Chapter 11. Monitor and Update Models

Once a model is deployed, its performance should be monitored just like any other software system. As they did in "Test Your ML Code", regular software best practices apply. And just like in "Test Your ML Code", there are additional things to consider when dealing with ML models.

In this chapter, we will describe key aspects to keep in mind when monitoring ML models. More specifically, we will answer three questions:

1. Why should we monitor our models?
2. How do we monitor our models?
3. What actions should our monitoring drive?

Let's start by covering how monitoring models can help decide when to deploy a new version or surface problems in production.

Monitoring Saves Lives

The goal of monitoring is to track the health of a system. For models, this means monitoring their performance and the quality of their predictions.

If a change in user habits suddenly causes a model to produce subpar results, a good monitoring system will allow you to notice and react as soon as possible. Let's cover some key issues that monitoring can help us catch.

Monitoring to Inform Refresh Rate

We saw in "Freshness and Distribution Shift" that most models need to be regularly updated to maintain a given level of performance. Monitoring can be used to detect when a model is not fresh anymore and needs to be retrained.

For example, let's say that we use the implicit feedback that we get from our

users (whether they click on recommendations, for example) to estimate the accuracy of a model. If we continuously monitor the accuracy of the model, we can train a new model as soon as accuracy drops below a defined threshold. Figure 11-1 shows a timeline of this process, with retraining events happening when accuracy dips below a threshold.

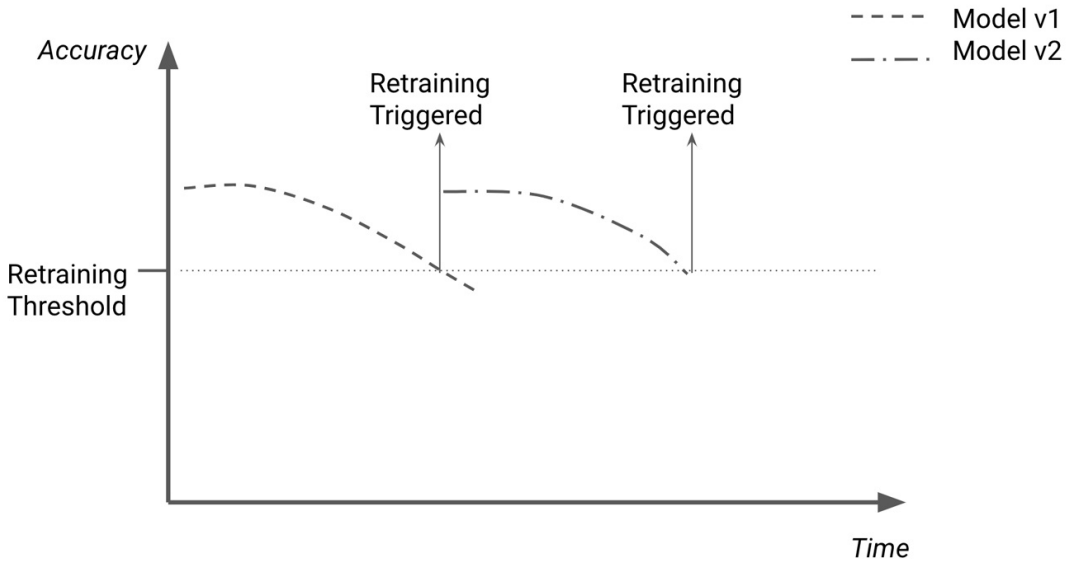

Figure 11-1. Monitoring to trigger redeploy

Before redeploying an updated model, we would need to verify that the new model is better. We will cover how to do this later in this section, "CI/CD for ML". First, let's tackle other aspects to monitor, such as potential abuse.

Monitor to Detect Abuse

In some cases such as when building abuse prevention or fraud detection systems, a fraction of users are actively working to defeat models. In these cases, monitoring becomes a key way to detect attacks and estimate their success rate.

A monitoring system can use anomaly detection to detect attacks. When tracking every attempt to log in to a bank's online portal, for example, a monitoring system could raise an alert if the number of login attempts suddenly increased tenfold, which could be a sign of an attack.

