Designing ML Systems – Chip Huyen

ML systems are complex and unique. They are complex because they consist of many different components (ML algorithms, data, business logic, evaluation metrics, underlying infrastructure, etc.) and involve many different stakeholders (data scientists, ML engineers, business leaders, users, even society at large). ML systems are unique because they are data dependent, and data varies wildly from one use case to the next. Two companies might be in the same domain (e-commerce) and have the same problem they want ML to solve (recommender system), but their resulting ML systems can have different model architecture, use different sets of features, be evaluated on different metrics, and bring different returns on investment.

It is not really possible to consider all these points in isolation. Changes in one component will likely affect other components. It is necessary to consider the system as a whole while attempting to make any design decision.

1. Overview of ML Systems

The algorithm is only a small part of an ML system in production. The system also includes the business requirements that gave birth to the ML project in the first place, the interface where users and developers interact with your system, the data stack, and the logic for developing, monitoring, and updating your models, as well as the infrastructure that enables the delivery of that logic.

When to Use ML

It is not a magic tool to solve all problems. Even for problems that ML can solve, ML solutions might not be the optimal solutions. Consider whether ML is necessary or cost-effective.

"Machine learning is an approach to learn complex patterns from existing data and use these patterns to make predictions on unseen data."

1. *Learn: the system has the capacity to learn.* For an ML system to learn, there must be something for it to learn from – usually data. In supervised learning, based on example input and output pairs, ML systems learn how to generate outputs from arbitrary inputs.
2. *Complex patterns: there are patterns to learn, and they are complex*. ML solutions are only really useful when there are patterns to learn. Even if there are patterns, your dataset might not be sufficient to capture them.
3. *Existing data: data is available, or it's possible to collect data.*
4. *Predictions: it's a predictive problem*. ML models make predictions, so they can only solve problems that require predictive answers.
5. *Unseen data*: *unseen data shares patterns with the training data.* The patterns your model learns from existing data are only useful if unseen data also share these patterns. Your unseen data and training data should come from similar distributions. We don't know the distribution of our unseen data, but we can make assumptions – such as we can assume that users' behaviours tomorrow won't be too different from

their behaviour today – and hope that our assumptions hold. If they don't we'll have a model that performs poorly.

ML Use Cases

Search engines, recommender systems, predictive typing, machine translation are famous use cases.

Mostly used for reducing costs, generating customer insights and intelligence, improving customer experience, predicting demand fluctuations, detecting fraud, internal processing automation, reducing customer churn, etc.

Understanding ML Systems

ML in Research vs in Production

	Research	Production
Requirements	State-of-the-art model performance on benchmark datasets.	Different stakeholders have different requirements.
Computational priority	Fast training, high throughput.	Fast inference, low latency.
Data	Static.	Constantly shifting.
Fairness	Often not a focus.	Must be considered.
Interpretability	Often not a focus.	Must be considered.

Different stakeholders and requirements

People involved in a research project usually have model performance as their objectives. To edge out a small improvement in performance, they resort to techniques that make models too complex to be useful. There are many stakeholders involved in bringing an ML system into production, each with their own, sometimes conflicting, requirements, which can make it difficult to design, develop, and select an ML model that satisfies all the requirements.

The sales team will want a mode that brings in the most revenue. The product team might want a model with minimum latency. ML engineers may want a more complex model with more data. "Recommend the restaurants that users are most likely to click on" and "recommending the restaurants that will bring in the most money for the app" are two different objectives. We need a system that satisfies different objectives. When developing an ML project, it's important for ML engineers to understand requirements from all stakeholders involved and how strict these requirements are.

Computational Priorities

People often spend too much time on the model development part and not enough on the model deployment and maintenance part. During model development, training is the bottleneck. Once the model has been deployed, its job is to generate predictions, so inference is the bottleneck. Research usually prioritizes faster training, whereas production usually prioritizes fast inference.

A corollary of this is that research prioritizes high throughput whereas production prioritizes low latency. Latency refers to the time it takes from receiving a query to return the result. Throughput refers to how many queries are processed within a specific period of time. If your system always processes one query at a time, higher latency means lower throughput. If the average latency is 10ms, which means it takes 10ms to process a query, the throughput is 100 queries/second. If the average latency is 100ms, the throughput is 10 queries/second. However, because most modern distributed systems batch queries to process them together, higher latency might also mean higher throughput. If you process 10 queries at a time and it takes 10ms to run a batch, the average latency is still 10ms, but the throughput is now 10x higher – 1000 queries/second.

