Summarize

Chapter 9. Continual Learning and Test in Production

In Chapter 8, we discussed various ways an ML system can fail in production. We focused on one especially thorny problem that has generated much discussion among both researchers and practitioners: data distribution shifts. We also discussed multiple monitoring techniques and tools to detect data distribution shifts.

This chapter is a continuation of this discussion: how do we adapt our models to data distribution shifts? The answer is by continually updating our ML models. We'll start with a discussion on what continual learning is and its challenges—spoiler: continual learning is largely an infrastructural problem. Then we'll lay out a four-stage plan to make continual learning a reality.

After you've set up your infrastructure to allow you to update your models as frequently as you want, you might want to consider the question that I've been asked by almost every single ML engineer I've met: "How often should I retrain my models?" This question is the focus of the next section of the book.

If the model is retrained to adapt to the changing environment, evaluating it on a stationary test set isn't enough. We'll cover a seemingly terrifying but necessary concept: test in production. This process is a way to test your systems with live data in production to ensure that your updated model indeed works without catastrophic consequences.

Topics in this chapter and the previous chapter are tightly coupled. Test in production is complementary to monitoring. If monitoring means passively keeping track of the outputs of whatever model is being used, test in production means proactively choosing which model to produce out-

puts so that we can evaluate it. The goal of both monitoring and test in production is to understand a model's performance and figure out when to update it. The goal of continual learning is to safely and efficiently automate the update. All of these concepts allow us to design an ML system that is maintainable and adaptable to changing environments.

This is the chapter I'm most excited to write about, and I hope that I can get you excited about it too!

Continual Learning

When hearing "continual learning," many people think of the training paradigm where a model updates itself with every incoming sample in production. Very few companies actually do that. First, if your model is a neural network, learning with every incoming sample makes it susceptible to catastrophic forgetting. Catastrophic forgetting refers to the tendency of a neural network to completely and abruptly forget previously learned information upon learning new information.[1]

Second, it can make training more expensive—most hardware backends today were designed for batch processing, so processing only one sample at a time causes a huge waste of compute power and is unable to exploit data parallelism.

Companies that employ continual learning in production update their models in micro-batches. For example, they might update the existing model after every 512 or 1,024 examples—the optimal number of examples in each micro-batch is task dependent.

The updated model shouldn't be deployed until it's been evaluated. This means that you shouldn't make changes to the existing model directly. Instead, you create a replica of the existing model and update this replica on new data, and only replace the existing model with the updated replica if the updated replica proves to be better. The existing model is called the champion model, and the updated replica, the challenger. This process is shown in Figure 9-1. This is an oversimplification of the process for the sake of understanding. In reality, a company might have multiple challengers at the same time, and handling the failed challenger is a lot more sophisticated than simply discarding it.

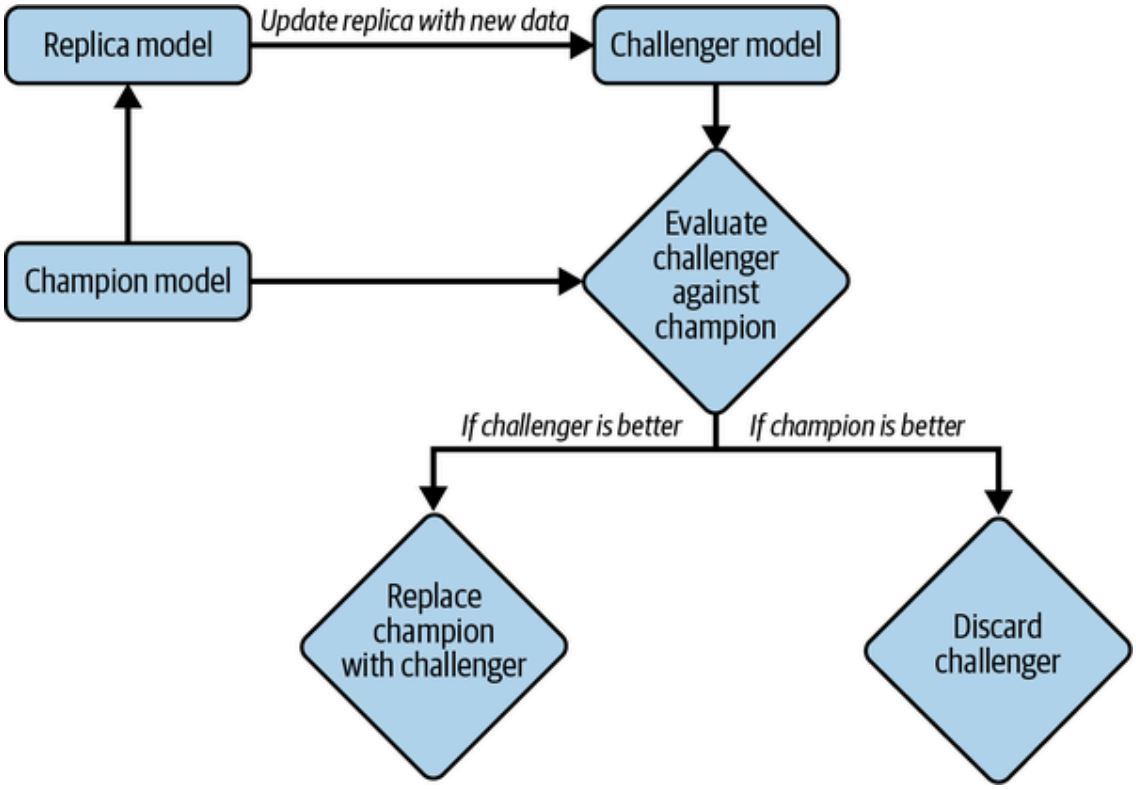

Figure 9-1. A simplification of how continual learning might work in production. In reality, the process of handling the failed challenger is a lot more sophisticated than simply discarding it.

Still, the term "continual learning" makes people imagine updating models very frequently, such as every 5 or 10 minutes. Many people argue that most companies don't need to update their models that frequently because of two reasons. First, they don't have enough traffic (i.e., enough new data) for that retraining schedule to make sense. Second, their models don't decay that fast. I agree with them. If changing the retraining schedule from a week to a day gives no return and causes more overhead, there's no need to do it.

Stateless Retraining Versus Stateful Training

However, continual learning isn't about the retraining frequency, but the manner in which the model is retrained. Most companies do *stateless retraining*—the model is trained from scratch each time. Continual learning means also allowing *stateful training*—the model continues training on new data.[2] Stateful training is also known as fine-tuning or incremental learning. The difference between stateless retraining and stateful training is visualized in Figure 9-2.

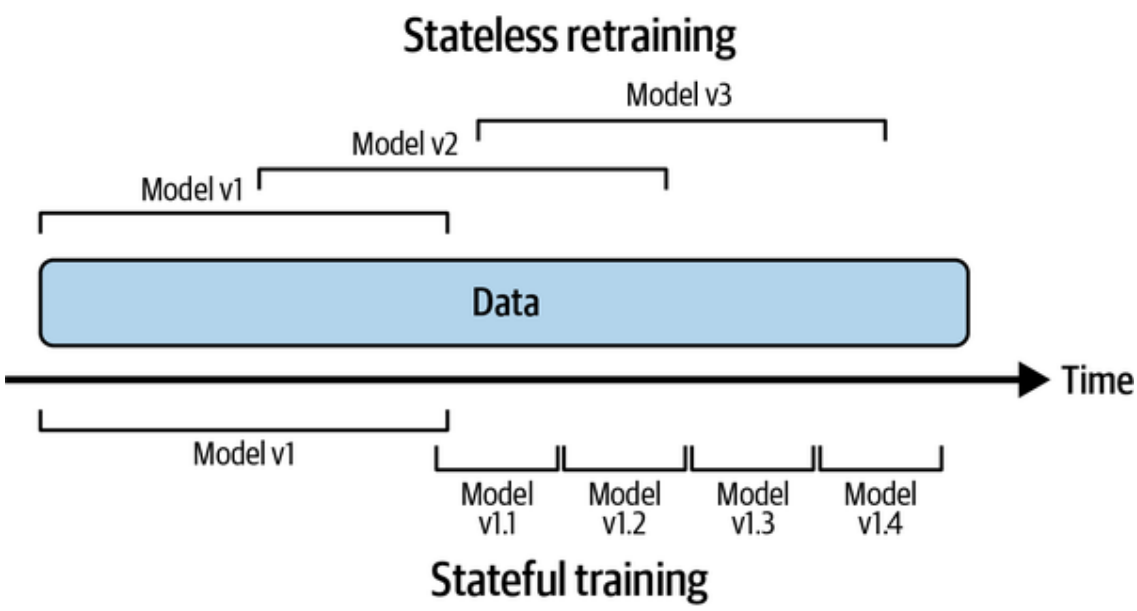

Figure 9-2. Stateless retraining versus stateful training

Stateful training allows you to update your model with less data. Training a model from scratch tends to require a lot more data than fine-tuning the same model. For example, if you retrain your model from scratch, you might need to use all data from the last three months. However, if you fine-tune your model from yesterday's checkpoint, you only need to use data from the last day.

