# Designing Machine Learning Systems - Chip Huyen

## Chapter 6 – Model Development and Offline Evaluation

* Now that we’ve discussed how to create training data for your model and how to engineer features from that training data, with this initial set of features, we’ll move to the ML algorithm part of ML systems
* This is a fun step that it allows one to play around with different algorithms and techniques, even the latest ones
* This is also the first step where one can see all the hard work they’ve put into data and feature engineering transformed into a system whose outputs (predictions) they can use to evaluate the success of their 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
* We start with 6 tips for selecting the best algorithms for your task, then discuss 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**
* We then go over how to evaluate a model before deploying it to production, covering a range of evaluation techniques including **perturbation tests**, **invariance tests**, **model calibration**, and **slide-based evaluation**.

### Model Development and Training

* Necessary aspects to help develop and train a model, such as how to evaluate different ML models for a problem, creating **ensembles** of models, **experiment tracking** and **versioning**, and **distributed training** (necessary for the scale at which models today are usually trained at), not to mention the optional **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 even leverage ML in its solution*, you might **wonder what ML algorithm you should use** for it
* Ex: Should you start with logistic regression, an algorithm you’re already familiar with? Or should you try out a new fancy model that’s supposed to be the new state of the art (SOTA) 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 think in terms of **classical ML algorithms versus NN’s**
* There are a lot of interests and media coverage for NN’s, especially DL, which is understandable given that most AI progress in the last decade happened due to NN’s getting bigger and deeper
* These interests + coverage might give off the impression that DL is replacing classical ML algorithms
* However, **even though DL is finding more use cases in production, classical ML algorithms are not going away**
* Ex: 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 NN’s *are* deployed, classic ML algorithms are still being used in tandem**
* Ex: NN’s 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 NN
* Vice versa, a pretrained NN (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***
* Ex: To build a system to detect toxic tweets, you know 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, RNN’s, and transformer-based models such as BERT, GPT, + their variants.
* Ex: To build a system to detect fraudulent transactions, you know 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 KNN, isolation forest, clustering, and NN’s
* **Knowledge of common ML tasks + 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 NN’s
* **When considering what model to use**, it’s important to **consider not *only* the model’s performance** as measured by metrics such as accuracy, F1 score, and log loss, **but *also* its *other* properties, such as how much data, compute, + time it needs to train, what’s its inference latency, and interpretability**
* Ex: A **simple logistic regression model** might have lower accuracy than a complex NN, but it **requires less labeled data to start, is much faster to train, is much easier to deploy,** and it’s **also much easier to explain why it’s making certain predictions**
* **No matter how good a comparison of ML models is, it will be outdated as soon as new algorithms come out**
* Ex: 2016, LSTM-RNNs were all the rage + 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 2 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

##### 6 tips for model selection

###### 1) Avoid the state-of-the-art trap

* Many believe that SOTA 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 SOTA 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, and 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 actually *solve* that problem**
* **If there’s a solution that can solve your problem that is much cheaper and simpler than SOTA models, use the simpler solution**

###### 2) Start with the simplest models

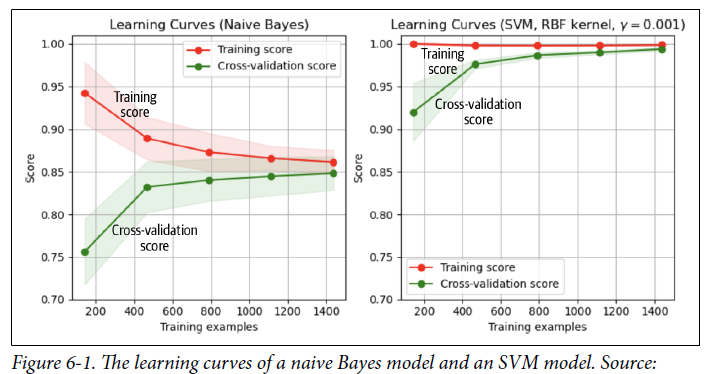
* Zen of Python = “**simple is better than complex**”, and this principle is **applicable to ML as well**
* Simplicity serves 3 purposes.
* **1) Simpler models are easier to deploy**, and **deploying models early allows you to validate that your prediction pipeline is consistent with your training pipeline**
* **2) Starting with something simple and adding more complex components step-by-step makes it easier to understand your model and debug it**
* **3) The simplest model serves as a baseline to which you can compare more complex models**
* **Simplest models are *NOT* always the same as models with the least effort**
* Ex: Pre-trained BERT models are complex, but 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 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**
* Pre-trained 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**

