Summarize

Chapter 6. Model Development and Offline Evaluation

In Chapter 4, we discussed how to create training data for your model, and in Chapter 5, we discussed how to engineer features from that training data. With the initial set of features, we'll move to the ML algorithm part of ML systems. For me, this has always been the most fun step, as it allows me to play around with different algorithms and techniques, even the latest ones. This is also the first step where I can see all the hard work I've put into data and feature engineering transformed into a system whose outputs (predictions) I can use to evaluate the success of my effort.

To build an ML model, we first need to select the ML model to build. There are so many ML algorithms out there, with more actively being developed. This chapter starts with six tips for selecting the best algorithms for your task.

The section that follows discusses different aspects of model development, such as debugging, experiment tracking and versioning, distributed training, and AutoML.

Model development is an iterative process. After each iteration, you'll want to compare your model's performance against its performance in previous iterations and evaluate how suitable this iteration is for production. The last section of this chapter is dedicated to how to evaluate your model before deploying it to production, covering a range of evaluation techniques including perturbation tests, invariance tests, model calibration, and slide-based evaluation.

I expect that most readers already have an understanding of common ML algorithms such as linear models, decision trees, *k*-nearest neighbors, and different types of neural networks. This chapter will discuss techniques surrounding these algorithms but won't go into details of how they work. Because this chapter deals with ML algorithms, it requires a lot more ML knowledge than other chapters. If you're not familiar with them, I recommend taking an online course or reading a book on ML algorithms before reading this chapter. Readers wanting a quick refresh on basic ML con-

cepts might find helpful the section “Basic ML Reviews” in the book’s GitHub repository.

Model Development and Training

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

Evaluating ML Models

There are many possible solutions to any given problem. Given a task that can leverage ML in its solution, you might wonder what ML algorithm you should use for it. For example, should you start with logistic regression, an algorithm that you’re already familiar with? Or should you try out a new fancy model that is supposed to be the new state of the art for your problem? A more senior colleague mentioned that gradient-boosted trees have always worked for her for this task in the past—should you listen to her advice?

If you had unlimited time and compute power, the rational thing to do would be to try all possible solutions and see what is best for you. However, time and compute power are limited resources, and you have to be strategic about what models you select.

When talking about ML algorithms, many people think in terms of classical ML algorithms versus neural networks. There are a lot of interests and media coverage for neural networks, especially deep learning, which is understandable given that most of the AI progress in the last decade happened due to neural networks getting bigger and deeper.

These interests and coverage might give off the impression that deep learning is replacing classical ML algorithms. However, even though deep learning is finding more use cases in production, classical ML algorithms are not going away. Many recommender systems still rely on collaborative filtering and matrix factorization. Tree-based algorithms, including gradient-boosted trees, still power many classification tasks with strict latency requirements.

Even in applications where neural networks are deployed, classic ML algorithms are still being used in tandem. For example, neural networks

and decision trees might be used together in an ensemble. A k-means clustering model might be used to extract features to input into a neural network. Vice versa, a pretrained neural network (like BERT or GPT-3) might be used to generate embeddings to input into a logistic regression model.

When selecting a model for your problem, you don't choose from every possible model out there, but usually focus on a set of models suitable for your problem. For example, if your boss tells you to build a system to detect toxic tweets, you know that this is a text classification problem—given a piece of text, classify whether it's toxic or not—and common models for text classification include naive Bayes, logistic regression, recurrent neural networks, and transformer-based models such as BERT, GPT, and their variants.

If your client wants you to build a system to detect fraudulent transactions, you know that this is the classic abnormality detection problem—fraudulent transactions are abnormalities that you want to detect—and common algorithms for this problem are many, including *k*-nearest neighbors, isolation forest, clustering, and neural networks.

Knowledge of common ML tasks and the typical approaches to solve them is essential in this process.

Different types of algorithms require different numbers of labels as well as different amounts of compute power. Some take longer to train than others, whereas some take longer to make predictions. Non-neural network algorithms tend to be more explainable (e.g., what features contributed the most to an email being classified as spam) than neural networks.

When considering what model to use, it's important to consider not only the model's performance, measured by metrics such as accuracy, F1 score, and log loss, but also its other properties, such as how much data, compute, and time it needs to train, what's its inference latency, and interpretability. For example, a simple logistic regression model might have lower accuracy than a complex neural network, but it requires less labeled data to start, it's much faster to train, it's much easier to deploy, and it's also much easier to explain why it's making certain predictions.

Comparing ML algorithms is out of the scope of this book. No matter how good a comparison is, it will be outdated as soon as new algorithms come out. Back in 2016, LSTM-RNNs were all the rage and the backbone of the architecture seq2seq (Sequence-to-Sequence) that powered many NLP tasks from machine translation to text summarization to text classification. However, just two years later, recurrent architectures were largely replaced by transformer architectures for NLP tasks.

To understand different algorithms, the best way is to equip yourself with basic ML knowledge and run experiments with the algorithms you're interested in. To keep up to date with so many new ML techniques and models, I find it helpful to monitor trends at major ML conferences such as NeurIPS, ICLR, and ICML, as well as following researchers whose work has a high signal-to-noise ratio on Twitter.

Six tips for model selection

Without getting into specifics of different algorithms, here are six tips that might help you decide what ML algorithms to work on next.

Avoid the state-of-the-art trap

While helping companies as well as recent graduates get started in ML, I usually have to spend a nontrivial amount of time steering them away from jumping straight into state-of-the-art models. I can see why people want state-of-the-art models. Many believe that these models would be the best solutions for their problems—why try an old solution if you believe that a newer and superior solution exists? Many business leaders also want to use state-of-the-art models because they want to make their businesses appear cutting edge. Developers might also be more excited getting their hands on new models than getting stuck into the same old things over and over again.

Researchers often only evaluate models in academic settings, which means that a model being state of the art often means that *it performs better than existing models on some static datasets*. It doesn't mean that this model will be fast enough or cheap enough for *you* to implement. It doesn't even mean that this model will perform better than other models on *your* data.

While it's essential to stay up to date with new technologies and beneficial to evaluate them for your business, the most important thing to do when solving a problem is finding solutions that can solve that problem. If there's a solution that can solve your problem that is much cheaper and simpler than state-of-the-art models, use the simpler solution.

Start with the simplest models

Zen of Python states that "simple is better than complex," and this principle is applicable to ML as well. Simplicity serves three purposes. First, simpler models are easier to deploy, and deploying your model early allows you to validate that your prediction pipeline is consistent with your training pipeline. Second, starting with something simple and adding more complex components step-by-step makes it easier to understand

your model and debug it. Third, the simplest model serves as a baseline to which you can compare your more complex models.

Simplest models are not always the same as models with the least effort. For example, pretrained BERT models are complex, but they require little effort to get started with, especially if you use a ready-made implementation like the one in Hugging Face's Transformer. In this case, it's not a bad idea to use the complex solution, given that the community around this solution is well developed enough to help you get through any problems you might encounter. However, you might still want to experiment with simpler solutions to ensure that pretrained BERT is indeed better than those simpler solutions for your problem. Pretrained BERT might be low effort to start with, but it can be quite high effort to improve upon. Whereas if you start with a simpler model, there'll be a lot of room for you to improve upon your model.

Avoid human biases in selecting models

Imagine an engineer on your team is assigned the task of evaluating which model is better for your problem: a gradient-boosted tree or a pretrained BERT model. After two weeks, this engineer announces that the best BERT model outperforms the best gradient-boosted tree by 5%. Your team decides to go with the pretrained BERT model.

A few months later, however, a seasoned engineer joins your team. She decides to look into gradient-boosted trees again and finds out that this time, the best gradient-boosted tree outperforms the pretrained BERT model you currently have in production. What happened?

There are a lot of human biases in evaluating models. Part of the process of evaluating an ML architecture is to experiment with different features and different sets of hyperparameters to find the best model of that architecture. If an engineer is more excited about an architecture, they will likely spend a lot more time experimenting with it, which might result in better-performing models for that architecture.

When comparing different architectures, it's important to compare them under comparable setups. If you run 100 experiments for an architecture, it's not fair to only run a couple of experiments for the architecture you're evaluating it against. You might need to run 100 experiments for the other architecture too.

Because the performance of a model architecture depends heavily on the context it's evaluated in—e.g., the task, the training data, the test data, the hyperparameters, etc.—it's extremely difficult to make claims that a

model architecture is better than another architecture. The claim might be true in a context, but unlikely true for all possible contexts.

Evaluate good performance now versus good performance later

The best model now does not always mean the best model two months from now. For example, a tree-based model might work better now because you don't have a ton of data yet, but two months from now, you might be able to double your amount of training data, and your neural network might perform much better.[1]

A simple way to estimate how your model's performance might change with more data is to use learning curves. A learning curve of a model is a plot of its performance—e.g., training loss, training accuracy, validation accuracy—against the number of training samples it uses, as shown in Figure 6-1. The learning curve won't help you estimate exactly how much performance gain you can get from having more training data, but it can give you a sense of whether you can expect any performance gain at all from more training data.

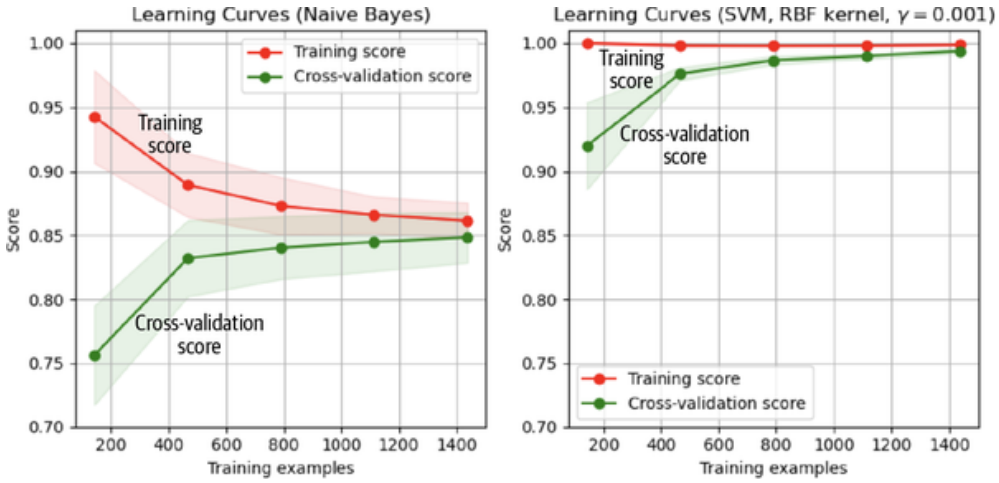

Figure 6-1. The learning curves of a naive Bayes model and an SVM model. Source: scikit-learn

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

While evaluating models, you might want to take into account their potential for improvements in the near future, and how easy/difficult it is to achieve those improvements.