Grubhub found out that stateful training allows their models to converge faster and require much less compute power. Going from daily stateless retraining to daily stateful training reduced their training compute cost 45 times and increased their purchase-through rate by 20%.[3]

One beautiful property that is often overlooked is that with stateful training, it might be possible to avoid storing data altogether. In the traditional stateless retraining, a data sample might be reused during multiple training iterations of a model, which means that data needs to be stored. This isn't always possible, especially for data with strict privacy requirements. In the stateful training paradigm, each model update is trained using only the fresh data, so a data sample is used only once for training, as shown in Figure 9-2. This means that it's possible to train your model without having to store data in permanent storage, which helps eliminate many concerns about data privacy. However, this is overlooked because today's let's-keep-track-of-everything practice still makes many companies reluctant to throw away data.

Stateful training doesn't mean no training from scratch. The companies that have most successfully used stateful training also occasionally train

their model from scratch on a large amount of data to calibrate it. Alternatively, they might also train their model from scratch in parallel with stateful training and then combine both updated models using techniques such as parameter server.[4]

Once your infrastructure is set up to allow both stateless retraining and stateful training, the training frequency is just a knob to twist. You can update your models once an hour, once a day, or whenever a distribution shift is detected. How to find the optimal retraining schedule will be discussed in the section "How Often to Update Your Models".

Continual learning is about setting up infrastructure in a way that allows you, a data scientist or ML engineer, to update your models whenever it is needed, whether from scratch or fine-tuning, and to deploy this update quickly.

You might wonder: stateful training sounds cool, but how does this work if I want to add a new feature or another layer to my model? To answer this, we must differentiate two types of model updates:

Model iteration

A new feature is added to an existing model architecture or the model architecture is changed.

Data iteration

The model architecture and features remain the same, but you refresh this model with new data.

As of today, stateful training is mostly applied for data iteration, as changing your model architecture or adding a new feature still requires training the resulting model from scratch. There has been research showing that it might be possible to bypass training from scratch for model iteration by using techniques such as knowledge transfer (Google, 2015) and model surgery (OpenAI, 2019). According to OpenAI, "Surgery transfers trained weights from one network to another after a selection process to determine which sections of the model are unchanged and which must be re-initialized."[5] Several large research labs have experimented with this; however, I'm not aware of any clear results in the industry.

TERMINOLOGY AMBIGUITY

I use the term “continual learning” instead of “online learning” because when I say “online learning,” people usually think of online education. If you enter “online learning” on Google, the top results will likely be about online courses.

Some people use “online learning” to refer to the specific setting where a model learns from each incoming new sample. In this setting, continual learning is a generalization of online learning.

I also use the term “continual learning” instead of “continuous learning.” Continuous learning refers to the regime in which your model continuously learns with each incoming sample, whereas with continual learning, the learning is done in a series of batches or micro-batches.

Continuous learning is sometimes used to refer to continuous delivery of ML, which is closely related to continual learning as both help companies to speed up the iteration cycle of their ML models. However, the difference is that “continuous learning,” when used in this sense, is from the DevOps perspective about setting up the pipeline for continuous delivery, whereas “continual learning” is from the ML perspective.

Due to the ambiguity of the term “continuous learning,” I hope that the community can stay away from this term altogether.

Why Continual Learning?

We discussed that continual learning is about setting up infrastructure so that you can update your models and deploy these changes as fast as you want. But why would you need the ability to update your models as fast as you want?

The first use case of continual learning is to combat data distribution shifts, especially when the shifts happen suddenly. Imagine you’re building a model to determine the prices for a ride-sharing service like Lyft.[6] Historically, the ride demand on a Thursday evening in this particular neighborhood is slow, so the model predicts low ride prices, which makes it less appealing for drivers to get on the road. However, on this Thursday evening, there’s a big event in the neighborhood, and suddenly the ride

demand surges. If your model can't respond to this change quickly enough by increasing its price prediction and mobilizing more drivers to that neighborhood, riders will have to wait a long time for a ride, which causes negative user experience. They might even switch to a competitor, which causes you to lose revenue.

Another use case of continual learning is to adapt to rare events. Imagine you work for an ecommerce website like Amazon. Black Friday is an important shopping event that happens only once a year. There's no way you will be able to gather enough historical data for your model to be able to make accurate predictions on how your customers will behave throughout Black Friday this year. To improve performance, your model should learn throughout the day with fresh data. In 2019, Alibaba acquired Data Artisans, the team leading the development of the stream processing framework Apache Flink, for $103 million so that the team could help them adapt Flink for ML use cases.[7] Their flagship use case was making better recommendations on Singles Day, a shopping occasion in China similar to Black Friday in the US.

A huge challenge for ML production today that continual learning can help overcome is the *continuous cold start* problem. The cold start problem arises when your model has to make predictions for a new user without any historical data. For example, to recommend to a user what movies they might want to watch next, a recommender system often needs to know what that user has watched before. But if that user is new, you won't have their watch history and will have to generate them something generic, e.g., the most popular movies on your site right now.[8]

Continuous cold start is a generalization of the cold start problem,[9] as it can happen not just with new users but also with existing users. For example, it can happen because an existing user switches from a laptop to a mobile phone, and their behavior on a phone is different from their behavior on a laptop. It can happen because users are not logged in—most news sites don't require readers to log in to read.

It can also happen when a user visits a service so infrequently that whatever historical data the service has about this user is outdated. For example, most people only book hotels and flights a few times a year. Coveo, a company that provides search engine and recommender systems to ecommerce websites, found that it is common for an ecommerce site to

have more than 70% of their shoppers visit their site less than three times a year.[10]

If your model doesn't adapt quickly enough, it won't be able to make recommendations relevant to these users until the next time the model is updated. By that time, these users might have already left the service because they don't find anything relevant to them.

If we could make our models adapt to each user within their visiting session, the models would be able to make accurate, relevant predictions to users even on their first visit. TikTok, for example, has successfully applied continual learning to adapt their recommender system to each user within minutes. You download the app and, after a few videos, TikTok's algorithms are able to predict with high accuracy what you want to watch next.[11] I don't think everyone should try to build something as addictive as TikTok, but it's proof that continual learning can unlock powerful predictive potential.

"Why continual learning?" should be rephrased as "why not continual learning?" Continual learning is a superset of batch learning, as it allows you to do everything the traditional batch learning can do. But continual learning also allows you to unlock use cases that batch learning can't.

If continual learning takes the same effort to set up and costs the same to do as batch learning, there's no reason not to do continual learning. As of writing this book, there are still a lot of challenges in setting up continual learning, as we'll go deeper into in the following section. However, MLOps tooling for continual learning is maturing, which means, one day not too far in the future, it might be as easy to set up continual learning as batch learning.

Continual Learning Challenges

Even though continual learning has many use cases and many companies have applied it with great success, continual learning still has many challenges. In this section, we'll discuss three major challenges: fresh data access, evaluation, and algorithms.

Fresh data access challenge

The first challenge is the challenge to get fresh data. If you want to update your model every hour, you need new data every hour. Currently, many companies pull new training data from their data warehouses. The speed at which you can pull data from your data warehouses depends on the speed at which this data is deposited into your data warehouses. The speed can be slow, especially if data comes from multiple sources. An alternative is to allow pull data before it's deposited into data warehouses, e.g., directly from real-time transports such as Kafka and Kinesis that transport data from applications to data warehouses,[12] as shown in Figure 9-3.