###### 3) Avoid human biases in selecting models

* Ex: 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 2 weeks, this engineer announces 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 + decides to look into gradient-boosted trees again + 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 your 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.**

###### 4) Evaluate good performance now versus good performance later

* **The best model *now* does not always mean the best model 2 months from now**
* Ex: A tree-based model might work better *now* because you don’t have a ton of data yet, but 2 months from now, you might be able to 2X your amount of training data, and a NN might perform much better
* **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**
* A **simple way to estimate how your model’s performance might change with more data is to use learning curves** = **a plot of a model’s performance** (e.g.*, training loss, training accuracy, validation accuracy*, etc.) **against the number of training samples it uses**
* **A 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**



* Ex: A team evaluates a simple NN against a collaborative filtering model for making recommendations
* When evaluating both models *offline*, collaborative filtering outperformed
* **However, the simple NN 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 NN
* They used the collaborative filtering model to make predictions for users, and continually trained the simple NN in production with new, incoming data
* After 2 weeks, the simple NN 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**

###### 5) Evaluate trade-offs

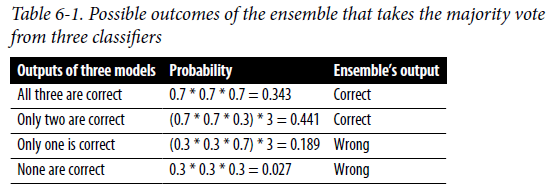
* There are **many trade-offs you have to make when selecting models**.
* **Understanding what’s more important in the performance of your ML system will help you choose the most suitable model**
* Classic example of trade-off = **FP’s vs. FN’s trade-off**
* *Reducing the number of FP’s might increase the number of FN’s, and vice versa*
* In a task where FP’s are more dangerous than FN’s, like fingerprint unlocking (unauthorized people shouldn’t be classified as authorized + given access), you might prefer a model that makes fewer FP’s
* Similarly, in a task where FN’s are more dangerous than FP’s, 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: **Compute requirement vs. accuracy**
* A more complex model might deliver higher accuracy but require a more powerful machine, such as a GPU instead of a CPU, to generate predictions with acceptable inference latency
* Many also care about the **interpretability vs. performance trade-**off
* A more *complex model can give a better performance, but its results are less interpretable*

###### 6) Understand your model’s assumptions

* George Box, 1976: “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 evaluate which model works best for a use case**
* Some common assumptions:
* **Prediction assumption =** Every model that aims to predict an output Y from an input Xmakes the **assumption that it’s even *possible* to predict Y based on X**
* **IID =** NN’s assume that its examples are **independent and identically distributed** = that **all the examples are independently drawn from the *same* joint distribution**
* **Smoothness =** Every supervised ML 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: it’s tractable to compute 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

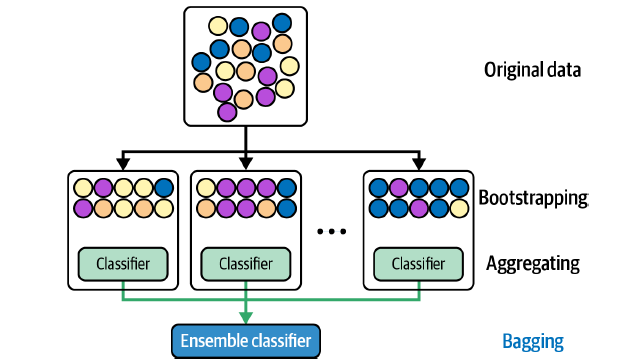
* When considering an ML solution to your problem, you might want to start with a system that contains just *one* model
* After developing one single model, you might think about how to continue improving its performance
* One method that has **consistently given a performance boost is to use an ensemble of multiple models** instead of just an individual model to make predictions
* **Each model in the ensemble is called a base learner**
* Ex: For predicting whether an email is SPAM or NOT SPAM, you might have 3 different models, and the final prediction for each email is the majority vote of all 3 models
* So, if at least 2 base learners output SPAM, the email will be classified as SPAM
* **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 CTR for ads**
* Example to give you the intuition of why ensembling works: you have 3 email spam classifiers, each with an accuracy of 70%
* Assuming each classifier has an equal probability of making a correct prediction for each email, and that these 3 classifiers are *not* correlated, we’ll show that by taking the majority vote of these 3 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* 2 classifiers are correct
* The table below 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%



* **This calculation only holds if the classifiers in an ensemble are *uncorrelated***
* If all classifiers are **perfectly correlated** (all 3 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**
* Ex: Might create an ensemble that consists of 1 transformer model, 1 RNN, and 1 gradient-boosted tree
* There are 3 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