This monitoring could raise an alert based on a threshold value being crossed, as you can see in Figure 11-2, or include more nuanced metrics such as the rate of

increase of login attempts. Depending on the complexity of attacks, it may be valuable to build a model to detect such anomalies with more nuance than a simple threshold could.

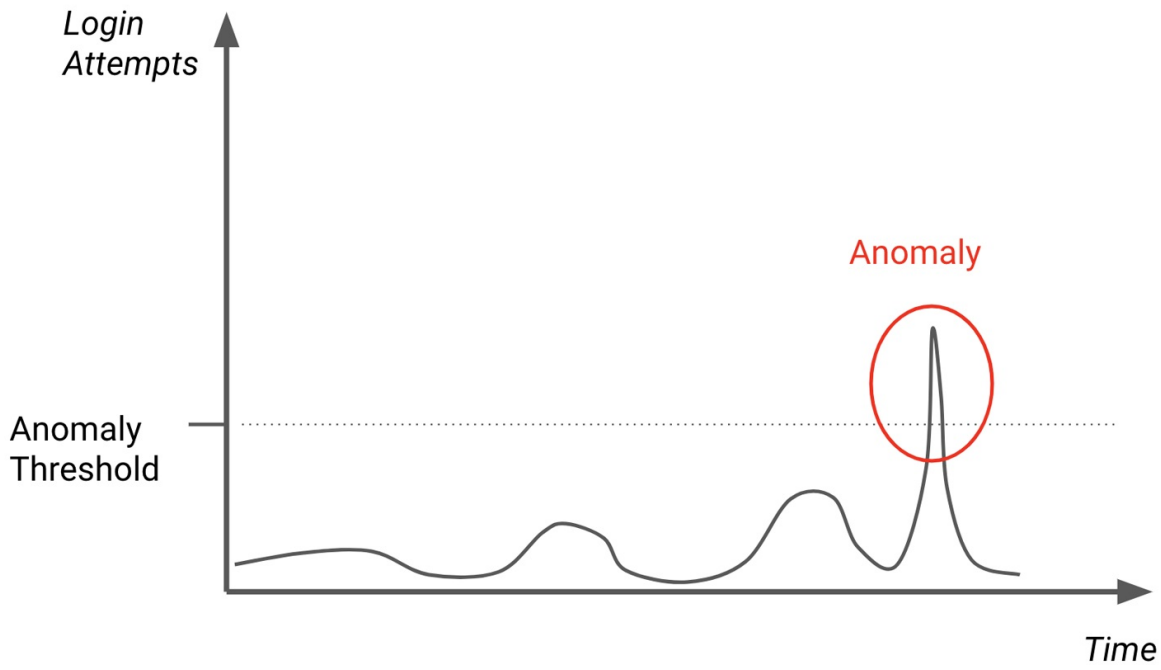

Figure 11-2. An obvious anomaly on a monitoring dashboard. You could build an additional ML model to automatically detect it.

In addition to monitoring freshness and detecting anomalies, which other metrics should we monitor?

Choose What to Monitor

Software applications commonly monitor metrics such as the average time it takes to process a request, the proportion of requests that fail to be processed, and the amount of available resources. These are useful to track in any production service and allow for proactive remediation before too many users are impacted.

Next, we will cover more metrics to monitor to detect when a model's performance is starting to decline.

Performance Metrics

A model can become stale if the distribution of data starts to change. You can see this illustrated in Figure 11-3.

