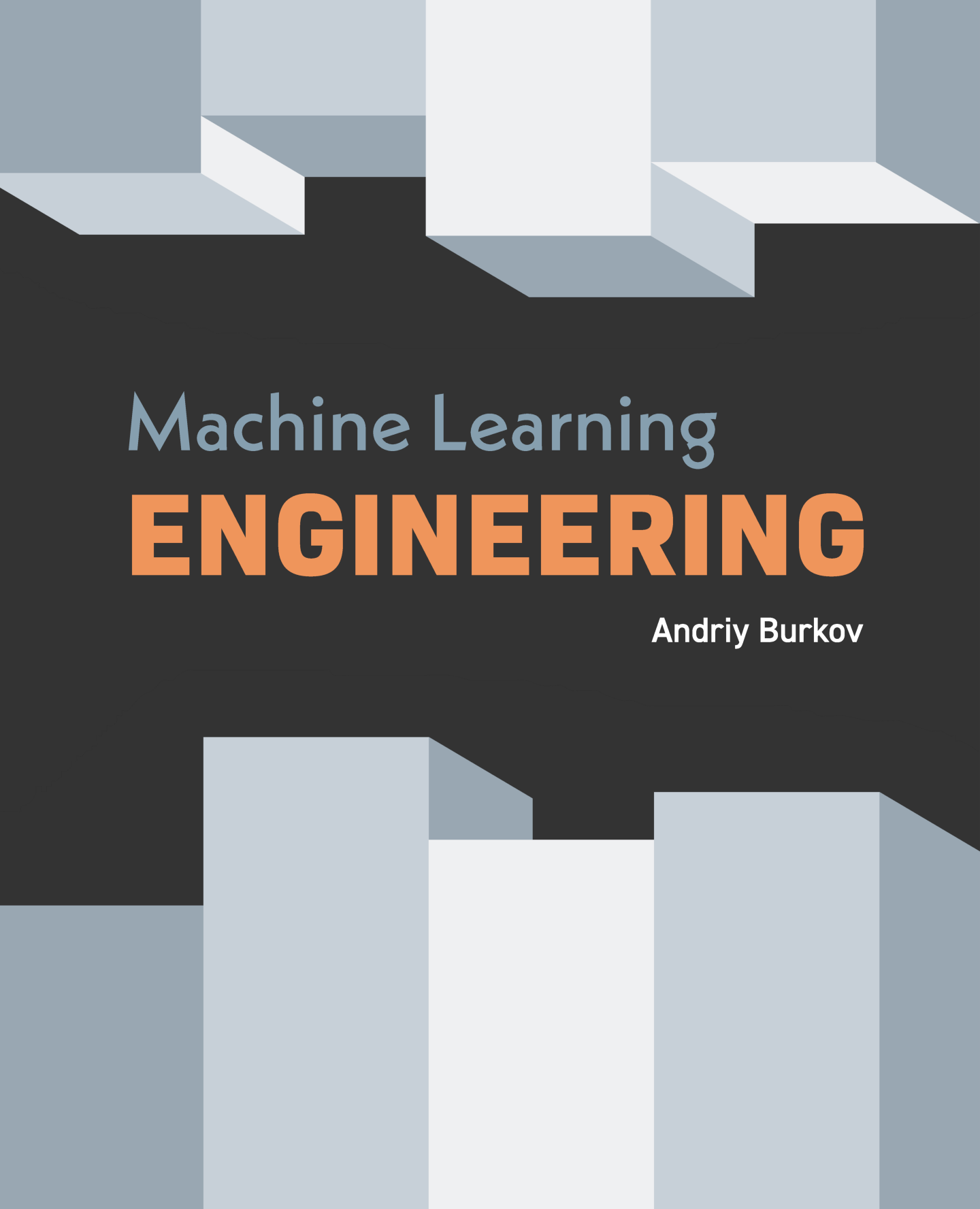
Machine Learning
ENGINEERING
Andriy Burkov

“In theory, there is no difference between theory and practice. But in practice, there is.”

— Benjamin Brewster

“The perfect project plan is possible if one first documents a list of all the unknowns.”

— Bill Langley

“When you’re fundraising, it’s AI. When you’re hiring, it’s ML. When you’re implementing, it’s linear regression. When you’re debugging, it’s printf().”

— Baron Schwartz

The book is distributed on the “read first, buy later” principle.

9 Model Serving, Monitoring, and Maintenance

In this chapter, we consider the best practices of serving, monitoring, and maintaining models in production. These are the last three stages in the machine learning project life cycle:

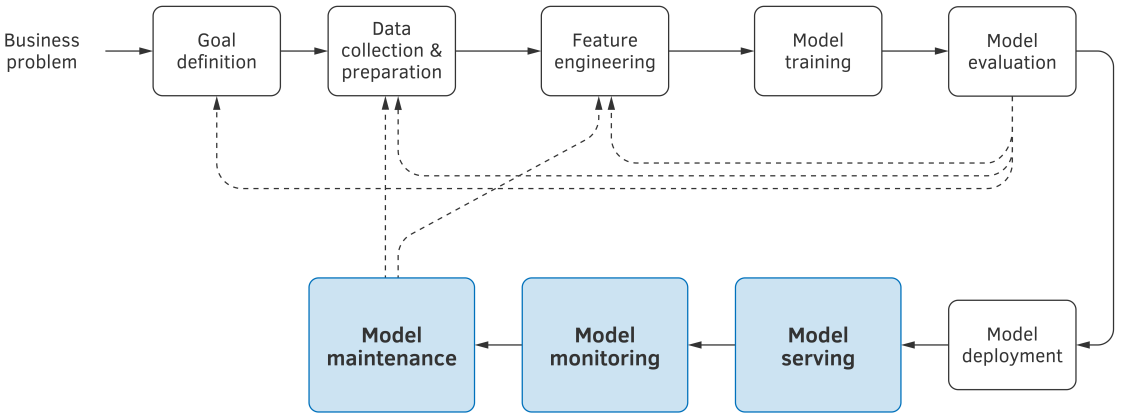

Figure 1: Machine learning project life cycle.

In particular, we characterize the properties of a machine learning runtime, the environment in which the input data is applied to the model, and the modes of model serving, such as batch and on demand. Furthermore, we consider three major challenges of serving a model in real world: errors, change, and human nature. We describe what should be monitored in the production environment, and when and how update the model.

9.1 Properties of the Model Serving Runtime

The model serving runtime is the environment in which the model is applied to the input data. The runtime properties are dictated by the model **deployment pattern**. However, an effective runtime will have several additional properties that we discuss here.

