Chapter 2. Introduction to Machine Learning Systems Design

Now that we've walked through an overview of ML systems in the real world, we can get to the fun part of actually designing an ML system. To reiterate from the first chapter, ML systems design takes a system approach to MLOps, which means that we'll consider an ML system holistically to ensure that all the components—the business requirements, the data stack, infrastructure, deployment, monitoring, etc.—and their stakeholders can work together to satisfy the specified objectives and requirements.

We'll start the chapter with a discussion on objectives. Before we develop an ML system, we must understand why this system is needed. If this system is built for a business, it must be driven by business objectives, which will need to be translated into ML objectives to guide the development of ML models.

Once everyone is on board with the objectives for our ML system, we'll need to set out some requirements to guide the development of this system. In this book, we'll consider the four requirements: reliability, scalability, maintainability, and adaptability. We will then introduce the iterative process for designing systems to meet those requirements.

You might wonder: with all these objectives, requirements, and processes in place, can I finally start building my ML model yet? Not so soon! Before using ML algorithms to solve your problem, you first need to frame your problem into a task that ML can solve. We'll continue this chapter with how to frame your ML problems. The difficulty of your job can change significantly depending on how you frame your problem.

Because ML is a data-driven approach, a book on ML systems design will be amiss if it fails to discuss the importance of data in ML systems. The last part of this chapter touches on a debate that has consumed much of

the ML literature in recent years: which is more important—data or intelligent algorithms?

Let's get started!

Business and ML Objectives

We first need to consider the objectives of the proposed ML projects. When working on an ML project, data scientists tend to care about the ML objectives: the metrics they can measure about the performance of their ML models such as accuracy, F1 score, inference latency, etc. They get excited about improving their model's accuracy from 94% to 94.2% and might spend a ton of resources—data, compute, and engineering time—to achieve that.

But the truth is: most companies don't care about the fancy ML metrics. They don't care about increasing a model's accuracy from 94% to 94.2% unless it moves some business metrics. A pattern I see in many short-lived ML projects is that the data scientists become too focused on hacking ML metrics without paying attention to business metrics. Their managers, however, only care about business metrics and, after failing to see how an ML project can help push their business metrics, kill the projects prematurely (and possibly let go of the data science team involved).[1]

So what metrics do companies care about? While most companies want to convince you otherwise, the sole purpose of businesses, according to the Nobel-winning economist Milton Friedman, is to maximize profits for shareholders.[2]

The ultimate goal of any project within a business is, therefore, to increase profits, either directly or indirectly: directly such as increasing sales (conversion rates) and cutting costs; indirectly such as higher customer satisfaction and increasing time spent on a website.

For an ML project to succeed within a business organization, it's crucial to tie the performance of an ML system to the overall business performance. What business performance metrics is the new ML system supposed to

influence, e.g., the amount of ads revenue, the number of monthly active users?

Imagine that you work for an ecommerce site that cares about purchase-through rate and you want to move your recommender system from batch prediction to online prediction.[3] You might reason that online prediction will enable recommendations more relevant to users right now, which can lead to a higher purchase-through rate. You can even do an experiment to show that online prediction can improve your recommender system's predictive accuracy by *X*% and, historically on your site, each percent increase in the recommender system's predictive accuracy led to a certain increase in purchase-through rate.

One of the reasons why predicting ad click-through rates and fraud detection are among the most popular use cases for ML today is that it's easy to map ML models' performance to business metrics: every increase in click-through rate results in actual ad revenue, and every fraudulent transaction stopped results in actual money saved.

Many companies create their own metrics to map business metrics to ML metrics. For example, Netflix measures the performance of their recommender system using *take-rate*: the number of quality plays divided by the number of recommendations a user sees.[4] The higher the take-rate, the better the recommender system. Netflix also put a recommender system's take-rate in the context of their other business metrics like total streaming hours and subscription cancellation rate. They found that a higher take-rate also results in higher total streaming hours and lower subscription cancellation rates.[5]

The effect of an ML project on business objectives can be hard to reason about. For example, an ML model that gives customers more personalized solutions can make them happier, which makes them spend more money on your services. The same ML model can also solve their problems faster, which makes them spend less money on your services.

To gain a definite answer on the question of how ML metrics influence business metrics, experiments are often needed. Many companies do that with experiments like A/B testing and choose the model that leads to bet-

ter business metrics, regardless of whether this model has better ML metrics.

Yet, even rigorous experiments might not be sufficient to understand the relationship between an ML model's outputs and business metrics. Imagine you work for a cybersecurity company that detects and stops security threats, and ML is just a component in their complex process. An ML model is used to detect anomalies in the traffic pattern. These anomalies then go through a logic set (e.g., a series of if-else statements) that categorizes whether they constitute potential threats. These potential threats are then reviewed by security experts to determine whether they are actual threats. Actual threats will then go through another, different process aimed at stopping them. When this process fails to stop a threat, it might be impossible to figure out whether the ML component has anything to do with it.