##### 1) Bagging

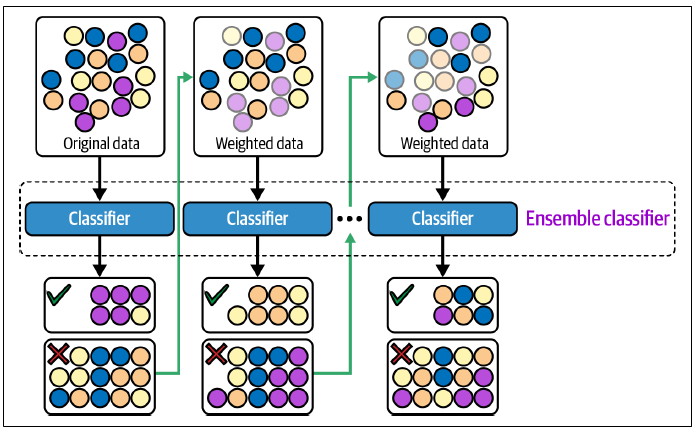
* **Bagging (bootstrap aggregating)** = **designed to improve both the training stability (less fluctuation in the training loss) and accuracy of ML algorithms**
* Bagging also **reduces variance and helps to avoid overfitting**
* Given a dataset, **instead of training 1 classifier on the entire dataset, sample with replacement to create *different* datasets** (called **bootstraps**) and **train a classification/regression model on each of these bootstraps**
* ***Sampling with replacement ensures each bootstrap is created independently from its peers***



* If the problem is **classification**, the **final prediction is decided by the majority vote** of all models
* Ex: 10 classifiers vote SPAM 6 models vote NOT SPAM = 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 NN’s, classification and regression trees, and subset selection in linear regression.
* However, it **can mildly degrade the performance of stable methods such as KNN**
* A **random forest** is an example of bagging = **a collection of decision trees constructed by *both* bagging + feature randomness, where each tree can pick only from a random subset of features to use**

##### 2) Boosting

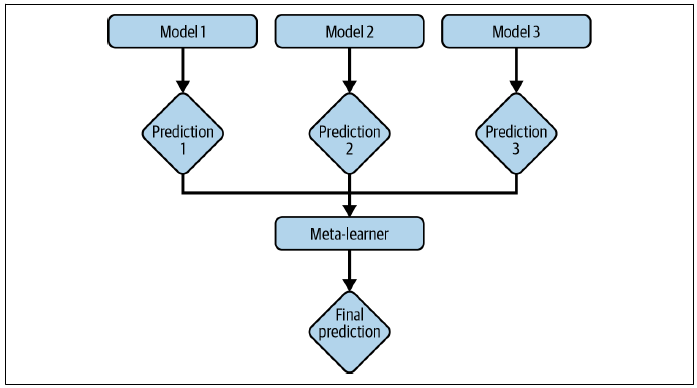
* **Boosting** = a **family of iterative ensemble algorithms that convert weak learners to strong ones**
* **Each learner in this ensemble is trained on the *same* set of samples, but the samples are weighted differently among iterations**
* As a result, ***future* weak learners focus more on examples that previous weak learners misclassified**
* Boosting steps
* **1. Train the first *weak* classifier on the original dataset.**
* **2. Samples are reweighted based on how well the 1st classifier classifies them (misclassified samples are given higher weight)**
* **3. Train the 2nd classifier on this *reweighted* dataset, + now the ensemble consists of the 1st *and* 2nd classifiers**
* **4. Samples are weighted based on how well the *ensemble* classifies them**
* **5. Train the *3rd* classifier on *this* reweighted dataset**, **+ add the 3rd classifier to the ensemble**
* ***6. Repeat for as many iterations as needed***
* **7. Form a 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 and been used in a wide range of tasks from classification, ranking, to the discovery of the Higgs Boson
* 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**

##### 3) Stacking

* **Stacking** = you **train base learners from the training data then create a meta-learner that *combines* the outputs of the base learners to output final prediction**



* The **meta-learner can be as simple as a heuristic** (take the majority vote (classification tasks) or the average vote (regression tasks) from all base learners), or **it can be another model, such as a logistic or linear regression model**
* For more great advice on how to create an ensemble: <https://github.com/MLWave/Kaggle-Ensemble-Guide>