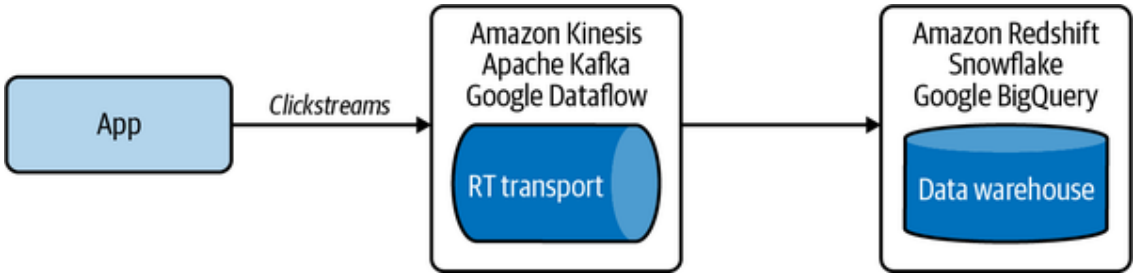

Figure 9-3. Pulling data directly from real-time transports, before it's deposited into data warehouses, can allow you to access fresher data

Being able to pull fresh data isn't enough. If your model needs labeled data to update, as most models today do, this data will need to be labeled as well. In many applications, the speed at which a model can be updated is bottlenecked by the speed at which data is labeled.

The best candidates for continual learning are tasks where you can get natural labels with short feedback loops. Examples of these tasks are dynamic pricing (based on estimated demand and availability), estimating time of arrival, stock price prediction, ads click-through prediction, and recommender systems for online content like tweets, songs, short videos, articles, etc.

However, these natural labels are usually not generated as labels, but rather as behavioral activities that need to be extracted into labels. Let's walk through an example to make this clear. If you run an ecommerce website, your application might register that at 10:33 p.m., user A clicks on the product with the ID of 32345. Your system needs to look back into the logs to see if this product ID was ever recommended to this user, and if yes, then what query prompted this recommendation, so that your sys-

tem can match this query to this recommendation and label this recommendation as a good recommendation, as shown in Figure 9-4.

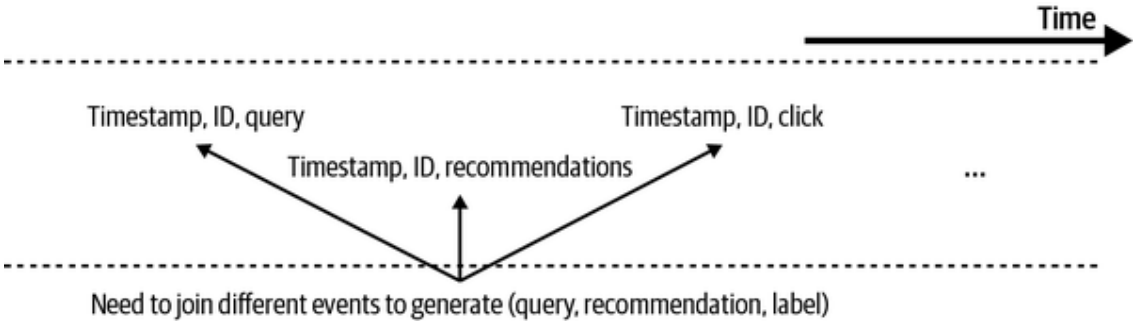

Figure 9-4. A simplification of the process of extracting labels from user feedback

The process of looking back into the logs to extract labels is called label computation. It can be quite costly if the number of logs is large. Label computation can be done with batch processing: e.g., waiting for logs to be deposited into data warehouses first before running a batch job to extract all labels from logs at once. However, as discussed previously, this means that we'd need to wait for data to be deposited first, then wait for the next batch job to run. A much faster approach would be to leverage stream processing to extract labels from the real-time transports directly.[13]

If your model's speed iteration is bottlenecked by labeling speed, it's also possible to speed up the labeling process by leveraging programmatic labeling tools like Snorkel to generate fast labels with minimal human intervention. It might also be possible to leverage crowdsourced labels to quickly annotate fresh data.

Given that tooling around streaming is still nascent, architecting an efficient streaming-first infrastructure for accessing fresh data and extracting fast labels from real-time transports can be engineering-intensive and costly. The good news is that tooling around streaming is growing fast. Confluent, the platform built on top of Kafka, is a $16 billion company as of October 2021. In late 2020, Snowflake started a team focusing on streaming.[14] As of September 2021, Materialize has raised $100 million to develop a streaming SQL database.[15] As tooling around streaming matures, it'll be much easier and cheaper for companies to develop a streaming-first infrastructure for ML.

Evaluation challenge

The biggest challenge of continual learning isn't in writing a function to continually update your model—you can do that by writing a script! The biggest challenge is in making sure that this update is good enough to be deployed. In this book, we've discussed how ML systems make catastrophic failures in production, from millions of minorities being unjustly denied loans, to drivers who trust autopilot too much being involved in fatal crashes.[16]

The risks for catastrophic failures amplify with continual learning. First, the more frequently you update your models, the more opportunities there are for updates to fail.

Second, continual learning makes your models more susceptible to coordinated manipulation and adversarial attack. Because your models learn online from real-world data, it makes it easier for users to input malicious data to trick models into learning wrong things. In 2016, Microsoft released Tay, a chatbot capable of learning through "casual and playful conversation" on Twitter. As soon as Tay launched, trolls started tweeting the bot racist and misogynist remarks. The bot soon began to post inflammatory and offensive tweets, causing Microsoft to shut down the bot 16 hours after its launch.[17]

To avoid similar or worse incidents, it's crucial to thoroughly test each of your model updates to ensure its performance and safety before deploying the updates to a wider audience. We already discussed model offline evaluation in Chapter 6, and will discuss online evaluation (test in production) in this chapter.

When designing the evaluation pipeline for continual learning, keep in mind that evaluation takes time, which can be another bottleneck for model update frequency. For example, a major online payment company I worked with has an ML system to detect fraudulent transactions.[18] The fraud patterns change quickly, so they'd like to update their system quickly to adapt to the changing patterns. They can't deploy the new model before it's been A/B tested against the current model. However, due to the imbalanced nature of the task—most transactions aren't fraud—it takes them approximately two weeks to see enough fraud transac-

tions to be able to accurately assess which model is better.[19] Therefore, they can only update their system every two weeks.

Algorithm challenge

Compared to the fresh data challenge and the evaluation, this is a "softer" challenge as it only affects certain algorithms and certain training frequencies. To be precise, it only affects matrix-based and tree-based models that want to be updated very fast (e.g., hourly).

To illustrate this point, consider two different models: a neural network and a matrix-based model, such as a collaborative filtering model. The collaborative filtering model uses a user-item matrix and a dimension reduction technique.

You can update the neural network model with a data batch of any size. You can even perform the update step with just one data sample. However, if you want to update the collaborative filtering model, you first need to use the entire dataset to build the user-item matrix before performing dimensionality reduction on it. Of course, you can apply dimensionality reduction to your matrix each time you update the matrix with a new data sample, but if your matrix is large, the dimensionality reduction step would be too slow and expensive to perform frequently. Therefore, this model is less suitable for learning with a partial dataset than the preceding neural network model.[20]

It's much easier to adapt models like neural networks than matrix-based and tree-based models to the continual learning paradigm. However, there have been algorithms to create tree-based models that can learn from incremental amounts of data, most notably Hoeffding Tree and its variants Hoeffding Window Tree and Hoeffding Adaptive Tree,[21] but their uses aren't yet widespread.

Not only does the learning algorithm need to work with partial datasets, but the feature extract code has to as well. We discussed in the section "Scaling" that it's often necessary to scale your features using statistics such as the min, max, median, and variance. To compute these statistics for a dataset, you often need to do a pass over the entire dataset. When your model can only see a small subset of data at a time, in theory, you can compute these statistics for each subset of data. However, this means

that these statistics will fluctuate a lot between different subsets. The statistics computed from one subset might differ wildly from the next subset, making it difficult for the model trained on one subset to generalize to the next subset.

To keep these statistics stable across different subsets, you might want to compute these statistics online. Instead of using the mean or variance from all your data at once, you compute or approximate these statistics incrementally as you see new data, such as the algorithms outlined in "Optimal Quantile Approximation in Streams."[22] Popular frameworks today offer some capacity for computing running statistics—for example, sklearn's StandardScaler has a `partial_fit` that allows a feature scaler to be used with running statistics—but the built-in methods are slow and don't support a wide range of running statistics.

Four Stages of Continual Learning

We've discussed what continual learning is, why continual learning matters, and the challenges of continual learning. Next, we'll discuss how to overcome these challenges and make continual learning happen. As of the writing of this book, continual learning isn't something that companies start out with. The move toward continual learning happens in four stages, as outlined next. We'll go over what happens in each stage as well as the requirements necessary to move from a previous stage to this stage.

