's common to choose very different types of models for an ensemble. For example, you might create an ensemble that consists of one transformer model, one recurrent neural network, and one gradient boosted tree.

There are three ways to create an ensemble:

bagging to reduce variance, boosting to reduce bias, and stacking to help with generalization. Other than to help boost performance, according to several survey papers, ensemble methods such as boosting and bagging, together with resampling, have shown to help with imbalanced datasets[12], [13]. We'll go over each of these three methods, starting with bagging.

Bagging

Bagging, shortened from **b**ootstrap **agg**regat**ing**, is designed to improve both the training stability[14] and accuracy of ML algorithms. It reduces variance and helps to avoid overfitting.

Given a dataset, instead of training one classifier on the entire dataset, you sample with replacement to create different datasets, called bootstraps, and train a classification or regression model on each of these bootstraps. Sampling with replacement ensures that each bootstrap is independent from its peers.

Figure 5-2 shows an illustration of bagging.

If the problem is classification, the final prediction is decided by the majority vote of all models. For example, if 10 classifiers vote SPAM and 6 models vote NOT SPAM, the final prediction is SPAM.

If the problem is regression, the final prediction is the average of all models' predictions.

Bagging generally improves unstable methods, such as neural networks, classification and regression trees, and subset selection in linear regression. However, it can mildly degrade the performance of stable methods such as k-nearest neighbors[15].

Figure 5-2. Bagging illustration by Sirakorn

A random forest is an example of bagging. A random forest is a collection of decision trees constructed by both bagging and feature randomness, where each tree can pick only from a random subset of features to use.

Boosting

Boosting is a family of iterative ensemble algorithms that convert weak learners to strong ones. Each learner in this ensemble is trained on the same set of samples but the samples are weighted differently among iterations. As a result, future weak learners focus more on the examples that previous

weak learners misclassified. Figure 5-3 shows an illustration of boosting.

1. You start by training the first weak classifier on the original dataset.
2. Samples are reweighted based on how well the first classifier classifies them, e.g. misclassified samples are given higher weight.
3. Train the second classifier on this reweighted dataset. Your ensemble now consists of the first and the second classifiers.
4. Samples are weighted based on how well the ensemble classifies them.
5. Train the third classifier on this reweighted dataset. Add the third classifier to the ensemble.
6. Repeat for as many iterations as needed.
7. Form the final strong classifier as a

weighted combination of the existing classifiers -- classifiers with smaller training errors have higher weights.

Figure 5-3. Boosting illustration by Sirakorn

An example of a boosting algorithm is Gradient Boosting Machine which produces a prediction model typically from weak decision trees. It builds the model in a stage-wise fashion like other boosting methods do, and it generalizes them by allowing optimization of an arbitrary differentiable loss function.

XGBoost, a variant of GBM, used to be the algorithm of choice for many winning teams

of machine learning competitions. It's been used in a wide range of tasks from classification, ranking, to the discovery of the Higgs Boson[16]. However, many teams have been opting for LightGBM, a distributed gradient boosting framework that allows parallel learning which generally allows faster training on large datasets.

Stacking

Stacking means that you train base learners from the training data then create a meta-learner that combines the outputs of the base learners to output final predictions, as shown in Figure 5-4. The meta-learner can be as simple as a heuristic: you take the majority vote (for classification tasks) or the average vote (for regression tasks) from all base learners. It can be another model, such as a logistic regression model or a linear regression model.

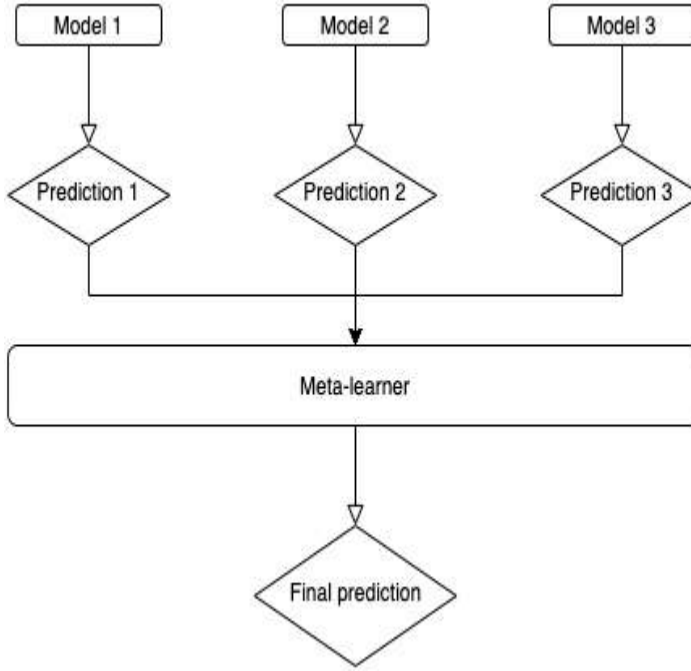


Figure 5-4. A visualization of a stacked ensemble from 3 base learners

For more great advice on how to create an ensemble, refer to this awesome ensemble guide by one of Kaggle's legendary team MLWave.

AutoML

There's a joke that a good ML researcher is

someone who will automate themselves out of job, designing an AI algorithm intelligent enough to design itself. It was funny until the TensorFlow DevSummit 2018, where Jeff Dean took the stage and declared that Google intended on replacing ML expertise with 100 times more computational power, introducing AutoML to the excitement and horror of the community. Instead of paying a group of 100 ML researchers/engineers to fiddle with various models and eventually select a sub-optimal one, why not use that money on compute to search for the optimal model? A screenshot from the recording of the event is shown in Figure 5-5.

TensorFlow

Current:

Solution = ML expertise + data + computation

Can we turn this into:

Solution = data + TPUX computation

???

TensorFlow

Figure 5-5. Jeff Dean unveiling Google's AutoML at TensorFlow Dev Summit 2018

Soft AutoML: Hyperparameter Tuning

AutoML refers to the process of applying ML to real-world problems. One mild form, and the most popular form, of AutoML in production is hyperparameter tuning. A hyperparameter is a parameter supplied by users whose value is used to control the learning process, e.g. learning rate, batch size, number of hidden layers, number of hidden units, dropout probability, and in Adam optimizer, etc. Even quantization level — e.g. mixed-precision, fixed-point — can be considered a hyperparameter to tune.

With different sets of hyperparameters, the same model can give drastically different performances on the same dataset. Melis et al. showed in their 2018 paper On the State of the Art of Evaluation in Neural Language Models that weaker models with well-tuned hyperparameters can outperform stronger,

fancier models. The goal of hyperparameter tuning is to find the optimal set of hyperparameters for a given model within a search space — the performance of each set evaluated on the validation set.

Despite knowing its importance, many still ignore systematic approaches to hyperparameter tuning in favor of a manual, gut-feeling approach. The most popular is arguably Graduate Student Descent (GSD), a technique in which a graduate student fiddles around with the hyperparameters until the model works[17].