Once you deploy your model into the real world, latency matters a lot. A study found that a 100ms delay can hurt conversion rates by 7%. To reduce latency in production, you might have to reduce the number of queries you can process on the same hardware at a time. When thinking about latency, it is important to keep in mind that it is not an individual number but a distribution. It is tempting to reduce this distribution by using a single number, like the average latency of all requests within a time window, but this can be misleading (if using the mean, this is susceptible to outliers, so the average latency doesn't reflect the length of time most queries take). It is usually better to think in percentiles. The most common is the median – if the median is 100ms, half of the requests take longer, half take less. Higher percentiles also help you discover outliers – say look at the 90th percentile and see what values are above. Higher percentiles are important because even though they account for a small percentage of your users, they can be the most important ones. On Amazon, the customers with the slowest requests are often those with the most data on their accounts because they made many purchases – they're the most valuable customers! It is common practice to use high percentiles to specify the performance requirements for your system; a product manager might specify that the 90th percentile latency of a system must be below a certain number.

Data

During the research phase, this is clean and well-formatted so you can focus on the models. Data is static so that the community can use them to benchmark new architectures and techniques. In production, data is a lot more messy. It's noisy, possibly unstructured, and constantly shifting. It's likely biased. Labels might be sparse, incorrect, or imbalanced. If you work with users' data, you have to worry about privacy and regulatory concerns.

Fairness

In research, since a model is not yet used on people, fairness can be an afterthought. There are metrics for performance, but not for fairness. People are often victims of algorithms, which reject loan applications, CVs, set mortgage rates based on biases. ML algorithms don't predict the future, but encode the past, thus perpetuating the biases in the data. When ML algorithms are deployed at scale, they can discriminate against people at scale. This can especially hurt minority groups since misclassification on them could only have a minor effect on models' overall performance metrics.

Interpretability

Again, most ML research focuses on performance, not interpretability. In industry, however, this isn't optional. It is important for users and business leaders to understand why a decision is made so they can trust the model and detect biases. It is also important in order to debug and improve a model.

2. Introduction to ML Systems Design

ML systems design takes a system approach to MLOps, which means the ML system will be considered holistically to ensure that all the components – the business requirements, the data stack, infrastructure, deployment, monitoring, etc. – and their stakeholders can work together to satisfy the specified objectives and requirements.

Data scientists tend to care more about the ML objectives: the metrics they can measure about the performance of their models. However they need to remember the metrics important to the business – maximizing profits for shareholders, whether directly through increasing sales/conversion and cutting costs, or indirectly such as higher customer satisfaction and increasing time spent on a website.

For an ML project to succeed within a business, the two objectives need to be tied together. Specifically, what business performance metrics is the new ML system supposed to influence? Business objectives -> ML objectives, in order to guide the development of ML models.

To gain a definite answer on the question of how ML metrics influence business metrics, experiments are needed. This can be done with A/B testing, choosing the model that leads to better business metrics, regardless of whether this model has better ML metrics.

Success won't happen overnight. ROI in ML depend a lot on the maturity stage of adoption. The longer you've adopted it, the more efficient your pipeline will run, the faster your development cycle will be, the less engineering time you'll need, the lower your cloud bills will be, which all lead to higher returns.

Requirements for ML Systems

- *Reliability.* The system should continue to perform the correct function at the desired level of performance even in the face of adversity (hardware or software faults, human error).
- *Scalability.* Whichever way your system grows, there should be reasonable ways of dealing with that growth. Mostly resource scaling, consisting of up-scaling (expanding the resources to handle growth) and down-scaling (reducing the resources when not needed). There is also artifact management – managing the several models that are running.
- *Maintainability.* Lots of different teams working on this. Important to structure your workloads and set up your infrastructure so that different contributors can work using tools they are comfortable with. Code should be documented. Code, data, and artifacts should be versioned. Models should be reproducible.

- *Adaptability.* To adapt to shifting data distributions and business requirements.