Many companies like to say that they use ML in their systems because "being AI-powered" alone already helps them attract customers, regardless of whether the AI part actually does anything useful.[6]

When evaluating ML solutions through the business lens, it's important to be realistic about the expected returns. Due to all the hype surrounding ML, generated both by the media and by practitioners with a vested interest in ML adoption, some companies might have the notion that ML can magically transform their businesses overnight.

Magically: possible. Overnight: no.

There are many companies that have seen payoffs from ML. For example, ML has helped Google search better, sell more ads at higher prices, improve translation quality, and build better Android applications. But this gain hardly happened overnight. Google has been investing in ML for decades.

Returns on investment in ML depend a lot on the maturity stage of adoption. The longer you've adopted ML, the more efficient your pipeline will run, the faster your development cycle will be, the less engineering time you'll need, and the lower your cloud bills will be, which all lead to

higher returns. According to a 2020 survey by Algorithmia, among companies that are more sophisticated in their ML adoption (having had models in production for over five years), almost 75% can deploy a model in under 30 days. Among those just getting started with their ML pipeline, 60% take over 30 days to deploy a model (see Figure 2-1).[7]

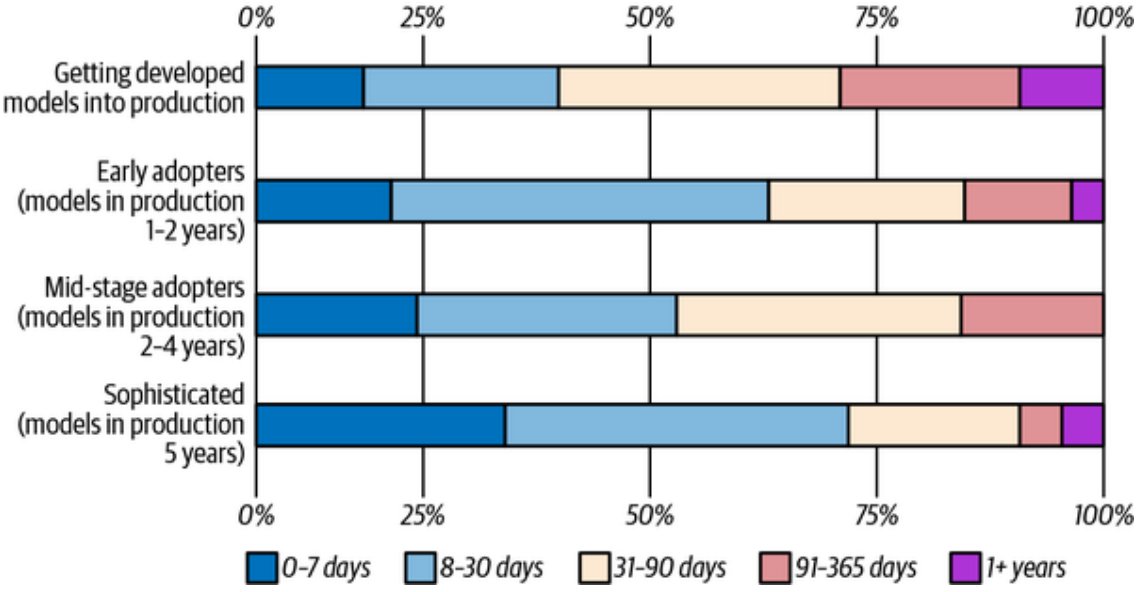

Figure 2-1. How long it takes for a company to bring a model to production is proportional to how long it has used ML. Source: Adapted from an image by Algorithmia

Requirements for ML Systems

We can't say that we've successfully built an ML system without knowing what requirements the system has to satisfy. The specified requirements for an ML system vary from use case to use case. However, most systems should have these four characteristics: reliability, scalability, maintainability, and adaptability. 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 can fail silently. End users don't even know that the system has failed and might have kept on using it as if it were 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 8.

Scalability

There are multiple ways an ML system can grow. It can grow in complexity. Last year you used a logistic regression model that fit into an Amazon Web Services (AWS) free tier instance with 1 GB of RAM, but this year, you switched to a 100-million-parameter neural network that requires 16 GB 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 8,000 models in production for their 8,000 enterprise customers.

Whichever way your system grows, there should be reasonable ways of dealing with that growth. When talking about scalability most people think of resource scaling, which consists of up-scaling (expanding the resources to handle growth) and down-scaling (reducing the resources when not needed).[8]

For example, at peak, your system might require 100 GPUs (graphics processing units). However, most of the time, it 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 of downtime was estimated to cost Amazon between $72 million and $99 million.[9]

However, handling growth isn't just resource scaling, but also artifact management. Managing one hundred models is very different from managing one 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 file that helps you reproduce this model whenever needed. However, with one hundred 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 adequately reproduce a model when you need to.

Because scalability is such an important topic throughout the ML project workflow, we'll discuss it in different parts of the book. Specifically, we'll touch on the resource scaling aspect in the section "Distributed Training", the section "Model optimization", and the section "Resource Management". We'll discuss the artifact management aspect in the section "Experiment Tracking and Versioning" and the section "Development Environment".

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 programming languages and tools, and might own different parts of the process.