9.1.1 Security and Correctness

The runtime is responsible for authenticating the user's identity, and authorizing their requests.

Things to check are:

- whether a specific user has authorized access to the models they want to run,

- whether the names and the values of parameters passed correspond to the model's specification, and
- whether those parameters and their values are currently available to the user.

9.1.2 Ease of Deployment

The runtime must allow the model to be updated with minimal effort and, ideally, without affecting the entire application. If the model was deployed as a web service on a physical server, then a model update must be as simple as replacing one model file with another, and restarting the web service.

If the model was deployed as a virtual machine instance or container, then the instances or containers running the old version of the model should be replaceable by gradually stopping the running instances and starting new instances from a new image. The same principle applies to the orchestrated containers.

Typically, a model streaming-based application is updated by streaming the new version of the model. To enable this, the streaming application must be stateful. Once a new version and the related components (such as feature extractor and scoring code) are streamed into the application, the state of the application changes, and now contains the new version of these assets. Modern **stream-processing engines** support stateful applications. The described architecture is schematically shown in Figure 2.

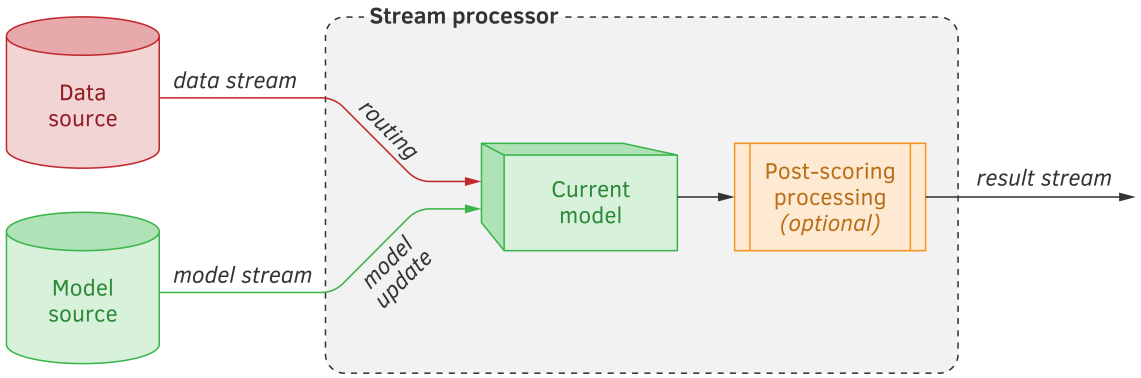

Figure 2: Model streaming high-level architecture.

9.1.3 Guarantees of Model Validity

An effective runtime will automatically make sure that the model it executes is valid. Furthermore, it makes sure the model, the feature extractor, and other components are in sync. It must be validated on each startup of the web service or the streaming application, and periodically during the runtime. As discussed in Section **??** of Chapter 8, each model should be deployed accompanied by the following four assets: an **end-to-end set**, a **confidence test set**, a **performance metric**, and its **range of acceptable values**.

The model should not be served in production (and must be immediately stopped if it is running) in either of the two conditions:

- at least one of the end-to-end test examples was not scored correctly, or
- the value of the metric, calculated on the confidence test set examples, is not within the acceptable range.

9.1.4 Ease of Recovery

An effective runtime allows easy recovery from errors by rolling back to previous versions.

The recovery from an unsuccessful deployment should be produced in the same way, and with the same ease, as the deployment of an updated model. The only difference is that, instead of the new model, the previous working version will be deployed.

9.1.5 Avoidance of Training/Serving Skew

It is strongly recommended to avoid using two different codebases, one for training the model, and one for scoring in production. When it concerns **feature extraction**, even a tiny difference between two versions of feature extractor code may lead to suboptimal or incorrect model performance.

The engineering team may reimplement the feature extractor code for production for many reasons. The most common is that the data analyst's code is inefficient or incompatible with the production ecosystem.

Thus, the runtime should allow easy access to the feature extraction code for various needs, including model retraining, ad-hoc model calls, and production. One way to implement it is by wrapping the feature extraction object into a separate web service.

If you cannot avoid using two different codebases to generate features for training and production, then the runtime should allow for the logging of feature values generated in the production environment. Those values should then be used as training values.

9.1.6 Avoidance of Hidden Feedback Loops

In Section **??** of Chapter 4, we saw one example of a **hidden feedback loop**. Model m_B used the output of model m_A as a feature, without knowing that model m_A also used the output of model m_B as its feature.

Another kind of hidden feedback loop only involves one model. Let's say we have a model that classifies incoming email messages as spam or not spam. Let the user interface allow the user to mark messages as spam or not spam. Obviously, we want to use those marked messages to improve our model. However, by so doing, we risk creating a hidden feedback loop, and here is why.

In our application, the user will only mark a message as spam when they see it. However, users only see the messages that our model classified as not spam. Also, it is unlikely that the user will regularly go to the spam folder and mark some messages as not spam. So, the action of the user is significantly affected by our model, which makes the data we get from the user skewed: we influence the phenomenon from which we learn.

To avoid the skew, mark a small percentage of examples as "held-out," and show all of them to the user without pre-applying the model. Then use only these held-out examples as additional training examples, including those to which the user didn't react.

In a more general scenario, one model can indirectly affect the data used to train another model. Let one model decide the order of books to display, while the other decides which reviews to display near each book. If the first model puts a review of a certain book at the bottom of the list, the absence of a user's response to the second model's review may be caused by its low position on the page, and not by the quality of the review.

9.2 Modes of Model Serving

Machine learning models are served in either batch or on-demand mode. On-demand, a model can be served to either a human client or a machine.

9.2.1 Serving in Batch Mode

A model is usually served in batch mode when it is applied to large quantities of input data. One example could be when the model is used to exhaustively process the data of all users of a product or service. Or, when it systematically applies to all incoming events, such as tweets, or comments to online publications. Batch mode is more resource-efficient compared to an on-demand mode, and is employed when some latency can be tolerated.

When served in batch mode, the model usually accepts between a hundred and a thousand feature vectors at once. Experiment to find the optimal batch size for speed. Typical sizes are powers of two: 32, 64, 128, etc.

The outputs for the batch are usually saved to the database, as opposed to sending them to specific consumers. You would use the batch mode to:

- generate the list of weekly recommendations of new songs to all users of a music streaming service,
- classify the flow of incoming comments to online news articles and blog posts as spam or not spam,
- extract named entities from documents indexed by a search engine, and so on.

9.2.2 Serving on Demand to a Human

The six steps of serving the model **on demand** to a human are as follows:

1) validate the request,
2) gather the context,
3) transform the context into model input,
4) apply the model to the input, and get the output,
5) make sure that the output makes sense,
6) present the output to the user.