Iterative Process

One workflow you may encounter when building an ML model to predict whether and should be shown when users enter a search query:

1. Choose a metric to optimize, e.g. impressions – the number of times an ad is shown.
2. Collect data and obtain labels.
3. Engineer features.
4. Train models.
5. During error analysis, you might notice errors caused by wrong labels, so you relabel the data.
6. Train the model again.
7. During error analysis, you realize your model always predicts that an ad shouldn't be shown, because 99.99% of your data have negative labels. So you have to collect more data of ads that should be shown.
8. Train the model again.
9. The model performs well on existing test data, which is now 2 months old. It performed poorly on data from yesterday. Your model is now stale, so you need to update it on more recent data.
10. Train the model again.
11. Deploy the model.
12. Model is performing well, but businesspeople ask why revenue is decreasing. The ads are being shown, but few people click on them. So you need to change your model to optimize for ad click-through rate instead.
13. Go to step 1.

General pipeline:

1. *Project scoping.* Laying out goals, objectives and constraints. Stakeholders should be identified and involved. Resources should be estimated and allocated.
2. *Data engineering.* Handling data from different sources and formats, sampling and generating labels.
3. *ML model development.* Extract features and develop initial models leveraging these features.
4. *Deployment.* The model needs to be made accessible to users.
5. *Monitoring and continual learning.* Models need to be monitored for performance decay and maintained to be adaptive to changing environments and changing requirements.
6. *Business analysis.* Model performance needs to be evaluated against business goals and analyzed to generate business insights. These insights can be used to eliminate unproductive projects or scope out new projects.

Framing ML Problems

An ML problem is defined by inputs, outputs, and the objective function that guides the learning process, so not everything is an ML problem.

Objective Functions

To learn, an ML model needs an objective function to guide the learning process. It is also called a loss function, because the objective of the learning process is usually to minimize (or optimize) the loss caused by wrong predictions. For supervised ML, this loss can be computed by comparing the model's outputs with the ground truth labels using a measurement like RMSE or cross entropy.

Decoupling Objectives

Framing ML problems can be tricky when you want to minimize multiple objective functions. Sometimes, objectives are in conflict with each other. One approach is to try to combine two losses into one loss and train a model to minimize that loss. E.g. loss = α quality_loss + β engagement_loss. You can randomly test out different values of α and β to find the values that work best. To be more systematic, check out Pareto optimization, which is concerned with optimizing more than one objective function simultaneously. A problem with this approach is that each time you tune one of the parameters, you'll have to retrain your model.

Another approach is to train two different models, each optimizing one loss. You can combine the models' outputs and check combined scored. Now you can tweak the parameters α and β without retraining your models. In general, it's a good idea to decouple objectives first because it makes model development and maintenance easier.

3. Data Engineering Principles

Data Sources

An ML system can work with data from different sources. They have different characteristics, can be used for different purposes, and require different processing methods. Understanding the sources your data comes from can help you use your data more efficiently.

One source is *user input data*, data explicitly input by users. This can be text, uploaded files, images, etc. Users often input wrong information, so this data can be easily malformatted. E.g. where numerical values are expected, users might accidentally enter text; they may upload an incorrect file format, etc. This data requires heavy-duty checking and processing.

Another source is *system-generated data*. This is data generated by different components of your systems, which include various types of logs and system outputs such as model predictions, and user behaviours such as clicking, scrolling, zooming, etc. Logs can record the state and significant events of the system, such as memory usage, number of instances, packages used, etc. They provide visibility into how the system is doing. The main purpose of this is for debugging and potentially improving the application. Only really used when the

system has a problem. Since they are system generated, they are far less likely to be malformatted. Whereas user input data may need to be processed as soon as they arrive, system generated data can be processed less periodically, such as hourly or daily.

There are also *internal databases*, generated by various services and enterprise applications in a company. These manage their assets such as inventory, customer relationship, users, and more. This data can be used by ML models directly.

There is also *third-party data*. First-party data is that which your company collects about your users or customers. Second-party data is that which is collected by another company on their own users that they make available to you. Third-party data companies collect data on the public who aren't their direct customers.

Data Formats

Data can be stored in different formats. To retrieve stored data, it's important to know not only how it's formatted but also how it's structured. It is also important to think about how the data will be used in the future so that the format you use will make sense. How can you store your data so that it's cheap and still fast to access?