Stage 1: Manual, stateless retraining

In the beginning, the ML team often focuses on developing ML models to solve as many business problems as possible. For example, if your company is an ecommerce website, you might develop four models in the following succession:

1. A model to detect fraudulent transactions
2. A model to recommend relevant products to users
3. A model to predict whether a seller is abusing a system
4. A model to predict how long it will take to ship an order

Because your team is focusing on developing new models, updating existing models takes a backseat. You update an existing model only when the

following two conditions are met: the model's performance has degraded to the point that it's doing more harm than good, and your team has time to update it. Some of your models are being updated once every six months. Some are being updated once a quarter. Some have been out in the wild for a year and haven't been updated at all.

The process of updating a model is manual and ad hoc. Someone, usually a data engineer, has to query the data warehouse for new data. Someone else cleans this new data, extracts features from it, retrains that model from scratch on both the old and new data, and then exports the updated model into a binary format. Then someone else takes that binary format and deploys the updated model. Oftentimes, the code encapsulating data, features, and model logic was changed during the retraining process but these changes failed to be replicated to production, causing bugs that are hard to track down.

If this process sounds painfully familiar to you, you're not alone. A vast majority of companies outside the tech industry—e.g., any company that adopted ML less than three years ago and doesn't have an ML platform team—are in this stage.[23]

Stage 2: Automated retraining

After a few years, your team has managed to deploy models to solve most of the obvious problems. You have anywhere between 5 and 10 models in production. Your priority is no longer to develop new models, but to maintain and improve existing models. The ad hoc, manual process of updating models mentioned from the previous stage has grown into a pain point too big to be ignored. Your team decides to write a script to automatically execute all the retraining steps. This script is then run periodically using a batch process such as Spark.

Most companies with somewhat mature ML infrastructure are in this stage. Some sophisticated companies run experiments to determine the optimal retraining frequency. However, for most companies in this stage, the retraining frequency is set based on gut feeling—e.g., "once a day seems about right" or "let's kick off the retraining process each night when we have idle compute."

When creating scripts to automate the retraining process for your system, you need to take into account that different models in your system might require different retraining schedules. For example, consider a recommender system that consists of two models: one model to generate embeddings for all products, and another model to rank the relevance of each product given a query. The embedding model might need to be retrained a lot less frequently than the ranking model. Because products' characteristics don't change that often, you might be able to get away with retraining your embeddings once a week,[24] whereas your ranking models might need to be retrained once a day.

The automating script might get even more complicated if there are dependencies among your models. For example, because the ranking model depends on the embeddings, when the embeddings change, the ranking model should be updated too.

Requirements

If your company has ML models in production, it's likely that your company already has most of the infrastructure pieces needed for automated retraining. The feasibility of this stage revolves around the feasibility of writing a script to automate your workflow and configure your infrastructure to automatically:

1. Pull data.
2. Downsample or upsample this data if necessary.
3. Extract features.
4. Process and/or annotate labels to create training data.
5. Kick off the training process.
6. Evaluate the newly trained model.
7. Deploy it.

How long it will take to write this script depends on many factors, including the script writer's competency. However, in general, the three major factors that will affect the feasibility of this script are: scheduler, data, and model store.

A scheduler is basically a tool that handles task scheduling, which we'll cover in the section "Cron, Schedulers, and Orchestrators". If you don't already have a scheduler, you'll need time to set up one. However, if you al-

ready have a scheduler such as Airflow or Argo, wiring the scripts together shouldn't be that hard.

The second factor is the availability and accessibility of your data. Do you need to gather data yourself into your data warehouse? Will you have to join data from multiple organizations? Do you need to extract a lot of features from scratch? Will you also need to label your data? The more questions you answer yes to, the more time it will take to set up this script. Stefan Krawczyk, ML/data platform manager at Stitch Fix, commented that he suspects most people's time might be spent here.

The third factor you'll need is a model store to automatically version and store all the artifacts needed to reproduce a model. The simplest model store is probably just an S3 bucket that stores serialized blobs of models in some structured manner. However, blob storage like S3 is neither very good at versioning artifacts nor human-readable. You might need a more mature model store like Amazon SageMaker (managed service) and Databricks' MLflow (open source). We'll go into detail on what a model store is and evaluate different model stores in the section "Model Store".

FEATURE REUSE (LOG AND WAIT)

When creating training data from new data to update your model, remember that the new data has already gone through the prediction service. This prediction service has already extracted features from this new data to input into models for predictions. Some companies reuse these extracted features for model retraining, which both saves computation and allows for consistency between prediction and training. This approach is known as "log and wait." It's a classic approach to reduce the train-serving skew discussed in Chapter 8 (see the section "Production data differing from training data").

Log and wait isn't yet a popular approach, but it's getting more popular. Faire has a great blog post discussing the pros and cons of their "log and wait" approach.

Stage 3: Automated, stateful training

In stage 2, each time you retrain your model, you train it from scratch (stateless retraining). It makes your retraining costly, especially for retraining with a higher frequency. You read the section "Stateless Retraining Versus Stateful Training" and decide that you want to do state-

ful training—why train on data from the last three months every day when you can continue training using only data from the last day?

So in this stage, you reconfigure your automatic updating script so that, when the model update is kicked off, it first locates the previous checkpoint and loads it into memory before continuing training on this checkpoint.

Requirements

The main thing you need in this stage is a change in the mindset: retraining from scratch is such a norm—many companies are so used to data scientists handing off a model to engineers to deploy from scratch each time—that many companies don't think about setting up their infrastructure to enable stateful training.

Once you're committed to stateful training, reconfiguring the updating script is straightforward. The main thing you need at this stage is a way to track your data and model lineage. Imagine you first upload model version 1.0. This model is updated with new data to create model version 1.1, and so on to create model 1.2. Then another model is uploaded and called model version 2.0. This model is updated with new data to create model version 2.1. After a while, you might have model version 3.32, model version 2.11, model version 1.64. You might want to know how these models evolve over time, which model was used as its base model, and which data was used to update it so that you can reproduce and debug it. As far as I know, no existing model store has this model lineage capacity, so you'll likely have to build the solution in-house.

If you want to pull fresh data from the real-time transports instead of from data warehouses, as discussed in the section "Fresh data access challenge", and your streaming infrastructure isn't mature enough, you might need to revamp your streaming pipeline.

Stage 4: Continual learning

At stage 3, your models are still updated based on a fixed schedule set out by developers. Finding the optimal schedule isn't straightforward and can be situation-dependent. For example, last week, nothing much happened in the market, so your models didn't decay that fast. However, this week,

a lot of events happen, so your models decay much faster and require a much faster retraining schedule.

Instead of relying on a fixed schedule, you might want your models to be automatically updated whenever data distributions shift and the model's performance plummets.

The holy grail is when you combine continual learning with edge deployment. Imagine you can ship a base model with a new device—a phone, a watch, a drone, etc.—and the model on that device will continually update and adapt to its environment as needed without having to sync with a centralized server. There will be no need for a centralized server, which means no centralized server cost. There will also be no need to transfer data back and forth between device and cloud, which means better data security and privacy!

Requirements

The move from stage 3 to stage 4 is steep. You'll first need a mechanism to trigger model updates. This trigger can be:

Time-based

For example, every five minutes

Performance-based

For example, whenever model performance plummets

Volume-based

For example, whenever the total amount of labeled data increases by 5%

Drift-based

For example, whenever a major data distribution shift is detected

For this trigger mechanism to work, you'll need a solid monitoring solution. We discussed in the section "Monitoring and Observability" that the hard part is not to detect the changes, but to determine which of these changes matter. If your monitoring solution gives a lot of false alerts,

your model will end up being updated much more frequently than it needs to be.

You'll also need a solid pipeline to continually evaluate your model updates. Writing a function to update your models isn't much different from what you'd do in stage 3. The hard part is to ensure that the updated model is working properly. We'll go over various testing techniques you can use in the section "Test in Production".

How Often to Update Your Models

Now that your infrastructure has been set up to update a model quickly, you started asking the question that has been haunting ML engineers at companies of all shapes and sizes: "How often should I update my models?" Before attempting to answer that question, we first need to figure out how much gain your model will get from being updated with fresh data. The more gain your model can get from fresher data, the more frequently it should be retrained.

Value of data freshness

The question of how often to update a model becomes a lot easier if we know how much the model performance will improve with updating. For example, if we switch from retraining our model every month to every week, how much performance gain can we get? What if we switch to daily retraining? People keep saying that data distributions shift, so fresher data is better, but how much better is fresher data?