#### 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 1 hyperparameter (such as one model using a learning rate of 0.003 and another model using a learning rate of 0.002), 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** = **files generated during an experiment (ex: 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 + results of an experiment is = 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 =** **versioning**
* **These two go hand in hand with each other** 🡪 Many tools originally set out to be experiment tracking tools (MLflow, Weights & Biases), have grown to incorporate versioning, + many tools originally set out to be versioning tools (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 training**, 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**
* Started in ML = track only loss and speed, fastforward several years = tracking so many things that experiment tracking boards look both beautiful and terrifying at the same time
* Short list of things you might want to consider tracking for each experiment during training:
* **Loss curve**corresponding *to the train split and each of the eval splits*
* **Model performance metrics**that you care about on *all* NON-test splits (ex: accuracy, F1, perplexity, etc.)
* The **log of corresponding sample, prediction, and ground truth label** 🡪 comes in handy for ad-hoc analytics and sanity check.
* **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**
* 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*
* **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 experiments is to automatically make copies of all the code files** **needed** for an experiment and **log all outputs with their timestamps**
* Using third-party experiment tracking tools, however, can give you nice dashboards + allow you to share your experiments with your coworkers.

##### Versioning

* Ex: 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 = 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**
* **1)** 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**
* Ex: **Code versioning is done by keeping track of all the changes made to a codebase**
* **A change is known as a** **diff** (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 data, especially if stored in a binary format, can be indefinitely long**
* Saying “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**
* **2)** 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**
* Ex: 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.
* **3)** If you use **user data** to train your model, regulations like GDPR **might make versioning this data complicated**
* Ex: 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** (Notable examples include atomic operations in CUDA where nondeterministic orders of operations lead to different floating point rounding errors between runs), 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, as the field progresses, we’ll hopefully gain more understanding into different models and can reason about what model will work best instead of running hundreds or thousands of experiments

##### Debugging ML Models

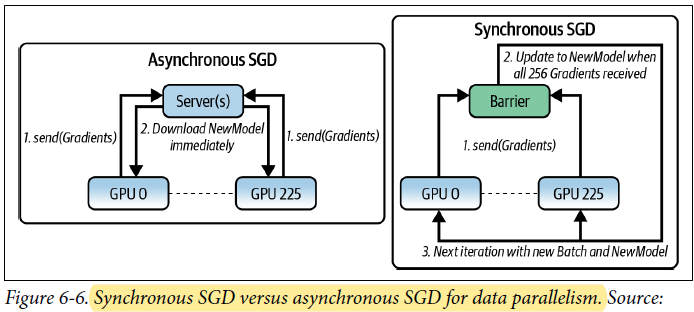
* **Debugging** = an inherent part of developing any piece of software, and ML models aren’t an exception
* Debugging ML models can be especially frustrating for the following 3 reasons
* **1) ML models fail silently**
* Code compiles, loss decreases as it should, the correct functions are called, predictions 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
* **2)** 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***
* **3)** 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**
* Ex: Data is managed by data engineers, labels by SME’s, ML algorithms by data scientists, and infrastructure by MLE’s 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**
* **Some of the things that might cause an ML model to fail:**
* **Theoretical constraints**
* 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
* Ex: you use a linear model for the data whose decision boundaries aren’t linear
* **Poor implementation of model**
* A model might be a good fit for the data, but the **bugs are in its implementation**
* Ex: In PyTorch, might’ve forgotten to stop gradient updates during evaluation
* **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 + 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 SOTA 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**
* Many things could go wrong in data collection + preprocessing that might cause models to perform poorly (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 models to overfit to training data or cause data leakage**
* **Too few features might lack predictive power to allow models to make good predictions**
* **Debugging should be both preventive *and* curative**
* You should **have healthy practices to minimize 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 + the debugging procedure is crucial in developing, implementing, + 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 MLE’s and researchers
* If interested in learning more, check out Andrej Karpathy’s post ”A Recipe for Training Neural Networks” (<https://karpathy.github.io/2019/04/25/recipe/>)
* The following are 3 tried-and-true debugging techniques
* **1) Start with the simplest model and gradually add more components to see if it helps or hurts the performance**
* Ex: To build an RNN, start with just 1 level of RNN cell before stacking multiple together, or adding more regularization
* Ex: If you want to use a BERT-like model, which uses both a **masked language model (MLM)** and **next sentence prediction (NSP)** loss, maybe use *only* the MLM loss before adding NSP loss
* Currently, many start out cloning an open-source implementation of a SOTA model + plugging in their own data
* On the off-chance that it works, great
* But if not, it’s very hard to debug the system because the problem could have been caused by any of the many components in the model
* **2) 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 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 the implementation**
* **3) 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** (no idea if 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** +also allows you to reproduce errors and other people to reproduce your results.

