Lecture 6. Model Offline Evaluation

Note: *This note is a work-in-progress, created for the course CS 329S: Machine Learning Systems Design (Stanford, 2022). For the fully developed text, see the book Designing Machine Learning Systems (Chip Huyen, O'Reilly 2022).*

Errata, questions, and feedback -- please send to chip@huyenchip.com. Thank you!

Table of contents

Distributed Training

As models are getting bigger and more resource-intensive, companies care a lot more about training at scale[1]. 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 a dataset that doesn't fit into memory. It happens a lot when dealing with medical data such as CT scans or genome sequences. It can also happen with text data if you're a tech giant with enough compute resources to work with a massive dataset (cue OpenAI, Google, NVIDIA, Facebook).

When your data doesn't fit into memory, you will first need algorithms for preprocessing (e.g. zero-centering, normalizing, whitening), shuffling, and batching data out-of-memory and in parallel. When a sample of your data is too 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 tradeoff to make your system do more computation with less memory. According to the authors of the open-source package gradient-checkpointing, "*for feed-forward model, we were able to fit more than 10x larger models onto our GPU, at only a 20% increase in computation time*"[2]. 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 (CPUs, GPUs, TPUs). Modern ML frameworks make it easy to do distributed training. The most common parallelization method 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 (SSGD) — stragglers will cause the

[1] 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 a machine learning project so not covered in this book.

[2] gradient-checkpointing repo. Tim Salimans, Yaroslav Bulatov and contributors, 2017.

entire model to slow down, wasting time and resources[3]. 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 [4], [5], [6].

If your model updates the weight using the gradient from each machine separately — Asynchronous SGD (ASGD) — 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[7].

The difference between SSGD and ASGD is illustrated in Figure 5-7.

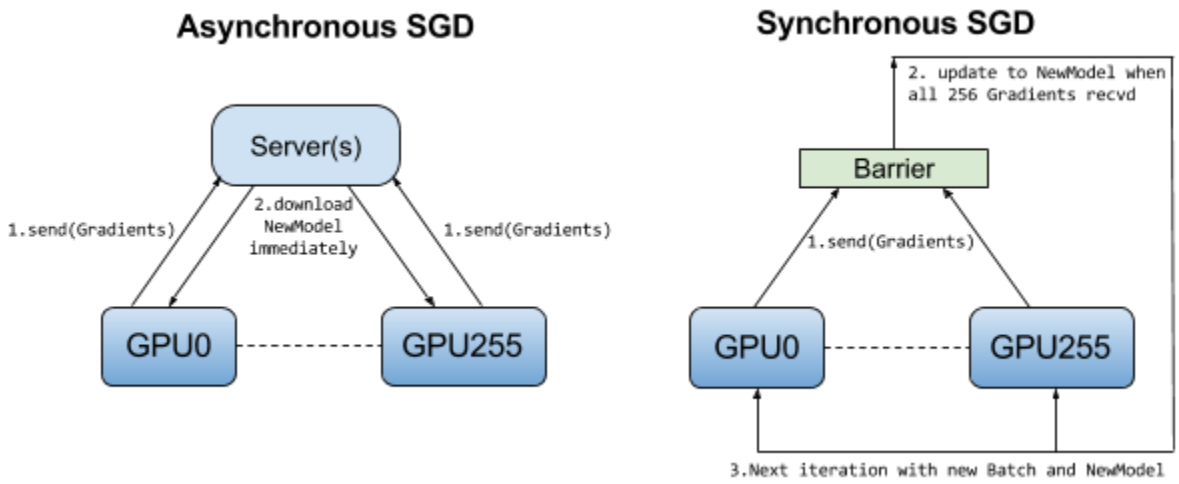

Figure 5-7: ASGD vs. SSGD for data parallelism.
Image by Jim Dowling[8]

In theory, ASGD converges but requires more steps than SSGD. However, in practice, when gradient updates are sparse, meaning most gradient updates only modify small fractions of the parameters, the model converges similarly[9].