However, more and more people are adopting hyperparameter tuning as part of their standard pipelines. Popular ML frameworks either come with built-in utilities or have third-party utilities for hyperparameter tuning, e.g. scikit-learn with auto-sklearn[18], TensorFlow with Keras Tuner. Popular methods for hyperparameter tuning including random search[19], grid search, Bayesian optimization. The book AutoML: Methods,

Systems, Challenges by the AutoML group at the University of Freiburg dedicates its first chapter to hyperparameter optimization, which you can read online for free.

When tuning hyperparameters, keep in mind that a model's performance might be more sensitive to the change in one hyperparameter than another, and therefore sensitive hyperparameters should be more carefully tuned.

> **WARNING**
>
> One important thing is to never use your test split to tune hyperparameters. Choose the best set of hyperparameters for a model based on its performance on a valid split, then report the model's final performance on the test split. If you use your test split to tune hyperparameters, you risk overfitting your model to the test split.

Hard AutoML: Architecture search and learned optimizer

Some teams take hyperparameter tuning to

the next level: what if we treat other components of a model or the entire model as hyperparameters. The size of a convolution layer or whether or not to have a skip layer can be considered a hyperparameter. Instead of manually putting a pooling layer after a convolutional layer or ReLu after linear, you give your algorithm these building blocks and let it figure out how to combine them. This area of research is known as architectural search, or neural architecture search (NAS) for neural networks, as it searches for the optimal model architecture.

A NAS setup consists of three components:

- a search space that defines possible neural networks, e.g. building blocks to choose from and constraints on how they can be combined.
- a performance estimation strategy to evaluate the performance of a candidate architecture. Even though the final architecture resulting from

the research might need retraining, the estimation for all candidate architectures should be done without having to re-construct or re-train them from scratch.

- a search strategy to explore the search space. A simple approach is random search — randomly choosing from all possible configurations — which is unpopular because it's prohibitively expensive even for NAS. Common approaches include reinforcement learning[20] (rewarding the choices that improve the performance estimation) and evolution[21] (adding mutations to an architecture, choosing the best-performing ones, adding mutations to them, and so on).

For NAS, the search space is discrete — the final architecture uses only one of the available options for each layer/operation[22],

and you have to provide the set of building blocks. The common building blocks are various convolutions of different sizes, linear, various activations, pooling, identity, zero, etc.. The set of building blocks varies based on the base architecture, e.g. convolutional neural networks or recurrent neural networks.

In a typical ML training process, you have a model and then a learning algorithm, a set of functions that specifies how to update the weights of the model. Learning algorithms are also called optimizers, and popular optimizers are, as you probably already know, Adam, Momentum, SGD, etc. In theory, you can include existing learning algorithms as building blocks in NAS and search for one that works best. In practice, this is tricky since optimizers are sensitive to the setting of their hyperparameters, and the default hyperparameters don't often work well across architectures.

This leads to an exciting research direction: what if we replace the functions that specify

the learning rule with a neural network? How much to update the model's weights will be calculated by this neural network. This approach results in learned optimizers, as opposed to designed optimizers.

Since learned optimizers are neural networks, they need to be trained. You can train your learned optimizer on the same dataset you're training the rest of your neural network on, but this requires you to train an optimizer every time you have a task.

Another approach is to train a learned optimizer once on a set of existing tasks — using aggregated loss on those tasks as the loss function and existing designed optimizers as the learning rule — and use it for every new task after that. For example, Metz et al. constructed a set of thousands of tasks to train learned optimizers. Their learned optimizer was able to generalize to both new datasets and domains as well as new architectures[23]. And the beauty of this approach is that the learned optimizer can

then be used to train a better-learned optimizer, an algorithm that improves on itself.

Whether it's architecture search or meta-learning learning rules, the upfront training cost is expensive enough that only a handful of companies in the world can afford to pursue them. However, it's important for people interested in ML in production to be aware of the progress in AutoML for two reasons. First, the resulting architectures and learned optimizers can allow ML algorithms to work off-the-shelf on multiple real-world tasks, saving production time and cost, during both training and inferencing. For example, EffecientNets, a family of models produced by Google's AutoML team, surpass state-of-the-art accuracy with up to 10x better efficiency[24]. Second, they might be able to solve many real-world tasks previously impossible with existing architectures and optimizers.

FOUR PHASES OF ML MODEL DEVELOPMENT

Before we transition to model training, let's take a look at the four phases of ML model development. Once you've decided to explore ML, your strategy depends on which phase of ML adoption you are in. There are four phases of adopting ML. The solutions from a phase can be used as baselines to evaluate the solutions from the next phase.

Phase 1. Before machine learning

> If this is your first time trying to make this type of prediction from this type of data, start with non-ML solutions. Your first stab at the problem can be the simplest heuristics. For example, to predict what letter users are going to type next in English, you can show the top three most common English letters, "e", "t", and "a", which is correct 30% of the time.

Facebook newsfeed was introduced in 2006 without any intelligent algorithms — posts were shown in chronological order. It wasn't until 2011 that Facebook started displaying news updates you were most interested in at the top of the feed, as shown in Figure 5-6[25].

Figure 5-6. Facebook newsfeed circa 2006

According to Martin Zinkevich in his magnificent Rules of Machine Learning: Best Practices for ML Engineering:

> *"If you think that machine learning will give you a 100% boost, then a heuristic will get you 50% of the way there.*[26]*"*

You might find out that non-ML solutions work just fine and you don't need ML yet.

Phase 2. Simplest machine learning models

For your first ML model, you want to start with a simple algorithm, something that gives you visibility into its working to allow you to validate the usefulness of your problem framing and your data. Logistic regression XGBoost, K-

nearest neighbors can be great for that.

They are also easier to implement and deploy which allows you to quickly build out a framework from data management to development to deployment that you can test and trust.

Phase 3. Optimizing simple models

Once you've had your ML framework in place, you can focus on optimizing the simple ML models with different objective functions, hyperparameter search, feature engineering, more data, and ensembles.

This phase will allow you to answer questions such as how quickly your model decays in production and update your infrastructure accordingly.

Phase 4. Complex systems

> Once you've reached the limit of your simple models and your use case demands significant model improvement, experiment with more complex models.

Model Training

In the previous section, we've discussed different approaches to select both individual models and ensembles of models for your problem. During a project life cycle, you'll likely experiment with multiple model architectures or different iterations of the same model architecture. In this section, we'll discuss techniques essential to working with multiple ML models at different scales, including distributed training, and experiment tracking and versioning.

Distributed Training

As models are getting bigger and more resource-intensive, companies care a lot more about training at scale[27]. Expertise in scalability is hard to acquire because it requires having regular access to massive compute resources. Scalability is a topic that merits a series of books. This section covers some notable issues to highlight the challenges of doing ML at scale and provide a scaffold to help you plan the resources for your project accordingly.

It's common to train a model using a dataset that doesn't fit into memory. It happens a lot when dealing with medical data such as CT scans or genome sequences. It can also happen with text data if you're a tech giant with enough compute resources to work with a massive dataset (cue OpenAI, Google, NVIDIA, Facebook).

When your data doesn't fit into memory, you will first need algorithms for preprocessing (e.g. zero-centering, normalizing, whitening), shuffling, and batching data out-of-memory

and in parallel. When a sample of your data is too large, e.g. one machine can handle a few samples at a time, you might only be able to work with a small batch size which leads to instability for gradient descent-based optimization.