#### Distributed Training

* As **models are getting bigger + more resource-**intensive, companies care a lot more about **training at scale** (For products that serve a large number of users, you also have to care about **scalability in serving a model**)
* **Expertise in scalability is hard to acquire because it requires having regular access to massive compute resources**
* Scalability = a topic that merits a series of books, this section covers some notable issues to highlight challenges of doing ML at scale + provides a scaffold to help plan resources for a project accordingly
* **It’s common to train a model using data that doesn’t fit into memory** (especially when dealing with medical data such as CT scans or genome sequences, or with text data if you work for teams that train LLM’s (OpenAI, Google, NVIDIA, Cohere, etc.))
* **When data doesn’t fit into memory, preprocessing algorithms (e.g., zero-centering, normalizing, whitening), shuffling, and batching data will need to run out of core and in parallel** (Wikipedia: *“Out-of-core algorithms are algorithms designed to process data that are too large to fit into a computer’s main memory at once”* (s.v. “External memory algorithm”))
* **When a *sample* of your data is large (e.g. 1 machine can handle only 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 authors of the open source package gradientcheckpointing, “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”
* Even when a sample *does* fit 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

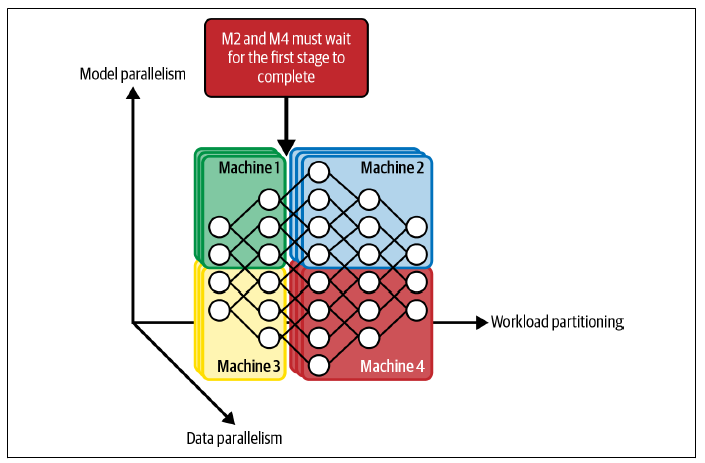
* It’s now the **norm to train ML models on multiple machines**
* The **most common parallelization method supported by modern ML frameworks is data parallelism: split your data on multiple machines, train your model on all of them, and accumulate gradients**
* ***This gives rise to a couple of issues***
* 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**
* 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**
* **If a model updates the weight using the gradient from each machine *separately* (a-synch 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**



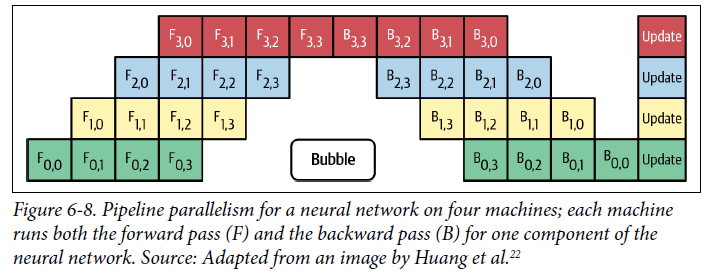
* **In theory, asynchronous SGD converges but requires more steps than synchronous SGD**
* **However, *in practice*, *when the number of weights is large*, gradient updates tend to be sparse, meaning most gradient updates only modify small fractions of the parameters, and it’s less likely that 2 gradient 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*
* *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 will process a batch size of 1M (OpenAI’s GPT-3 175B uses a batch size of 3.2M in 2020)
* 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**
* 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

* **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** **= when *different components of your model* are trained on different machine**



* Ex: Machine 0 handles the computation for the first 2 layers while machine 1 handles the next 2 layers, or some machines can handle the forward pass while several others handle the backward pass
* 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
* With **pipeline parallelism**, each mini-batch is broken into four micro-batches
* Machine 1 computes the 1st layer on the 1st micro-batch, then machine 2 computes the 2nd layer on machine 1’s results while machine 1 computes the 1st layer on the 2nd micro-batch, and so on
* Below shows what 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



* **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

* 2018: AutoML = 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?