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 1000, then 1000 machines process a batch size of 1M (OpenAI's GPT-3 175B uses a batch size of 3.2M in 2020[10]). If training an epoch on a

---

[3] Distributed Deep Learning Using Synchronous Stochastic Gradient Descent, Das et al., 2016.
[4] Revisiting Distributed Synchronous SGD, Chen et al., ICLR 2017.
[5] Improving MapReduce Performance in Heterogeneous Environments, Zaharia et al., 2008.
[6] Addressing the straggler problem for iterative convergent parallel ML, Harlap et al., SoCC 2016.
[7] Large Scale Distributed Deep Networks, Dean et al., NIPS 2012.
[8] Distributed TensorFlow (Jim Dowling, O'Reilly 2017)
[9] Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent (Niu et al., 2011)
[10] Language Models are Few-Shot Learners (Brown et al., 2020)

machine takes 1M steps, training on 1000 machines takes 1000 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[11],[12].

Last but not least, with the same model setup, the master-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 master-worker and a larger batch size on other workers.

Model Parallelism

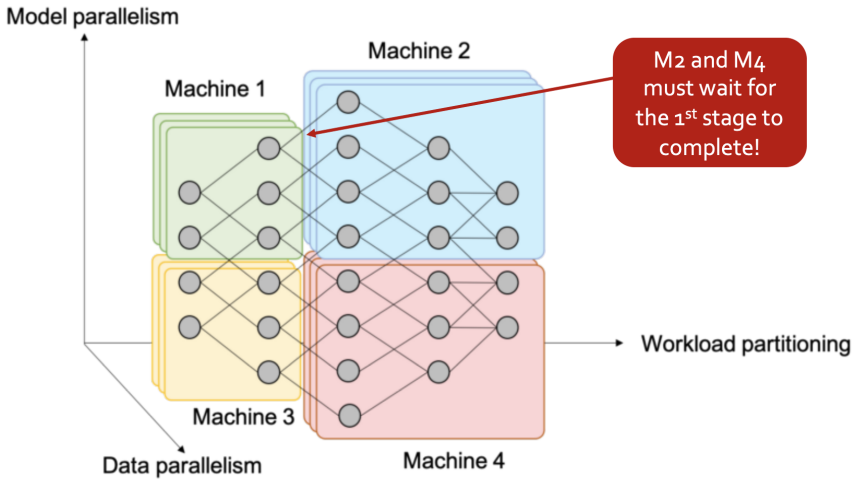

Figure 5-8: Data parallelism and model parallelism.
Image by Jure Leskovec.[13]

With data parallelism, each worker has its own copy of the model and does all the computation necessary for the model. Model parallelism is when different components of your model are trained on different machines, as shown in Figure 5-8. 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.

Model parallelism can be misleading in some cases when parallelism doesn't mean that different parts of the model in different machines are executed in parallel. For example, if your model is a

[11] An Empirical Model of Large-Batch Training (McCandlish et al., 2018)
[12] [1811.03600] Measuring the Effects of Data Parallelism on Neural Network Training (Shallue et al., 2018)
[13] Mining Massive Datasets course,Stanford, lecture 13. Jure Leskovec. 2020.

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, and when machine 1 finishes the first part of its computation, it passes the result onto machine 2, then continues executing the second part, and so on. Machine 2 now can execute its computation on part 1 while machine 1 executes its computation on part 2.

To make this concrete, consider you have 4 different machines and the first, second, third, and fourth layers are on machine 1, 2, 3, and 4 respectively. Given a mini-batch, you break it into 4 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 for the first micro-batch while machine 1 computes the first layer on the second micro-batch, and so on. Figure 5-9 shows how pipeline parallelism looks like on 4 machines, each machine runs both the forward pass and the backward pass for one component of a neural network.