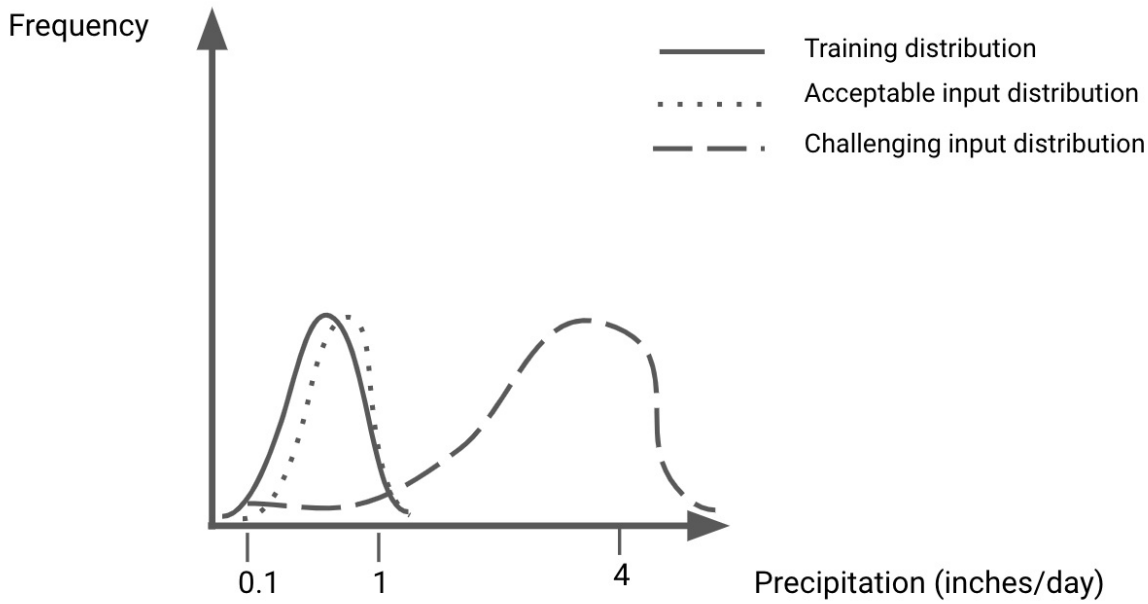

Figure 11-3. Example of drift in a feature's distribution

When it comes to distribution shifts, both the input and the output distribution of data can change. Consider the example of a model that tries to guess which movie a user will watch next. Given the same user history as an input, the model's prediction should change based on new entries in a catalog of available movies.

- *Tracking changes in the input distribution* (also called feature drift) is easier than tracking the output distribution, since it can be challenging to access the ideal value of outputs to satisfy users.
- *Monitoring the input distribution* can be as simple as monitoring summary statistics such as the mean and variance of key features and raising an alert if these statistics drift away from the values in the training data by more than a given threshold.
- *Monitoring distribution shifts* can be more challenging. A first approach is to monitor the distribution of model outputs. Similarly to inputs, a significant change in the distribution of outputs may be a sign that model performance has degraded. The distribution of the results users

would have liked to see, however, can be harder to estimate.

One of the reasons for why estimating ground truth can be hard is that a model's actions can often prevent us from observing it. To see why that may be the case, consider the illustration of a credit card fraud detection model in Figure 11-4. The distribution of the data that the model will receive is on the left side. As the model makes predictions on the data, application code acts on these predictions by blocking any transaction predicted as fraudulent.

Once a transaction is blocked, we are thus unable to observe what would have happened if we had let it through. This means that we are not be able to know whether the blocked transaction was actually fraudulent or not. We are only able to observe and label the transactions we let through. Because of having acted on a model's predictions, we are only able to observe a skewed distribution of nonblocked transactions, represented on the right side.