##### Soft AutoML: Hyperparameter tuning

* **AutoML** = **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** = **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, 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 hyperparameters to tune
* **With different sets of hyperparameters, the same model can give drastically different performances on the *same* dataset**
* Melis et al. 2018: “On the State of the Art of Evaluation in Neural Language Models” = **weaker models with well-tuned hyperparameters can outperform stronger, fancier models**
* The **goal of hyperparameter tuning** = **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**
* Most popular is arguably graduate student descent (GSD) = a graduate student fiddles around with the hyperparameters until the model “works”
* However, **more and more people are adopting hyperparameter tuning as part of their standard pipelines**
* Popular ML frameworks either have built-in utilities or have 3rd-party utilities for hyperparameter tuning (scikit-learn with auto-sklearn, TensorFlow with Keras Tuner, Ray with Tune, etc.)
* **Popular methods for hyperparameter tuning include random search,** **grid search, and Bayesian optimization**
* A common practice = *start with coarse-to-fine random search, then experiment with Bayesian or grid search once the search space has been significantly reduced*
* See: the book AutoML: Methods, Systems, Challengesby the AutoML group at the University of Freiburg (<https://www.automl.org/wp-content/uploads/2019/05/AutoML_Book_Chapter1.pdf>)
* 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**
* **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 convolutionlayer 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, **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 3 components:
* **1) A search space**
* **Defines possible model architectures** (i.e., building blocks to choose from and constraints on how they can be combined)
* **2) 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
* **3) 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 RL (rewarding 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)**
* For NAS, the **search space is discrete** (the final architecture uses only one of the available options for each layer/operation, and **you have to provide the set of building blocks**
* 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”)
* The **common building blocks are various convolutions of different sizes, linear, various activations, pooling, identity, zero, etc.**
* The **set of building blocks varies based on the base architecture (e.g., convolutional NN’s or transformers)**
* In a **typical ML training process, you have a model + 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 NN’s today is gradient descent, which leverages an optimizer to specify how to update a model’s weights given gradient updates**
* **Popular optimizers are 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 NN?***
* **How much to update the model’s weights will be calculated by this NN**
* This approach **results in *learned* optimizers, as opposed to hand-designed optimizers**
* Since **learned optimizers are NN’s, they need to be trained**
* You can train your learned optimizer on the same dataset you’re training the rest of your NN 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 + existing designed optimizers as the learning rule) and use it for every new task after that**
* Ex: 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
* 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 **up-front training cost for AutoML is expensive enough that only a handful of companies 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 2 reasons**
* **1) 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**
* Ex: EfficientNets, a family of models produced by Google’s AutoML team, surpass SOTA accuracy with up to 10x better efficiency
* **2) They might be able to solve many real-world tasks previously impossible with existing architectures and optimizers**

#### Four Phases of ML Model Development

* Before we transition to model training, let’s take a look at the **4 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, + **solutions from one 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**
* A first stab at the problem can be the **simplest heuristics**
* Ex: To predict what letter users are going to type next in English, you can show the top 3 most common English letters, “e”, “t”, and “a”, which might get your accuracy to be 30%
* Ex: FB newsfeed was introduced in 2006 *without* any intelligent algorithms (posts were shown in chronological order)
* It wasn’t until 2011 that FB started displaying news updates you were most interested in at the top of the feed.
* Martin Zinkevich’s “Rules of Machine Learning: Best Practices for ML Engineering”: **“If you think that ML will give you a 100% boost, then a heuristic will get you 50% of the way there”**
* 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 *visibility into its workings* to allow you to validate the usefulness of your problem framing + your data**
* Logistic regression, gradient-boosted trees, KNN can be great for that
* They are also **easier to implement + deploy, which allows you to quickly build out a framework from data engineering to development to deployment that you can test out + gain confidence on**

##### Phase 3. Optimizing simple models

* Once you have an **ML framework in place**, you can **focus on optimizing simple ML models with different objective functions, hyperparameter search, feature engineering, more data, + ensembles**

##### Phase 4. Complex models

* Once you’ve **reached the limit of your simple models** and your **use case demands significant model improvement, then it’s time to experiment with more complex models**.
* You’ll **also want to run some experiments 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**

### Model Offline Evaluation

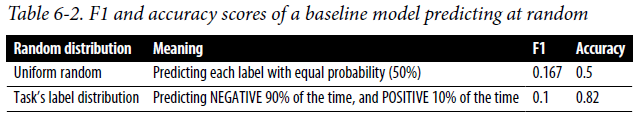
* One common but quite difficult question from companies concerning their ML strategies: **“How do I know that our ML models are any good?”**
* Ex: A company deployed ML to detect intrusions to 100 surveillance drones, but had no way of measuring how many intrusions their system failed to detect, + couldn’t 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**
* 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***
* Ex: 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**
* **Once your model is deployed, you’ll need to continue monitoring and testing your model in production**
* **There are methods to evaluate your model’s performance *before* it’s deployed**, such as the **baselines against which we will evaluate our models**, and some common evaluation **methods to evaluate your model *beyond* overall accuracy metrics**