Evaluate trade-offs

There are many trade-offs you have to make when selecting models. Understanding what's more important in the performance of your ML system will help you choose the most suitable model.

One classic example of trade-off is the false positives and false negatives trade-off. Reducing the number of false positives might increase the number of false negatives, and vice versa. In a task where false positives are more dangerous than false negatives, such as fingerprint unlocking (unauthorized people shouldn't be classified as authorized and given access), you might prefer a model that makes fewer false positives. Similarly, in a task where false negatives are more dangerous than false positives, such as COVID-19 screening (patients with COVID-19 shouldn't be classified as no COVID-19), you might prefer a model that makes fewer false negatives.

Another example of trade-off is compute requirement and accuracy—a more complex model might deliver higher accuracy but might require a more powerful machine, such as a GPU instead of a CPU, to generate predictions with acceptable inference latency. Many people also care about the interpretability and performance trade-off. A more complex model can give a better performance, but its results are less interpretable.

Understand your model's assumptions

The statistician George Box said in 1976 that "all models are wrong, but some are useful." The real world is intractably complex, and models can only approximate using assumptions. Every single model comes with its own assumptions. Understanding what assumptions a model makes and whether our data satisfies those assumptions can help you evaluate which model works best for your use case.

Following are some of the common assumptions. It's not meant to be an exhaustive list, but just a demonstration:

Prediction assumption

Every model that aims to predict an output Y from an input X makes the assumption that it's possible to predict Y based on X.

IID

Neural networks assume that the examples are independent and identically distributed, which means that all the examples are independently drawn from the same joint distribution.

Smoothness

Every supervised machine learning method assumes that there's a set of functions that can transform inputs into outputs such that similar inputs are transformed into similar outputs. If an input X produces an output Y, then an input close to X would produce an output proportionally close to Y.

Tractability

Let X be the input and Z be the latent representation of X. Every generative model makes the assumption that it's tractable to compute the probability $P(Z|X)$.

Boundaries

A linear classifier assumes that decision boundaries are linear.

Conditional independence

A naive Bayes classifier assumes that the attribute values are independent of each other given the class.

Normally distributed

Many statistical methods assume that data is normally distributed.

Ensembles

When considering an ML solution to your problem, you might want to start with a system that contains just one model (the process of selecting one model for your problem was discussed earlier in the chapter). After developing one single model, you might think about how to continue improving its performance. One method that has consistently given a performance boost is to use an ensemble of multiple models instead of just an individual model to make predictions. Each model in the ensemble is called a *base learner*. For example, for the task of predicting whether an email is SPAM or NOT SPAM, you might have three different models. The final prediction for each email is the majority vote of all three models. So if at least two base learners output SPAM, the email will be classified as SPAM.

Twenty out of 22 winning solutions on Kaggle competitions in 2021, as of August 2021, use ensembles.[2] As of January 2022, 20 top solutions on

SQuAD 2.0, the Stanford Question Answering Dataset, are ensembles, as shown in Figure 6-2.

Ensembling methods are less favored in production because ensembles are more complex to deploy and harder to maintain. However, they are still common for tasks where a small performance boost can lead to a huge financial gain, such as predicting click-through rate for ads.

Rank	Model	EM	F1
	Human Performance *Stanford University* (Rajpurkar & Jia et al. '18)	86.831	89.452
1 Jun 04, 2021	IE-Net (ensemble) *RICOH_SRCB_DML*	**90.939**	**93.214**
2 Feb 21, 2021	FPNet (ensemble) *Ant Service Intelligence Team*	90.871	93.183
3 May 16, 2021	IE-NetV2 (ensemble) *RICOH_SRCB_DML*	90.860	93.100
4 Apr 06, 2020	SA-Net on Albert (ensemble) *QIANXIN*	90.724	93.011
5 May 05, 2020	SA-Net-V2 (ensemble) *QIANXIN*	90.679	92.948
5 Apr 05, 2020	Retro-Reader (ensemble) *Shanghai Jiao Tong University* http://arxiv.org/abs/2001.09694	90.578	92.978
5 Feb 05, 2021	FPNet (ensemble) *YuYang*	90.600	92.899
6 Apr 18, 2021	TransNets + SFVerifier + SFEnsembler (ensemble) *Senseforth AI Research*	90.487	92.894

Figure 6-2. As of January 2022, the top 20 solutions on SQuAD 2.0 are all ensembles

We'll go over an example to give you the intuition of why ensembling works. Imagine you have three email spam classifiers, each with an accuracy of 70%. Assuming that each classifier has an equal probability of making a correct prediction for each email, and that these three classifiers are not correlated, we'll show that by taking the majority vote of these three classifiers, we can get an accuracy of 78.4%.

For each email, each classifier has a 70% chance of being correct. The ensemble will be correct if at least two classifiers are correct. Table 6-1 shows the probabilities of different possible outcomes of the ensemble given an email. This ensemble will have an accuracy of 0.343 + 0.441 = 0.784, or 78.4%.

Table 6-1. Possible outcomes of the ensemble that takes the majority vote from three classifiers

Outputs of three models	Probability	Ensemble's output
All three are correct	0.7 * 0.7 * 0.7 = 0.343	Correct
Only two are correct	(0.7 * 0.7 * 0.3) * 3 = 0.441	Correct
Only one is correct	(0.3 * 0.3 * 0.7) * 3 = 0.189	Wrong
None are correct	0.3 * 0.3 * 0.3 = 0.027	Wrong

This calculation only holds if the classifiers in an ensemble are uncorrelated. If all classifiers are perfectly correlated—all three of them make the same prediction for every email—the ensemble will have the same accuracy as each individual classifier. When creating an ensemble, the less correlation there is among base learners, the better the ensemble will be. Therefore, it's common to choose very different types of models for an ensemble. For example, you might create an ensemble that consists of one transformer model, one recurrent neural network, and one gradient-boosted tree.

There are three ways to create an ensemble: bagging, boosting, and stacking. In addition to helping boost performance, according to several survey papers, ensemble methods such as boosting and bagging, together with resampling, have shown to help with imbalanced datasets.[3] We'll go over each of these three methods, starting with bagging.

Bagging

Bagging, shortened from *bootstrap aggregating*, is designed to improve both the training stability and accuracy of ML algorithms.[4] It reduces variance and helps to avoid overfitting.

Given a dataset, instead of training one classifier on the entire dataset, you sample with replacement to create different datasets, called bootstraps, and train a classification or regression model on each of these bootstraps. Sampling with replacement ensures that each bootstrap is created independently from its peers. Figure 6-3 shows an illustration of bagging.

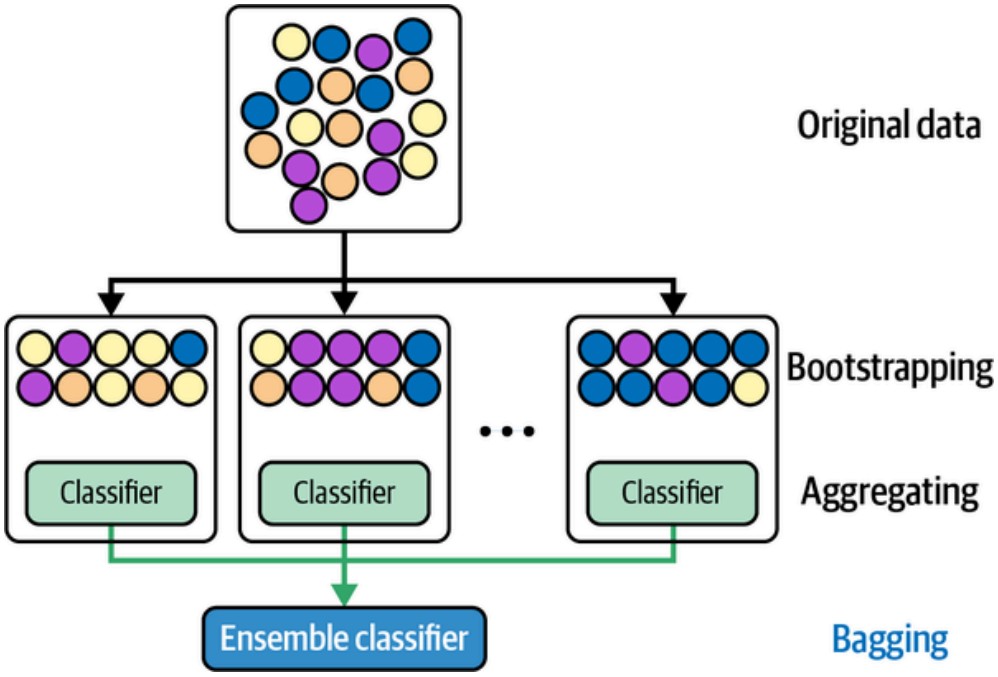

Figure 6-3. Bagging illustration. Source: Adapted from an image by Sirakorn

If the problem is classification, the final prediction is decided by the majority vote of all models. For example, if 10 classifiers vote SPAM and 6 models vote NOT SPAM, the final prediction is SPAM.

If the problem is regression, the final prediction is the average of all models' predictions.

Bagging generally improves unstable methods, such as neural networks, classification and regression trees, and subset selection in linear regression. However, it can mildly degrade the performance of stable methods such as *k*-nearest neighbors.[5]

A random forest is an example of bagging. A random forest is a collection of decision trees constructed by both bagging and feature randomness, where each tree can pick only from a random subset of features to use.

Boosting

Boosting is a family of iterative ensemble algorithms that convert weak learners to strong ones. Each learner in this ensemble is trained on the same set of samples, but the samples are weighted differently among iterations. As a result, future weak learners focus more on the examples that previous weak learners misclassified. Figure 6-4 shows an illustration of boosting, which involves the steps that follow.