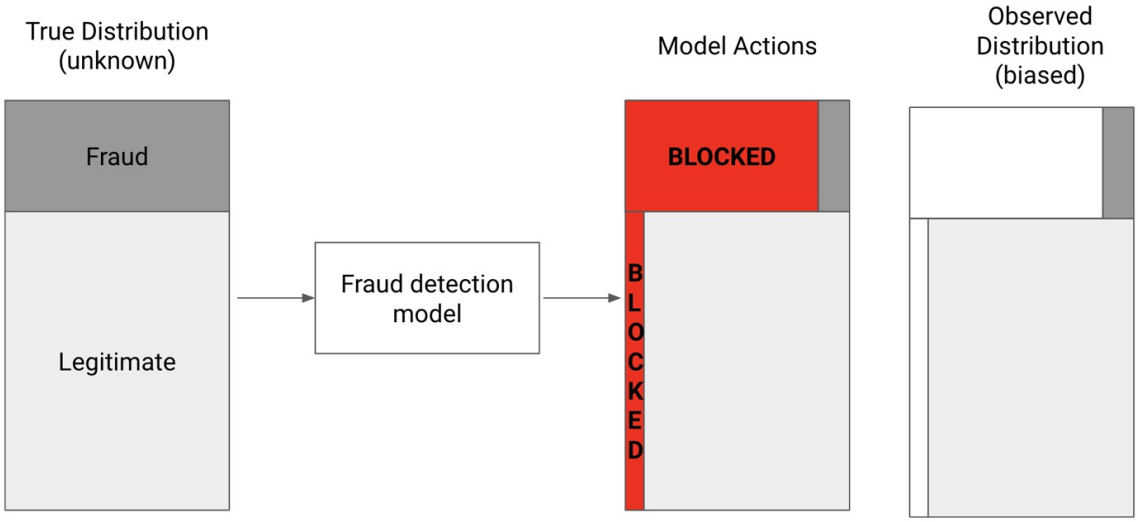

Figure 11-4. Taking action based on a model's predictions can bias the observed distribution of data

Only having access to a skewed sample of the true distribution makes it impossible to correctly evaluate a model's performance. This is the focus of *counterfactual evaluation*, which aims to evaluate what would have happened if we hadn't actioned a model. To perform such evaluation in practice, you can withhold running a model on a small subset of examples (see the article by Lihong Li et al., "Counterfactual Estimation and Optimization of Click Metrics for Search Engines"). Not acting on a random subset of examples will then allow us to observe an unbiased distribution of fraudulent transactions. By comparing

model predictions to true outcomes for the random data, we can begin to estimate a model's precision and recall.

This approach provides a way to evaluate models but comes at the cost of letting a proportion of fraudulent transactions go through. In many cases, this trade-off can be favorable since it allows for model benchmarking and comparisons. In some cases, such as in medical domains where outputting a random prediction is not acceptable, this approach should not be used.

In "CI/CD for ML", we'll cover other strategies to compare models and decide which ones to deploy, but first, let's cover the other key types of metrics to track.

Business Metrics

As we've seen throughout this book, the most important metrics are the ones related to product and business goals. They are the yardstick against which we can judge our model's performance. If all of the other metrics are in the green and the rest of the production system is performing well but users don't click on search results or use recommendations, then a product is failing by definition.

For this reason, product metrics should be closely monitored. For systems such as search or recommendation systems, this monitoring could track the CTR, the ratio at which people that have seen a model's recommendation clicked on it.

Some applications may benefit from modifications to the product to more easily track product success, similarly to the feedback examples we saw in "Ask for Feedback". We discussed adding a share button, but we could track feedback at a more granular level. If we can have users click on recommendations in order to implement them, we can track whether each recommendation was used and use this data to train a new version of the model. Figure 11-5 shows an illustrated comparison between the aggregate approach on the left side and the granular one on the right.

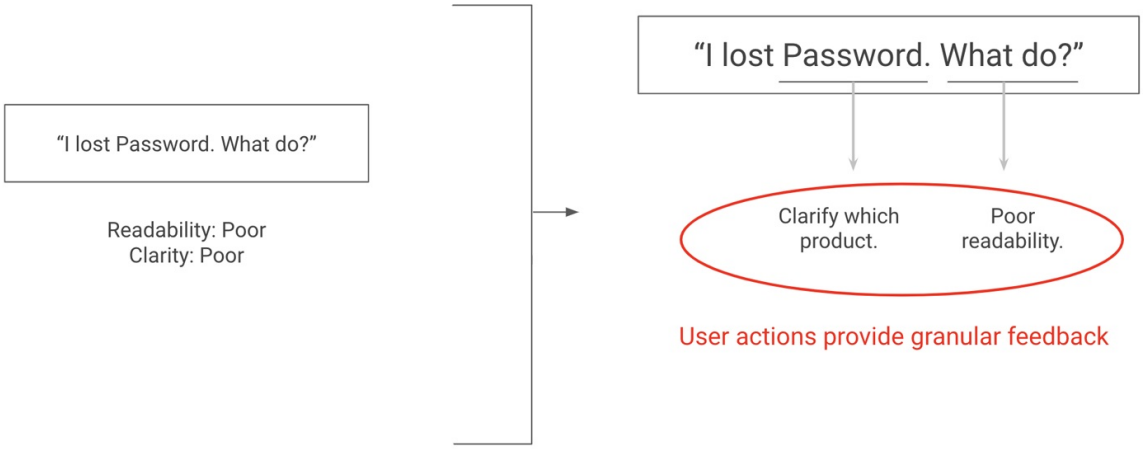

Figure 11-5. Proposing word-level suggestions gives us more opportunities to collect user feedback