In some cases, a data sample is so large it can't even fit into memory and you will have to use something like gradient checkpointing, a technique that leverages the memory footprint and compute tradeoff to make your system do more computation with less memory. According to the authors of the open-source package gradient-checkpointing, "*for feed-forward model, we were able to fit more than 10x larger models onto our GPU, at only a 20% increase in computation time*"[28]. Even when a sample fits into memory, using checkpointing can allow you to fit more samples into a batch, which might allow you to train your model faster.

Data Parallelism

It’s now the norm to train ML models on multiple machines (CPUs, GPUs, TPUs). Modern ML frameworks make it easy to do distributed training. The most common parallelization method is data parallelism: you split your data on multiple machines, train your model on all of them, and accumulate gradients. This gives rise to a couple of issues.

A challenging problem is how to accurately and effectively accumulate gradients from different machines. As each machine produces its own gradient, if your model waits for all of them to finish a run — Synchronous stochastic gradient descent (SSGD) — stragglers will cause the entire model to slow down, wasting time and resources[29]. The straggler problem grows with the number of machines, as the more workers, the more likely that at least one worker will run unusually slowly in a given iteration. However, there have been many algorithms that effectively address this

problem [30], [31], [32].

If your model updates the weight using the gradient from each machine separately — Asynchronous SGD (ASGD) — gradient staleness might become a problem because the gradients from one machine have caused the weights to change before the gradients from another machine have come in[33].

The difference between SSGD and ASGD is illustrated in Figure 5-7.

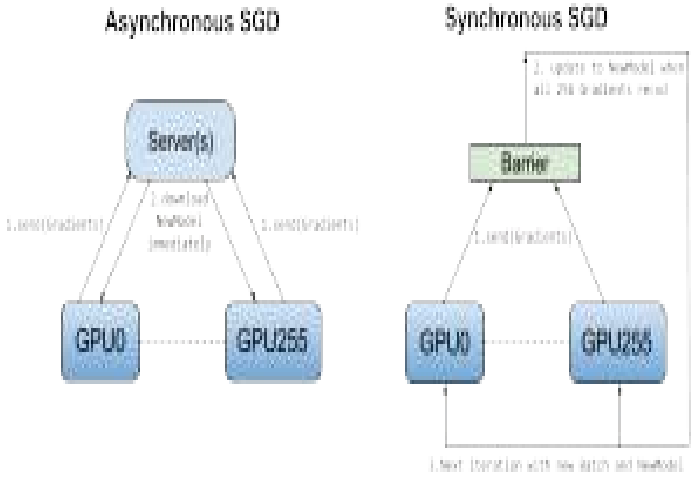


Figure 5-7. ASGD vs. SSGD for data parallelism. Image by Jim Dowling[34]

In theory, ASGD converges but requires more steps than SSGD. However, in practice, when gradient updates are sparse, meaning most gradient updates only modify small fractions of the parameters, the model converges similarly[35].

Another problem is that spreading your model on multiple machines can cause your batch size to be very big. If a machine processes a batch size of 1000, then 1000 machines process a batch size of 1M (OpenAI's GPT-3 175B uses a batch size of 3.2M in 2020[36]). If training an epoch on a machine takes 1M steps, training on 1000 machines takes 1000 steps. An intuitive approach is to scale up the learning rate to account for more learning at each step, but we also can't make the learning rate too big as it will lead to unstable convergence. In practice, increasing the batch size past a certain point yields diminishing returns[37],[38].

Last but not least, with the same model setup, the master-worker sometimes uses a lot more

resources than other workers. If that’s the case, to make the most use out of all machines, you need to figure out a way to balance out the workload among them. The easiest way, but not the most effective way, is to use a smaller batch size on the master-worker and a larger batch size on other workers.

Model Parallelism

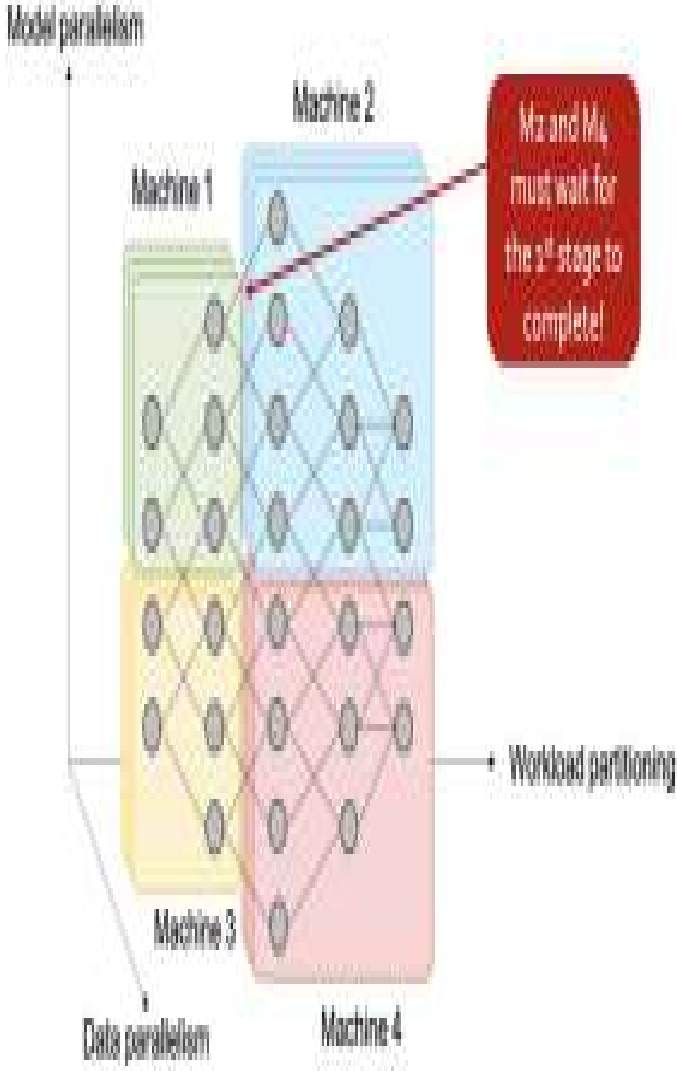

Model parallelism
Machine 2
Machine 1
M2 and M4 must wait for the 2nd stage to complete!
Workload partitioning
Machine 3
Data parallelism
Machine 4

Figure 5-8. Data parallelism and model parallelism. Image by Jure Leskovec.[39]

With data parallelism, each worker has its own copy of the model and does all the computation necessary for the model. Model parallelism is when different components of your model are trained on different machines, as shown in Figure 5-8. For example, machine 0 handles the computation for the first two layers while machine 1 handles the next two layers, or some machines can handle the forward pass while several others handle the backward pass.

Model parallelism can be misleading in some cases when parallelism doesn't mean that different parts of the model in different machines are executed in parallel. For example, if your model is a massive matrix and the matrix is split into two halves on two machines, then these two halves might be executed in parallel. However, if your model is a neural network and you put the first layer on machine 1 and the second layer on

machine 2, and layer 2 needs outputs from layer 1 to execute, then machine 2 has to wait for machine 1 to finish first to run.