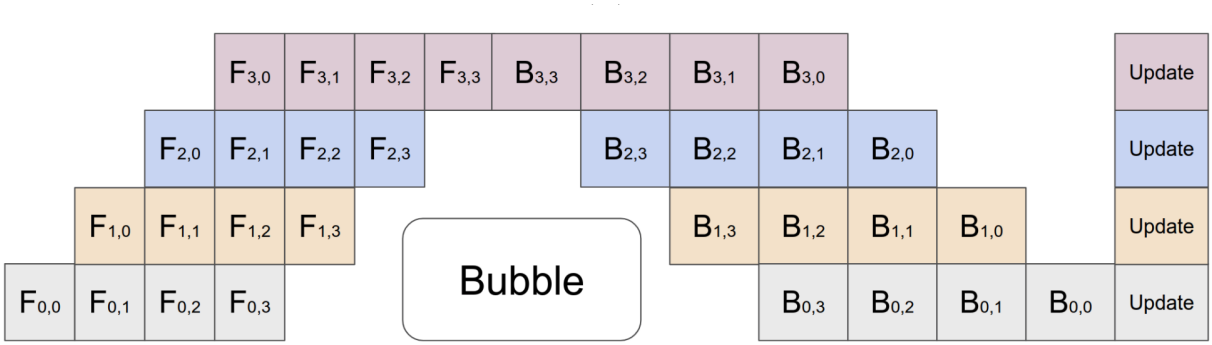

Figure 5-9: Pipeline parallelism for a neural network on 4 machines, each machine runs both the forward pass (F) and the backward pass (B) for one component of the neural network. Image by Huang et al.

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.

Model Offline Evaluation

One common but quite difficult question I often encounter when helping companies with their AI strategies is: "How do I know that the ML model is 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 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.

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

For certain tasks, it's possible to infer or approximate ground truths in production based on user's feedback. 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. The section Continual Learning in Chapter 8 will cover how to leverage users' feedback to improve your systems in production.

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 in your model's performance in particular and to your system in general. We'll cover the causes of changes in your model's performance in Chapter 7 and the tools you can use to detect these changes in the section Monitoring in chapter 8.

Both monitoring and continual learning can happen once your model has been deployed. 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[14] score of 10.3 on ImageNet. I had no idea what this number meant or whether her model would be useful for my problem.

[14] 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.

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 if his model randomly outputted the positive class 90% of the time, its F1 score would also be around 0.90[15]. His model might as well be making predictions at random, which meant it probably didn't learn anything much.

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.

1. **Random baseline**
   If our model just predicts at random, what's the expected performance? At random means both "following a uniform random distribution" or "following the same distribution as 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 5-2 shows the F1 and accuracy scores of baseline models 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.

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

Table 5-2: F1 and accuracy scores of a baseline model predicting at random for a task that has NEGATIVE that appears 90% of the time and POSITIVE that appears 10% of the time.

2. **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,

[15] The accuracy, in this case, would be around 0.80.

how much time would a user spend on it if you just rank all the items in reverse chronological order, with the latest one shown first?

3. **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.

4. **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.

5. **Existing solutions**
   In some 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.

Picking up the usefulness thread from the last section, a good system isn't necessarily useful. A system meant to replace human experts often has to perform at least as well as human experts to be useful. In some cases, even if it's better than human experts, people might still not trust it, as in the case of self-driving cars. On the contrary, a system that predicts what word a user will type next on their phone can perform much worse than a native speaker and still be useful.

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 the above 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 2-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 you have to rely on the data collected by someone else. As a result, inputs in production are often noisy compared to inputs used in development[16]. The model that performs best on the training data isn't necessarily the model that performs best on noisy inputs in production.

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 how the recordings might be in production. 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 and get much higher quality microphones, your model's performance might degrade. It also makes your model susceptible to adversarial attack.

Invariance Tests

A Berkeley study found out that between 2008 and 2015, 1.3 million creditworthy black and Latino applicants had their mortgage applications rejected because of their races. When the researchers used the income and credit scores of the rejected applications but deleted the race identifiers, the applications were accepted.