Before running a model in production for a request coming from a user, it might be necessary to verify whether that user has the correct permissions for this model.

The **context** represents the user's situation when they send a request to the machine learning system, and in which the user will receive the system's response.

The user can send the request to the machine learning system explicitly or implicitly. An example of an explicit request is when a music-streaming service's user requests recommendations for similar songs to a given song. On the other hand, an implicit request is sent by a direct messenger application for suggested replies to the most recent message received by the user.

A good context may be collected in real or near-real time. It will contain the information needed by the feature extractor to generate all the feature values the model expects. It also contains enough information for debugging, is compact enough to be saved in a log, and contains information that will be used to improve the model over time.

Let's see examples of a good context for several problems.

Device malfunctioning
: When detecting device malfunction, a good context contains vibration and noise levels, the task executed by the device, the user ID, the firmware version, the time passed since manufacturing and the last maintenance, and the number of uses since manufacturing and the last maintenance.

Emergency room hospitalization
: To decide whether the new patient should be admitted to an intensive care unit, a good context would include age, blood pressure, temperature, heart rate, pulse oximetry

level, complete blood count, chemistry profile, arterial blood gas test, blood alcohol level, medical history, and pregnancy.

Credit risk assessment
To make an approval/rejection decision for a credit card application, a good context would include age, education, employment status, country residency status, annual salary, family status, outstanding debts, availability of other credit cards, whether the person is a homeowner or tenant, whether the person has declared bankruptcy, and whether, and how many times, the person missed past credit payments. Even if certain information is not needed for feature extraction, it is still pertinent for logging and debugging: client's ID, date, and time of the day.

Advertisement display
To decide whether a specific advertisement should be displayed to a website user, a good context would include the webpage title, the user's position on the web page, the screen resolution, the text on the webpage and the text visible to the user, how the user reached the webpage, and the time spent on it. For logging and debugging purposes, the context might include the browser version, operating system version, connection information, and date and time.

A **feature extractor** transforms the context into the model input. Sometimes, the feature extractor is a part of the machine learning **pipeline**, as we discussed in Section **??** of Chapter 5. However, it's common to build the feature extractor as a separate object.

When the result of the scoring is to be served to a human client, it's rarely presented directly. Usually, the scoring code transforms the model's prediction into a form more easily interpreted, and that adds value to the client.

Before serving the model to a human, it's common to measure the prediction confidence score. If the confidence is low, you can decide to not present anything: users tend to complain less about the errors they don't see. Or, if the user expects an output, inform them about the low confidence. Then prompt, "Are you sure?"

Prompting is especially important when the system might initiate an action based on the prediction. If you are able to estimate the error's possible cost and if the prediction confidence is bounded by $(0, 1)$, then multiply (1 — confidence) by the cost to see the possible impact of making a wrong action. For example, let the cost of making an error is estimated as 1000 dollars and the model outputs the confidence score equal to 0.95, then the **expected error cost** value is $(1 - 0.95) \times 1000 = 50$ dollars. You might put a threshold on the expected cost value for different actions recommended by the model, and prompt the user if the expected cost is above the threshold.

In addition to measuring the model's confidence, calculate whether the value of the prediction makes sense. In Section 9.3, we will further detail what to check, and what the system's reaction should be, if the output doesn't make sense.

It is convenient to log the context in which the model was served, as well as the reaction of the user. This can help both to debug eventual problems, and improve the model by creating

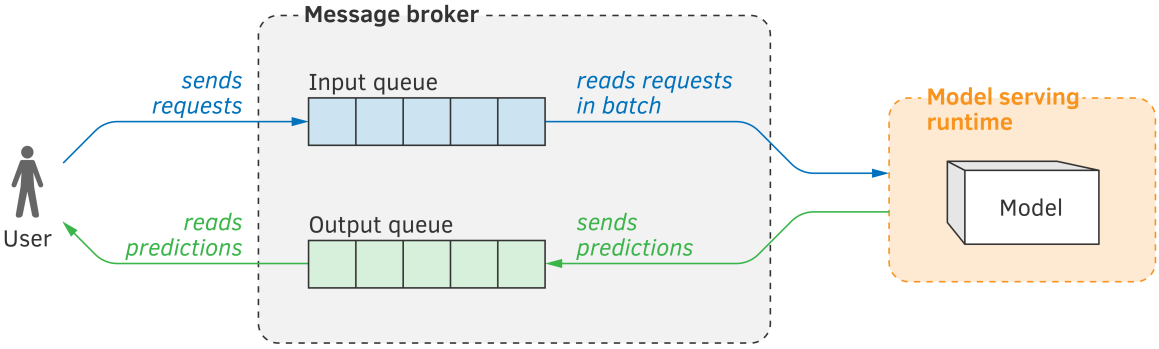

Figure 3: On-demand model serving with a message broker.

new training examples.

9.2.3 Serving on Demand to a Machine

While building a REST API is appropriate for many cases, we often serve a machine by streaming. Indeed, a machine's data requirements are usually standard and predetermined. A well-designed, fixed topology of a streaming application allows an efficient use of available resources.

Serving on demand, to either machine or human, can be tricky. The demand may vary, from very high during the day, to very low during the night. If you use virtual resources in the cloud, **autoscaling** can help with adding more resources when needed, and then freeing them when demand decreases. However, autoscaling is not nimble enough to cope with accidental spikes.

To deal with such situations, on-demand architectures include a **message broker**, such as **RabbitMQ** or **Apache Kafka**. A message broker allows one process to write messages in a queue, and another to read from that queue. On-demand requests are placed in the input queue. The model runtime process periodically connects to the broker. It reads a batch of input data elements from the input queue and generates predictions for each element in batch mode. It then writes the predictions to the output queue. Another process periodically connects to the broker, reads the predictions from the output queue, and pushes them to users who sent the requests (Figure 3). In addition to allowing us to cope with demand spikes, such an approach is more resource-efficient.

9.3 Model Serving in Real World

When real people interact with a software system in the real world, serving the model gets complicated. It's usually impossible to predict all user actions and reactions. The architecture of a software system intended for the real world must be ready for three phenomena: errors, change, and human nature.

9.3.1 Being Ready for Errors

Errors are inevitable in any software. In machine-learning-based software, errors are an integral part of the solution: no model is perfect. Because we cannot fix all errors, the only option is to embrace them.

Embracing errors means designing the software system in such a way that when an error happens, the system continues operating normally.

There are three "cannots" we must accept and embrace:

1. We cannot always explain why an error happened.
2. We cannot reliably predict when it will happen, and even a high confidence prediction can be false.
3. We cannot always know how to fix a specific error. If it's fixable, what kind and how much training data is needed?