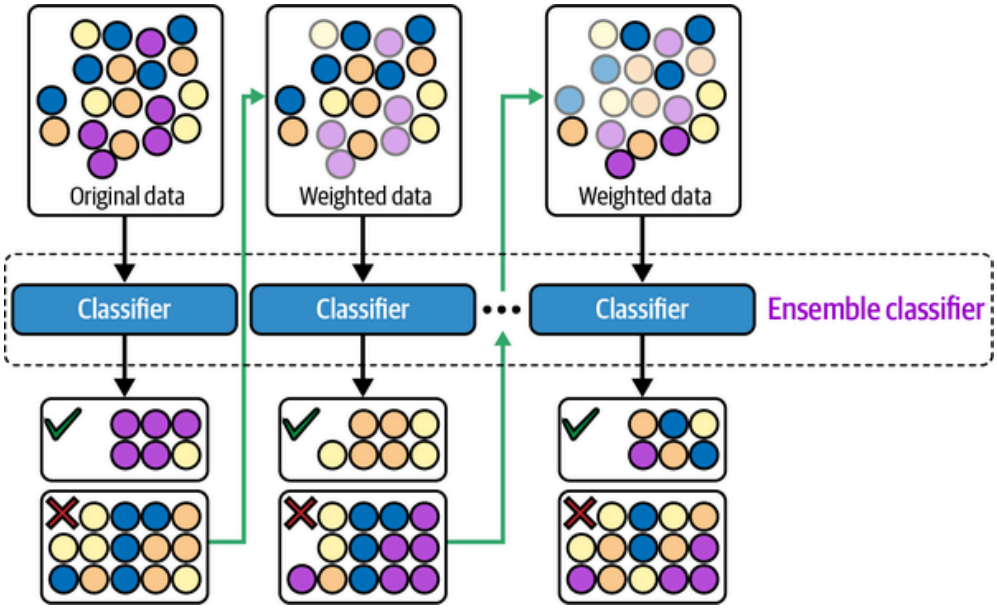

Figure 6-4. Boosting illustration. Source: Adapted from an image by Sirakorn

1. You start by training the first weak classifier on the original dataset.
2. Samples are reweighted based on how well the first classifier classifies them, e.g., misclassified samples are given higher weight.
3. Train the second classifier on this reweighted dataset. Your ensemble now consists of the first and the second classifiers.
4. Samples are weighted based on how well the ensemble classifies them.
5. Train the third classifier on this reweighted dataset. Add the third classifier to the ensemble.
6. Repeat for as many iterations as needed.
7. Form the final strong classifier as a weighted combination of the existing classifiers—classifiers with smaller training errors have higher weights.

An example of a boosting algorithm is a gradient boosting machine (GBM), which produces a prediction model typically from weak decision trees. It builds the model in a stage-wise fashion like other boosting methods do, and it generalizes them by allowing optimization of an arbitrary differentiable loss function.

XGBoost, a variant of GBM, used to be the algorithm of choice for many winning teams of ML competitions.[6] It's been used in a wide range of tasks from classification, ranking, to the discovery of the Higgs Boson.[7] However, many teams have been opting for LightGBM, a distributed gradient boosting framework that allows parallel learning, which generally allows faster training on large datasets.

Stacking

Stacking means that you train base learners from the training data then create a meta-learner that combines the outputs of the base learners to

output final predictions, as shown in Figure 6-5. The meta-learner can be as simple as a heuristic: you take the majority vote (for classification tasks) or the average vote (for regression tasks) from all base learners. It can be another model, such as a logistic regression model or a linear regression model.

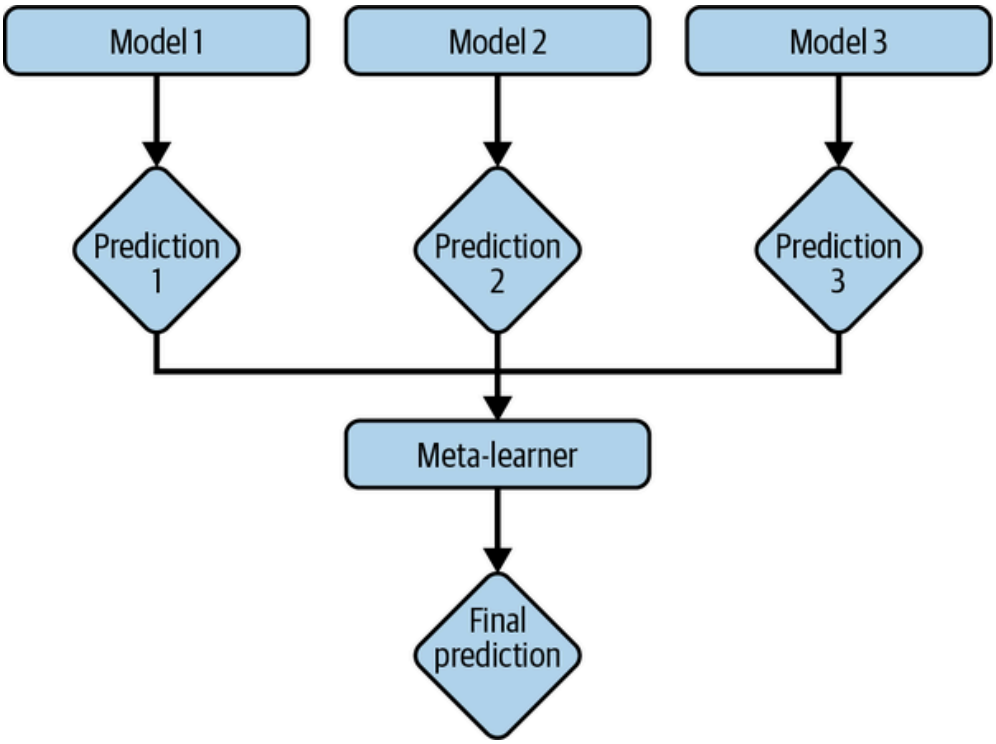

Figure 6-5. A visualization of a stacked ensemble from three base learners

For more great advice on how to create an ensemble, refer to the awesome ensemble guide by one of Kaggle's legendary teams, MLWave.

Experiment Tracking and Versioning

During the model development process, you often have to experiment with many architectures and many different models to choose the best one for your problem. Some models might seem similar to each other and differ in only one hyperparameter—such as one model using a learning rate of 0.003 and another model using a learning rate of 0.002—and yet their performances are dramatically different. It's important to keep track of all the definitions needed to re-create an experiment and its relevant artifacts. An artifact is a file generated during an experiment—examples of artifacts can be files that show the loss curve, evaluation loss graph, logs, or intermediate results of a model throughout a training process. This enables you to compare different experiments and choose the one best suited for your needs. Comparing different experiments can also help you understand how small changes affect your model's performance, which, in turn, gives you more visibility into how your model works.

The process of tracking the progress and results of an experiment is called experiment tracking. The process of logging all the details of an experiment for the purpose of possibly recreating it later or comparing it with other experiments is called versioning. These two go hand in hand with each other. Many tools originally set out to be experiment tracking tools, such as MLflow and Weights & Biases, have grown to incorporate versioning. Many tools originally set out to be versioning tools, such as DVC, have also incorporated experiment tracking.

Experiment tracking

A large part of training an ML model is babysitting the learning processes. Many problems can arise during the training process, including loss not decreasing, overfitting, underfitting, fluctuating weight values, dead neurons, and running out of memory. It's important to track what's going on during training not only to detect and address these issues but also to evaluate whether your model is learning anything useful.

When I just started getting into ML, all I was told to track was loss and speed. Fast-forward several years, and people are tracking so many things that their experiment tracking boards look both beautiful and terrifying at the same time. Following is just a short list of things you might want to consider tracking for each experiment during its training process:

- The *loss curve* corresponding to the train split and each of the eval splits.
- The *model performance metrics* that you care about on all nontest splits, such as accuracy, F1, perplexity.
- The log of *corresponding sample, prediction, and ground truth label*. This comes in handy for ad hoc analytics and sanity check.
- The *speed* of your model, evaluated by the number of steps per second or, if your data is text, the number of tokens processed per second.
- *System performance metrics* such as memory usage and CPU/GPU utilization. They're important to identify bottlenecks and avoid wasting system resources.
- The values over time of any *parameter and hyperparameter* whose changes can affect your model's performance, such as the learning rate if you use a learning rate schedule; gradient norms (both globally and per layer), especially if you're clipping your gradient norms; and weight norm, especially if you're doing weight decay.

In theory, it's not a bad idea to track everything you can. Most of the time, you probably don't need to look at most of them. But when something does happen, one or more of them might give you clues to understand and/or debug your model. In general, tracking gives you observability

into the state of your model.[8] However, in practice, due to the limitations of tooling today, it can be overwhelming to track too many things, and tracking less important things can distract you from tracking what is really important.

Experiment tracking enables comparison across experiments. By observing how a certain change in a component affects the model's performance, you gain some understanding into what that component does.

A simple way to track your experiments is to automatically make copies of all the code files needed for an experiment and log all outputs with their timestamps.[9] Using third-party experiment tracking tools, however, can give you nice dashboards and allow you to share your experiments with your coworkers.

Versioning

Imagine this scenario. You and your team spent the last few weeks tweaking your model, and one of the runs finally showed promising results. You wanted to use it for more extensive tests, so you tried to replicate it using the set of hyperparameters you'd noted down somewhere, only to find out that the results weren't quite the same. You remembered that you'd made some changes to the code between that run and the next, so you tried your best to undo the changes from memory because your reckless past self had decided that the change was too minimal to be committed. But you still couldn't replicate the promising result because there are just too many possible ways to make changes.

This problem could have been avoided if you versioned your ML experiments. ML systems are part code, part data, so you need to not only version your code but your data as well. Code versioning has more or less become a standard in the industry. However, at this point, data versioning is like flossing. Everyone agrees it's a good thing to do, but few do it.

There are a few reasons why data versioning is challenging. One reason is that because data is often much larger than code, we can't use the same strategy that people usually use to version code to version data.

For example, code versioning is done by keeping track of all the changes made to a codebase. A change is known as a diff, short for difference. Each change is measured by line-by-line comparison. A line of code is usually short enough for line-by-line comparison to make sense. However, a line of your data, especially if it's stored in a binary format, can be indefinitely long. Saying that this line of 1,000,000 characters is different from the other line of 1,000,000 characters isn't going to be that helpful.

Code versioning tools allow users to revert to a previous version of the codebase by keeping copies of all the old files. However, a dataset used might be so large that duplicating it multiple times might be unfeasible.

Code versioning tools allow for multiple people to work on the same codebase at the same time by duplicating the codebase on each person's local machine. However, a dataset might not fit into a local machine.

Second, there's still confusion in what exactly constitutes a diff when we version data. Would diffs mean changes in the content of any file in your data repository, only when a file is removed or added, or when the checksum of the whole repository has changed?

As of 2021, data versioning tools like DVC only register a diff if the checksum of the total directory has changed and if a file is removed or added.

Another confusion is in how to resolve merge conflicts: if developer 1 uses data version X to train model A and developer 2 uses data version Y to train model B, it doesn't make sense to merge data versions X and Y to create Z, since there's no model corresponding with Z.

Third, if you use user data to train your model, regulations like General Data Protection Regulation (GDPR) might make versioning this data complicated. For example, regulations might mandate that you delete user data if requested, making it legally impossible to recover older versions of your data.

Aggressive experiment tracking and versioning helps with reproducibility, but it doesn't ensure reproducibility. The frameworks and hardware you use might introduce nondeterminism to your experiment results,[10] making it impossible to replicate the result of an experiment without knowing everything about the environment your experiment runs in.