[16] Other examples of noisy data include images with different lighting or texts with accidental typos or intentional text modifications such as typing "long" as "looooooong."

Certain changes to the inputs shouldn't lead to changes in the output. In the case above, 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[17].

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 the output. 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 the times this prediction is made, this event happens 70% of the time. If a model predicts that team A will beat team B with a 70% probability, and out of the 1000 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 system that makes predictions. 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 to watch next. Suppose user A watches romance movies 80% of the time and comedy 20% of the time. If you choose your recommendations to consist of only the movies A will most likely to watch, the recommendations will only consist of romance movies because A is much more likely to watch romance than comedy movies. You might want a calibrated recommendation system whose

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

recommendations are representative of users' actual watching habits. In this case, they should consist of 80% romance and 20% comedy[18].

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 2 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 a 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 than this model, you might want to consider this other model.

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. In scikit-learn, you can plot the calibration curve of a binary classifier with the method `sklearn.calibration.calibration_curve`, as shown in Figure 5-10.

[18] For more information on calibrated recommendations, check out the paper Calibrated recommendations by Harald Steck in 2018 based on his work at Netflix.

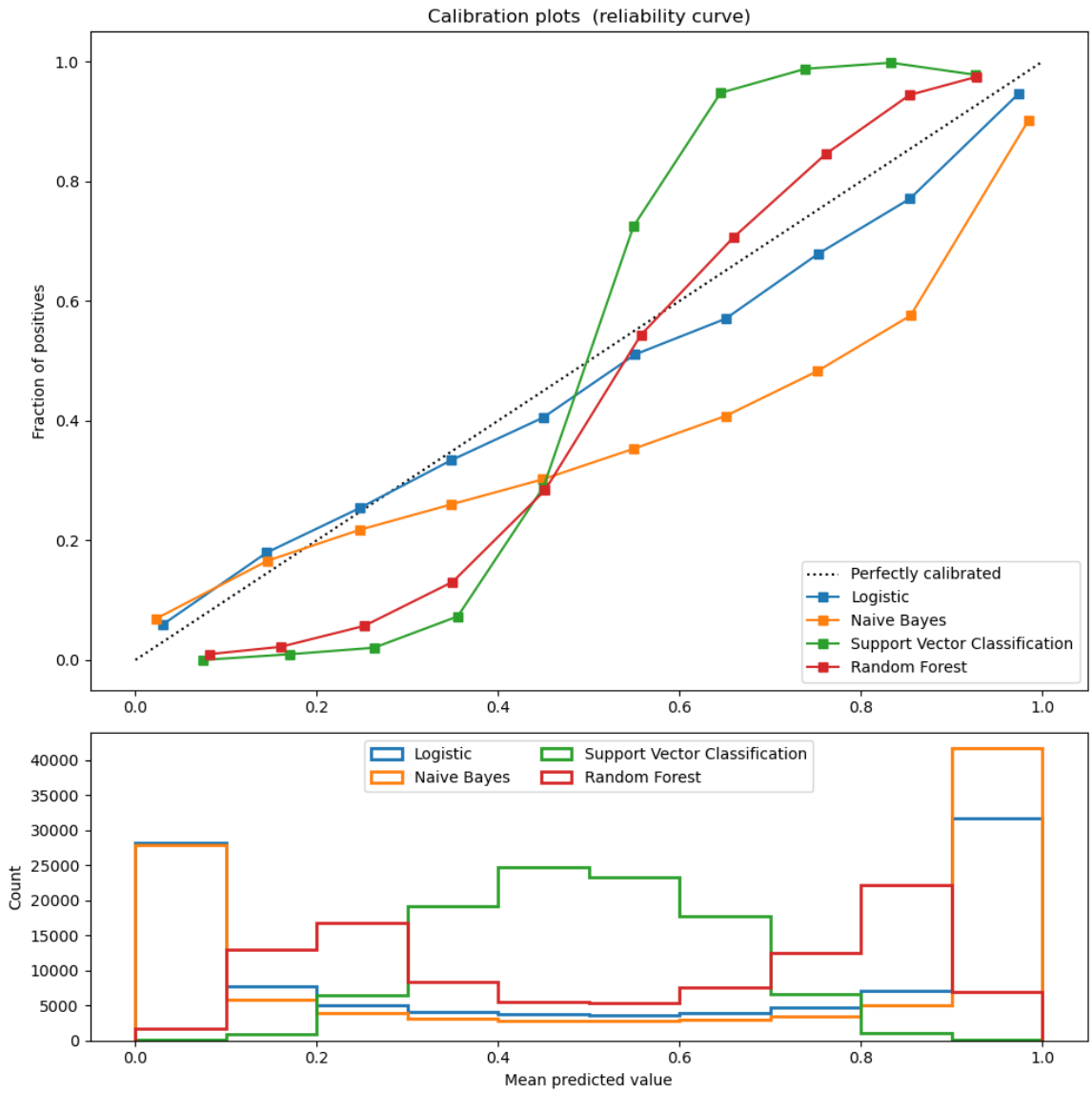

Figure 5-10: The calibration curves of different models on a toy task. The Logistic Regression model is the best calibrated model because it directly optimizes logistic loss. Image by scikit-learn.

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 showing all 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 it, loop in humans, or ask for more information from users?

While most other metrics deal with system-level measuring system's performance on average, confidence measurement is a metric for each individual instance. System-level measurement is useful to get a sense of overall performance, but instance-level metrics are crucial when you care about your system's performance on every instance, and the model's failure in just one instance can be catastrophic.

Slice-based Evaluation

Slicing means to separate your data into subgroups and look at your model's performance on those subgroups separately. A common mistake that I've seen in many companies is that they are focused only on coarse-grained metrics like overall F1 or accuracy on the entire datasets. This can lead to two problems.