Furthermore, when an error happens, we cannot always expect that the incorrect prediction will at least be close or similar to the correct prediction. An error can be arbitrarily "crazy." For example, a model for a self-driving car, at the speed of 120 km/h (~74 mph) with no obstacles, may predict that the best action is to stop and drive backward.

Tiny changes in the context may result in unexpected error patterns. For example, the model that recognizes dangerous situations on the factory floor may start making errors after the lightbulb near the camera is replaced. The previous lightbulb was incandescent, and the new one is fluorescent.

Even rare errors may impact users, if the number of users is large. Let the model have a 99% accuracy. If you have a million users, one percent of prediction errors will affect thousands.

It's rare that fixing one error in a model results in new errors. However, there's no guarantee.

How to design a system in the presence of inevitable errors?

9.3.2 Dealing With Errors

First of all, have a strategy that mitigates, at least, partially, a situation in which your system looks or acts "stupid." For example, if your system talks to the user, like a personal assistant or a chatbot, it's better to say, "I don't know," than to say something random. If the error

will be directly visible to the user, calculate the **expected cost of the error**, as discussed above, and do not display the prediction to the user if the cost is above a threshold.

Alternatively, train a second model m_B that predicts, for an input, that the first model m_A is likely to make an error on that input. The presence of a "safeguard" model m_B is especially relevant if model m_A is used in a mission critical system.

The error's visibility is an important factor in deciding whether and how to hide it. For example, consider a system that downloads web pages from the internet and extracts some entities from them. Let the user be interested in being alerted when a kind of entity is detected. The model can make two kinds of errors: 1) extract an entity even if the document doesn't contain it (false positive, FP), and 2) not extract an entity that is present in the document (false negative, FN). When the former error happens, the user receives an irrelevant alert and gets frustrated. If the latter, the user doesn't receive any alert, remains unaware of the error, and avoids frustration. In this situation, you might prefer to optimize the model for **precision**, by keeping **recall** reasonably high.

When you train a model, decide which kind of errors you would most like to avoid, and then optimize your hyperparameters, including the prediction threshold, accordingly.

When your confidence for the best prediction is low, consider presenting several options. This is why Google presents 10 search results at once. There are much higher chances for the most relevant link to be among those 10 search results, than for it to be in the first position.

Another way to avoid user's frustration with model errors is to dose the user's exposure to the model. Measure the number of errors your model makes, and estimate how many errors per minute (day, week, or month) a user is ready to tolerate. Then limit the interactions the user will have with the model to keep the number of perceived errors below that level.

For situations when an error happened and might have been perceived, add a possibility for the user to report the error. Once the report is received, log the context in which the model was used, as well as the prediction of the model. Explain to the user what actions will be taken to prevent a similar error from happening in the future.

It's appropriate to measure the user's engagement with the system, log all interactions, and then analyze suspicious interactions offline. This includes:

- whether the user interacts with the system less than before,
- whether the user ignored certain recommendations, and
- whether the user spent adequate time in various settings.

To reduce an error's negative impact even further, if the system allows it, give the user an option to undo an action recommended by the system. Extend this, if possible, to any automated action executed by the system on the user's behalf.

Software applications that act on their user's behalf must be especially limited in their possible actions. Recall that machine-learning models' errors can be arbitrarily "crazy" like in the example of a self-driving car that can suddenly decide to drive backward. Caution must be exercised in other critical scenarios involving health, safety, or money, such as bidding in

auctions or prescribing medication. If the model predicts to buy or sell more stocks than the moving average plus one standard deviations, it's a good idea to send an alert and put an otherwise "automatic" action on hold. The same logic should apply if the model predicts to serve an unreasonably high dose of a drug to a patient, or change the speed of the car to a value substantially above or below usual.

If your system can automatically reject the model prediction, it's best to implement some fallback strategy, in addition to informing the user about a failure (Figure 4). A less sophisticated model or a handcrafted heuristic may be used as a fallback. Of course, the output of the fallback strategy should also be validated, and also rejected if it seems unreasonable. In this case, an error message should be sent to the user.

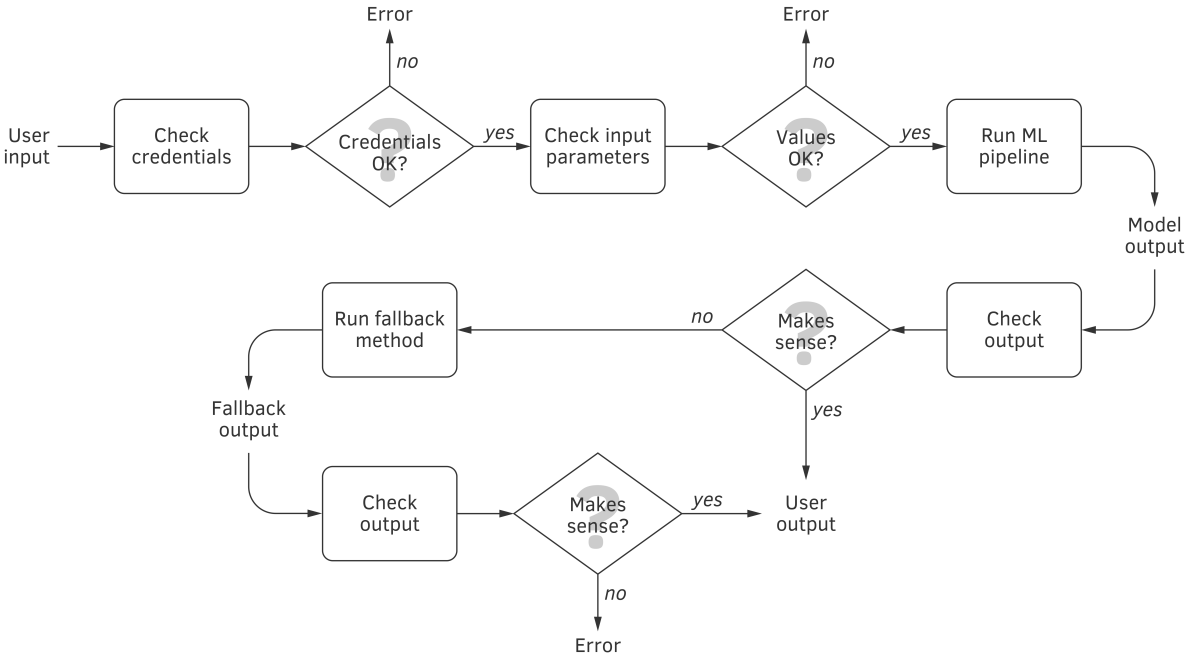

Figure 4: Real-world model serving flowchart.