One way to figure out the gain is by training your model on the data from different time windows in the past and evaluating it on the data from today to see how the performance changes. For example, consider that you have data from the year 2020. To measure the value of data freshness, you can experiment with training model version A on the data from January to June 2020, model version B on the data from April to September, and model version C on the data from June to November, then test each of these model versions on the data from December, as shown in Figure 9-5. The difference in the performance of these versions will give you a sense of the performance gain your model can get from fresher data. If the model trained on data from a quarter ago is much

worse than the model trained on data from a month ago, you know that you shouldn't wait a quarter to retrain your model.

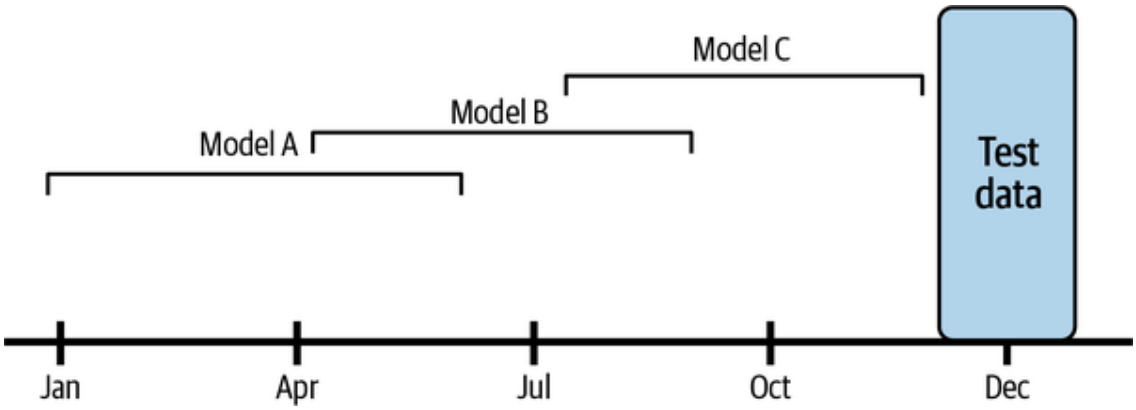

Figure 9-5. To get a sense of the performance gain you can get from fresher data, train your model on data from different time windows in the past and test on data from today to see how the performance changes

This is a simple example to illustrate how the data freshness experiment works. In practice, you might want your experiments to be much more fine-grained, operating not in months but in weeks, days, even hours or minutes. In 2014, Facebook did a similar experiment for ad click-through-rate prediction and found out that they could reduce the model's loss by 1% by going from retraining weekly to retraining daily, and this performance gain was significant enough for them to switch their retraining pipeline from weekly to daily.[25] Given that online contents today are so much more diverse and users' attention online changes much faster, we can imagine that the value of data freshness for ad click-through rate is even higher. Some of the companies with sophisticated ML infrastructure have found enough performance gain to switch their retraining pipeline to every few minutes.[26]

Model iteration versus data iteration

We discussed earlier in this chapter that not all model updates are the same. We differentiated between model iteration (adding a new feature to an existing model architecture or changing the model architecture) and data iteration (same model architecture and features but you refresh this model with new data). You might wonder not only how often to update your model, but also what kind of model updates to perform.

In theory, you can do both types of updates, and in practice, you should do both from time to time. However, the more resources you spend in one approach, the fewer resources you can spend in another.

On the one hand, if you find that iterating on your data doesn't give you much performance gain, then you should spend your resources on finding a better model. On the other hand, if finding a better model architecture requires 100X compute for training and gives you 1% performance whereas updating the same model on data from the last three hours requires only 1X compute and also gives 1% performance gain, you'll be better off iterating on data.

Maybe in the near future, we'll get more theoretical understanding to know in what situation an approach will work better (cue "call for research"), but as of today, no book can give you the answer on which approach will work better for your specific model on your specific task. You'll have to do experiments to find out.

The question on how often to update your model is a difficult one to answer, and I hope that this section has sufficiently explained its nuances. In the beginning, when your infrastructure is nascent and the process of updating a model is manual and slow, the answer is: as often as you *can*.

However, as your infrastructure matures and the process of updating a model is partially automated and can be done in a matter of hours, if not minutes, the answer to this question is contingent on the answer to the following question: "How much performance gain would I get from fresher data?" It's important to run experiments to quantify the value of data freshness to your models.

Test in Production

Throughout this book, including this chapter, we've talked about the danger of deploying models that haven't been sufficiently evaluated. To sufficiently evaluate your models, you first need a mixture of offline evaluation discussed in Chapter 6 and online evaluation discussed in this section. To understand why offline evaluation isn't enough, let's go over two major test types for offline evaluation: test splits and backtests.

The first type of model evaluation you might think about is the good old test splits that you can use to evaluate your models offline, as discussed in Chapter 6. These test splits are usually static and have to be static so that you have a trusted benchmark to compare multiple models. It'll be hard

to compare the test results of two models if they are tested on different test sets.

However, if you update the model to adapt to a new data distribution, it's not sufficient to evaluate this new model on test splits from the old distribution. Assuming that the fresher the data, the more likely it is to come from the current distribution, one idea is to test your model on the most recent data that you have access to. So, after you've updated your model on the data from the last day, you might want to test this model on the data from the last hour (assuming that data from the last hour wasn't included in the data used to update your model). The method of testing a predictive model on data from a specific period of time in the past is known as a *backtest*.

The question is whether backtests are sufficient to replace static test splits. Not quite. If something went wrong with your data pipeline and some data from the last hour is corrupted, evaluating your model solely on this recent data isn't sufficient.

With backtests, you should still evaluate your model on a static test set that you have extensively studied and (mostly) trust as a form of sanity check.

Because data distributions shift, the fact that a model does well on the data from the last hour doesn't mean that it will continue doing well on the data in the future. The only way to know whether a model will do well in production is to deploy it. This insight led to one seemingly terrifying but necessary concept: test in production. However, test in production doesn't have to be scary. There are techniques to help you evaluate your models in production (mostly) safely. In this section, we'll cover the following techniques: shadow deployment, A/B testing, canary analysis, interleaving experiments, and bandits.

Shadow Deployment

Shadow deployment might be the safest way to deploy your model or any software update. Shadow deployment works as follows:

1. Deploy the candidate model in parallel with the existing model.

2. For each incoming request, route it to both models to make predictions, but only serve the existing model's prediction to the user.
3. Log the predictions from the new model for analysis purposes.

Only when you've found that the new model's predictions are satisfactory do you replace the existing model with the new model.

Because you don't serve the new model's predictions to users until you've made sure that the model's predictions are satisfactory, the risk of this new model doing something funky is low, at least not higher than the existing model. However, this technique isn't always favorable because it's expensive. It doubles the number of predictions your system has to generate, which generally means doubling your inference compute cost.

A/B Testing

A/B testing is a way to compare two variants of an object, typically by testing responses to these two variants, and determining which of the two variants is more effective. In our case, we have the existing model as one variant, and the candidate model (the recently updated model) as another variant. We'll use A/B testing to determine which model is better according to some predefined metrics.

A/B testing has become so prevalent that, as of 2017, companies like Microsoft and Google each conduct over 10,000 A/B tests annually.[27] It is many ML engineers' first response to how to evaluate ML models in production. A/B testing works as follows:

1. Deploy the candidate model alongside the existing model.
2. A percentage of traffic is routed to the new model for predictions; the rest is routed to the existing model for predictions. It's common for both variants to serve prediction traffic at the same time. However, there are cases where one model's predictions might affect another model's predictions—e.g., in ride-sharing's dynamic pricing, a model's predicted prices might influence the number of available drivers and riders, which, in turn, influence the other model's predictions. In those cases, you might have to run your variants alternatively, e.g., serve model A one day and then serve model B the next day.

3. Monitor and analyze the predictions and user feedback, if any, from both models to determine whether the difference in the two models' performance is statistically significant.

To do A/B testing the right way requires doing many things right. In this book, we'll discuss two important things. First, A/B testing consists of a randomized experiment: the traffic routed to each model has to be truly random. If not, the test result will be invalid. For example, if there's a selection bias in the way traffic is routed to the two models, such as users who are exposed to model A are usually on their phones whereas users exposed to model B are usually on their desktops, then if model A has better accuracy than model B, we can't tell whether it's because A is better than B or whether "being on a phone" influences the prediction quality.

Second, your A/B test should be run on a sufficient number of samples to gain enough confidence about the outcome. How to calculate the number of samples needed for an A/B test is a simple question with a very complicated answer, and I'd recommend readers reference a book on A/B testing to learn more.