Since I do not expect the ML Editor prototype to be used frequently enough for the method described to provide a large enough dataset, we will abstain from building it here. If we were building a product we were intending to maintain, collecting such data would allow us to get precise feedback about which recommendations the user found the most useful.

Now that we have discussed reasons and methods to monitor models, let's cover ways to address any issues detected by monitoring.

CI/CD for ML

CI/CD stands for continuous integration (CI) and continuous delivery (CD). Roughly speaking, CI is the process of letting multiple developers regularly merge their code back into a central codebase, while CD focuses on improving the speed at which new versions of software can be released. Adopting CI/CD practices allows individuals and organizations to quickly iterate and improve on an application, whether they are releasing new features or fixing existing bugs.

CI/CD for ML thus aims to make it easier to deploy new models or update existing ones. Releasing updates quickly is easy; the challenge comes in guaranteeing their quality.

When it comes to ML, we saw that having a test suite is not enough to guarantee that a new model improves upon a previous one. Training a new model and

testing that it performs well on held-out data is a good first step, but ultimately, as we saw earlier, there is no substitute for live performance to judge the quality of a model.

Before deploying a model to users, teams will often deploy them in what Schelter et al., in their paper, "On Challenges in Machine Learning Model Management", refer to as *shadow mode*. This refers to the process of deploying a new model in parallel to an existing one. When running inference, both models' predictions are computed and stored, but the application only uses the prediction of the existing model.

By logging the new predicted value and comparing it both to the old version and to ground truth when it is available, engineers can estimate a new model's performance in a production environment without changing the user experience. This approach also allows to test the infrastructure required to run inference for a new model that may be more complex than the existing one. The only thing shadow mode doesn't provide is the ability to observe the user's response to the new model. The only way to do that is to actually deploy it.

Once a model has been tested, it is a candidate for deployment. Deploying a new model comes with the risk of exposing users to a degradation of performance. Mitigating that risk requires some care and is the focus of the field of experimentation.

Figure 11-6 shows a visualization of each of the three approaches we covered here, from the safest one of evaluating a mode on a test set to the most informative and dangerous one of deploying a model live in production. Notice that while shadow mode does require engineering effort in order to be able to run two models for each inference step, it allows for the evaluation of a model to be almost as safe as using a test set and provides almost as much information as running it in production.