9.3.3 Being Ready for, and Dealing With, Change

The performance of a system based on machine learning usually changes over time. In some applications, it can change in near-real-time.

There are two types of model change:

1. Its quality could become better or worse.

2. The predictions for some inputs can become different.

A typical reason for the model performance degradation over time is **concept drift** that we already considered in Section **??** of Chapter 3. The notion of what is a correct prediction may change because of the users' preferences and interests. This would require retraining the model, using more recently labeled data.

Some change can be perceived by the user as positive. Sometimes, the change can be negatively perceived, even if the system's performance improved, from the engineering point of view. You might have added training examples, retrained the model and observed a better performance metric value. However, by adding new data, you involuntarily induced a data imbalance. Some classes are now underrepresented. Users interested in those classes' predictions see decreased performance, and complain or even abandon your system.

Users become accustomed to certain behaviors. They might know what query to submit to the search engine to get an often-used document or a web application. That query was not necessarily the most optimal for the purpose, but it worked. Suppose you improved the relevancy of your search-result ranking algorithm. Now that query doesn't return that specific document or application, or it puts it on the second page of the search results. The user can no longer find the resource they once found easily, and get frustrated.

If you expect that the user might negatively perceive the change, give them time to adapt. Educate the user about the changes and what to expect from the new model. Or, it can be done by gradually introducing the changes. You might mix the predictions of the old model and the new model, and slowly decrease the proportion for the old model. Alternatively, you can run both the new and the old model in parallel, and let the user switch to the old model for some time before sunsetting it.

9.3.4 Being Ready for, and Dealing With, Human Nature

Human nature is what makes effective system engineering such a hard endeavor. Humans are unpredictable, often irrational, inconsistent, and have unclear expectations. A solid software system must anticipate that.

Avoid Confusion

The system must be designed in such a way that the user doesn't feel confused interacting with it. A model's output must be served in an intuitive way, without assuming that the user knows anything about machine learning and AI. In fact, many users will assume that they work with typical software and will be surprised to see errors.

Manage Expectations

On the other hand, some users will have too high expectations. The main reason for that is advertisement. To attract attention, a product or a system based on machine learning is often displayed in advertisements as being "intelligent." For example, personal assistants such as Apple Siri, Google Home, and Amazon Alexa are often shown in advertisements as having

human intelligence. Indeed, any machine-learning-based system might look very intelligent when inputs are carefully selected. Users can look at such advertisements and extrapolate what they see to situations in which the system isn't designed to operate effectively.

Another common reason users expect something spectacular (even without being promised) is that they worked with a similar (in their understanding) system that looked "very intelligent" to them. Such users would expect the same level of "intelligence" from your system.

Gain Trust

Some users, especially experienced ones, will mistrust any system if they know it contains some "intelligence." The main reason for that mistrust is past experience. Most so-called intelligent systems fail to deliver, and, because of that, some users expect failure when they first encounter your system.

As a consequence, your system must gain each user's confidence, and this must be done early.

A user experienced with "intelligent" systems will most likely make several simple tests of your system's abilities. If your system fails, the user will not trust it. For example, if your system is a search engine, then a user would query their name or a document they authored to test your system. Or, if your system provides intelligence on organizations to corporate customers, a user will check how much your system knows about their organization, and whether the intelligence makes sense. A driver of a self-driving car will most likely test commands like "start the engine," "follow that car," "keep the current speed," or "park on that street." Depending on the nature of the service, you should anticipate such simple tests and make sure that your system passes them.

Manage User Fatigue

User fatigue can be another reason why you see decreasing interest in your system. Make sure that the system doesn't excessively interrupt user experience with recommendations or requests for approval. Avoid showing everything you have to show in one shot. Whenever possible, let the user explicitly express their interest.

Furthermore, not all actions that the system can handle automatically have to be handled this way. For example, if the system automates user's interactions with other people, it might send private or restricted data as an email reply, or post it to an open forum. Before sharing on a user's behalf, it makes sense to evaluate the information's sensitivity. Use a model trained to detect such potentially sensitive texts and images. On the other extreme, a system can be too conservative and automatically filter out relevant information or ask the user to confirm too many decisions which might result in user fatigue.

Beware of the Creep Factor

When users interact with a learning system, there's a phenomenon known as **creep factor**. It means that the user perceives the model's predictive capacity as too high. The user feels uncomfortable, especially when a prediction concerns their very private details. Make sure that the system doesn't feel like "Big Brother" and doesn't take too much responsibility.

9.4 Model Monitoring

A deployed model must be constantly monitored. Monitoring helps make sure that,

- the model is served correctly, and
- the performance of the model remains within acceptable limits.

9.4.1 What Can Go Wrong?

Monitoring should be designed to provide early warnings about issues with the model in production. More specifically, this includes:

- new training data used to update the model made it perform worse;
- the live data in production changed, but the model didn't;
- the feature extraction code was significantly updated, but the model didn't adapt;
- a resource needed to generate a feature changed or became unavailable;
- the model is being abused or under an adversarial attack.

Additional training data is not always good. A **labeler** may have incorrectly interpreted the labeling instructions. Or, one labeler's decisions might be in contradiction with another labeler. Data automatically gathered to improve the model may be biased. Reasons for that could be, for example, a **hidden feedback loop** considered in Section 9.1.6 or a **systematic value distortion** discussed in Section **??** of Chapter 3.

Sometimes, the properties of the data in production gradually change, but the model doesn't adapt. It remains based on older data, which is no longer representative. One reason for this is **concept drift** that we discussed in Section 9.3.

A software engineer could fix a bug in the feature extraction code, and update the feature extractor in production. But if the engineer fails to also update the production model, the performance may change in an unpredictable manner.

Even if the feature extraction and the model are in sync, a disappearance or a change of some resource (database connection, database table, or external API) may affect some of the features generated by that feature extractor.

Some models, especially those deployed in e-commerce and media platforms, often become targets of adversarial attacks. Bad actors, such as unfair competitors, fraudsters, criminals, and foreign governments, may actively seek out weaknesses in a model and adjust their attacks accordingly. If your machine learning system learns from the user's actions, then some may act to change the model behavior in their favor.

Furthermore, attackers may want to examine the trained model in order to obtain information about the model's training data. That training data might contain confidential information about people and organizations.