The gist here is that if your A/B test result shows that a model is better than another with statistical significance, you can determine which model is indeed better. To measure statistical significance, A/B testing uses statistical hypothesis testing such as two-sample tests. We saw two-sample tests in Chapter 8 when we used them to detect distribution shifts. As a reminder, a two-sample test is a test to determine whether the difference between these two populations is statistically significant. In the distribution shift use case, if a statistical difference suggests that the two populations come from different distributions, this means that the original distribution has shifted. In the A/B testing use case, statistical differences mean that we've gathered sufficient evidence to show that one variant is better than the other variant.

Statistical significance, while useful, isn't foolproof. Say we run a two-sample test and get the result that model A is better than model B with the p-value of $p = 0.05$ or 5%, and we define statistical significance as $p \leq 0.5$. This means that if we run the same A/B testing experiment multiple times, (100 – 5 =) 95% of the time, we'll get the result that A is better than B, and the other 5% of the time, B is better than A. So even if the result is

statistically significant, it's possible that if we run the experiment again, we'll pick another model.

Even if your A/B test result isn't statistically significant, it doesn't mean that this A/B test fails. If you've run your A/B test with a lot of samples and the difference between the two tested models is statistically insignificant, maybe there isn't much difference between these two models, and it's probably OK for you to use either.

For readers interested in learning more about A/B testing and other statistical concepts important in ML, I recommend Ron Kohav's book *Trustworthy Online Controlled Experiments (A Practical Guide to A/B Testing)* (Cambridge University Press) and Michael Barber's great introduction to statistics for data science (much shorter).

Often, in production, you don't have just one candidate but multiple candidate models. It's possible to do A/B testing with more than two variants, which means we can have A/B/C testing or even A/B/C/D testing.

Canary Release

Canary release is a technique to reduce the risk of introducing a new software version in production by slowly rolling out the change to a small subset of users before rolling it out to the entire infrastructure and making it available to everybody.[28] In the context of ML deployment, canary release works as follows:

1. Deploy the candidate model alongside the existing model. The candidate model is called the canary.
2. A portion of the traffic is routed to the candidate model.
3. If its performance is satisfactory, increase the traffic to the candidate model. If not, abort the canary and route all the traffic back to the existing model.
4. Stop when either the canary serves all the traffic (the candidate model has replaced the existing model) or when the canary is aborted.

The candidate model's performance is measured against the existing model's performance according to the metrics you care about. If the can-

didate model's key metrics degrade significantly, the canary is aborted and all the traffic will be routed to the existing model.

Canary releases can be used to implement A/B testing due to the similarities in their setups. However, you can do canary analysis without A/B testing. For example, you don't have to randomize the traffic to route to each model. A plausible scenario is that you first roll out the candidate model to a less critical market before rolling out to everybody.

For readers interested in how canary release works in the industry, Netflix and Google have a great shared blog post on how automated canary analysis is used at their companies.

Interleaving Experiments

Imagine you have two recommender systems, A and B, and you want to evaluate which one is better. Each time, a model recommends 10 items users might like. With A/B testing, you'd divide your users into two groups: one group is exposed to A and the other group is exposed to B. Each user will be exposed to the recommendations made by one model.

What if instead of exposing a user to recommendations from a model, we expose that user to recommendations from both models and see which model's recommendations they will click on? That's the idea behind interleaving experiments, originally proposed by Thorsten Joachims in 2002 for the problems of search rankings.[29] In experiments, Netflix found that interleaving "reliably identifies the best algorithms with considerably smaller sample size compared to traditional A/B testing."[30]

Figure 9-6 shows how interleaving differs from A/B testing. In A/B testing, core metrics like retention and streaming are measured and compared between the two groups. In interleaving, the two algorithms can be compared by measuring user preferences. Because interleaving can be decided by user preferences, there's no guarantee that user preference will lead to better core metrics.

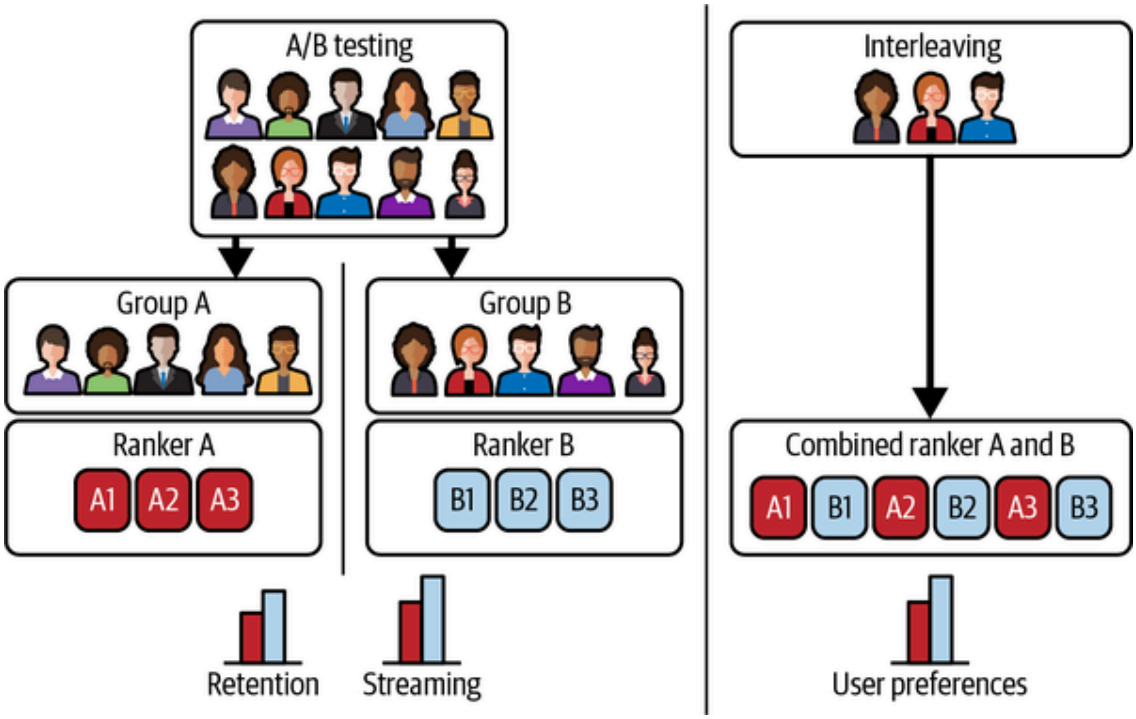

Figure 9-6. An illustration of interleaving versus A/B testing. Source: Adapted from an image by Parks et al.

When we show recommendations from multiple models to users, it's important to note that the position of a recommendation influences how likely a user will click on it. For example, users are much more likely to click on the top recommendation than the bottom recommendation. For interleaving to yield valid results, we must ensure that at any given position, a recommendation is equally likely to be generated by A or B. To ensure this, one method we can use is team-draft interleaving, which mimics the drafting process in sports. For each recommendation position, we randomly select A or B with equal probability, and the chosen model picks the top recommendation that hasn't already been picked.[31] A visualization of how this team-drafting method works is shown in Figure 9-7.