Pipeline parallelism is a clever technique to make different components of a model on different machines run more in parallel. There are multiple variants to this, but the key idea is to break the computation of each machine into multiple parts, and when machine 1 finishes the first part of its computation, it passes the result onto machine 2, then continues executing the second part, and so on. Machine 2 now can execute its computation on part 1 while machine 1 executes its computation on part 2.

To make this concrete, consider you have 4 different machines and the first, second, third, and fourth layers are on machine 1, 2, 3, and 4 respectively. Given a mini-batch, you break it into 4 micro-batches. Machine 1 computes the first layer on the first micro-batch, then machine 2 computes the second layer on machine 1’s results for the first micro-batch

while machine 1 computes the first layer on the second micro-batch, and so on. Figure 5-9 shows how pipeline parallelism looks like on 4 machines, each machine runs both the forward pass and the backward pass for one component of a neural network.

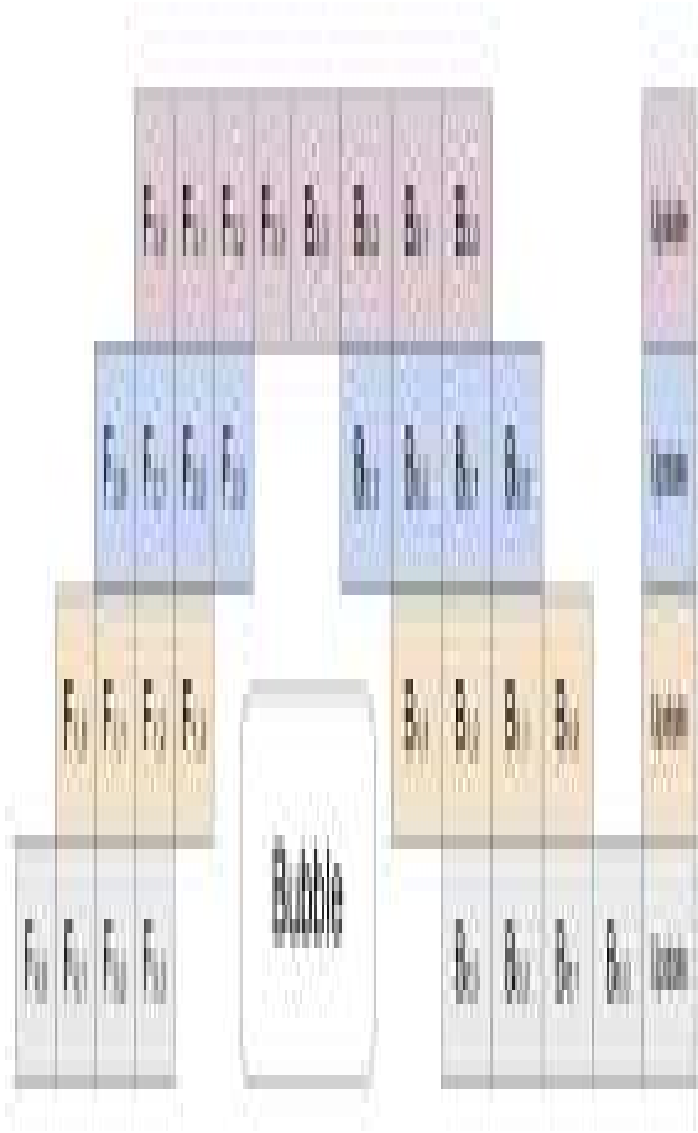


Figure 5-9. Pipeline parallelism for a neural network on 4

machines, each machine runs both the forward pass (F) and the backward pass (B) for one component of the neural network. Image by Huang et al.

Model parallelism and data parallelism aren't mutually exclusive. Many companies use both methods for better utilization of their hardware, even though the setup to use both methods can require significant engineering effort.

Experiment Tracking and Versioning

During the model development process, you often have to experiment with many architectures and many different models to choose the best one for your problem. Some models might seem similar to each other and differ in only one hyperparameter — such as one model uses the learning rate of 0.003 while the other model uses the learning rate of 0.002 — and yet their performances are dramatically different. It's important to keep track of all the defining characteristics of an

experiment and its relevant artifacts. An artifact is a file generated during an experiment — examples of artifacts can be files that show the loss curve, evaluation loss graph, logs, or intermediate results of a model throughout a training process. This enables you to compare different experiments and choose the one best suited for your needs. Comparing different experiments can also help you understand how small changes affect your model's performance, which, in turn, gives you more visibility into how your model works.

The process of tracking the progress and results of an experiment is called experiment tracking. The process of logging all the details of an experiment for the purpose of possibly recreating it later or comparing it with other experiments is called versioning. These two go hand-in-hand with each other. Many tools originally set out to be experiment tracking tools, such as Weights & Biases, have grown to incorporate versioning.

Many tools originally set out to be versioning tools, such as DVC, have also incorporated experiment tracking.

Experiment tracking

A large part of training an ML model is babysitting the learning processes. Many problems can arise during the training process, including loss not decreasing, overfitting, underfitting, fluctuating weight values, dead neurons, and running out of memory. It's important to track what's going on during training not only to detect and address these issues but also to evaluate whether your model is learning anything useful.

When I just started getting into ML, all I was told to track was loss and speed. Fast forward several years, people are tracking so many things that their experiment tracking boards look both beautiful and terrifying at the same time. Below is just a short list of things you might want to consider tracking for each

experiment during its training process.

- The **loss curve** corresponding to the train split and each of the eval splits.
- The **model performance metrics** that you care about on all non-test splits, such as accuracy, F1, perplexity.
- The **speed** of your model, evaluated by the number of steps per second or, if your data is text, the number of tokens processed per second.
- **System performance metrics** such as memory usage and CPU/GPU utilization. They're important to identify bottlenecks and avoid wasting system resources.
- The values over time of any **parameter and hyperparameter** whose changes can affect your model's performance, such as the learning rate if you use a learning

rate schedule, gradient norms (both globally and per layer) if you're clipping your gradient norms, weight norm especially if you're doing weight decay.

In theory, it's not a bad idea to track everything you can. Most of the time, you probably don't need to look at most of them. But when something does happen, one or more of them might give you clues to understand and/or debug your model. However, in practice, due to limitations of tooling today, it can be overwhelming to track too many things, and tracking less important things can distract you from tracking really important things..

Experiment tracking enables comparison across experiments. By observing how a certain change in a component affects the model's performance, you gain some understanding into what that component does.

A simple way to track your experiments is to

automatically make copies of all the code files needed for an experiment and log all outputs with their timestamps[40]. However, using third-party experiment tracking tools can give you nice dashboards and allow you to share your experiments with your coworkers.

Versioning

Imagine this scenario. You and your team spent the last few weeks tweaking your model and one of the runs finally showed promising results. You wanted to use it for more extensive tests so you tried to replicate it using the set of hyperparameters you'd noted down somewhere, only to find out that the results weren't quite the same. You remembered that you'd made some changes to the code between that run and the next, so you tried your best to undo the changes from memory because your reckless past self had decided that the change was too minimal to be committed. But you still couldn't replicate

the promising result because there are just too many possible ways to make change to remember.