The way we have to run so many experiments right now to find the best possible model is the result of us treating ML as a black box. Because we can't predict which configuration will work best, we have to experiment with multiple configurations. However, I hope that as the field progresses, we'll gain more understanding into different models and can reason about what model will work best instead of running hundreds or thousands of experiments.

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

First, ML models fail silently, a topic we'll cover in depth in Chapter 8. The code compiles. The loss decreases as it should. The correct functions are called. The predictions are made, but the predictions are wrong. The developers don't notice the errors. And worse, users don't either and use the predictions as if the application was functioning as it should.

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

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

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

Theoretical constraints

As discussed previously, each model comes with its own assumptions about the data and the features it uses. A model might fail because the data it learns from doesn't conform to its assumptions. For example, you use a linear model for the data whose decision boundaries aren't linear.

Poor implementation of model

The model might be a good fit for the data, but the bugs are in the implementation of the model. For example, if you use PyTorch, you might have forgotten to stop gradient updates during evaluation when you should. The more components a model has, the more things that can go wrong, and the harder it is to figure out which

goes wrong. However, with models being increasingly commoditized and more and more companies using off-the-shelf models, this is becoming less of a problem.

Poor choice of hyperparameters

With the same model, one set of hyperparameters can give you the state-of-the-art result but another set of hyperparameters might cause the model to never converge. The model is a great fit for your data, and its implementation is correct, but a poor set of hyperparameters might render your model useless.

Data problems

There are many things that could go wrong in data collection and preprocessing that might cause your models to perform poorly, such as data samples and labels being incorrectly paired, noisy labels, features normalized using outdated statistics, and more.

Poor choice of features

There might be many possible features for your models to learn from. Too many features might cause your models to overfit to the training data or cause data leakage. Too few features might lack predictive power to allow your models to make good predictions.

Debugging should be both preventive and curative. You should have healthy practices to minimize the opportunities for bugs to proliferate as well as a procedure for detecting, locating, and fixing bugs. Having the discipline to follow both the best practices and the debugging procedure is crucial in developing, implementing, and deploying ML models.

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

Start simple and gradually add more components

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

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

Overfit a single batch

After you have a simple implementation of your model, try to overfit a small amount of training data and run evaluation on the same data to make sure that it gets to the smallest possible loss. If it's for image recognition, overfit on 10 images and see if you can get the accuracy to be 100%, or if it's for machine translation, overfit on 100 sentence pairs and see if you can get to a BLEU score of near 100. If it can't overfit a small amount of data, there might be something wrong with your implementation.

Set a random seed

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

Distributed Training

As models are getting bigger and more resource-intensive, companies care a lot more about training at scale.[11] Expertise in scalability is hard to acquire because it requires having regular access to massive compute resources. Scalability is a topic that merits a series of books. This section covers some notable issues to highlight the challenges of doing ML at scale and provide a scaffold to help you plan the resources for your project accordingly.

It's common to train a model using data that doesn't fit into memory. It's especially common when dealing with medical data such as CT scans or genome sequences. It can also happen with text data if you work for teams that train large language models (cue OpenAI, Google, NVIDIA, Cohere).

When your data doesn't fit into memory, your algorithms for preprocessing (e.g., zero-centering, normalizing, whitening), shuffling, and batching data will need to run out of core and in parallel.[12] When a sample of your

data is large, e.g., one machine can handle a few samples at a time, you might only be able to work with a small batch size, which leads to instability for gradient descent-based optimization.

In some cases, a data sample is so large it can't even fit into memory and you will have to use something like gradient checkpointing, a technique that leverages the memory footprint and compute trade-off to make your system do more computation with less memory. According to the authors of the open source package gradient-checkpointing, "For feed-forward models we were able to fit more than 10x larger models onto our GPU, at only a 20% increase in computation time."[13] Even when a sample fits into memory, using checkpointing can allow you to fit more samples into a batch, which might allow you to train your model faster.

Data parallelism

It's now the norm to train ML models on multiple machines. The most common parallelization method supported by modern ML frameworks is data parallelism: you split your data on multiple machines, train your model on all of them, and accumulate gradients. This gives rise to a couple of issues.

A challenging problem is how to accurately and effectively accumulate gradients from different machines. As each machine produces its own gradient, if your model waits for all of them to finish a run—synchronous stochastic gradient descent (SGD)—stragglers will cause the entire system to slow down, wasting time and resources.[14] The straggler problem grows with the number of machines, as the more workers, the more likely that at least one worker will run unusually slowly in a given iteration. However, there have been many algorithms that effectively address this problem.[15]

If your model updates the weight using the gradient from each machine separately—asynchronous SGD—gradient staleness might become a problem because the gradients from one machine have caused the weights to change before the gradients from another machine have come in.[16]

The difference between synchronous SGD and asynchronous SGD is illustrated in Figure 6-6.

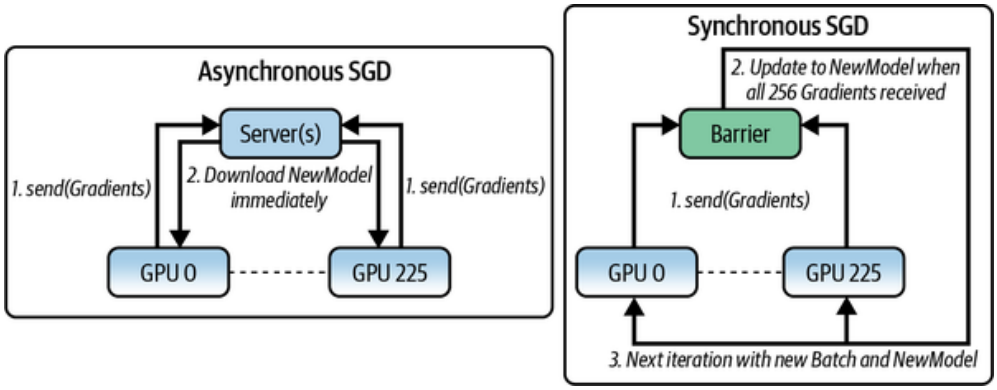

Figure 6-6. Synchronous SGD versus asynchronous SGD for data parallelism. Source: Adapted from an image by Jim Dowling[17]

In theory, asynchronous SGD converges but requires more steps than synchronous SGD. However, in practice, when the number of weights is large, gradient updates tend to be sparse, meaning most gradient updates only modify small fractions of the parameters, and it's less likely that two gradient updates from different machines will modify the same weights. When gradient updates are sparse, gradient staleness becomes less of a problem and the model converges similarly for both synchronous and asynchronous SGD.[18]

Another problem is that spreading your model on multiple machines can cause your batch size to be very big. If a machine processes a batch size of 1,000, then 1,000 machines process a batch size of 1M (OpenAI's GPT-3 175B uses a batch size of 3.2M in 2020).[19] To oversimplify the calculation, if training an epoch on a machine takes 1M steps, training on 1,000 machines might take only 1,000 steps. An intuitive approach is to scale up the learning rate to account for more learning at each step, but we also can't make the learning rate too big as it will lead to unstable convergence. In practice, increasing the batch size past a certain point yields diminishing returns.[20]

Last but not least, with the same model setup, the main worker sometimes uses a lot more resources than other workers. If that's the case, to make the most use out of all machines, you need to figure out a way to balance out the workload among them. The easiest way, but not the most effective way, is to use a smaller batch size on the main worker and a larger batch size on other workers.

Model parallelism

With data parallelism, each worker has its own copy of the whole model and does all the computation necessary for its copy of the model. Model parallelism is when different components of your model are trained on different machines, as shown in Figure 6-7. For example, machine 0 handles the computation for the first two layers while machine 1 handles the

next two layers, or some machines can handle the forward pass while several others handle the backward pass.

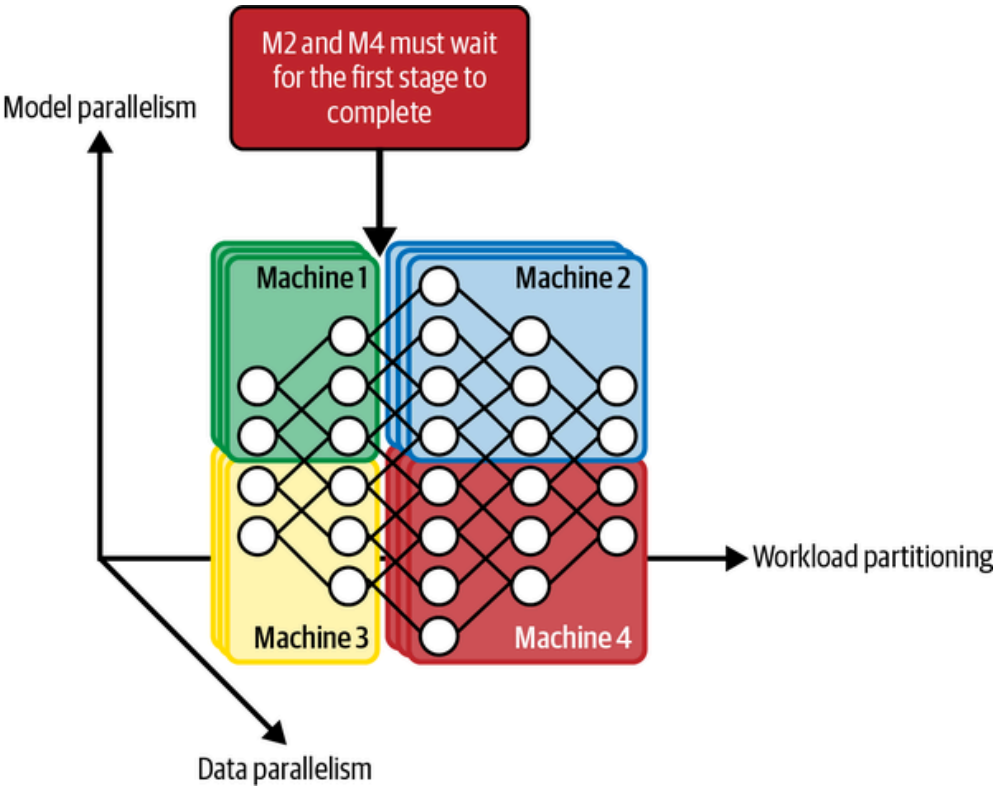

Figure 6-7. Data parallelism and model parallelism. Source: Adapted from an image by Jure Leskovec[21]

Model parallelism can be misleading because in some cases parallelism doesn't mean that different parts of the model in different machines are executed in parallel. For example, if your model is a massive matrix and the matrix is split into two halves on two machines, then these two halves might be executed in parallel. However, if your model is a neural network and you put the first layer on machine 1 and the second layer on machine 2, and layer 2 needs outputs from layer 1 to execute, then machine 2 has to wait for machine 1 to finish first to run.