#### Baselines

* Ex: Imagine someone tells you their new generative model achieved a FID score of 10.3 on ImageNet
* **FID = Frechet inception distance**, a common metric for measuring the quality of synthesized images
* The smaller the value, the higher the quality is supposed to be
* **Do you have any idea what this number meant or whether that model would be useful for your problem?**
* Ex: A company implements a classification model where the positive class appears 90% of the time
* An MLE on the team tells you me, all excited, that their initial model achieved an F1 score of 0.90
* Asked 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** (The accuracy, in this case, would be around 0.80.)
* *His model might as well be making predictions at random*
* **Evaluation metrics, by themselves, mean little**
* When evaluating your model, it’s **essential to know the baseline you’re evaluating it against**
* **Exact baselines should vary from one use case to another, but here are the 5 baselines that might be useful across use cases**:

##### 1) 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.
* Ex: Consider the task that has 2 labels, NEGATIVE that appears 90% of the time, and POSITIVE that appears 10% of the time
* The table below shows the F1 and accuracy scores of baseline models making predictions at random
* *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*



##### 2) Simple heuristic

* **Forget ML**. If you just make predictions based on simple heuristics, what performance would you expect?
* Ex: 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?

##### 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***
* Ex: 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
* Ex: For 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 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**
* Ex: 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
* Ex: 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 performance metrics
* However, **in production, we also want our models to be robust, fair, calibrated, and overall make sense**
* Here are some evaluation methods that help with measuring these characteristics of a model

##### Perturbation tests

* Ex: A team built 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 deployed 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, 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**
* **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**
* Other examples of noisy data include images with different lighting or texts with accidental typos or intentional text modifications such as typing “long” as “loooooong”)
* **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 user behaviors change just slightly** (i.e., they change phones), **your model’s performance might degrade**
* It **also** **makes your model susceptible to** **adversarial attack**

##### Invariance tests

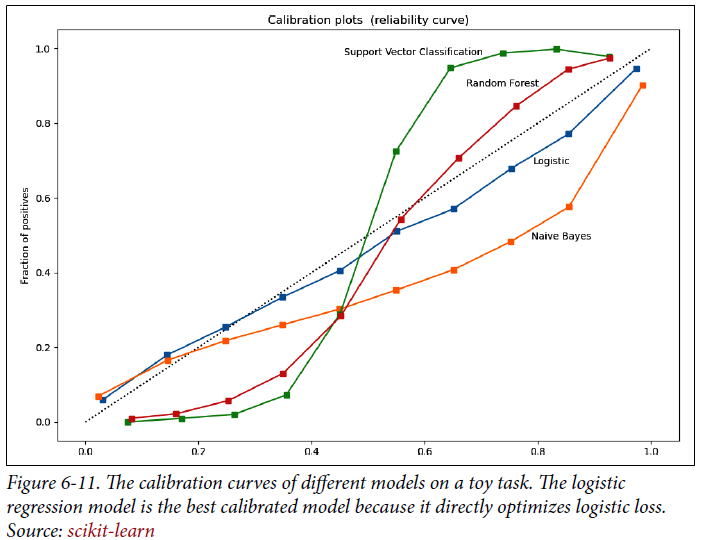
* **Certain changes to the inputs shouldn’t lead to changes in the output**
* Ex: Changes to race information shouldn’t affect the mortgage application outcome
* Similarly, changes to applicants’ names shouldn’t affect resume screening results nor should someone’s gender affect how much they should be 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 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** (It might also be mandated by law to exclude sensitive information from the model training process.)

##### Directional expectation tests

* **Certain changes to the inputs *should*, however, cause predictable changes in outputs**.
* Ex: When developing a model to predict housing prices, keeping all features the same but increasing lot size shouldn’t decrease the predicted price, and decreasing 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 team A will beat team B with a 70% probability, and out of 1,000 games, 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**
* 2 examples to show why model calibration is important
* 1) Consider the task of building a recommender system to recommend movies to users
* 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
* 2) 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 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 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 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 Xand the frequency Yof that prediction coming true, and plot X against Y**
* **The graph for a perfectly calibrated model will have X equal Yat all data points**
* In scikit-learn, you can plot the **calibration curve** of a binary classifier with the method `sklearn.calibration.calibration\_curve`:



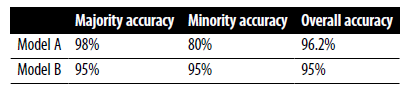
* **To calibrate your models, a common method is Platt scaling** (implemented in scikit-learn with sklearn.calibration.CalibratedClassifierCV)
* Another good open-source implementation by Geoff Pleiss can be found on GitHub
* <https://github.com/gpleiss/temperature_scaling>
* For readers who want to learn more about the importance of model calibration and how to calibrate NN’s, Lee Richardson and Taylor Pospisil have an excellent blog post based on their work at Google
* <https://www.unofficialgoogledatascience.com/2021/04/why-model-calibration-matters-and-how.html>

##### Confidence measurement

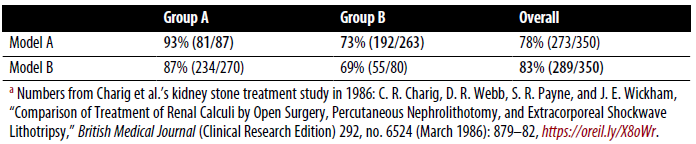
* **Confidence measurement can be considered a way to think about the usefulness threshold for each individual prediction**
* Indiscriminately showing all a model’s predictions to users, even the predictions that the model is unsure about, can, at best, cause annoyance and make users lose trust in the 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, ask for more information from users, etc.)?
* **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

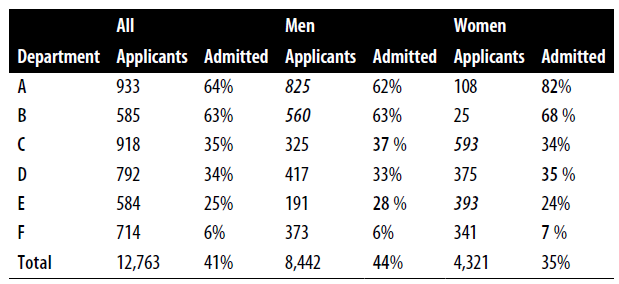
* **Slicing** **= separate your data into subsets and look at your model’s performance on each subset *separately***
* **A common mistake in companies = they are focused too much on coarse-grained metrics like overall F1 or accuracy on the *entire* data and *not* enough on sliced-based metrics**
* This can lead to 2 problems.
* **1) Their model performs differently on different slices of data when the model *should perform the same***
* Ex: Their data has 2 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, so its overall accuracy is 95%



* *Which model would you choose?*
* If a company focuses *only* on *overall* metrics, they might go with model A, and 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
* **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**
* Ex: 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**
* **2) Their model performs the same on different slices of data when the model should perform *differently***
* **Some subsets of data are more critical**
* Ex: For a model for user churn prediction, paid users = more critical than nonpaid users
* **Focusing on a model’s *overall* performance might hurt performance on these critical slices**
* A fascinating and seemingly counterintuitive reason why slice-based evaluation is crucial is **Simpson’s paradox** **= a trend appears in several groups of data but disappears or reverses when the groups are combined**
* This means 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 below
* *Model A outperforms model B for both group A and B, but when combined, model B outperforms model A.*



* Simpson’s paradox is more common than you’d think 🡪 1973 Berkeley graduate statistics showed 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 4/6 departments



* 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, + might also help reveal *non*-ML problems**
* Ex: A team discovered that their model performed great overall but very poorly on traffic from mobile users
* After investigating, 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 a boss or 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 (EDA)**
* 3 main approaches:
* **1) Heuristics-based**
* **Slice your data using domain knowledge** you have **of the data and the task at hand**
* Ex: 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
* **2) Error analysis**
* **Manually go through misclassified examples and find patterns among them**
* From the prior example, the team discovered their model’s problem with mobile users when they saw that most of the misclassified examples were from mobile users
* **3) 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” (2019) in and 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

* **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)**
* **For how to select ML models best suited for our tasks, instead of going into pros and cons of each individual model architecture** (a fool’s errand given the growing pools of existing models), we **outlined the** **aspects you need to consider to make an informed decision on which model is best for your objectives, constraints, and requirements**
* Different aspects of model development, such as individual models vs. ensembles of models
* 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 MLE’s 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**
* **As models today are getting bigger and consuming more data, distributed training is becoming an essential skill for ML model developers**, and there are techniques for parallelism including **data parallelism**, **model parallelism**, and **pipeline parallelism**
* **Making 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**
* **Then we think about 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 there are different types of baselines you might want to consider for evaluation
* There’s also 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**