This problem could have been avoided if you versioned your ML experiments. ML systems are part code, part data so you need to not only version your code but your data as well. Code versioning has more or less become a standard in the industry. However, at this point, data versioning is like floss. Everyone agrees it's a good thing to do but few do it.

There are a few reasons why data versioning is challenging. One reason is that because data is often much larger than code, we can't use the same strategy that people usually use to version code to version data.

For example, code versioning is done by keeping track of all the changes made to a codebase. A change is known as a diff, short for difference. Each change is measured by line-by-line comparison. A line of code is usually short enough for line-by-line

comparison makes sense. However, a line of your data, especially if it's stored in a binary format, can be indefinitely long. Saying that this line of 1000000 characters is different from the other line of 1000000 characters isn't going to be much helpful.

To allow users to revert to a previous version of the codebase, code versioning tools do that by keeping copies of all the old files. However, a dataset used might be so large that duplicating it multiple times might be unfeasible.

To allow for multiple people to work on the same code base at the same time, code versioning tools duplicate the code base on each person's local machine. However, a dataset might not fit into a local machine.

Second, there's still confusion in what exactly constitutes a diff when we version data. Would diffs mean changes in the content of any file in your data repository, only when a file is removed or added, or

when the checksum of the whole repository has changed?

As of 2021, data versioning tools like DVC only register a diff if the checksum of the total directory has changed and if a file is removed or added.

Another confusion is in how to resolve merge conflicts: if developer 1 uses data version X to train model A and developer 2 uses data version Y to train model B, it doesn't make sense to merge data versions X and Y to create Z, since there's no model corresponding with Z.

Third, if you use user data to train your model, regulations like GDPR might make versioning this data complicated. For example, regulations might mandate that you delete user data if requested, making it legally impossible to recover older versions of your data.

Aggressive experiment tracking and versioning helps with reproducibility, but

doesn’t ensure reproducibility. The frameworks and hardware you use might introduce non-determinism to your experiment results[41], making it impossible to replicate the result of an experiment without knowing everything about the environment your experiment runs in.

The way we have to run so many experiments right now to find the best possible model is the result of us treating ML as a blackbox. Because we can’t predict which configuration will work best, we have to experiment with multiple configurations. However, I hope that as the field progresses, we’ll gain more understanding into different models and can reason about what model will work best instead of running hundreds, if not thousands of experiments.

Model Offline Evaluation

One common but quite difficult question I often encounter when consulting companies

on their AI strategies is: “How do I know that the ML model is good?” In one case, a company deployed ML to detect intrusions to 100 surveillance drones, but they had no way of measuring how many intrusions their system failed to detect, and couldn’t decide if one ML algorithm was better than another for their needs.

Lacking a clear understanding of how to evaluate your ML systems is not necessarily a reason for your ML project to fail, but it might make it impossible to find the best solution for your need, and make it harder to convince your managers to adopt ML.

Ideally, the evaluation methods should be the same in both the development and production environments. But in many cases, the ideal is impossible because during development, you have ground truths, but in production, you don’t have ground truths.

For certain tasks, it’s possible to infer or approximate ground truths in production

based on user's feedback. For example, for the recommendation task, it's possible to infer if a recommendation is good by whether users click on it. However, there are many biases associated with this. The section Continual Learning in chapter 7 will cover how to leverage users' feedback to improve your systems in production.

For other tasks, you might not be able to evaluate your model's performance in production directly, and might have to rely on extensive monitoring to detect changes in your model's performance in particular and to your system in general. We'll cover monitoring in the section Monitoring in chapter 7.

Both monitoring and continual learning can happen once your model has been deployed. In this section, we'll discuss methods to evaluate your model's performance before it's deployed. We'll start with the baselines against which we will evaluate our models. Then we'll cover some of the common

methods to evaluate your model beyond overall accuracy metrics.

Baselines

Someone once told me that her new generative model achieved the FID[42] score of 10.3 on ImageNet. I had no idea what this number meant or whether her model would be useful for my problem.

Another time, I helped a company implement a classification model where the positive class appears 90% of the time. An ML engineer on the team told me, all excited, that their initial model achieved an F1 score of 0.90. I asked him how it was compared to random. He had no idea. It turned out that if his model randomly outputted the positive class 90% of the time, its F1 score would also be around 0.90[43]. His model might as well be making predictions at random, which meant it probably didn't learn anything much.

Evaluation metrics, by themselves, mean

little. When evaluating your model, it's essential to know the baseline you're evaluating it against. The exact baselines should vary from one use case to another, but here are the five baselines that might be useful across use cases.

Random baseline

If our model just predicts at random, what's the expected performance? At random means both "following a uniform random distribution" or "following the same distribution as the task's label distribution."

For example, consider the task that has two labels, NEGATIVE that appears 90% of the time and POSITIVE that appears 10% of the time. Table 5-2 shows the F1 and accuracy scores of baseline models making predictions at random. However, as an exercise to see how challenging it is for most people to have an intuition for these values, try to calculate these raw

numbers in your head before looking at the table.

Table 5-4. F1 and accuracy scores of a baseline model predicting at random for a task that has NEGATIVE that appears 90% of the time and POSITIVE that appears 10% of the time.

Random distribution	Meaning	F1
Uniform random	Predicting each label with equal probability (50%)	0.167
Task's label distribution	Predicting NEGATIVE 90% of the time, and POSITIVE 10% of the time	0.1

Simple heuristic

Forget ML. If you just make predictions based on simple heuristics, what performance would you expect? For example, if you want to build a ranking system to rank items on a user's newsfeed with the goal of getting that user to spend more time on the newsfeed, how much time would a user spend on it if you just rank all the items in reverse chronological order, with the latest one shown first?

Zero rule baseline

The zero rule baseline is a special case of the simple heuristic baseline when your baseline model always predicts the most common class.

For example, for the task of recommending the app a user is most likely to use next on their phone, the simplest model would be to recommend

their most frequently used app. If this simple heuristic can predict the next app accurately 70% of the time, any model you build has to outperform it significantly to justify the added complexity.

Human baseline

In many cases, the goal of ML is to automate what would have been otherwise done by humans, so it's useful to know how your model performs compared to human experts. For example, if you work on a self-driving system, it's crucial to measure your system's progress compared to human drivers, because otherwise you might never be able to convince your users to trust this system. Even if your system isn't meant to replace human experts and only to aid them in improving their productivity, it's still important to know in what scenarios this system would be useful to humans.

Existing solutions

> In some cases, ML systems are designed to replace existing solutions, which might be business logic with a lot of if/else statements or third-party solutions. It's crucial to compare your new model to these existing solutions. Your ML model doesn't always have to be better than existing solutions to be useful. A model whose performance is a little bit inferior can still be useful if it's much easier or cheaper to use.

Picking up the usefulness thread from the last section, a good system isn't necessarily useful. A system meant to replace human experts often has to perform at least as well as human experts to be useful. In some cases, even if it's better than human experts, people might still not trust it, as in the case of self-driving cars. On the contrary, a system that predicts what word a user will type next on their phone can perform much worse than a

native speaker and still be useful.