Pipeline parallelism is a clever technique to make different components of a model on different machines run more in parallel. There are multiple variants to this, but the key idea is to break the computation of each machine into multiple parts. When machine 1 finishes the first part of its computation, it passes the result onto machine 2, then continues to the second part, and so on. Machine 2 now can execute its computation on the first part while machine 1 executes its computation on the second part.

To make this concrete, consider you have four different machines and the first, second, third, and fourth layers are on machine 1, 2, 3, and 4 respectively. With pipeline parallelism, each mini-batch is broken into four micro-batches. Machine 1 computes the first layer on the first micro-batch, then machine 2 computes the second layer on machine 1's results while

machine 1 computes the first layer on the second micro-batch, and so on. Figure 6-8 shows what pipeline parallelism looks like on four machines; each machine runs both the forward pass and the backward pass for one component of a neural network.

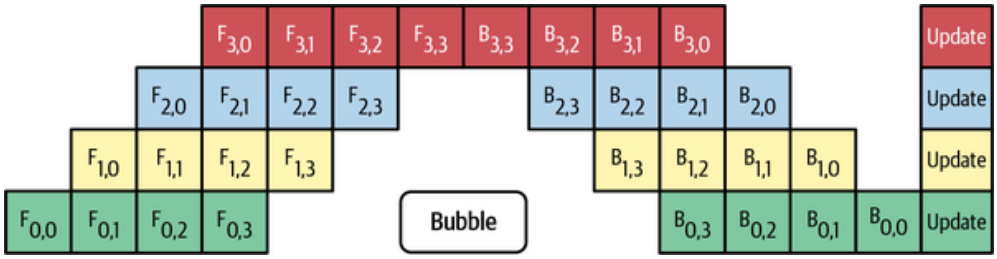

Figure 6-8. Pipeline parallelism for a neural network on four machines; each machine runs both the forward pass (F) and the backward pass (B) for one component of the neural network. Source: Adapted from an image by Huang et al.[22]

Model parallelism and data parallelism aren't mutually exclusive. Many companies use both methods for better utilization of their hardware, even though the setup to use both methods can require significant engineering effort.

AutoML

There's a joke that a good ML researcher is someone who will automate themselves out of job, designing an AI algorithm intelligent enough to design itself. It was funny until the TensorFlow Dev Summit 2018, where Jeff Dean took the stage and declared that Google intended on replacing ML expertise with 100 times more computational power, introducing AutoML to the excitement and horror of the community. Instead of paying a group of 100 ML researchers/engineers to fiddle with various models and eventually select a suboptimal one, why not use that money on compute to search for the optimal model? A screenshot from the recording of the event is shown in Figure 6-9.

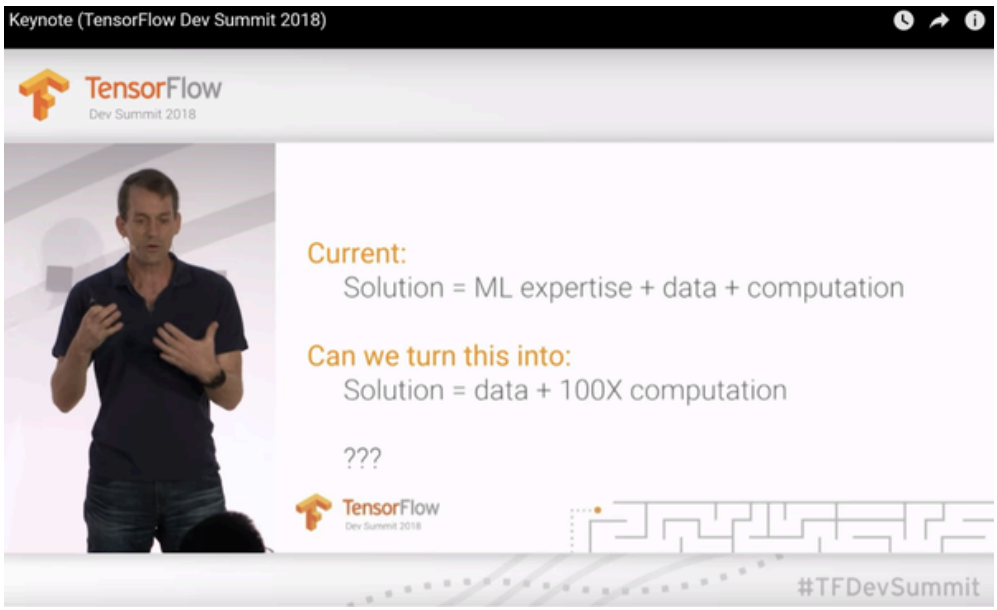

Figure 6-9. Jeff Dean unveiling Google's AutoML at TensorFlow Dev Summit 2018

Soft AutoML: Hyperparameter tuning

AutoML refers to automating the process of finding ML algorithms to solve real-world problems. One mild form, and the most popular form, of AutoML in production is hyperparameter tuning. A hyperparameter is a parameter supplied by users whose value is used to control the learning process, e.g., learning rate, batch size, number of hidden layers, number of hidden units, dropout probability, β_1 and β_2 in Adam optimizer, etc. Even quantization—e.g., whether to use 32 bits, 16 bits, or 8 bits to represent a number or a mixture of these representations—can be considered a hyperparameter to tune.[23]

With different sets of hyperparameters, the same model can give drastically different performances on the same dataset. Melis et al. showed in their 2018 paper "On the State of the Art of Evaluation in Neural Language Models" that weaker models with well-tuned hyperparameters can outperform stronger, fancier models. The goal of hyperparameter tuning is to find the optimal set of hyperparameters for a given model within a search space—the performance of each set evaluated on a validation set.

Despite knowing its importance, many still ignore systematic approaches to hyperparameter tuning in favor of a manual, gut-feeling approach. The most popular is arguably graduate student descent (GSD), a technique in which a graduate student fiddles around with the hyperparameters until the model works.[24]

However, more and more people are adopting hyperparameter tuning as part of their standard pipelines. Popular ML frameworks either come with built-in utilities or have third-party utilities for hyperparameter tuning—for example, scikit-learn with auto-sklearn,[25] TensorFlow with Keras Tuner, and Ray with Tune. Popular methods for hyperparameter tuning include random search,[26] grid search, and Bayesian optimization.[27] The book *AutoML: Methods, Systems, Challenges* by the AutoML group at the University of Freiburg dedicates its first chapter (which you can read online for free) to hyperparameter optimization.

When tuning hyperparameters, keep in mind that a model's performance might be more sensitive to the change in one hyperparameter than another, and therefore sensitive hyperparameters should be more carefully tuned.

WARNING

It's crucial to never use your test split to tune hyperparameters. Choose the best set of hyperparameters for a model based on its performance on a validation split, then report the model's final performance on the test split. If you use your test split to tune hyperparameters, you risk overfitting your model to the test split.

Hard AutoML: Architecture search and learned optimizer

Some teams take hyperparameter tuning to the next level: what if we treat other components of a model or the entire model as hyperparameters. The size of a convolution layer or whether or not to have a skip layer can be considered a hyperparameter. Instead of manually putting a pooling layer after a convolutional layer or ReLu (rectified linear unit) after linear, you give your algorithm these building blocks and let it figure out how to combine them. This area of research is known as architectural search, or neural architecture search (NAS) for neural networks, as it searches for the optimal model architecture.

A NAS setup consists of three components:

A search space

Defines possible model architectures—i.e., building blocks to choose from and constraints on how they can be combined.

A performance estimation strategy

To evaluate the performance of a candidate architecture without having to train each candidate architecture from scratch until convergence. When we have a large number of candidate architectures, say 1,000, training all of them until convergence can be costly.

A search strategy

To explore the search space. A simple approach is random search—randomly choosing from all possible configurations—which is unpopular because it's prohibitively expensive even for NAS. Common approaches include reinforcement learning (rewarding the choices that improve the performance estimation) and evolution (adding mutations to an architecture, choosing the best-performing ones, adding mutations to them, and so on).[28]

For NAS, the search space is discrete—the final architecture uses only one of the available options for each layer/operation,[29] and you have to provide the set of building blocks. The common building blocks are various convolutions of different sizes, linear, various activations, pooling, iden-

tity, zero, etc. The set of building blocks varies based on the base architecture, e.g., convolutional neural networks or transformers.

In a typical ML training process, you have a model and then a learning procedure, an algorithm that helps your model find the set of parameters that minimize a given objective function for a given set of data. The most common learning procedure for neural networks today is gradient descent, which leverages an optimizer to specify how to update a model's weights given gradient updates.[30] Popular optimizers are, as you probably already know, Adam, Momentum, SGD, etc. In theory, you can include optimizers as building blocks in NAS and search for one that works best. In practice, this is difficult to do, since optimizers are sensitive to the setting of their hyperparameters, and the default hyperparameters don't often work well across architectures.

This leads to an exciting research direction: what if we replace the functions that specify the update rule with a neural network? How much to update the model's weights will be calculated by this neural network. This approach results in learned optimizers, as opposed to hand-designed optimizers.

Since learned optimizers are neural networks, they need to be trained. You can train your learned optimizer on the same dataset you're training the rest of your neural network on, but this requires you to train an optimizer every time you have a task.

Another approach is to train a learned optimizer once on a set of existing tasks—using aggregated loss on those tasks as the loss function and existing designed optimizers as the learning rule—and use it for every new task after that. For example, Metz et al. constructed a set of thousands of tasks to train learned optimizers. Their learned optimizer was able to generalize to both new datasets and domains as well as new architectures.[31] And the beauty of this approach is that the learned optimizer can then be used to train a better-learned optimizer, an algorithm that improves on itself.

Whether it's architecture search or meta-learning learning rules, the upfront training cost is expensive enough that only a handful of companies in the world can afford to pursue them. However, it's important for people interested in ML in production to be aware of the progress in AutoML for two reasons. First, the resulting architectures and learned optimizers can allow ML algorithms to work off-the-shelf on multiple real-world tasks, saving production time and cost, during both training and inferencing. For example, EfficientNets, a family of models produced by Google's AutoML team, surpass state-of-the-art accuracy with up to 10x better

efficiency.[32] Second, they might be able to solve many real-world tasks previously impossible with existing architectures and optimizers.

Before we transition to model training, let's take a look at the four phases of ML model development. Once you've decided to explore ML, your strategy depends on which phase of ML adoption you are in. There are four phases of adopting ML. The solutions from a phase can be used as baselines to evaluate the solutions from the next phase:

Phase 1. Before machine learning

If this is your first time trying to make this type of prediction from this type of data, start with non-ML solutions. Your first stab at the problem can be the simplest heuristics. For example, to predict what letter users are going to type next in English, you can show the top three most common English letters, "e," "t," and "a," which might get your accuracy to be 30%.