It's important to structure your workloads and set up your infrastructure in such a way that different contributors can work using tools that they are comfortable with, instead of one group of contributors forcing their tools onto other groups. Code should be documented. Code, data, and artifacts should be versioned. Models should be sufficiently reproducible so that even when the original authors are not around, other contributors can have sufficient contexts to build on their work. 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 the section "Team Structure".

Adaptability

To adapt to shifting 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", and how to continually update your model with new data in the section "Continual Learning".

Iterative Process

Developing an ML system is an iterative and, in most cases, never-ending process.[10] Once a system is put into production, it'll need to be continually monitored and updated.

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

For example, here is one workflow that you might encounter when building an ML model to predict whether an ad should be shown when users enter a search query:[11]

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 the wrong labels, so you relabel the data.
6. Train the 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 have NEGATIVE labels (ads that shouldn't be shown). So you have to collect more data of ads that should be shown.
8. Train the model again.
9. The model performs well on your existing test data, which is by now two months old. However, it performs poorly on the data from yesterday. Your model is now stale, so you need to update it on more recent data.
10. Train the model again.
11. Deploy the model.
12. The model seems to be performing well, but then the businesspeople 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 ad click-through rate instead.
13. Go to step 1.

Figure 2-2 shows an oversimplified representation of what the iterative process for developing ML systems in production looks like from the perspective of a data scientist or an ML engineer. This process looks different from the perspective of an ML platform engineer or a DevOps engineer, as they might not have as much context into model development and might spend a lot more time on setting up infrastructure.

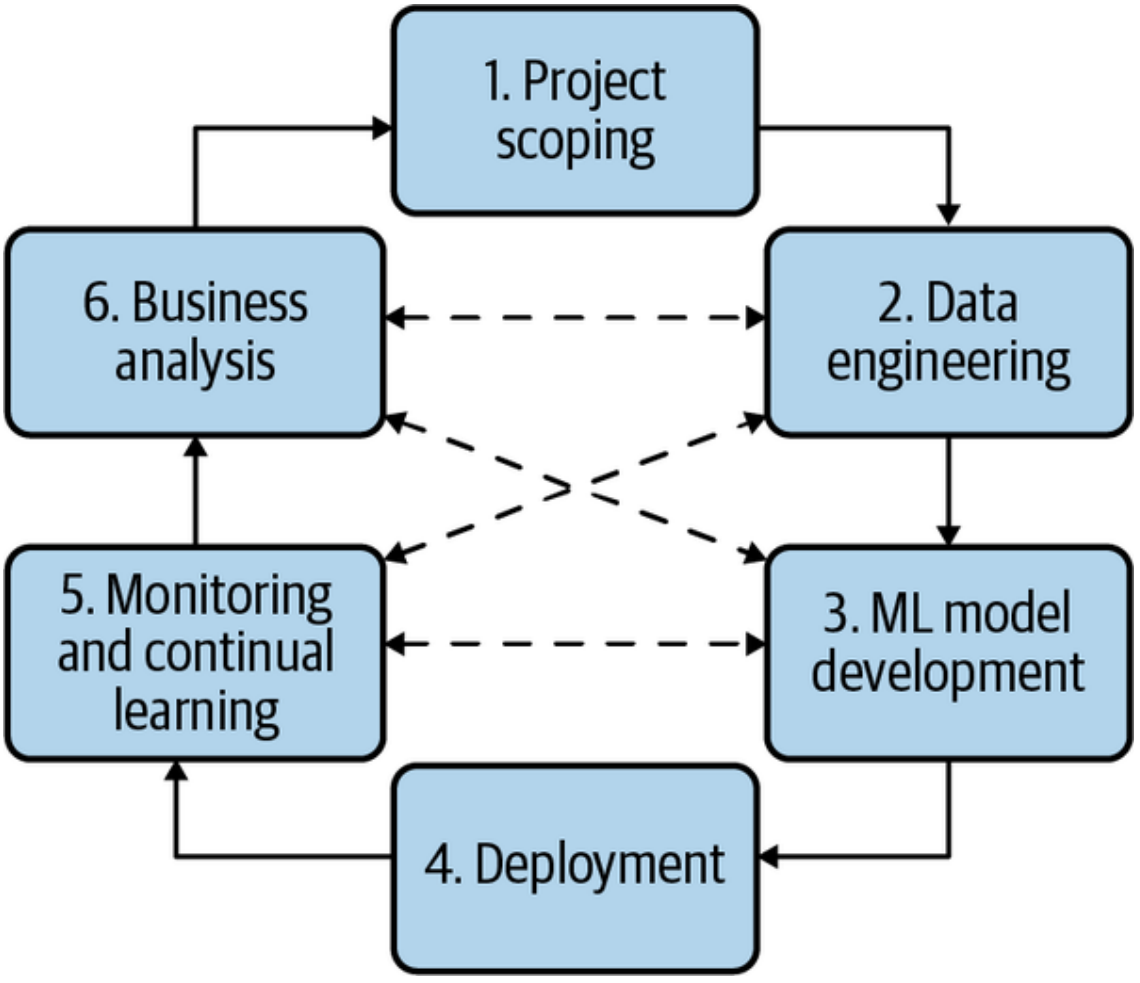

Figure 2-2. The process of developing an ML system looks more like a cycle with a lot of back and forth between steps

Later chapters will dive deeper into what each of these steps requires in practice. Here, let's take a brief look at what they mean:

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 foci for ML projects in production in Chapter 1. We also already discussed how to scope an ML project in the context of a business earlier in this

chapter. We'll discuss how to organize teams to ensure the success of an ML project in Chapter 11.