Another form of abuse, which may be the hardest to prevent, is model **dual use**. As any software, a machine learning model can be used for good (as you intended) or for bad (often without your consent). For example, you might create and publicly release a model that makes one's voice sound like a cartoon character. Fraudsters may adapt your result to fake the voice of a bank client, and execute a phone transaction on their behalf. Alternatively, you might create a model that recognizes pedestrians on the street. An automatic weapon manufacturer could use your model to detect people on the battlefield.

9.4.2 What and How to Monitor

Monitoring must allow us to make sure that the model generates reasonable performance metrics when applied to the **confidence test set**. This set should be regularly updated with new data to avoid possible **distribution shift**. Additionally, the model must be regularly tested on the examples from the **end-to-end set**.

While it's obvious that accuracy, precision, and recall are good candidates for monitoring, one metric is especially useful for measuring the change over time: **prediction bias**.

In a static world where nothing changes, the distribution of predicted classes would roughly equal the distribution of observed classes. This is especially true when the model is **well-calibrated**. If you observe otherwise, the model is exhibiting prediction bias. The latter might mean that the distribution of the training data labels and the production's current class distribution are now different. You must investigate the reasons for this change and make the necessary adjustments.

Monitoring allows us to stay alert of abandoned or repurposed data sources. Some database columns might stop being populated. The definition or format of the data in some columns might change, while the unadapted models still assume the previous definitions and formats. To avoid that, the distribution of the values of every feature extracted from a database table must be monitored for a significant shift. A shift of the distribution of both feature values and predictions can be detected by applying statistical tests such as the **Chi-square independence test** and **Kolmogorov–Smirnov test**. If a significant distribution shift is detected, an alert must be sent to the stakeholders.

The **numerical stability** of the model should also be monitored. An alert should be triggered if NaNs (not-a-numbers) or an infinity is observed.

It's important to monitor computational performance of a machine learning system. Both dramatic and slow-leak regression should be detected, and warnings must be sent.

Monitor and send alerts when the usage fluctuations look suspicious. In particular:

- monitor the number of model servings during an hour, and compare it to the corresponding value calculated one day earlier. Send a warning alert to the stakeholders if the number has changed by 30% or more. This threshold must be tuned for your use case to avoid generating excessive warnings;

- monitor the daily number of model servings and compare it to the corresponding value calculated one week earlier. Send a warning alert to the stakeholders if the number has changed by 15% or more. Tune the value for your use case.

Monitoring these numbers helps detect undesirable change:

- minimal and maximal prediction values,
- median, mean, and standard deviation prediction values over a given timeframe,
- latency when calling the model API, and
- memory consumption and CPU usage when performing predictions.

Additionally, to prevent distribution shift, the monitoring automation must:

1) accumulate inputs by randomly putting some aside during a certain time period,
2) send those inputs for labeling,
3) run the model, and calculate the value of the performance metric,
4) alert the stakeholders if there is significant performance degradation.

Recommender systems need additional monitoring. These models offer recommendations to website or application users. It can be useful to monitor click-though rate (CTR), that is, ratio of users who clicked on a recommendation to the number of total users who received recommendations from that model. If CTR is decreasing, the model must be updated.

It's important to note that there is a tricky tradeoff between being too conservative versus frequently alerting stakeholders about small changes in the metrics. If you alert too often, people might become tired of receiving alerts and eventually will start ignoring them. In non-mission-critical cases, it can be appropriate to allow the stakeholders to define their own thresholds that trigger alerts.

Log monitoring events so the entire process is traceable. For visual model performance analysis, the monitoring tool's user interface should provide trend charts showing how the model degradation evolves over time.

One of the monitoring tool's properties should be the ability to compute and visualize metrics on slices of data. A slice is a subset of the data that includes only such examples in which a specific attribute has a certain value. For example, one slice could contain only the examples where the state attribute is Florida; another slice might contain only the data for women, and so on. The degradation of the model might only be observed in some slices, yet remain insignificant in others.

Besides real-time monitoring, it's important to also log data that:

- might help find a problem's source,
- is impossible to analyze in real-time, or
- is helpful for improving existing models or training new ones.

9.4.3 What to Log

It is important to log enough information to reproduce any erratic system behavior during a future analysis. If the model is served to a front-end user, such as a website visitor or a mobile application user, it's worth saving the user's **context** at the moment of the model serving. As discussed in Section 9.2, the context might include: the content of the webpage (or the state of the application), the user's position on the web page, time of the day, where the user came from, and what they clicked before the model prediction was served.

Additionally, it is useful to include the model input, that is, the features extracted from the context, and the time it took to generate those features.

The log could also include:

- the model's output, and time it took to generate it,
- the new context of the user, once they observed the model's output,
- the user's reaction to the output.

The user's reaction is the immediate action that followed the observation of the model output: what was clicked, and how much time after the output was served.

In large systems with thousands of users, where the model is served to each user hundreds of times a day, it can be prohibitive to log every event. It would be more practical to do **stratified sampling**. You first decide which groups of events you want to log, and then you log only a certain percentage of events in each group. The groups can be groups of users or groups of contexts. Users can be grouped by age, gender, or seniority with the service (new clients vs. long-time clients). The groups of contexts could be early-morning, business-day, and late-night interactions.

When you store users' activity data in logs, the users should know what, when, and how it is stored, and for how long. If possible, data should be anonymized or aggregated without loss of utility. Access to sensitive data must be restricted only to those assigned to solve a specific problem during a specific time period. Avoid letting any analyst access sensitive data to solve unrelated business problems. It could lead to legal problems.

Make sure users may opt-out from logging and analysis of their activity data. Different data retention policies will apply to different countries. Each country imposes its own restrictions on what can and cannot be stored about their citizens, or used for analysis.

9.4.4 Monitor for Abuse

Some people or organizations may use your model for their own business. Such users might send millions of daily requests, while a typical user would only send a dozen. Alternatively, some users might want to reverse-engineer the training data, or learn how to make the model produce a desired output.

Ways to prevent such abuse include,

- making users pay per request,
- creating progressively longer pauses before responding to requests, or even
- blocking some users.

To reach their own business goals, some attackers might try to manipulate your model. An attacker might submit data that changes the model in a way that only benefits the attacker. As a result, the overall quality of the model might degrade.

Ways to prevent such abuse include,

- not trusting the data from a user unless similar data comes from multiple users,
- assigning a reputation score to each user, and not trusting the data obtained from users with low reputations, and
- classifying user behavior as either normal or abnormal, and not accepting the data coming from users demonstrating abnormal behavior.

The attackers will try to bypass your defence by adapting their behavior. To effectively defend your system, update your models regularly. Add both new data and new features that detect fraudulent transactions.