One is that their model performs differently on different slices (subsets) of data when the model should perform the same. For example, if 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 5-3.

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[19]. The focus on overall performance is harmful not only

[19] Google Photos Tags Two African-Americans As Gorillas Through Facial Recognition Software (Maggie Zhang, Forbes 2015) d

because of the potential public's backlash, but also because it blinds the company to huge potential model's improvements. If the company sees the two models' performance on different subgroups, they can use different strategies regarding these two models' performance: improve model A's performance on the minority subgroup while improving model's performance overall, and choose one after having weighed the pros and cons of both.

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

Table 5-3: Two models' performance on the majority subgroup, which accounts for 90% of the data, and the minority subgroup, which accounts for 10% of the data. Which model would you choose?

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 non-paid 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 A can perform better than model B on all data together but model B performs better than model A on each subgroup separately. Consider model A's and model B's performance on group A and group B as shown in Table 5-4. Model A outperforms model B for both group A and B, but when combined, model B outperforms model 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)**

Table 5-4: An example of Simpson's paradox. Model A outperforms model B for both group A and B, but when combined, model B outperforms model A.
Numbers from Numbers from Charig et al.'s kidney stone treatment study in 1986.

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 4 out of 6 departments, as shown in Figure 5-11.

	All		Men		Women	
	Applicants	Admitted	Applicants	Admitted	Applicants	Admitted
Total	12,763	41%	8442	**44%**	4321	35%

Department	All		Men		Women	
	Applicants	Admitted	Applicants	Admitted	Applicants	Admitted
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%**

Figure 5-11: The overall graduate admission rate for men and women at Berkeley in 1973 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 4 out of 6 departments. Data from Sex Bias in Graduate Admissions: Data from Berkeley (Bickel et al., 1975). Screenshotted on Wikipedia.

Whether this paradox happens in our work or not, 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-machine learning 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, like 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 existing 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[20]. This approach might require subject matter expertise.
- **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 prune out clearly bad candidates for slices, and then rank the candidates that are left.

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