Step 2. Data engineering

A vast majority of ML models today learn from data, so developing ML models starts with engineering data. In Chapter 3, 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 4.

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 5, we'll discuss feature engineering. In Chapter 6, 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 7.

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 Chapters 8 and 9.

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. This step is closely related to the first step.

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 two times faster. He orders your team to 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—none 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 as an ML problem.

Upon investigation, you discover that the bottleneck in responding to customer requests lies in routing customer requests to the right department among four departments: accounting, inventory, HR (human resources), and IT. You can alleviate this bottleneck by developing an ML model to predict which of these four 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'll discuss extensively how to extract features from raw data to input into your ML model in Chapter 5. 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 2-3. We'll go over each of these task types.

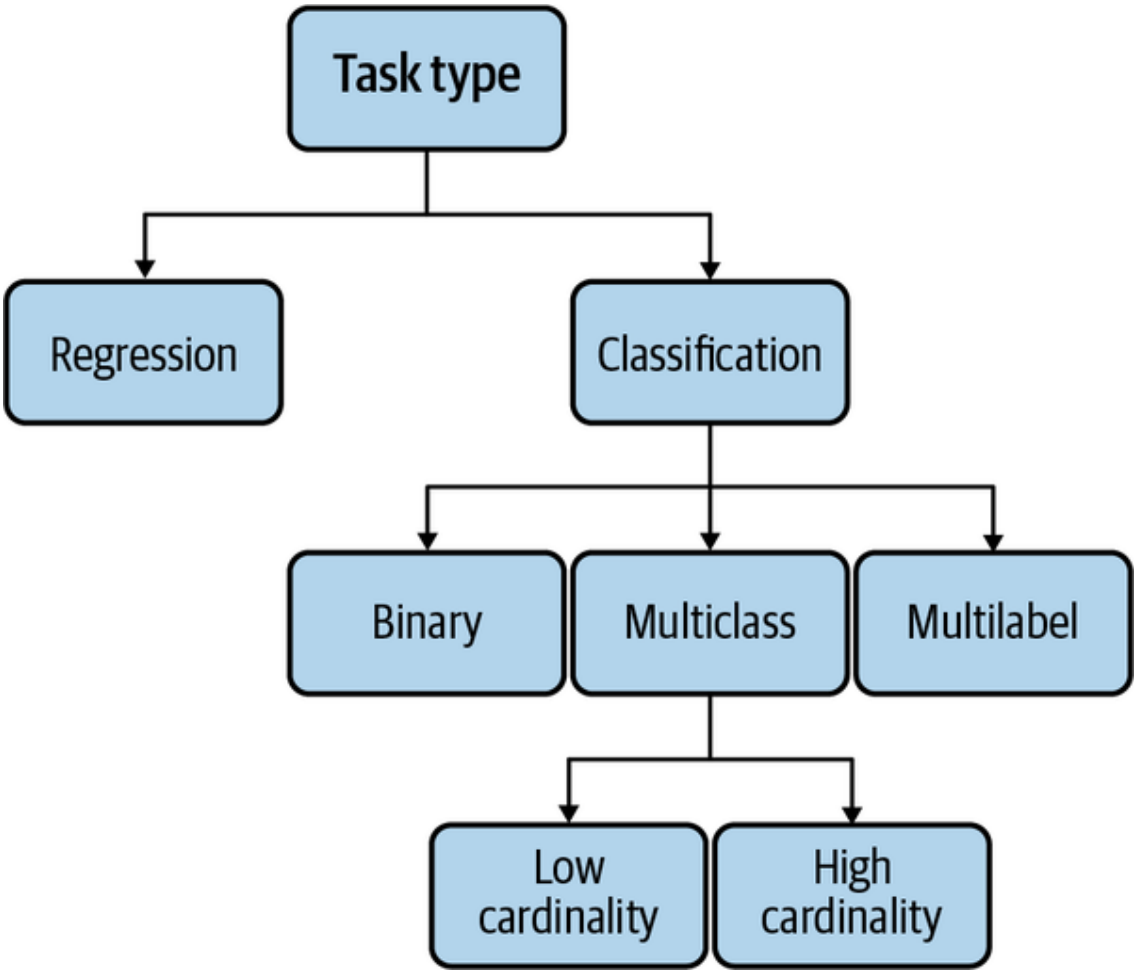

Figure 2-3. Common task types in ML

Classification versus 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 so forth 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 decide on a threshold to determine which values should be SPAM (for example, if the value is above 0.5, the email is spam), as shown in Figure 2-4.