9.5 Model Maintenance

Most production models must be regularly updated. The rate depends on several factors:

- how often it makes errors and how critical they are,
- how "fresh" the model should be, so as to be useful,
- how fast new training data becomes available,
- how much time it takes to retrain a model,
- how costly it is to deploy the model, and
- how much a model update contributes to the product and the achievement of user goals.

In this section, we talk about model maintenance: when and how to update the model after it's deployed in production.

9.5.1 When to Update

When a model is deployed in production for the first time, it's often far from perfect. Inevitably, the model makes prediction errors. Some of them could be critical, so the model needs an update. Over time, a model could become more solid, and require fewer updates. However, some models should be constantly updated, so to speak, always be "fresh."

Model freshness depends on the business needs and the needs of the user. The recommender model on an e-commerce website must be updated after each purchase. If the user utilizes a model to get recommended content on a news website, the model might need to be updated weekly. On the other hand, a voice recognition/synthesis or a machine translation model could be updated less frequently.

The speed of availability of new training data also affects the rate of model updates. Even if new data comes in fast, such as the stream of comments on a popular website, it may take time and require significant investment to get labeled data. Sometimes, labeling is automated but delayed, as in **churn prediction**, where the user's decision to stay with or leave the service happens far in the future.

Some models take significant time to build, especially if the **hyperparameter search** is needed. It's not uncommon to wait for days or even weeks to get a new version of the model. Use parallelizable machine learning algorithms and graphical processing units (GPU) to speed up the training. Modern libraries, such as **thundersvm** and **cuML**, allow the analyst to run shallow learning algorithms on GPUs, with a significant gain in training time. If you cannot afford to wait for days or weeks to get an updated model, using a less complex (and, therefore, less accurate) model might be your only choice.

You might decide to update the model less often if an update is costly. For example, in healthcare, getting labeled examples is complicated and expensive, due to regulations, privacy concerns, and expensive medical experts.

Not all models are worth deploying. Sometimes the potential performance gain is not worth the user's possible frustration. However, if the user disturbance is manageable, and the deployment is not costly, even a small improvement may result in a significant business outcome in the long run.

9.5.2 How to Update

As discussed, your software ideally allows the new model version to be deployed without stopping the entire system. In virtual or containerized infrastructure, this can be done by replacing the image of a virtual machine (VM) or a container in the repository, gradually closing VMs/containers, and letting the autoscaler instantiate a VM/container from an updated image.

An architecture of machine learning deployment and maintenance automation is schematically shown in Figure 5. Here, we have three repositories: data, code, and model; all three repositories are versioned. We also have two runtimes: model training and production. The model runs in the production runtime, which is load-balanced and auto-scaled. When an update of the model is needed, the model training runtime pulls the training data, as well as the model training code, from the data and code repositories, respectively. It then trains the new model and saves it in the model repository.

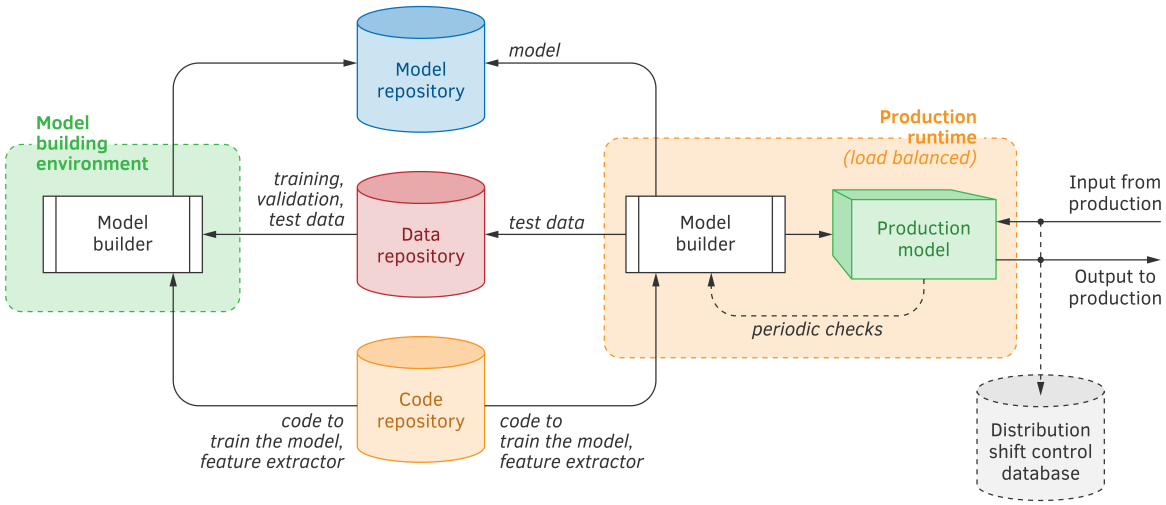

Figure 5: A machine learning deployment and maintenance automation architecture.

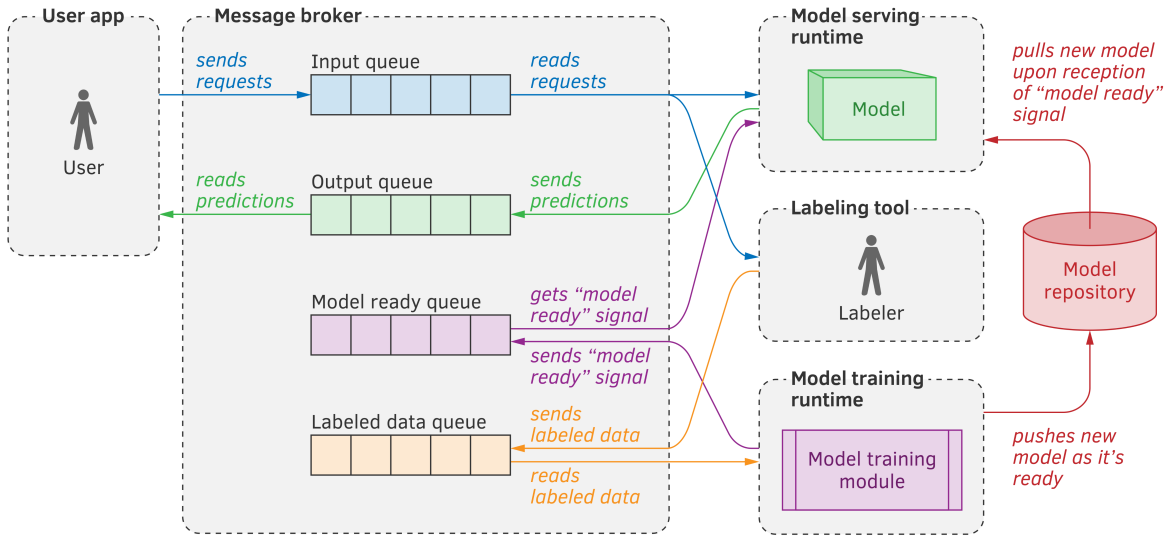

Figure 6: On-demand model serving and update with a message broker.