Facebook newsfeed was introduced in 2006 without any intelligent algorithms—posts were shown in chronological order, as shown in Figure 6-10.[33] It wasn't until 2011 that Facebook started displaying news updates you were most interested in at the top of the feed.

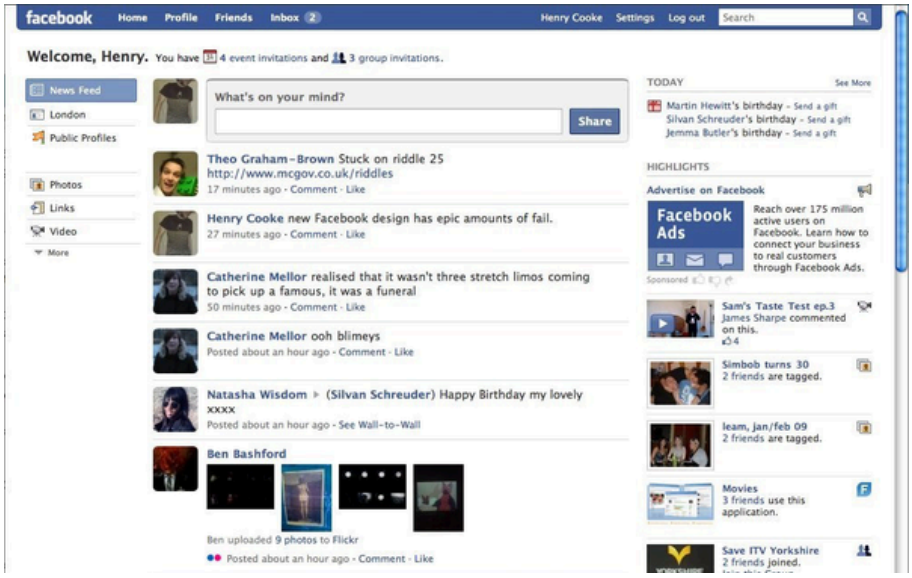

Figure 6-10. Facebook newsfeed circa 2006. Source: Iveta Ryšavá[34]

According to Martin Zinkevich in his magnificent "Rules of Machine Learning: Best Practices for ML Engineering": "If you think that machine learning will give you a 100% boost, then a heuristic will get you 50% of the way there."[35] You might even find that non-ML solutions work fine and you don't need ML yet.

Phase 2. Simplest machine learning models

For your first ML model, you want to start with a simple algorithm, something that gives you visibility into its working to allow you to validate the usefulness of your problem framing and your data.

Logistic regression, gradient-boosted trees, *k*-nearest neighbors can be great for that. They are also easier to implement and deploy, which allows you to quickly build out a framework from data engineering to development to deployment that you can test out and gain confidence on.

Phase 3. Optimizing simple models

Once you have your ML framework in place, you can focus on optimizing the simple ML models with different objective functions, hyperparameter search, feature engineering, more data, and ensembles.

Phase 4. Complex models

Once you've reached the limit of your simple models and your use case demands significant model improvement, experiment with more complex models.

You'll also want to experiment to figure out how quickly your model decays in production (e.g., how often it'll need to be retrained) so that you can build out your infrastructure to support this retraining requirement.[36]

Model Offline Evaluation

One common but quite difficult question I often encounter when helping companies with their ML strategies is: "How do I know that our ML models are any good?" In one case, a company deployed ML to detect intrusions to 100 surveillance drones, but they had no way of measuring how many intrusions their system failed to detect, and they couldn't decide if one ML algorithm was better than another for their needs.

Lacking a clear understanding of how to evaluate your ML systems is not necessarily a reason for your ML project to fail, but it might make it impossible to find the best solution for your need, and make it harder to convince your managers to adopt ML. You might want to partner with the business team to develop metrics for model evaluation that are more relevant to your company's business.[37]

Ideally, the evaluation methods should be the same during both development and production. But in many cases, the ideal is impossible because during development, you have ground truth labels, but in production, you don't.

For certain tasks, it's possible to infer or approximate labels in production based on users' feedback, as covered in the section "Natural Labels". For

example, for the recommendation task, it's possible to infer if a recommendation is good by whether users click on it. However, there are many biases associated with this.

For other tasks, you might not be able to evaluate your model's performance in production directly and might have to rely on extensive monitoring to detect changes and failures in your ML system's performance. We'll cover monitoring in Chapter 8.

Once your model is deployed, you'll need to continue monitoring and testing your model in production. In this section, we'll discuss methods to evaluate your model's performance before it's deployed. We'll start with the baselines against which we will evaluate our models. Then we'll cover some of the common methods to evaluate your model beyond overall accuracy metrics.

Baselines

Someone once told me that her new generative model achieved the FID score of 10.3 on ImageNet.[38] I had no idea what this number meant or whether her model would be useful for my problem.

Another time, I helped a company implement a classification model where the positive class appears 90% of the time. An ML engineer on the team told me, all excited, that their initial model achieved an F1 score of 0.90. I asked him how it was compared to random. He had no idea. It turned out that because for his task the POSITIVE class accounts for 90% of the labels, if his model randomly outputs the positive class 90% of the time, its F1 score would also be around 0.90.[39] His model might as well be making predictions at random.[40]

Evaluation metrics, by themselves, mean little. When evaluating your model, it's essential to know the baseline you're evaluating it against. The exact baselines should vary from one use case to another, but here are the five baselines that might be useful across use cases:

Random baseline

If our model just predicts at random, what's the expected performance? The predictions are generated at random following a specific distribution, which can be the uniform distribution or the task's label distribution.

For example, consider the task that has two labels, NEGATIVE that appears 90% of the time and POSITIVE that appears 10% of the time. Table 6-2 shows the F1 and accuracy scores of baseline mod-

els making predictions at random. However, as an exercise to see how challenging it is for most people to have an intuition for these values, try to calculate these raw numbers in your head before looking at the table.

Table 6-2. F1 and accuracy scores of a baseline model predicting at random

Random distribution	Meaning	F1	Accuracy
Uniform random	Predicting each label with equal probability (50%)	0.167	0.5
Task's label distribution	Predicting NEGATIVE 90% of the time, and POSITIVE 10% of the time	0.1	0.82

Simple heuristic

Forget ML. If you just make predictions based on simple heuristics, what performance would you expect? For example, if you want to build a ranking system to rank items on a user's newsfeed with the goal of getting that user to spend more time on the newsfeed, how much time would a user spend if you just rank all the items in reverse chronological order, showing the latest one first?

Zero rule baseline

The zero rule baseline is a special case of the simple heuristic baseline when your baseline model always predicts the most common class.

For example, for the task of recommending the app a user is most likely to use next on their phone, the simplest model would be to recommend their most frequently used app. If this simple heuristic can predict the next app accurately 70% of the time, any model you build has to outperform it significantly to justify the added complexity.

Human baseline

In many cases, the goal of ML is to automate what would have been otherwise done by humans, so it's useful to know how your model performs compared to human experts. For example, if you work on a self-driving system, it's crucial to measure your system's progress

compared to human drivers, because otherwise you might never be able to convince your users to trust this system. Even if your system isn't meant to replace human experts and only to aid them in improving their productivity, it's still important to know in what scenarios this system would be useful to humans.

Existing solutions

In many cases, ML systems are designed to replace existing solutions, which might be business logic with a lot of if/else statements or third-party solutions. It's crucial to compare your new model to these existing solutions. Your ML model doesn't always have to be better than existing solutions to be useful. A model whose performance is a little bit inferior can still be useful if it's much easier or cheaper to use.

When evaluating a model, it's important to differentiate between "a good system" and "a useful system." A good system isn't necessarily useful, and a bad system isn't necessarily useless. A self-driving vehicle might be good if it's a significant improvement from previous self-driving systems, but it might not be useful if it doesn't perform at least as well as human drivers. In some cases, even if an ML system drives better than an average human, people might still not trust it, which renders it not useful. On the other hand, a system that predicts what word a user will type next on their phone might be considered bad if it's much worse than a native speaker. However, it might still be useful if its predictions can help users type faster some of the time.

Evaluation Methods

In academic settings, when evaluating ML models, people tend to fixate on their performance metrics. However, in production, we also want our models to be robust, fair, calibrated, and overall make sense. We'll introduce some evaluation methods that help with measuring these characteristics of a model.

Perturbation tests

A group of my students wanted to build an app to predict whether someone has COVID-19 through their cough. Their best model worked great on the training data, which consisted of two-second long cough segments collected by hospitals. However, when they deployed it to actual users, this model's predictions were close to random.

One of the reasons is that actual users' coughs contain a lot of noise compared to the coughs collected in hospitals. Users' recordings might contain background music or nearby chatter. The microphones they use are

of varying quality. They might start recording their coughs as soon as recording is enabled or wait for a fraction of a second.

Ideally, the inputs used to develop your model should be similar to the inputs your model will have to work with in production, but it's not possible in many cases. This is especially true when data collection is expensive or difficult and the best available data you have access to for training is still very different from your real-world data. The inputs your models have to work with in production are often noisy compared to inputs in development.[41] The model that performs best on training data isn't necessarily the model that performs best on noisy data.

To get a sense of how well your model might perform with noisy data, you can make small changes to your test splits to see how these changes affect your model's performance. For the task of predicting whether someone has COVID-19 from their cough, you could randomly add some background noise or randomly clip the testing clips to simulate the variance in your users' recordings. You might want to choose the model that works best on the perturbed data instead of the one that works best on the clean data.

The more sensitive your model is to noise, the harder it will be to maintain it, since if your users' behaviors change just slightly, such as they change their phones, your model's performance might degrade. It also makes your model susceptible to adversarial attack.

Invariance tests

A Berkeley study found that between 2008 and 2015, 1.3 million creditworthy Black and Latino applicants had their mortgage applications rejected because of their races.[42] When the researchers used the income and credit scores of the rejected applications but deleted the race-identifying features, the applications were accepted.

Certain changes to the inputs shouldn't lead to changes in the output. In the preceding case, changes to race information shouldn't affect the mortgage outcome. Similarly, changes to applicants' names shouldn't affect their resume screening results nor should someone's gender affect how much they should be paid. If these happen, there are biases in your model, which might render it unusable no matter how good its performance is.

To avoid these biases, one solution is to do the same process that helped the Berkeley researchers discover the biases: keep the inputs the same but change the sensitive information to see if the outputs change. Better,

you should exclude the sensitive information from the features used to train the model in the first place.[43]

Directional expectation tests

Certain changes to the inputs should, however, cause predictable changes in outputs. For example, when developing a model to predict housing prices, keeping all the features the same but increasing the lot size shouldn't decrease the predicted price, and decreasing the square footage shouldn't increase it. If the outputs change in the opposite expected direction, your model might not be learning the right thing, and you need to investigate it further before deploying it.