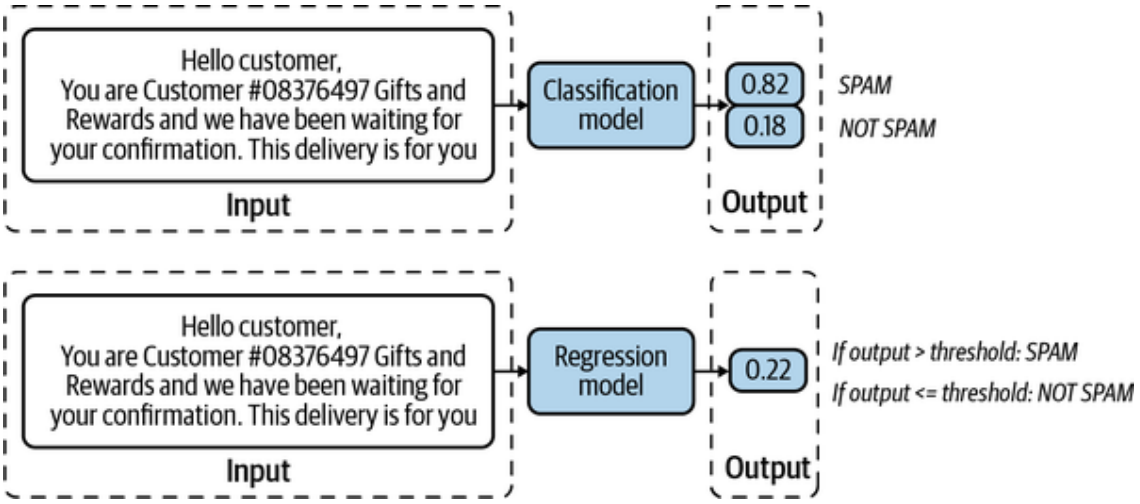

Figure 2-4. The email classification task can also be framed as a regression task

Binary versus 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, whether a lung scan shows signs of cancer, whether a transaction is fraudulent. It's unclear whether this type of problem is common in the industry 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 when there are only 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 1,000 classes, you already need at least 100,000 examples. The data collection can be especially difficult for rare classes. 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 and kitchen, fashion, or 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, shirts, jeans, or accessories.

Multiclass versus 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, and 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 a set of binary classification problems. For the article classification problem, you can have four models corresponding to four topics, each model outputting whether an article is in that topic or not.

Out of all task types, multilabel classification is usually the one 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 discuss in Chapter 4. For example, an annotator might believe an example belongs to two classes while another annotator might believe the same example to belong in only one 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. Imagine that, given an article, your model outputs this raw probability distribution: [0.45, 0.2, 0.02, 0.33]. In the multiclass setting, when 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 one prediction to make, and the prediction is a vector of the size N. This setup is visualized in Figure 2-5.

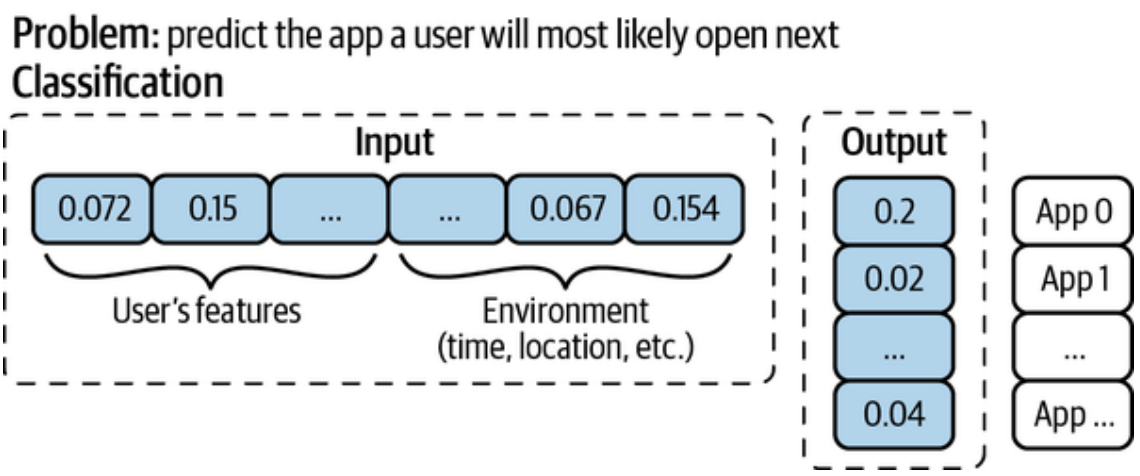

Figure 2-5. Given the problem of predicting the app a user will most likely open next, you can frame it as a classification problem. The input is the user's features and environment's features. The output is a distribution over all apps on the phone.

This is a bad approach because whenever a new app is added, you might have to retrain your model from scratch, or at least retrain 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. This improved setup is visualized in Figure 2-6.