Once a new version of the model is placed in the repository, the production runtime pulls,

- the new model, from the model repository;
- the test data, from the data repository; and,
- the code that applies the model to the test data, from the code repository.

If the new model passes the test, the old model is withdrawn from production. It is replaced with the new one with the appropriate deployment strategy, as discussed in Section **??**. **A/B testing** or a **multi-armed bandit** algorithm can help make the replacement decision.

The **distribution shift** control database accumulates the inputs received by the model, as well as their scoring results. Once a sufficient number of examples is accumulated, that data is sent for validation to a human[1] with the goal of detecting distribution shift.

In the **model streaming** scenario, the model update happens when the stream processor's state is updated (see Section 9.1.2 and Figure 2).

Model update in **on-demand model serving** with a **message broker** architecture is similar to that of model streaming (see Section 9.2.3 and Figure 3).

Figure 6 illustrates a message-broker-based architecture that allows not just serving the model and updating it, but also contains a human **labeler** in the loop. The labeler receives unlabeled examples, samples some of them, assigns labels to sampled examples, and sends the annotated examples back to the message broker. The model training module reads the labeled examples from a queue. When their quantity is sufficient to significantly update the model, it trains a new model, saves it in the model repository, and sends the "model ready" message to the broker. A model-serving process pulls the new model version from the repository, and discards the current model.

Let's outline a few additional considerations for successful model maintenance.

Many companies use a continuous integration workflow in which the models are trained automatically as soon as new training data becomes available. It is recommended to retrain the model from scratch, by using the entire training data, instead of fine-tuning an existing model on the new examples only.

For each training example, it's recommended to store the labeler's identity. Furthermore, attach the model version used to generate a specific value in the production database, to that value. Should a problem with the version model be discovered, knowing which database values it generated will allow reprocessing of those specific values only.

If a model is frequently re-trained, it is convenient to store pipeline's hyperparameters in a configuration system. Google recommends[2] the following for a good configuration system:

1. It should be easy to specify a configuration as a change from a previous configuration.
2. It should be hard to make manual errors, omissions, or oversights.

[1] Or to an automated tool, more accurate than the model, that cannot be deployed in production (e.g., too fragile, costly, or slow).

[2] "Hidden Technical Debt in Machine Learning Systems" by Sculley et al. (2015).

3. It should be easy to see, visually, the difference in configuration between two models.
4. It should be easy to automatically assert and verify basic facts about a configuration: number of features used, data dependencies, etc.
5. It should be possible to detect unused or redundant settings.
6. Configurations should undergo a full code review and be checked into a repository.

Make sure that the runtime environment has enough hard drive space and RAM for the updated model. Do not expect that the old version of the model and the new one will only differ in performance. Be ready for the situation where the new model is much larger than the previous one. Similarly, do not expect that the new model will run as fast as the previous one. Inefficiency in the feature extraction code, an additional stage in the pipeline, or a different choice of the algorithm may significantly affect the prediction speed.

Models will inevitably make prediction errors. However, to the business or client, some errors are more costly than others. Once a new model version is deployed, validate it doesn't make significantly more costly errors than the previous model.

Check that the errors are distributed uniformly across the user categories. It's undesirable if the new model negatively affects more users from a minority or specific location.

If any of the above validations fail, it is not recommended to deploy the new model. Roll it back if the failure is detected after deployment and initiate an investigation. As discussed in Section 9.1, rolling back to the previous model must be as easy as deploying the new model.

Beware of **model cascading**. As discussed in Section **??** of Chapter 6, if the one model's outputs become inputs for another model, changing one model will affect the performance the other. If your system is using model cascading, be sure to update all models in the cascade.

9.6 Summary

An effective runtime has the following properties. It is secure and correct, ensures ease of deployment and recovery, and provides guarantees of model validity. Furthermore, it avoids training/serving skew and hidden feedback loops.

Machine learning models are served in either batch or on-demand mode. In on-demand mode, a model can be served to either a human client or a machine. A model is usually served in batch mode when it will be applied to big data and some latency is tolerable.

When served on-demand to a human, a model is usually wrapped into a REST API. A machine's data requirements are usually standard and pre-determined, so we often serve it by streaming.

The architecture of a software system intended for the real world must be ready for three phenomena: errors, change, and human nature.

The model deployed in production must be constantly monitored. The goals of monitoring are to make sure that the model is served correctly, and that the performance of the model

remains within acceptable limits.

A variety of things might go wrong with the model in production, in particular:

- additional training data made the model perform worse;
- the properties of the production data changed, but the model didn't;
- the feature extraction code was significantly updated, but the model didn't adapt;
- a resource needed to generate a feature changed or became unavailable;
- the model is abused or is under an adversarial attack.

An automation must calculate values of the performance metrics critical for the business, and send alerts to the appropriate stakeholders if the values of those metrics change significantly or fall below a threshold. In addition, the monitoring must reveal the distribution shift, numerical instability, and a decreasing computational performance.

It is important to log enough information to reproduce any erratic system behavior during an analysis in the future. If the model is served to a front-end user, it's important to log the user's context at the moment of the model serving. Additionally, it is useful to include the model input, that is, the features extracted from the context, and the time it took to generate those features. The log could also include the outputs obtained from the model, and time it took to generate it, the new context of the user once they observed the output of the model, and the reaction of the user to the output.

Some users can utilize your model as a basis for their own business. They might reverse-engineer the training data, or learn how to "trick" your model. To prevent abuse:

- don't trust the data from a user unless similar data comes from multiple users,
- assign a reputation score to each user and don't trust the data obtained from users with low reputations,
- classify user behavior as normal or abnormal,
- make users pay per request,
- make progressively longer pauses, and
- block some users.

Most machine learning models must be regularly or occasionally updated. The rate of updates depends on several factors:

- how often it makes errors and how critical they are,
- how "fresh" the model should be to be useful,
- how fast new training data becomes available,
- how much time it takes to retrain a model,
- how costly it is to train and deploy the model, and
- how much a model update contributes to the achievement of user goals.

After a model update, a good practice is to run the model against the examples in the end-to-end and confidence test sets. It's important to make sure that the outputs are either the same as before, or that the changes are as expected. It's also important to validate that the new model doesn't make significantly more costly errors than the previous model.

Check also that the errors are distributed uniformly across the user categories. It's undesirable if the new model negatively affects users from a minority or specific location.