The process of converting a data structure or object state into a format that can be stored or transmitted and reconstructed later is *data serialization*. There are many, many data serialization formats. When considering a format to work with, you might want to consider different characteristics such as human readability, access patterns, and whether it's based on text or binary, which influences the size of its files.

Format	Binary/Text	Human-readable	Example use cases
JSON	Text	Yes	Everywhere
CSV	Text	Yes	Everywhere
Parquet	Binary	No	Databricks, Hadoop
Pickle	Binary	No	Python, PyTorch

JSON (JavaScript Object Notation) was derived from JavaScript, but it's language-agnostic – most modern programming language can generate and parse JSON. It's human-readable. Its key-value pair paradigm is simple but powerful, capable of handling data of different levels of structuredness. It can be stored like this:

```
{
  "firstName": "Boatie",
  "lastName": "McBoatFace",
  "isVibing": true,
  "age": 12,
  "address": {
    "streetAddress": "12 Ocean Drive",
    "city": "Port Royal",
    "postalCode": "10021-3100"
  }
}
```

But equally like this:

```
{
  "text": "Boatie McBoatFace, aged 12, is vibing, at 12 Ocean Drive, Port Royal,
           10021-3100"
}
```

JSON are text files, so they take up a lot of space.

Row-Major Versus Column-Major Format

Two common formats which represent two distinct paradigms are CSV and Parquet. CSV (comma-separated values) are row-major, which means consecutive elements in a row are stored next to each other in memory. Parquet is column-major, which means consecutive elements in a column are stored next to each other.

Computers process sequential data more efficiently than non-sequential data, so if a table is row-major, accessing its rows will be faster than its columns in expectation. Hence, for row-major formats, accessing data by rows is faster than doing so by columns. So, CSV files are better for accessing examples (e.g. all the examples collected today), and Parquet files are better for accessing features (e.g. all the timestamps of all your examples).

Column-major formats allow flexible column-based reads. Say you have 1000 features but only want to access 4. With column-major formats, you can read those four columns directly. However with row-major formats, if you don't know the size of the rows, you have to read in all the columns, then filter down to these four columns. Even if you know the size of the rows, it can be slow as you'll have to jump around the memory. Row-major formats allow faster data writes, such as adding new examples to your data. So: row-major formats are better when you have to do a lot of writes, whereas column-major formats are better when you have to do a lot of column-based reads.

Text Vs Binary Format

Parquet files are binary files. Text files are human-readable, binary files are the catchall that refers to all non-text files. Binary files are typically files that contain only 0s and 1s. They are more compact. An example CSV file for 17,654 rows and 10 columns are 14MB.In binary format (Parquet), the file is 6MB. The Parquet format is up to 2x faster to unload and consumes up to 6x less storage, compared to text formats.

Data Models

Data models define how the data stored in a particular format is structured. In a database, a car can be described using its make, its model, its year, its color, its price, etc. These attributes make up a data model for cars. Alternatively, you can also describe a car using its owner, its license plate, and its history of registered addresses. This is another data model for cars.

How you choose to represent data not only affects the way your systems are built, but also the problems your systems can solve. For example, the first data model above makes it

easier for people looking to buy cars, whereas the second data model makes it easier for police officers to track down criminals.

Relational Model

In this model, data is organized into relations; each relation is a set of tuples. A table is an accepted visual representation of a relation, and each row of a table makes up a tuple. Relations are unordered; you can shuffle the order of the rows or the columns in a relation and it's still the same relation. Data following the relational model is usually stored in file formats like CSV or Parquet.

It is often desirable for relations to be normalized. For example, in a 'Book' relation table, there may be lots of duplicates, differing only in format (paperback, hardback, e-book) and price. If the publisher information changes, we'd have to update multiple rows. However, if we separate publisher information to its own table, when this information changes we only have to update the "Publisher" relation. The "Book" relation would now have a "Publisher ID" column with an ID denoting the publisher, and this ID would be found in the "Publisher" relation. So, normalization makes it easier to make changes to values.

Databases built around the relational data model are relational databases. The language used to retrieve/specify the data you want from a database is called a *query language*. SQL is the most popular of these. The most important thing to note about SQL is that it's a declarative language, as opposed to Python, which is an imperative language. In this latter paradigm, you specify the steps needed for an action and the computer executes these steps to return the outputs. In the former, you specify the outputs you want, and the computer figures out the steps needed to get you the queried outputs.