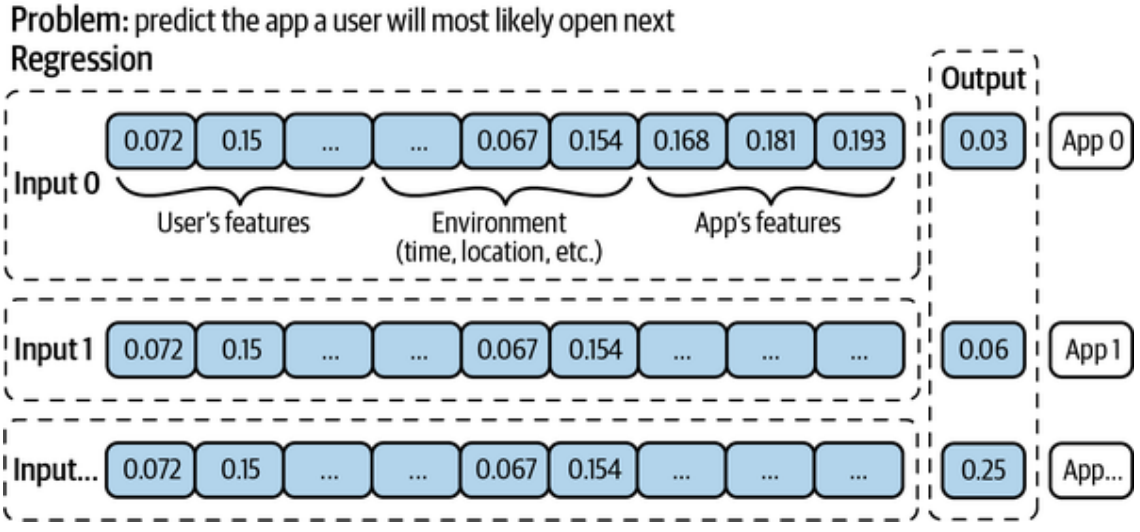

Figure 2-6. Given the problem of predicting the app a user will most likely open next, you can frame it as a regression problem. The input is the user's features, environment's features, and an app's features. The output is a single value between 0 and 1 denoting how likely the user will be to open the 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.[1][2] 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. 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 previous 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, though not 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' newsfeeds. Your original goal is to maximize users' engagement. You want to achieve this goal through the following three objectives:

- Filter out spam
- Filter out NSFW content
- Rank posts by engagement: how likely users will click on it

However, you quickly learned that optimizing for users' engagement alone can lead to questionable ethical concerns. Because extreme posts tend to get more engagements, your algorithm learned to prioritize extreme content.[13] 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:

- Filter out spam
- Filter out NSFW content
- Filter out misinformation
- Rank posts by quality
- Rank posts by engagement: how likely users will click on it

Now two objectives are in conflict with each other. If a post is 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 quality.[14]

Similarly, to rank posts by engagement, you first need to predict the number of clicks each post will get. You want to minimize *engagement_loss*: the difference between each post's predicted clicks and its actual number of clicks.

One approach is to combine these two losses into one loss and train one model to minimize that loss:

$loss = \alpha\ quality\_loss + \beta\ engagement\_loss$

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."[15]

A problem with this approach is that each time you tune α and β—for example, if the quality of your users' newsfeeds 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 models' outputs and rank posts by their combined scores:

$\alpha\ quality\_score + \beta\ engagement\_score$

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 previ-

ously explained. 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.

Mind Versus 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.[16]

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 five years, at every academic conference I attended, there were always some public debates on the power of mind versus 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 architectural designs and leverage large data and computation, but spending time on one often takes time away from another.[17]

In 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 a job in three to five 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."[18]

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 less data.[19]

Many people in ML today are in 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."[20]

When asked how Google Search was doing so well, Peter Norvig, Google's director of search quality, 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."[21]

Dr. Monica Rogati, former VP of data at Jawbone, argued that data lies at the foundation of data science, as shown in Figure 2-7. If you want to use data science, a discipline of which ML 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 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, it might be possible for us to look up the answer. Having a lot of data is different from having infinite data.