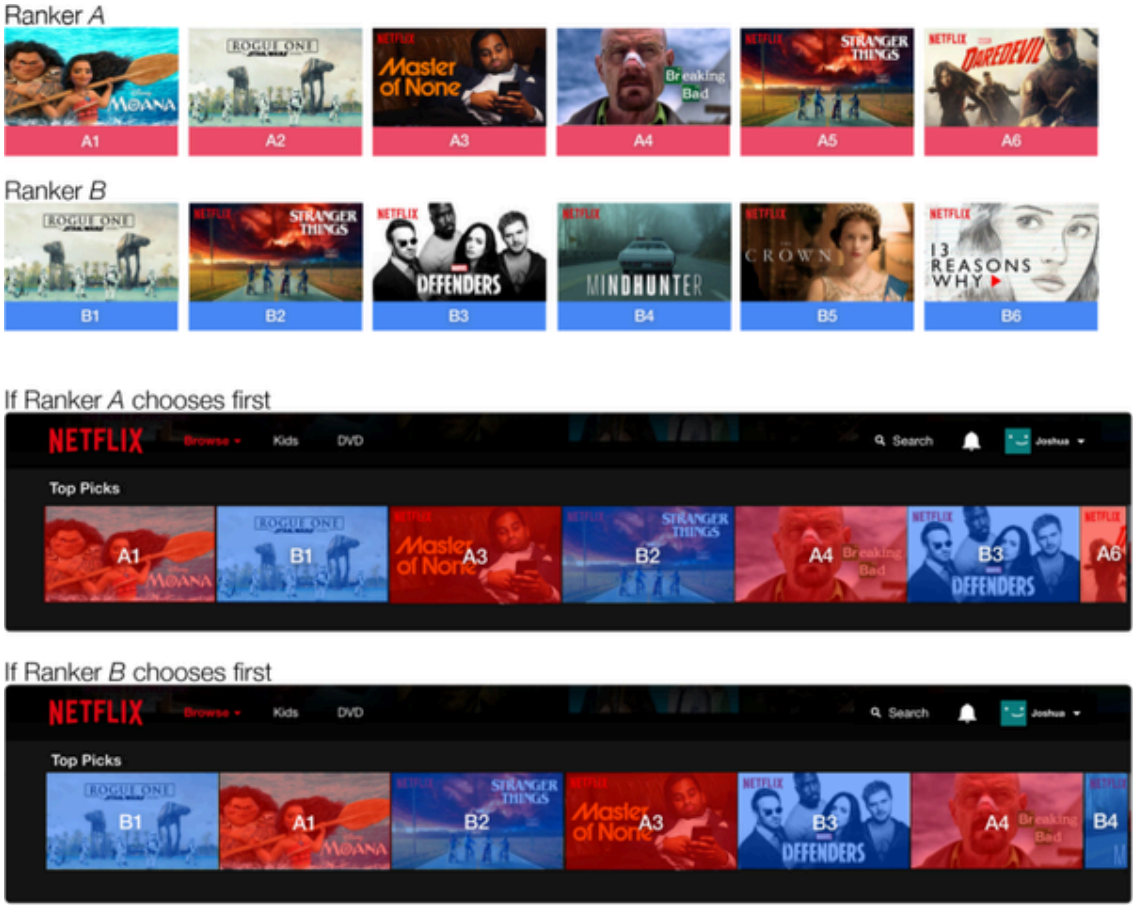

Figure 9-7. Interleaving video recommendations from two ranking algorithms using team draft. Source: Parks et al.[32]

Bandits

For those unfamiliar, bandit algorithms originated in gambling. A casino has multiple slot machines with different payouts. A slot machine is also known as a one-armed bandit, hence the name. You don't know which slot machine gives the highest payout. You can experiment over time to find out which slot machine is the best while maximizing your payout. Multi-armed bandits are algorithms that allow you to balance between exploitation (choosing the slot machine that has paid the most in the past) and exploration (choosing other slot machines that may pay off even more).

As of today, the standard method for testing models in production is A/B testing. With A/B testing, you randomly route traffic to each model for predictions and measure at the end of your trial which model works better. A/B testing is stateless: you can route traffic to each model without having to know about their current performance. You can do A/B testing even with batch prediction.

When you have multiple models to evaluate, each model can be considered a slot machine whose payout (i.e., prediction accuracy) you don't

know. Bandits allow you to determine how to route traffic to each model for prediction to determine the best model while maximizing prediction accuracy for your users. Bandit is stateful: before routing a request to a model, you need to calculate all models' current performance. This requires three things:

- Your model must be able to make online predictions.
- Preferably short feedback loops: you need to get feedback on whether a prediction is good or not. This is usually true for tasks where labels can be determined from users' feedback, like in recommendations—if users click on a recommendation, it's inferred to be good. If the feedback loops are short, you can update the payoff of each model quickly.
- A mechanism to collect feedback, calculate and keep track of each model's performance, and route prediction requests to different models based on their current performance.

Bandits are well-studied in academia and have been shown to be a lot more data-efficient than A/B testing (in many cases, bandits are even optimal). Bandits require less data to determine which model is the best and, at the same time, reduce opportunity cost as they route traffic to the better model more quickly. See discussions on bandits at LinkedIn, Netflix, Facebook, and Dropbox, Zillow, and Stitch Fix. For a more theoretical view, see Chapter 2 of *Reinforcement Learning* (Sutton and Barto 2020).

In an experiment by Google's Greg Rafferty, A/B testing required over 630,000 samples to get a confidence interval of 95%, whereas a simple bandit algorithm (Thompson Sampling) determined that a model was 5% better than the other with less than 12,000 samples.[33]

However, bandits are a lot more difficult to implement than A/B testing because it requires computing and keeping track of models' payoffs. Therefore, bandit algorithms are not widely used in the industry other than at a few big tech companies.

BANDIT ALGORITHMS

Many of the solutions for the multi-armed bandit problem can be used here. The simplest algorithm for exploration is ε-greedy. For a percentage of time, say 90% of the time or $\varepsilon = 0.9$, you route traffic to the model that is currently the best-performing one, and for the other 10% of the time, you route traffic to a random model. This means that for each of the predictions your system generates, 90% of them come from the best-at-that-point-in-time model.

Two of the most popular exploration algorithms are Thompson Sampling and Upper Confidence Bound (UCB). Thompson Sampling selects a model with a probability that this model is optimal given the current knowledge.[34] In our case, it means that the algorithm selects the model based on its probability of having a higher value (better performance) than all other models. On the other hand, UCB selects the item with the highest upper confidence bound.[35] We say that UCB implements *optimism in the face of uncertainty*, it gives an "uncertainty bonus," also called "exploration bonus," to the items it's uncertain about.

Contextual bandits as an exploration strategy

If bandits for model evaluation are to determine the payout (i.e., prediction accuracy) of each model, contextual bandits are to determine the payout of each action. In the case of recommendations/ads, an action is an item/ad to show to users, and the payout is how likely it is a user will click on it. Contextual bandits, like other bandits, are an amazing technique to improve the data efficiency of your model.

WARNING

Some people also call bandits for model evaluation "contextual bandits." This makes conversations confusing, so in this book, "contextual bandits" refer to exploration strategies to determine the payout of predictions.

Imagine that you're building a recommender system with 1,000 items to recommend, which makes it a 1,000-arm bandit problem. Each time, you can only recommend the top 10 most relevant items to a user. In bandit

terms, you'll have to choose the best 10 arms. The shown items get user feedback, inferred via whether the user clicks on them. But you won't get feedback on the other 990 items. This is known as the *partial feedback* problem, also known as *bandit feedback*. You can also think of contextual bandits as a classification problem with bandit feedback.

Let's say that each time a user clicks on an item, this item gets 1 value point. When an item has 0 value points, it could either be because the item has never been shown to a user, or because it's been shown but not clicked on. You want to show users the items with the highest value to them, but if you keep showing users only the items with the most value points, you'll keep on recommending the same popular items, and the never-before-shown items will keep having 0 value points.

Contextual bandits are algorithms that help you balance between showing users the items they will like and showing the items that you want feedback on.[36] It's the same exploration–exploitation trade-off that many readers might have encountered in reinforcement learning. Contextual bandits are also called "one-shot" reinforcement learning problems.[37] In reinforcement learning, you might need to take a series of actions before seeing the rewards. In contextual bandits, you can get bandit feedback right away after an action—e.g., after recommending an ad, you get feedback on whether a user has clicked on that recommendation.

Contextual bandits are well researched and have been shown to improve models' performance significantly (see reports by Twitter and Google). However, contextual bandits are even harder to implement than model bandits, since the exploration strategy depends on the ML model's architecture (e.g., whether it's a decision tree or a neural network), which makes it less generalizable across use cases. Readers interested in combining contextual bandits with deep learning should check out a great paper written by a team at Twitter: "Deep Bayesian Bandits: Exploring in Online Personalized Recommendations" (Guo et al. 2020).

Before we wrap up this section, there's one point I want to emphasize. We've gone through multiple types of tests for ML models. However, it's important to note that a good evaluation pipeline is not only about what tests to run, but also about who should run those tests. In ML, the evaluation process is often owned by data scientists—the same people who developed the model are responsible for evaluating it. Data scientists tend

to evaluate their new model ad hoc using the sets of tests that they like. First, this process is imbued with biases—data scientists have contexts about their models that most users don't, which means they probably won't use this model in a way most of their users will. Second, the ad hoc nature of the process means that the results might be variable. One data scientist might perform a set of tests and find that model A is better than model B, while another data scientist might report differently.

The lack of a way to ensure models' quality in production has led to many models failing after being deployed, which, in turn, fuels data scientists' anxiety when deploying models. To mitigate this issue, it's important for each team to outline clear pipelines on how models should be evaluated: e.g., the tests to run, the order in which they should run, the thresholds they must pass in order to be promoted to the next stage. Better, these pipelines should be automated and kicked off whenever there's a new model update. The results should be reported and reviewed, similar to the continuous integration/continuous deployment (CI/CD) process for traditional software engineering. It's crucial to understand that a good evaluation process involves not only what tests to run but also who should run those tests.