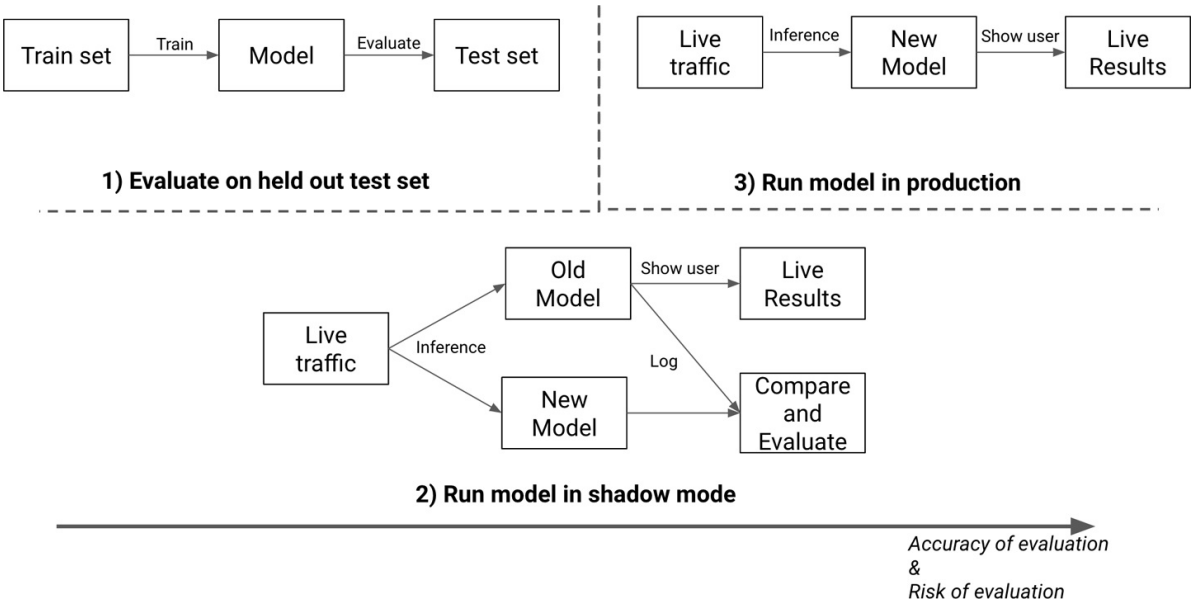

Figure 11-6. Ways to evaluate a model, from safest and least accurate to riskiest and most accurate

Since deploying models in production can be a risky process, engineering teams have developed methods to deploy changes incrementally, starting by showing new results to only a subset of users. We will cover this next.

A/B Testing and Experimentation

In ML, the goal of experimentation is to maximize chances of using the best model, while minimizing the cost of trying out suboptimal models. There are many experimentation approaches, the most popular being A/B testing.

The principle behind A/B testing is simple: expose a sample of users to a new model, and the rest to another. This is commonly done by having a larger "control" group being served the current model and a smaller "treatment" group being served a new version that we want to test. Once we have run an experiment for a sufficient amount of time, we compare the results for both groups and choose the better model.

In Figure 11-7, you can see how to randomly sample users from a total population to allocate them to a test set. At inference time, the model used for a given user is determined by their allocated group.

The idea behind A/B testing is simple, but experimental design concerns such as choosing the control and the treatment group, deciding which amount of time is sufficient, and evaluating which model performs better are all challenging issues.

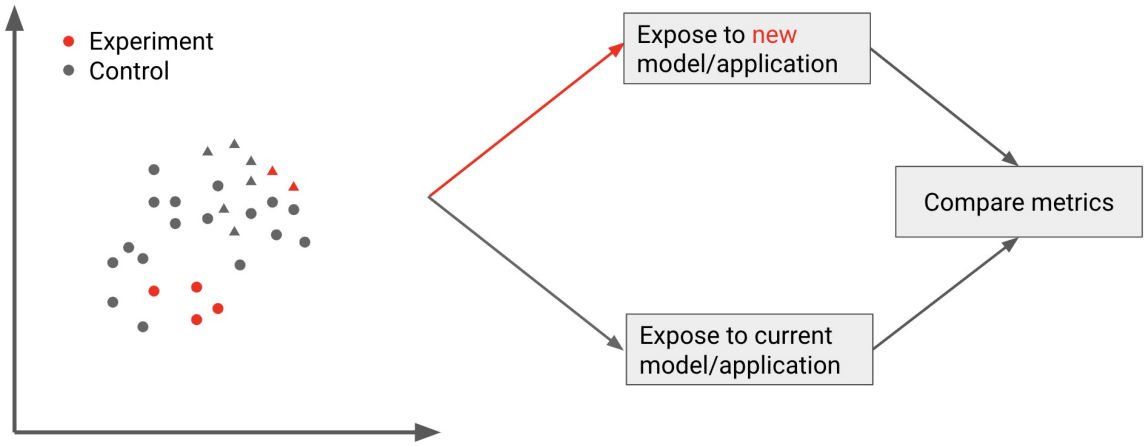

Figure 11-7. An example of an A/B test

In addition, A/B testing requires the building of additional infrastructure to support the ability to serve different models to different users. Let's cover each of these challenges in more detail.

Choosing groups and duration

Deciding which users should be served which model comes with a few requirements. Users in both groups should be as similar as possible so that any observed difference in outcome can be attributed to our model and not to a difference in cohorts. If all users in group A are power users and group B contains only occasional users, the results of an experiments will not be conclusive.