Evaluation Methods

In academic settings, when evaluating ML models, people tend to fixate on their performance metrics. However, in production, we also want our models to be robust, fair, calibrated, and overall, make sense. We'll introduce some evaluation methods that help with measuring the above characteristics of a model.

Perturbation Tests

A group of my students wanted to build an app to predict whether someone has covid-19 through their cough. Their best model worked great on the training data, which consisted of 2-second long cough segments collected by hospitals. However, when they deployed it to actual users, this model's predictions were close to random.

One of the reasons is that actual users'

coughs contain a lot of noise compared to the coughs collected in hospitals. Users' recordings might contain background music or nearby chatter. The microphones they use are of varying quality. They might start recording their coughs as soon as recording is enabled or wait for a fraction of a second.

Ideally, the inputs used to develop your model should be similar to the inputs your model will have to work with in production, but it's not possible in many cases. This is especially true when data collection is expensive or difficult and you have to rely on the data collected by someone else. As a result, inputs in production are often noisy compared to inputs used in development[44]. The model that performs best on the training data isn't necessarily the model that performs best on noisy inputs in production.

To get a sense of how well your model might perform with noisy data, you can make small changes to your test splits to see how these changes affect your model's performance.

For the task of predicting whether someone has covid-19 from their cough, you could randomly add some background noise or randomly clip the testing clips to simulate how the recordings might be in production. You might want to choose the model that works best on the perturbed data instead of the one that works best on the clean data.

The more sensitive your model is to noise, the harder it will be to maintain it since if your users' behaviors change just slightly, such as they change their phones and get much higher quality microphones, your model's performance might degrade. It also makes your model susceptible to adversarial attack.

Invariance Tests

A Berkeley study found out that between 2008 and 2015, 1.3 million creditworthy black and Latino applicants had their mortgage applications rejected because of their races. When the researchers used the

income and credit scores of the rejected applications but deleted the race identifiers, the applications were rejected.

Certain changes to the inputs shouldn’t lead to changes in the output. In the case above, changes to race information shouldn’t affect the mortgage outcome. Similarly, changes to applicants’ names shouldn’t affect their resume screening results nor should someone’s gender affect how much they should be paid. If these happen, there are biases in your model, which might render it unusable no matter how good its performance is.

To avoid these biases, one solution is to do the same process that helped the Berkeley researchers discover the biases: keep the inputs the same but change the sensitive information to see if the outputs change. Better, you might want to exclude the sensitive information from the features used to train the model in the first place.

Directional Expectation Tests

Certain changes to the inputs should, however, cause predictable changes in outputs. For example, when developing a model to predict housing prices, keeping all the features the same but increasing the lot size shouldn't decrease the predicted price, and decreasing the square footage shouldn't increase the output. If the outputs change in the opposite expected direction, your model might not be learning the right thing, and you need to investigate it further before deploying it.

Model Calibration

Model calibration is a subtle but crucial concept to grasp. Imagine that someone makes a prediction that something will happen with a probability of 70%. What this prediction means is that out of the times this prediction is made, this event happens 70% of the time. If a model predicts that team A will beat team B with a 70% probability, and out

of the 1000 times these two teams play together, team A only wins 60% of the time, then we say that this model isn't calibrated. A calibrated model should predict that team A wins with a 60% probability.

Model calibration is often overlooked by ML practitioners, but it's one of the most important properties of any system that makes predictions. To quote Nate Silver in his book *The Signal and the Noise,* calibration is "one of the most important tests of a forecast — I would argue that it is the single most important one."

We'll walk through two examples to show why model calibration is important. First, consider the task of building a recommender system to recommend what movies users will likely to watch next. Suppose user A watches romance movies 80% of the time and comedy 20% of the time. If you choose your recommendations to consist of only the movies A will most likely to watch, the recommendations will only consist of

romance movies because A is much more likely to watch romance than comedy movies. You might want a calibrated recommendation system whose recommendations are representative of users' actual watching habits. In this case, they should consist of 80% romance and 20% comedy[45].

Second, consider the task of building a model to predict how likely it is that a user will click on an ad. For the sake of simplicity, imagine that there are only 2 ads, ad A and ad B. Your model predicts that this user will click on ad A with a 10% probability and on ad B with a 8% probability. You don't need your model to be calibrated to rank ad A above ad B. However, if you want to predict how many clicks your ads will get, you'll need your model to be calibrated. If your model predicts that a user will click on ad A with a 10% probability but in reality, the ad is only clicked on 5% of the time, your estimated number of clicks will be way off. If you have another model that gives the same ranking

but is better calibrated than this model, you might want to consider this other model.

To measure a model's calibration, a simple method is counting: you count the number of times your model outputs the probability X and the frequency Y of that prediction coming true, and plot X against Y. In scikit-learn, you can plot the calibration curve of a binary classifier with the method sklearn.calibration.calibration_curve, as shown in Figure 5-10.

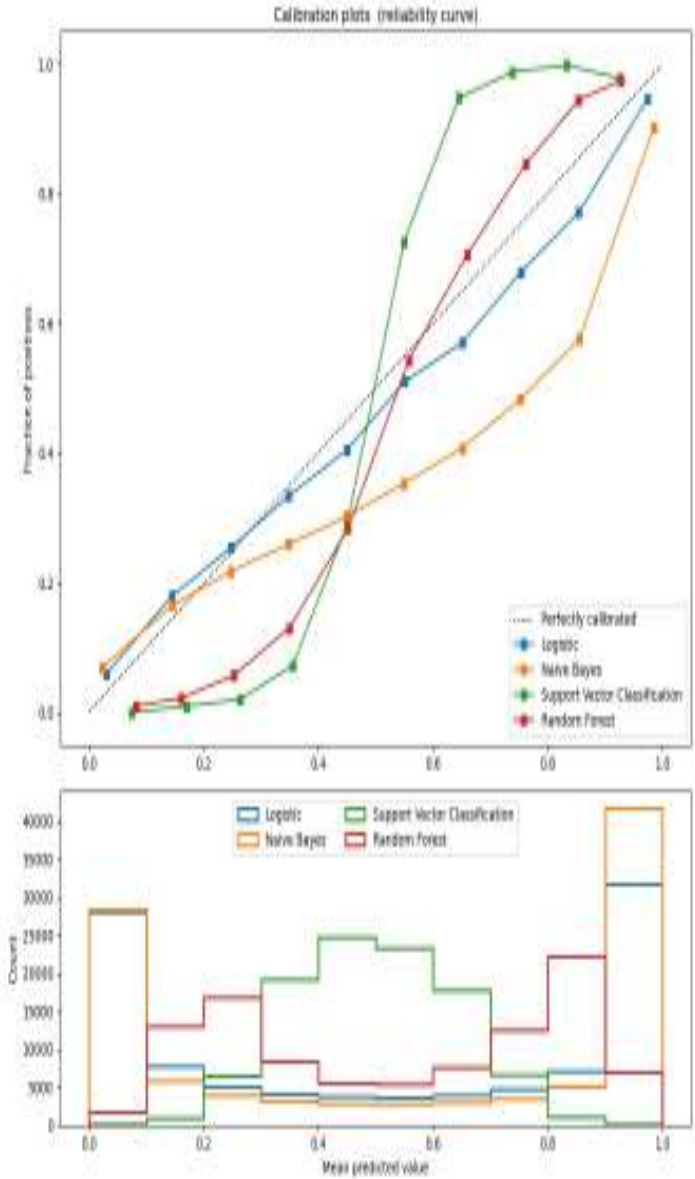

Calibration plots (reliability curve)
Fraction of positives
Perfectly calibrated
Logistic
Naive Bayes
Support Vector Classification
Random Forest
Count
Mean predicted value

Figure 5-10. The calibration curves of different models on a toy task. The Logistic Regression model is the best calibrated model because it directly optimizes logistic loss. Image by scikit-learn.