Summary

This chapter touches on a topic that I believe is among the most exciting yet underexplored topics: how to continually update your models in production to adapt them to changing data distributions. We discussed the four stages a company might go through in the process of modernizing their infrastructure for continual learning: from the manual, training from scratch stage to automated, stateless continual learning.

We then examined the question that haunts ML engineers at companies of all shapes and sizes, "How often *should* I update my models?" by urging them to consider the value of data freshness to their models and the trade-offs between model iteration and data iteration.

Similar to online prediction discussed in Chapter 7, continual learning requires a mature streaming infrastructure. The training part of continual learning can be done in batch, but the online evaluation part requires streaming. Many engineers worry that streaming is hard and costly. It

was true three years ago, but streaming technologies have matured significantly since then. More and more companies are providing solutions to make it easier for companies to move to streaming, including Spark Streaming, Snowflake Streaming, Materialize, Decodable, Vectorize, etc.

Continual learning is a problem specific to ML, but it largely requires an infrastructural solution. To be able to speed up the iteration cycle and detect failures in new model updates quickly, we need to set up our infrastructure in the right way. This requires the data science/ML team and the platform team to work together. We'll discuss infrastructure for ML in the next chapter.

1 Joan Serrà, Dídac Surís, Marius Miron, and Alexandros Karatzoglou, "Overcoming Catastrophic Forgetting with Hard Attention to the Task," *arXiv*, January 4, 2018, *https://oreil.ly/P95EZ*.

2 It's "stateful training" instead of "stateful retraining" because there's no *re*-training here. The model continues training from the last state.

3 Alex Egg, "Online Learning for Recommendations at Grubhub," *arXiv*, July 15, 2021, *https://oreil.ly/FBBUw*.

4 Mu Li, Li Zhou, Zichao Yang, Aaron Li, Fei Xia, David G. Andersen, and Alexander Smola, "Parameter Server for Distributed Machine Learning" (NIPS Workshop on Big Learning, Lake Tahoe, CA, 2013), *https://oreil.ly/xMmru*.

5 Jonathan Raiman, Susan Zhang, and Christy Dennison, "Neural Network Surgery with Sets," *arXiv*, December 13, 2019, *https://oreil.ly/SU0F1*.

6 This type of problem is also called "dynamic pricing."

7 Jon Russell, "Alibaba Acquires German Big Data Startup Data Artisans for $103M," *TechCrunch*, January 8, 2019, *https://oreil.ly/4tf5c*. An early reviewer mentioned that it's also possible that the main goal of this acquisition was to increase Alibaba's open source footprint, which is tiny compared to other tech giants.

8 The problem is also equally challenging if you want your model to figure out when to recommend a new movie that no one has watched and given feedback on yet.

9 Lucas Bernardi, Jaap Kamps, Julia Kiseleva, and Melanie J. I. Müller, "The Continuous Cold Start Problem in e-Commerce Recommender Systems," *arXiv*,

August 5, 2015, https://oreil.ly/GWUyD.

10 Jacopo Tagliabue, Ciro Greco, Jean-Francis Roy, Bingqing Yu, Patrick John Chia, Federico Bianchi, and Giovanni Cassani, "SIGIR 2021 E-Commerce Workshop Data Challenge," *arXiv*, April 19, 2021, https://oreil.ly/8QxmS.

11 Catherine Wang, "Why TikTok Made Its User So Obsessive? The AI Algorithm That Got You Hooked," *Towards Data Science*, June 7, 2020, https://oreil.ly/BDWf8.

12 See the section "Data Passing Through Real-Time Transport".

13 See the section "Batch Processing Versus Stream Processing".

14 Tyler Akidau, "Snowflake Streaming: Now Hiring! Help Design and Build the Future of Big Data and Stream Processing," Snowflake blog, October 26, 2020, https://oreil.ly/Knh2Y.

15 Arjun Narayan, "Materialize Raises a $60M Series C, Bringing Total Funding to Over $100M," *Materialize*, September 30, 2021, https://oreil.ly/dqxRb.

16 Khristopher J. Brooks, "Disparity in Home Lending Costs Minorities Millions, Researchers Find," *CBS News*, November 15, 2019, https://oreil.ly/SpZ1N; Lee Brown, "Tesla Driver Killed in Crash Posted Videos Driving Without His Hands on the Wheel," *New York Post*, May 16, 2021, https://oreil.ly/uku9S; "A Tesla Driver Is Charged in a Crash Involving Autopilot That Killed 2 People," *NPR*, January 18, 2022, https://oreil.ly/WWaRA.

17 James Vincent, "Twitter Taught Microsoft's Friendly AI Chatbot to Be a Racist Asshole in Less Than a Day," *The Verge*, May 24, 2016, https://oreil.ly/NJEVF.

18 Their fraud detection system consists of multiple ML models.

19 In the section "Bandits", we'll learn about how bandits can be used as a more data-efficient alternative to A/B testing.

20 Some people call this setting "learning with partial information," but learning with partial information refers to another setting, as outlined in the paper "Subspace Learning with Partial Information" by Gonen et al. (2016).

21 Pedro Domingos and Geoff Hulten, "Mining High-Speed Data Streams," in *Proceedings of the Sixth International Conference on Knowledge Discovery and Data Mining* (Boston: ACM Press, 2000), 71–80; Albert Bifet and Ricard Gavaldà, "Adaptive Parameter-free Learning from Evolving Data Streams," 2009, https://oreil.ly/XIMpl.

22 Zohar Karnin, Kevin Lang, and Edo Liberty, "Optimal Quantile Approximation in Streams," *arXiv*, March 17, 2016, https://oreil.ly/bUu4H.

23 We'll cover ML platforms in the section "ML Platform".

24 You might need to train your embedding model more frequently if you have a lot of new items each day.

25 Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Tanxin Shi, et al., "Practical Lessons from Predicting Clicks on Ads at Facebook," in *ADKDD '14: Proceedings of the Eighth International Workshop on Data Mining for Online Advertising* (August 2014): 1–9, https://oreil.ly/oS16J.

26 Qian Yu, "Machine Learning with Flink in Weibo," QCon 2019, video, 17:57, https://oreil.ly/Yia6v.

27 Ron Kohavi and Stefan Thomke, "The Surprising Power of Online Experiments," *Harvard Business Review*, September–October 2017, https://oreil.ly/OHfj0.

28 Danilo Sato, "CanaryRelease," June 25, 2014, MartinFowler.com, https://oreil.ly/YtKJE.

29 Thorsten Joachims, "Optimizing Search Engines using Clickthrough Data," KDD 2002, https://oreil.ly/XnH5G.

30 Joshua Parks, Juliette Aurisset, and Michael Ramm, "Innovating Faster on Personalization Algorithms at Netflix Using Interleaving," *Netflix Technology Blog*, November 29, 2017, https://oreil.ly/lnvDY.

31 Olivier Chapelle, Thorsten Joachims, Filip Radlinski, and Yisong Yue, "Large-Scale Validation and Analysis of Interleaved Search Evaluation," *ACM Transactions on Information Systems* 30, no. 1 (February 2012): 6, https://oreil.ly/lccvK.

32 Parks et al., "Innovating Faster on Personalization Algorithms."

33 Greg Rafferty, "A/B Testing—Is There a Better Way? An Exploration of Multi-Armed Bandits," *Towards Data Science*, January 22, 2020, https://oreil.ly/MsaAK.

34 William R. Thompson, "On the Likelihood that One Unknown Probability Exceeds Another in View of the Evidence of Two Samples," *Biometrika* 25, no. 3/4 (December 1933): 285–94, https://oreil.ly/TH1HC.

35 Peter Auer, "Using Confidence Bounds for Exploitation–Exploration Trade-offs," *Journal of Machine Learning Research* 3 (November 2002): 397–422,

https://oreil.ly/vp9mI.

36 Lihong Li, Wei Chu, John Langford, and Robert E. Schapire, "A Contextual-Bandit Approach to Personalized News Article Recommendation," *arXiv*, February 28, 2010, *https://oreil.ly/uaWHm*.

37 According to Wikipedia, *multi-armed bandit* is a classic reinforcement learning problem that exemplifies the exploration–exploitation trade-off dilemma (s.v., "Multi-armed bandit," *https://oreil.ly/ySjwo*). The name comes from imagining a gambler at a row of slot machines (sometimes known as "one-armed bandits") who has to decide which machines to play, how many times to play each machine and in which order to play them, and whether to continue with the current machine or try a different machine.