In addition, the treatment group B should be large enough to draw a statistically meaningful conclusion, but as small as possible to limit exposure to a potentially worse model. The duration of the test presents a similar trade-off: too short and we risk not having enough information, too long and we risk losing users.

These two constraints are challenging enough, but consider for a minute the case of large companies with hundreds of data scientists who run dozens of A/B tests in parallel. Multiple A/B tests may be testing the same aspect of the pipeline at the same time, making it harder to determine the effect of an individual test accurately. When companies get to this scale, this leads them to building experimentation platforms to handle the complexity. See Airbnb's ERF, as described in Jonathan Parks's article, "Scaling Airbnb's Experimentation

Platform"; Uber's XP as described in A. Deb et al.'s post, "Under the Hood of Uber's Experimentation Platform"; or the GitHub repo for Intuit's open source Wasabi.

Estimating the better variant

Most A/B tests choose a metric they would like to compare between groups such as CTR. Unfortunately, estimating which version performed better is more complex than selecting the group with the highest CTR.

Since we expect there to be natural fluctuations in any metric results, we first need to determine whether results are statistically significant. Since we are estimating a difference between two populations, the most common tests that are used are two-sample hypothesis tests.

For an experiment to be conclusive, it needs to be run on a sufficient amount of data. The exact quantity depends on the value of the variable we are measuring and the scale of the change we are aiming to detect. For a practical example, see Evan Miller's sample size calculator.

It is also important to decide on the size of each group and the length of the experiment before running it. If you instead continuously test for significance while an A/B test is ongoing and declare the test successful as soon as you see a significant result, you will be committing a repeated significance testing error. This kind of error consists of severely overestimating the significance of an experiment by opportunistically looking for significance (once again, Evan Miller has a great explanation here).

> **NOTE**
>
> While most experiments focus on comparing the value of a single metric, it is important to also consider other impacts. If the average CTR increases but the number of users who stop using the product doubles, we probably should not consider a model to be better.
>
> Similarly, results of A/B tests should take into account results for different segments of users. If the average CTR increases but the CTR for a given segment plummets, it may be better to not deploy the new model.

Implementing an experiment requires the ability to assign users to a group, track

each user's assignment, and present different results based on it. This necessitates building additional infrastructure, which we cover next.

Building the infrastructure

Experiments also come with infrastructure requirements. The simplest way to run an A/B test is to store each user's associated group with the rest of user-related information, such as in a database.

The application can then rely on branching logic that decides which model to run depending on the given field's value. This simple approach works well for systems where users are logged in but becomes significantly harder if a model is accessible to logged-out users.

This is because experiments usually assume that each group is independent and exposed to only one variant. When serving models to logged out users, it becomes harder to guarantee that a given user was always served the same variant across each session. If most users are exposed to multiple variants, this could invalidate the results of an experiment.

Other information to identify users such as browser cookies and IP addresses can be used to identify users. Once again, however, such approaches require building new infrastructure, which may be hard for small, resource-constrained teams.

Other Approaches

A/B testing is a popular experimentation method, but other approaches exist that try to address some of A/B testing's limitations.

Multiarmed bandits are a more flexible approach that can test variants continually and on more than two alternatives. They dynamically update which model to serve based on how well each option is performing. I've illustrated how multiarmed bandits work in Figure 11-8. Bandits continuously keep a tally of how each alternative is performing based on the success of each request they route. Most requests are simply routed to the current best alternative, as shown on the left. A small subset of requests gets routed to a random alternative, as you can see on the right. This allows bandits to update their estimate of which model is the best and detect if a model that is currently not being served is starting to perform better.