Model calibration

Model calibration is a subtle but crucial concept to grasp. Imagine that someone makes a prediction that something will happen with a probability of 70%. What this prediction means is that out of all the times this prediction is made, the predicted outcome matches the actual outcome 70% of the time. If a model predicts that team A will beat team B with a 70% probability, and out of the 1,000 times these two teams play together, team A only wins 60% of the time, then we say that this model isn't calibrated. A calibrated model should predict that team A wins with a 60% probability.

Model calibration is often overlooked by ML practitioners, but it's one of the most important properties of any predictive system. To quote Nate Silver in his book *The Signal and the Noise*, calibration is "one of the most important tests of a forecast—I would argue that it is the single most important one."

We'll walk through two examples to show why model calibration is important. First, consider the task of building a recommender system to recommend what movies users will likely watch next. Suppose user A watches romance movies 80% of the time and comedy 20% of the time. If your recommender system shows exactly the movies A will most likely watch, the recommendations will consist of only romance movies because A is much more likely to watch romance than any other type of movies. You might want a more calibrated system whose recommendations are representative of users' actual watching habits. In this case, they should consist of 80% romance and 20% comedy.[44]

Second, consider the task of building a model to predict how likely it is that a user will click on an ad. For the sake of simplicity, imagine that there are only two ads, ad A and ad B. Your model predicts that this user

will click on ad A with a 10% probability and on ad B with an 8% probability. You don't need your model to be calibrated to rank ad A above ad B. However, if you want to predict how many clicks your ads will get, you'll need your model to be calibrated. If your model predicts that a user will click on ad A with a 10% probability but in reality the ad is only clicked on 5% of the time, your estimated number of clicks will be way off. If you have another model that gives the same ranking but is better calibrated, you might want to consider the better calibrated one.

To measure a model's calibration, a simple method is counting: you count the number of times your model outputs the probability *X* and the frequency *Y* of that prediction coming true, and plot *X* against *Y*. The graph for a perfectly calibrated model will have *X* equal *Y* at all data points. In scikit-learn, you can plot the calibration curve of a binary classifier with the method `sklearn.calibration.calibration_curve`, as shown in Figure 6-11.

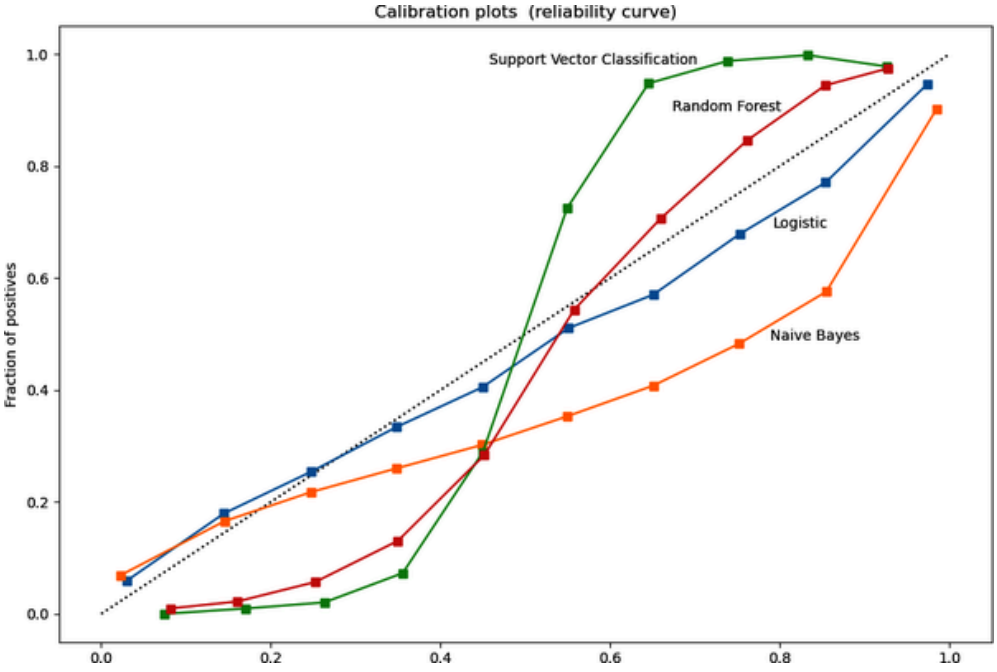

Figure 6-11. The calibration curves of different models on a toy task. The logistic regression model is the best calibrated model because it directly optimizes logistic loss. Source: scikit-learn

To calibrate your models, a common method is Platt scaling, which is implemented in scikit-learn with `sklearn.calibration.CalibratedClassifierCV`. Another good open source implementation by Geoff Pleiss can be found on GitHub. For readers who want to learn more about the importance of model calibration and how to calibrate neural networks, Lee Richardson and Taylor Pospisil have an excellent blog post based on their work at Google.

Confidence measurement

Confidence measurement can be considered a way to think about the usefulness threshold for each individual prediction. Indiscriminately show-

ing all a model's predictions to users, even the predictions that the model is unsure about, can, at best, cause annoyance and make users lose trust in the system, such as an activity detection system on your smartwatch that thinks you're running even though you're just walking a bit fast. At worst, it can cause catastrophic consequences, such as a predictive policing algorithm that flags an innocent person as a potential criminal.

If you only want to show the predictions that your model is certain about, how do you measure that certainty? What is the certainty threshold at which the predictions should be shown? What do you want to do with predictions below that threshold—discard them, loop in humans, or ask for more information from users?

While most other metrics measure the system's performance on average, confidence measurement is a metric for each individual sample. System-level measurement is useful to get a sense of overall performance, but sample-level metrics are crucial when you care about your system's performance on every sample.

Slice-based evaluation

Slicing means to separate your data into subsets and look at your model's performance on each subset separately. A common mistake that I've seen in many companies is that they are focused too much on coarse-grained metrics like overall F1 or accuracy on the entire data and not enough on sliced-based metrics. This can lead to two problems.

One is that their model performs differently on different slices of data when the model should perform the same. For example, their data has two subgroups, one majority and one minority, and the majority subgroup accounts for 90% of the data:

- Model A achieves 98% accuracy on the majority subgroup but only 80% on the minority subgroup, which means its overall accuracy is 96.2%.
- Model B achieves 95% accuracy on the majority and 95% on the minority, which means its overall accuracy is 95%.

These two models are compared in Table 6-3. Which model would you choose?

Table 6-3. Two models' performance on the majority and minority subgroups

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

If a company focuses only on overall metrics, they might go with model A. They might be very happy with this model's high accuracy until, one day, their end users discover that this model is biased against the minority subgroup because the minority subgroup happens to correspond to an underrepresented demographic group.[45] The focus on overall performance is harmful not only because of the potential public backlash, but also because it blinds the company to huge potential model improvements. If the company sees the two models' slice-based performance, they might follow different strategies. For example, they might decide to improve model A's performance on the minority subgroup, which leads to improving this model's performance overall. Or they might keep both models the same but now have more information to make a better-informed decision on which model to deploy.

Another problem is that their model performs the same on different slices of data when the model should perform differently. Some subsets of data are more critical. For example, when you build a model for user churn prediction (predicting when a user will cancel a subscription or a service), paid users are more critical than nonpaid users. Focusing on a model's overall performance might hurt its performance on these critical slices.

A fascinating and seemingly counterintuitive reason why slice-based evaluation is crucial is Simpson's paradox, a phenomenon in which a trend appears in several groups of data but disappears or reverses when the groups are combined. This means that model B can perform better than model A on all data together, but model A performs better than model B on each subgroup separately. Consider model A's and model B's performance on group A and group B as shown in Table 6-4. Model A outperforms model B for both group A and B, but when combined, model B outperforms model A.

Table 6-4. An example of Simpson's paradox[a]

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

a Numbers from Charig et al.'s kidney stone treatment study in 1986: C. R. Charig, D. R. Webb, S. R. Payne, and J. E. Wickham, "Comparison of Treatment of Renal Calculi by Open Surgery, Percutaneous Nephrolithotomy, and Extracorporeal Shockwave Lithotripsy," *British Medical Journal* (Clinical Research Edition) 292, no. 6524 (March 1986): 879–82, *https://oreil.ly/X8oWr*.

Simpson's paradox is more common than you'd think. In 1973, Berkeley graduate statistics showed that the admission rate for men was much higher than for women, which caused people to suspect biases against women. However, a closer look into individual departments showed that the admission rates for women were actually higher than those for men in four out of six departments,[46] as shown in Table 6-5.

Table 6-5. Berkeley's 1973 graduate admission data[a]

	All		Men		Women	
Department	**Applicants**	**Admitted**	**Applicants**	**Admitted**	**Applicants**	**Adm**
A	933	64%	**825**	62%	108	**82%**
B	585	63%	**560**	63%	25	**68 %**
C	918	35%	325	**37 %**	**593**	34%
D	792	34%	417	33%	375	**35 %**
E	584	25%	191	**28 %**	**393**	24%
F	714	6%	373	6%	341	**7 %**
Total	12,763	41%	8,442	**44%**	4,321	35%

a Data from Bickel et al. (1975)

Regardless of whether you'll actually encounter this paradox, the point here is that aggregation can conceal and contradict actual situations. To make informed decisions regarding what model to choose, we need to take into account its performance not only on the entire data, but also on individual slices. Slice-based evaluation can give you insights to improve your model's performance both overall and on critical data and help detect potential biases. It might also help reveal non-ML problems. Once, our team discovered that our model performed great overall but very poorly on traffic from mobile users. After investigating, we realized that it was because a button was half hidden on small screens (e.g., phone screens).

Even when you don't think slices matter, understanding how your model performs in a more fine-grained way can give you confidence in your model to convince other stakeholders, like your boss or your customers, to trust your ML models.

To track your model's performance on critical slices, you'd first need to know what your critical slices are. You might wonder how to discover critical slices in your data. Slicing is, unfortunately, still more of an art than a science, requiring intensive data exploration and analysis. Here are the three main approaches:

Heuristics-based

Slice your data using domain knowledge you have of the data and the task at hand. For example, when working with web traffic, you might want to slice your data along dimensions like mobile versus desktop, browser type, and locations. Mobile users might behave very differently from desktop users. Similarly, internet users in different geographic locations might have different expectations on what a website should look like.[47]

Error analysis

Manually go through misclassified examples and find patterns among them. We discovered our model's problem with mobile users when we saw that most of the misclassified examples were from mobile users.