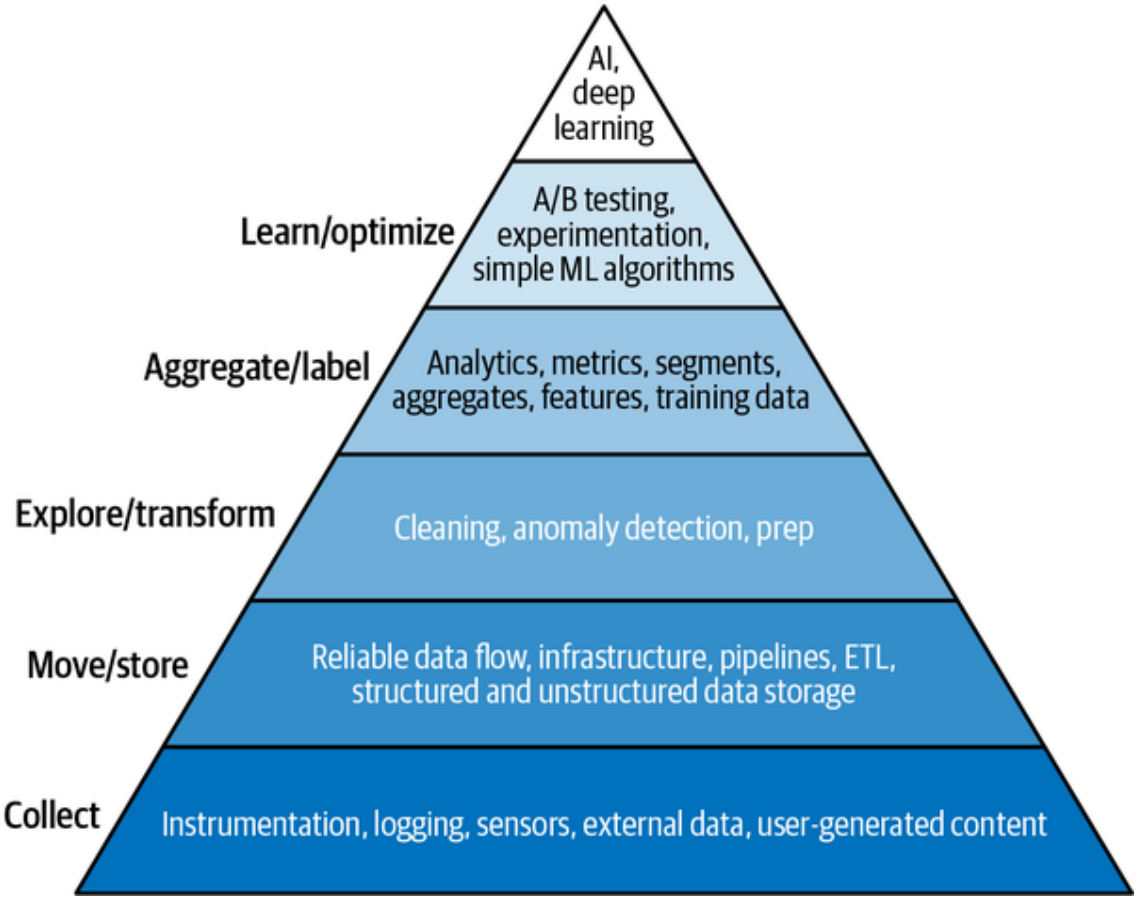

Figure 2-7. The data science hierarchy of needs. Source: Adapted from an image by Monica Rogati[22]

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 ML 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 Word Benchmark for Language Modeling was released, which contains 0.8 billion tokens.[23] Six years later, OpenAI's GPT-2 used a dataset of 10 billion tokens. And another year later, GPT-3 used 500 billion tokens. The growth rate of the sizes of datasets is shown in Figure 2-8.

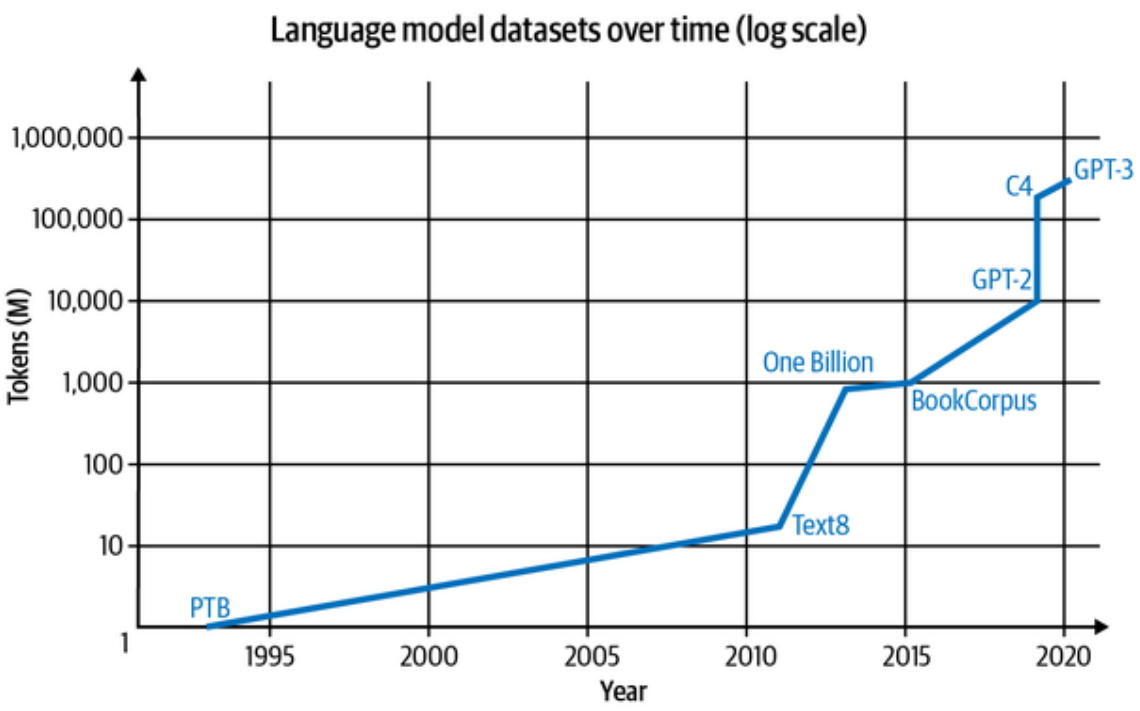

Figure 2-8. The size of the datasets (log scale) used for language models over time

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.

Summary

I hope that this chapter has given you an introduction to ML systems design and the considerations we need to take into account when designing an ML system.