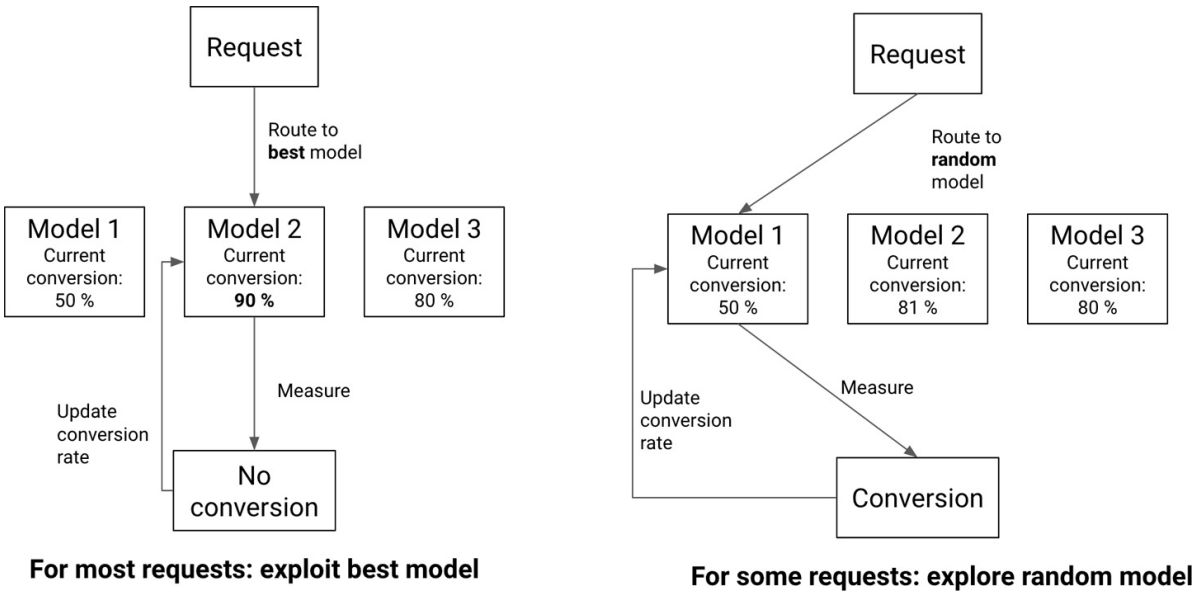

Figure 11-8. Multiarmed bandits in practice

Contextual multiarmed bandits take this process even further, by learning which model is a better option for each particular user. For more information, I recommend this overview by the Stitch Fix team.

> **NOTE**
>
> While this section covered the usage of experimentation to validate models, companies increasingly use experimentation methods to validate any significant change they make to their applications. This allows them to continuously evaluate which functionality users are finding useful and how new features are performing.

Because experimentation is such a hard and error-prone process, multiple startups have started offering "optimization services" allowing customers to integrate their applications with a hosted experimentation platform to decide which variants perform best. For organizations without a dedicated experimentation team, such solutions may be the easiest way to test new model versions.

Conclusion

Overall, deploying and monitoring models is still a relatively new practice. It is a

crucial way to verify that models are producing value but often requires significant efforts both in terms of infrastructure work and careful product design.

As the field has started to mature, experimentation platforms such as Optimizely have emerged to make some of this work easier. Ideally, this should empower builders of ML applications to make them continuously better, for everyone.

Looking back at all the systems described in this book, only a small subset aims to train models. The majority of work involved with building ML products consists of data and engineering work. Despite this fact, most of the data scientists I have mentored found it easier to find resources covering modeling techniques and thus felt unprepared to tackle work outside of this realm. This book is my attempt at helping bridge that gap.

Building an ML application requires a broad set of skills in diverse domains such as statistics, software engineering, and product management. Each part of the process is complex enough to warrant multiple books being written about it. The goal of this book is to provide you with a broad set of tools to help you build such applications and let you decide which topics to explore more deeply by following the recommendations outlined in "Additional Resources", for example.

With that in mind, I hope this book gave you tools to more confidently tackle the majority of the work involved with building ML-powered products. We've covered every part of the ML product life cycle, starting by translating a product goal to an ML approach, then finding and curating data and iterating on models, before validating their performance and deploying them.

Whether you've read this book cover to cover or dove into specific sections that were most relevant to your work, you should now have the required knowledge to start building your own ML-powered applications. If this book has helped you to build something or if you have any questions or comments about its content, please reach out to me by emailing mlpoweredapplications@gmail.com. I look forward to hearing from you and seeing your ML work.