Slice finder

There has been research to systemize the process of finding slices, including Chung et al.'s "Slice Finder: Automated Data Slicing for Model Validation" in 2019 and covered in Sumyea Helal's "Subgroup Discovery Algorithms: A Survey and Empirical Evaluation" (2016). The process generally starts with generating

slice candidates with algorithms such as beam search, clustering, or decision, then pruning out clearly bad candidates for slices, and then ranking the candidates that are left.

Keep in mind that once you have discovered these critical slices, you will need sufficient, correctly labeled data for each of these slices for evaluation. The quality of your evaluation is only as good as the quality of your evaluation data.

Summary

In this chapter, we've covered the ML algorithm part of ML systems, which many ML practitioners consider to be the most fun part of an ML project lifecycle. With the initial models, we can bring to life (in the form of predictions) all our hard work in data and feature engineering, and can finally evaluate our hypothesis (i.e., we can predict the outputs given the inputs).

We started with how to select ML models best suited for our tasks. Instead of going into pros and cons of each individual model architecture—which is a fool's errand given the growing pools of existing models—the chapter outlined the aspects you need to consider to make an informed decision on which model is best for your objectives, constraints, and requirements.

We then continued to cover different aspects of model development. We covered not only individual models but also ensembles of models, a technique widely used in competitions and leaderboard-style research.

During the model development phase, you might experiment with many different models. Intensive tracking and versioning of your many experiments are generally agreed to be important, but many ML engineers still skip it because doing it might feel like a chore. Therefore, having tools and appropriate infrastructure to automate the tracking and versioning process is essential. We'll cover tools and infrastructure for ML production in Chapter 10.

As models today are getting bigger and consuming more data, distributed training is becoming an essential skill for ML model developers, and we discussed techniques for parallelism including data parallelism, model parallelism, and pipeline parallelism. Making your models work on a large distributed system, like the one that runs models with hundreds of millions, if not billions, of parameters, can be challenging and require specialized system engineering expertise.

We ended the chapter with how to evaluate your models to pick the best one to deploy. Evaluation metrics don't mean much unless you have a baseline to compare them to, and we covered different types of baselines you might want to consider for evaluation. We also covered a range of evaluation techniques necessary to sanity check your models before further evaluating your models in a production environment.

Often, no matter how good your offline evaluation of a model is, you still can't be sure of your model's performance in production until that model has been deployed. In the next chapter, we'll go over how to deploy a model.

1 Andrew Ng has a great lecture where he explains that if a learning algorithm suffers from high bias, getting more training data by itself won't help much. Whereas if a learning algorithm suffers from high variance, getting more training data is likely to help.

2 I went through the winning solutions listed on Farid Rashidi's "Kaggle Solutions" web page. One solution used 33 models (Giba, "1st Place-Winner Solution-Gilberto Titericz and Stanislav Semenov," Kaggle, *https://oreil.ly/z5od8*).

3 Mikel Galar, Alberto Fernandez, Edurne Barrenechea, Humberto Bustince, and Francisco Herrera, "A Review on Ensembles for the Class Imbalance Problem: Bagging-, Boosting-, and Hybrid-Based Approaches," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42, no. 4 (July 2012): 463–84, *https://oreil.ly/ZBlgE*; G. Rekha, Amit Kumar Tyagi, and V. Krishna Reddy, "Solving Class Imbalance Problem Using Bagging, Boosting Techniques, With and Without Using Noise Filtering Method," *International Journal of Hybrid Intelligent Systems* 15, no. 2 (January 2019): 67–76, *https://oreil.ly/hchzU*.

4 Training stability here means less fluctuation in the training loss.

5 Leo Breiman, "Bagging Predictors," *Machine Learning* 24 (1996): 123–40, *https://oreil.ly/adzJu*.

6 "Machine Learning Challenge Winning Solutions," *https://oreil.ly/YjS8d*.

7 Tianqi Chen and Tong He, "Higgs Boson Discovery with Boosted Trees," *Proceedings of Machine Learning Research* 42 (2015): 69–80, *https://oreil.ly/ysBYO*.

8 We'll cover observability in Chapter 8.

9 I'm still waiting for an experiment tracking tool that integrates with Git commits and DVC commits.

10 Notable examples include atomic operations in CUDA where nondeterministic orders of operations lead to different floating point rounding errors between runs.

11 For products that serve a large number of users, you also have to care about scalability in serving a model, which is outside of the scope of an ML project so not covered in this book.

12 According to Wikipedia, "Out-of-core algorithms are algorithms that are designed to process data that are too large to fit into a computer's main memory at once" (s.v. "External memory algorithm," *https://oreil.ly/apv5m*).

13 Tim Salimans, Yaroslav Bulatov, and contributors, gradient-checkpointing repository, 2017, *https://oreil.ly/GTUgC*.

14 Dipankar Das, Sasikanth Avancha, Dheevatsa Mudigere, Karthikeyan Vaidynathan, Srinivas Sridharan, Dhiraj Kalamkar, Bharat Kaul, and Pradeep Dubey, "Distributed Deep Learning Using Synchronous Stochastic Gradient Descent," *arXiv*, February 22, 2016, *https://oreil.ly/ma8Y6*.

15 Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz, "Revisiting Distributed Synchronous SGD," ICLR 2017, *https://oreil.ly/dzVZ5*; Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica, "Improving MapReduce Performance in Heterogeneous Environments," 8th USENIX Symposium on Operating Systems Design and Implementation, *https://oreil.ly/FWswd*; Aaron Harlap, Henggang Cui, Wei Dai, Jinliang Wei, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing, "Addressing the Straggler Problem for Iterative Convergent Parallel ML" (SoCC '16, Santa Clara, CA, October 5–7, 2016), *https://oreil.ly/wZgOO*.

16 Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Marc'aurelio Ranzato, et al., "Large Scale Distributed Deep Networks," NIPS 2012, *https://oreil.ly/EWPun*.

17 Jim Dowling, "Distributed TensorFlow," O'Reilly Media, December 19, 2017, *https://oreil.ly/VYlOP*.

18 Feng Niu, Benjamin Recht, Christopher Ré, and Stephen J. Wright, "Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent," 2011, *https://oreil.ly/sAEbv*.

19 Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, et al., "Language Models Are Few-Shot Learners," *arXiv*, May 28, 2020, *https://oreil.ly/qjg2S*.

20 Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team, "An Empirical Model of Large-Batch Training," *arXiv*, December 14, 2018, *https://oreil.ly/mcjbV*; Christopher J. Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl, "Measuring the Effects of Data Parallelism on Neural Network Training," *Journal of Machine Learning Research* 20 (2019): 1–49, *https://oreil.ly/YAEOM*.

21 Jure Leskovec, Mining Massive Datasets course, Stanford, lecture 13, 2020, *https://oreil.ly/gZcja*.

22 Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, et al., "GPipe: Easy Scaling with Micro-Batch Pipeline Parallelism," *arXiv*, July 25, 2019, *https://oreil.ly/wehkx*.

23 We'll cover quantization in Chapter 7.

24 GSD is a well-documented technique. See "How Do People Come Up With All These Crazy Deep Learning Architectures?," Reddit, *https://oreil.ly/5vEsH*; "Debate About Science at Organizations like Google Brain/FAIR/DeepMind," Reddit, *https://oreil.ly/2K77r*; "Grad Student Descent," *Science Dryad*, January 25, 2014, *https://oreil.ly/dIR9r*; and Guy Zyskind (@GuyZys), "Grad Student Descent: the preferred #nonlinear #optimization technique #machinelearning," Twitter, April 27, 2015, *https://oreil.ly/SW1or*.

25 auto-sklearn 2.0 also provides basic model selection capacity.

26 Our team at NVIDIA developed Milano, a framework-agnostic tool for automatic hyperparameter tuning using random search.

27 A common practice I've observed is to start with coarse-to-fine random search, then experiment with Bayesian or grid search once the search space has been significantly reduced.

28 Barret Zoph and Quoc V. Le, "Neural Architecture Search with Reinforcement Learning," *arXiv*, November 5, 2016, *https://oreil.ly/FhsuQ*; Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le, "Regularized Evolution for Image Classifier Architecture Search," AAAI 2019, *https://oreil.ly/FWYjn*.

29 You can make the search space continuous to allow differentiation, but the resulting architecture has to be converted into a discrete architecture. See "DARTS: Differentiable Architecture Search" (Liu et al. 2018).

30 We cover learning procedures and optimizers in more detail in the section "Basic ML Reviews" in the book's GitHub repository.

31 Luke Metz, Niru Maheswaranathan, C. Daniel Freeman, Ben Poole, and Jascha Sohl-Dickstein, "Tasks, Stability, Architecture, and Compute: Training More Effective Learned Optimizers, and Using Them to Train Themselves," *arXiv*, September 23, 2020, *https://oreil.ly/IH7eT*.

32 Mingxing Tan and Quoc V. Le, "EfficientNet: Improving Accuracy and Efficiency through AutoML and Model Scaling," *Google AI Blog*, May 29, 2019, *https://oreil.ly/gonEn*.

33 Samantha Murphy, "The Evolution of Facebook News Feed," *Mashable*, March 12, 2013, *https://oreil.ly/1HMXh*.

34 Iveta Ryšavá, "What Mark Zuckerberg's News Feed Looked Like in 2006," Newsfeed.org, January 14, 2016, *https://oreil.ly/XZT6Q*.

35 Martin Zinkevich, “Rules of Machine Learning: Best Practices for ML Engineering,” Google, 2019, *https://oreil.ly/YtEsN*.

36 We’ll go in depth about how often to update your models in Chapter 9.

37 See the section “Business and ML Objectives”.

38 Fréchet inception distance, a common metric for measuring the quality of synthesized images. The smaller the value, the higher the quality is supposed to be.

39 The accuracy, in this case, would be around 0.80.

40 Revisit the section “Using the right evaluation metrics” for a refresh on the asymmetry of F1.

41 Other examples of noisy data include images with different lighting or texts with accidental typos or intentional text modifications such as typing “long” as “looooong.”

42 Khristopher J. Brooks, “Disparity in Home Lending Costs Minorities Millions, Researchers Find,” *CBS News*, November 15, 2019, *https://oreil.ly/TMPVl*.

43 It might also be mandated by law to exclude sensitive information from the model training process.

44 For more information on calibrated recommendations, check out the paper “Calibrated Recommendations” by Harald Steck in 2018 based on his work at Netflix.

45 Maggie Zhang, “Google Photos Tags Two African-Americans As Gorillas Through Facial Recognition Software,” *Forbes*, July 1, 2015, *https://oreil.ly/VYG2j*.

46 P. J. Bickel, E. A. Hammel, and J. W. O’Connell, “Sex Bias in Graduate Admissions: Data from Berkeley,” *Science* 187 (1975): 398–404, *https://oreil.ly/TeR7E*.

47 For readers interested in learning more about UX design across cultures, Jenny Shen has a great post.