Every project must start with why this project needs to happen, and ML projects are no exception. We started the chapter with an assumption that most businesses don't care about ML metrics unless they can move business metrics. Therefore, if an ML system is built for a business, it must be motivated by business objectives, which need to be translated into ML objectives to guide the development of ML models.

Before building an ML system, we need to understand the requirements that the system needs to meet to be considered a good system. The exact requirements vary from use case to use case, and in this chapter, we fo-

cused on the four most general requirements: reliability, scalability, maintainability, and adaptability. Techniques to satisfy each of these requirements will be covered throughout the book.

Building an ML system isn't a one-off task but an iterative process. In this chapter, we discussed the iterative process to develop an ML system that met those preceding requirements.

We ended the chapter on a philosophical discussion of the role of data in ML systems. There are still many people who believe that having intelligent algorithms will eventually trump having a large amount of data. However, the success of systems including AlexNet, BERT, and GPT showed that the progress of ML in the last decade relies on having access to a large amount of data.[24] 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.

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

1 Eugene Yan has a great post on how data scientists can understand the business intent and context of the projects they work on.

2 Milton Friedman, "A Friedman Doctrine—The Social Responsibility of Business Is to Increase Its Profits," *New York Times Magazine*, September 13, 1970, *https://oreil.ly/Fmbem*.

3 We'll cover batch prediction and online prediction in Chapter 7.

4 Ashok Chandrashekar, Fernando Amat, Justin Basilico, and Tony Jebara, "Artwork Personalization at Netflix," *Netflix Technology Blog*, December 7, 2017,

https://oreil.ly/UEDmw.

5 Carlos A. Gomez-Uribe and Neil Hunt, "The Netflix Recommender System: Algorithms, Business Value, and Innovation," *ACM Transactions on Management Information Systems* 6, no. 4 (January 2016): 13, https://oreil.ly/JkEPB.

6 Parmy Olson, "Nearly Half of All 'AI Startups' Are Cashing In on Hype," *Forbes*, March 4, 2019, https://oreil.ly/w5kOr.

7 "2020 State of Enterprise Machine Learning," Algorithmia, 2020, https://oreil.ly/FlIV1.

8 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 (Leah Schoeb, "Cloud Scalability: Scale Up vs Scale Out," *Turbonomic Blog*, March 15, 2018, https://oreil.ly/CFPtb).

9 Sean Wolfe, "Amazon's One Hour of Downtime on Prime Day May Have Cost It up to $100 Million in Lost Sales," *Business Insider*, July 19, 2018, https://oreil.ly/VBezI.

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

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

12 Note that objective functions are mathematical functions, which are different from the business and ML objectives we discussed earlier in this chapter.

13 Joe Kukura, "Facebook Employee Raises Powered by 'Really Dangerous' Algorithm That Favors Angry Posts," *SFist*, September 24, 2019, https://oreil.ly/PXtGi; Kevin Roose, "The Making of a YouTube Radical," *New York Times*, June 8, 2019, https://oreil.ly/KYqzF.

14 For simplicity, let's pretend for now that we know how to measure a post's quality.

15 Wikipedia, s.v. "Pareto optimization," https://oreil.ly/NdApy. While you're at it, you might also want to read Jin and Sendhoff's great paper on applying Pareto optimization for ML, in which the authors claimed that "machine learning is inherently a multiobjective task" (Yaochu Jin and Bernhard Sendhoff, "Pareto-Based Multiobjective Machine Learning: An Overview and Case Studies," *IEEE Transactions on Systems, Man, and Cybernetics—Part C: Applications and Reviews* 38, no. 3 [May 2008], https://oreil.ly/f1aKk).

16 Anand Rajaraman, “More Data Usually Beats Better Algorithms,” *Datawocky*, March 24, 2008, *https://oreil.ly/wNwhV*.

17 Rich Sutton, “The Bitter Lesson,” March 13, 2019, *https://oreil.ly/RhOp9*.

18 Tweet by Dr. Judea Pearl (@yudapearl), September 27, 2020, *https://oreil.ly/wFbHb*.

19 “Deep Learning and Innate Priors” (Chris Manning versus Yann LeCun debate), February 2, 2018, video, 1:02:55, *https://oreil.ly/b3hb1*.

20 Sutton, “The Bitter Lesson.”

21 Alon Halevy, Peter Norvig, and Fernando Pereira, “The Unreasonable Effectiveness of Data,” *IEEE Computer Society*, March/April 2009, *https://oreil.ly/WkN6p*.

22 Monica Rogati, “The AI Hierarchy of Needs,” *Hackernoon Newsletter*, June 12, 2017, *https://oreil.ly/3nxJ8*.

23 Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson, “One Billion Word Benchmark for Measuring Progress in Statistical Language Modeling,” *arXiv*, December 11, 2013, *https://oreil.ly/1AdO6*.

24 Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” in *Advances in Neural Information Processing Systems*, vol. 25, ed. F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Curran Associates, 2012), *https://oreil.ly/MFYp9*; Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” *arXiv*, 2019, *https://oreil.ly/TN8fN*; “Better Language Models and Their Implications,” OpenAI blog, February 14, 2019, *https://oreil.ly/SGV7g*.