To calibrate your models, a common method is Platt scaling, which is implemented in scikit-learn with sklearn.calibration.CalibratedClassifierCV. Another good open-source implementation by Geoff Pleiss can be found on GitHub. For readers who want to learn more about the importance of model calibration and how to calibrate neural networks, Lee Richardson and Taylor Pospisil have an excellent blog post based on their work at Google.

Confidence Measurement

Confidence measurement can be considered a way to think about the usefulness threshold for each individual prediction. Indiscriminately showing all model's predictions to users, even the predictions that the model is unsure about, can, at best, cause annoyance and make users lose trust in the

system, such as an activity detection system on your smartwatch that thinks you're running even though you're just walking a bit fast. At worst, it can cause catastrophic consequences, such as a predictive policing algorithm that flags an innocent person as a potential criminal).

If you only want to show the predictions that your model is certain about, how do you measure that certainty? What is the certainty threshold at which the predictions should be shown? What do you want to do with predictions below that threshold — discard it, loop in humans, or ask for more information from users?

While most other metrics deal with system-level measuring system's performance on average, confidence measurement is a metric for each individual instance. System-level measurement is useful to get a sense of overall performance, but instance-level metrics are crucial when you care about your system's performance on every instance, and

the model's failure in just one instance can be catastrophic.

Slice-based Evaluation

Slicing means to separate your data into subgroups and look at your model's performance on those subgroups separately. A common mistake that I've seen in many companies is that they are focused only on coarse-grained metrics like overall F1 or accuracy on the entire datasets. This can lead to two problems.

One is that their model performs differently on different slices (subsets) of data when the model should perform the same. For example, if their data has two subgroups, one majority and one minority, and the majority subgroup accounts for 90% of the data. Model A achieves 98% accuracy on the majority subgroup but only 80% on the minority subgroup, which means its overall accuracy is 96.2%. Model B achieves 95% accuracy on the majority and 95% on the minority, which

means its overall accuracy is 95%. These two models are compared in Table 5-3.

If a company focuses only on overall metrics, they might go with Model A. They might be very happy with this model's high accuracy until one day, their end users discover that this model is biased against the minority subgroup because the minority subgroup happens to correspond to an underrepresented demographic group[46]. The focus on overall performance is harmful not only because of the potential public's backlash, but also because it blinds the company to huge potential model's improvements. If the company sees the two models' performance on different subgroups, they can use different strategies regarding these two models' performance: improve model A's performance on the minority subgroup while improving model's performance overall, and choose one after having weighed the pros and cons of both.

Table 5-5. Two models' performance on the

majority subgroup, which accounts for 90% of the data, and the minority subgroup, which accounts for 10% of the data. Which model would you choose?

	Majority accuracy	Minority accuracy
Model A	98%	80%
Model B	95%	95%

Another problem is that their model performs the same on different slices of data when the model should perform differently. Some subsets of data are more critical. For example, when you build a model for user churn prediction (predicting when a user will cancel a subscription or a service), paid users are more critical than non-paid users. Focusing on a model's overall performance might hurt its performance on these critical slices.

A fascinating and seemingly counterintuitive reason why slice-based evaluation is crucial is Simpson's paradox, a phenomenon in

which a trend appears in several groups of data but disappears or reverses when the groups are combined. This means that model A can perform better than model B on all data together but model B performs better than model A on each subgroup separately. Consider model A's and model B's performance on group A and group B as shown in Table 5-4. Model A outperforms model B for both group A and B, but when combined, model B outperforms model A.

Table 5-6. An example of Simpson's paradox. Model A outperforms model B for both group A and B, but when combined, model B outperforms model A. Numbers from Numbers from Charig et al.'s kidney stone treatment study in 1986.

	Group A	Group B
Model A	93% (81/87)	73% (192/263)
Model B	87% (234/270)	69% (55/80)

Simpson's paradox is more common than

you’d think. In 1973, Berkeley graduate statistics showed that the admission rate for men was much higher than for women, which caused people to suspect biases against women. However, a closer look into individual departments showed that the admission rates for women were actually higher than those for men in 4 out of 6 departments, as shown in Figure 5-11.

	All		Men		Women	
	Applicants	Admitted	Applicants	Admitted	Applicants	Admitted
Total	12,763	41%	8442	44%	4321	35%

Department	All		Men		Women	
	Applicants	Admitted	Applicants	Admitted	Applicants	Admitted
A	933	64%	825	62%	108	82%
B	585	63%	560	63%	25	68%
C	918	35%	325	37%	593	34%
D	792	34%	417	33%	375	35%
E	584	25%	191	28%	393	24%
F	714	6%	373	6%	341	7%

Figure 5-11. The overall graduate admission rate for men and women at Berkeley in 1973 caused people to suspect biases against women. However, a closer look into individual departments showed that the admission rates for women were actually higher than those for men in 4 out of 6 departments. Data from Sex Bias in Graduate Admissions: Data from Berkeley (Bickel et al., 1975)

Whether this paradox happens in our work or not, the point here is that aggregation can conceal and contradict actual situations. To make informed decisions regarding what model to choose, we need to take into account its performance not only on the entire data, but also on individual slices. Slice-based evaluation can give you insights to improve your model's performance both overall and on critical data and help detect potential biases. It might also help reveal non-machine learning problems. Once, our team discovered that our model performed great overall but very poorly on traffic from mobile users. After investigating, we realized that it was because a button was half hidden on small screens, like phone screens.

Even when you don’t think slices matter, understanding how your model performs in a more fine-grained way can give you confidence in your model to convince other stakeholders, like your boss or your customers, to trust your ML models.

To track your model’s performance on critical slices, you’d first need to know what your critical slices are. You might wonder how to discover critical slices in your data. Slicing is, unfortunately, still more of an art than a science, requiring intensive data exploration and analysis. Here are the three main approaches:

- Heuristics-based: slice your data using existing knowledge you have of the data and the task at hand. For example, when working with web traffic, you might want to slice your data along dimensions like mobile versus desktop, browser type, and locations. Mobile users might behave very differently from desktop

users. Similarly, Internet users in different geographic locations might have different expectations on what a website should look like.[47] This approach might require subject matter expertise.

- Error analysis: manually go through misclassified examples and find patterns among them. We discovered our model's problem with mobile users when we saw that most of the misclassified examples were from mobile users.
- Slice finder: there has been research to systemize the process of finding slices, including Chung et al.'s Slice finder: Automated data slicing for model validation in 2019 and covered in Sumyea Helal's Subgroup Discovery Algorithms: A Survey and Empirical Evaluation (2016). The process generally starts with generating slice candidates with

algorithms such as beam search, clustering, or decision, then prune out clearly bad candidates for slices, and then rank the candidates that are left.

Summary

In this chapter, we've covered what many ML practitioners consider to be the most fun part of an ML project cycle: developing, training, and evaluating ML models. Not all parts are equally fun, however. Making your models work on a large distributed system, like the one that runs models with hundreds of millions, if not billions, of parameters, can be challenging and require specialized system engineering expertise. Intensive tracking and versioning your many experiments are generally agreed to be necessary, but doing it might feel like a chore. Evaluating your models' fitness for the production environment while you only have access to

training data is difficult. However, these methods are necessary to sanity check your models before further evaluating your models in a production environment.

Often, no matter how good your office evaluation of a model is, you still can't be sure of your model's performance in production until that model has been deployed. In the next chapter, we'll go over how to deploy a model. And in the chapter after that, we'll cover how to continually monitor and evaluate your model in production.

---

1 In the section AutoML later in this chapter, we'll cover how to use algorithms to automatically choose a function form from a predefined set of possible forms.

2 If you don't know what these terms mean, you should still be able to understand approximately 80% of this chapter. However, I recommend that you take an introductory course to Machine Learning or read an introductory book on Machine Learning in your free time.

3 The subfield that studies different learning procedures

is called optimization and it's a large, complex, and fascinating field. Readers interested in learning more can refer to the book Algorithms for Optimization (Kochenderfer and Wheeler, 2019).

4 In technical terms, you want optimizers that can generalize to unseen data.

5 Facebook Employee Raises Powered by 'Really Dangerous' Algorithm That Favors Angry Posts (SFist, 2019)

6 The Making of a YouTube Radical (NYT, 2019)

7 For simplicity, let's pretend for now that we know to measure a post's quality.

8 While you're at it, you might also want to read Jin and Sendhoff's great paper on applying Pareto optimization for ML where the authors claimed that "machine learning is inherently a multiobjective task."

9 Andrew Ng has a great lecture where he explains that if a learning algorithm suffers from high bias, getting more training data by itself won't help much. Whereas, if a learning algorithm suffers from high variance, getting more training data is likely to help.

10 Continual learning is almost magical. We'll cover it in detail in Chapter 7.

11 I went through the winning solutions listed at https://farid.one/kaggle-solutions/. One solution used 33 models

12 A Review on Ensembles for the Class Imbalance Problem: Bagging-, Boosting-, and Hybrid-Based

Approaches (Galar et al., 2011)

13 Solving class imbalance problem using bagging, boosting techniques, with and without using noise filtering method (Rekha et al., 2019)

14 Training stability here means less fluctuation in the training loss.

15 Bagging Predictors (Leo Breiman, 1996)

16 Higgs Boson Discovery with Boosted Trees (Tianqi Chen and Tong He, 2015)

17 GSD is a well-documented technique, see here, here, here, and here.

18 auto-sklearn 2.0 also provides basic model selection capacity.

19 Our team at NVIDIA developed Milano, a framework-agnostic tool for automatic hyperparameter tuning using random search. See the code at https://github.com/NVIDIA/Milano.

20 Neural architecture search with reinforcement learning, Zoph et Le. 2016.

21 Regularized Evolution for Image Classifier Architecture Search, Real et al., 2018.

22 You can make the search space continuous to allow differentiation, but the resulting architecture has to be converted into a discrete architecture. See DARTS: Differentiable Architecture Search, Liu et al., 2018.

23 [2009.11243] Tasks, stability, architecture, and compute: Training more effective learned optimizers,

and using them to train themselves (Metz et al. 2020)

24 EfficientNet: Improving Accuracy and Efficiency through AutoML and Model Scaling (Tan et Le, 2019)

25 The Evolution of Facebook News Feed (Samantha Murphy, Mashable 2013)

26 Rules of Machine Learning: Best Practices for ML Engineering (Martin Zinkevich, Google 2019)

27 For products that serve a large number of users, you also have to care about scalability in serving a model, which is outside of the scope of a machine learning project so not covered in this book.

28 gradient-checkpointing repo. Tim Salimans, Yaroslav Bulatov and contributors, 2017.

29 Distributed Deep Learning Using Synchronous Stochastic Gradient Descent, Das et al., 2016.

30 Revisiting Distributed Synchronous SGD, Chen et al., ICLR 2017.

31 Improving MapReduce Performance in Heterogeneous Environments, Zaharia et al., 2008.

32 Addressing the straggler problem for iterative convergent parallel ML, Harlap et al., SoCC 2016.

33 Large Scale Distributed Deep Networks, Dean et al., NIPS 2012.

34 Distributed TensorFlow (Jim Dowling, O'Reilly 2017)

35 Hogwild!: A Lock-Free Approach to Parallelizing

Stochastic Gradient Descent (Niu et al., 2011)

36 Language Models are Few-Shot Learners (Brown et al., 2020)

37 An Empirical Model of Large-Batch Training (McCandlish et al., 2018)

38 [1811.03600] Measuring the Effects of Data Parallelism on Neural Network Training (Shallue et al., 2018)

39 Mining Massive Datasets course,Stanford, lecture 13. Jure Leskovec. 2020.

40 I'm still waiting for an experiment tracking tool that integrates with git commits and data version control commits.

41 Notable examples include atomic operations in CUDA where non-deterministic orders of operations lead to different floating point rounding errors between runs.

42 Fréchet Inception Distance, a common metric for measuring the quality of synthesized images. The smaller the value, the higher the quality is supposed to be.

43 The accuracy, in this case, would be around 0.80.

44 Other examples of noisy data include images with different lighting or texts with accidental typos or intentional text modifications such as typing "long" as "looooooong."

45 For more information on calibrated

recommendations, check out the paper Calibrated recommendations by Harald Steck in 2018 based on his work at Netflix.

46 Google Photos Tags Two African-Americans As Gorillas Through Facial Recognition Software (Maggie Zhang, Forbes 2015) d

47 For readers interested in learning more about UX design across cultures, Jenny Shen has a great post

About the Author

Chip Huyen (*https://huyenchip.com*) is an engineer and founder who develops infrastructure for real-time machine learning. Through her work at Netflix, NVIDIA, Snorkel AI, and her current startup, she has helped some of the world's largest organizations develop and deploy machine learning systems. She is the founder of a startup that focuses on real-time machine learning.

In 2017, she created and taught the Stanford course TensorFlow for Deep Learning Research. She is currently teaching CS 329S: Machine Learning Systems Design at Stanford. This book is based on the course's lecture notes.

She is also the author of four Vietnamese books that have sold more than 100,000 copies. The first two books belong to the series Xách ba lô lên và Đi (Quảng Văn 2012, 2013). The first book in the series was the #1

best-selling book of 2012 on Tiki.vn. The series was among FAHASA's Top 10 Readers Choice Books in 2014.

Chip's expertise is in the intersection of software engineering and machine learning. LinkedIn included her among the 10 Top Voices in Software Development in 2019, and Top Voices in Data Science & AI in 2020.