NoSQL

For certain cases, the relational data model can be restrictive. It demands that your data follows a strict schema, and schema management is painful. It can also be difficult to write and execute SQL queries for specialized applications. NoSQL (Not Only SQL) is a movement against relational databases. Two major types of non-relational models are the document model and the graph model.

Document Model

This targets use cases where data comes in self-contained documents and relationships between one document and another are rare. It is built around the concept of "document". A document is often a single continuous string, encoded as JSON, XML, or a binary format like BSON (Binary JSON). All documents in a document database are assumed to be encoded in the same format. Each document has a unique key that represents that document, which can be used to retrieve it.

A collection of documents can be considered analogous to a table in a relational database, and a document analogous to a row. However, a collection of documents is more flexible than a table. All rows in a table must follow the same schema (have the same sequence of columns), while documents in the same collection can have completely different schemas.

The application that reads the document assumes some kind of structure of the documents. Document databases just shift the responsibility of assuming structures from the application that writes the data to the application that reads the data. Information is easier to retrieve in this way, as you don't have to query multiple tables. However, it's harder to execute joins across documents compared to across tables.

Graph Model

This targets use cases where relationships between data items are common and important. It is built around the concept of a "graph". A graph consists of nodes and edges, where edges represent the relationships between the nodes. If in document databases, the content of each document is the priority, then in graph databases, the relationships between data items are the priority. Thus, it's faster to retrieve data based on relationships.

Many queries that are easy to do in one data model are harder to do in another. Picking the right data model for your application can make your life easier.

Structured vs Unstructured Data

Structured data follows a predefined data model, aka a data schema. For example, the data model might specify that each data item consists of two values: the first, "name" is a string of at most 50 characters, the second, "age", is an 8-bit integer in the range between 0 and 200. The predefined structure makes your data easier to analyze. The disadvantage of structured data is that you have to commit your data to a predefined schema. If your schema changes, you'll have to retrospectively update all your data (e.g. adding e-mail addresses to all users).

Because business requirements change over time, committing to a predefined data schema can become too restricting. Or you might have data from multiple sources that are beyond your control, so it is impossible to make them follow the same schema. Unstructured data thus becomes appealing; it doesn't adhere to a predefined data schema. It is usually text, but can also be numbers, dates, images, audio, etc. Unstructured data also allows for more flexible data storage. You can convert all your data, regardless of types and formats, into bytestrings and store them together.

A repository for storing structure data is called a *data warehouse*. A repository for storing unstructured data is called a *data lake*. The latter are usually used to store raw data before processing, whereas the former are used to store data that has been processed into formats ready to be used.

Data Storage Engines and Processing

Typically, there are two types of workloads that databases are optimized for, transactional processing and analytical processing.

Transactional and Analytical Processing

In the digital world, a transaction refers to any kind of action: tweeting, ordering a ride through a ride-sharing service, uploading a new model, etc. Even though these transactions

involve different types of data, the way they're processed is similar across applications. The transactions are inserted as they are generated, and occasionally updated when something changes, or deleted when they are no longer needed. This type of processing is called *online transaction processing (OLTP)*.

Because these transactions often involve users, they need to be processed fast (low latency). The processing method needs to have high availability – it needs to be available any time a user wants to make a transaction. Transactional databases are designed for these requirements.

- *Atomicity*. To guarantee that all the steps in a transaction are completed successfully as a group. If any step in the transaction fails, all others steps will too. E.g. if a user's payment fails, you don't want to assign a driver to that user.
- *Consistency*. All transactions must follow predefined rules. E.g. a transaction must be made by a valid user.
- *Isolation*. To guarantee that two transactions happen at the same time as if they were isolated. E.g. you don't want two users to book the same driver at the same time.
- *Durability*. To guarantee that once a transaction has been committed, it will remain so even in the case of a system failure. E.g. if you've ordered a ride and your phone dies, you still want your ride to come.

Because each transaction is often processed as a unit separately from other transactions, transactional databases are often row-major. So, they might not be efficient for questions like "What's the average price for all rides in September in New York?" This kind of analytical question requires aggregating data in columns across multiple rows of data. Analytical databases are designed for this purpose – they type of processing is called *online analytical processing (OLAP).*

These terms appear to be outdated as there are databases that can handle both transactional and analytical queries.

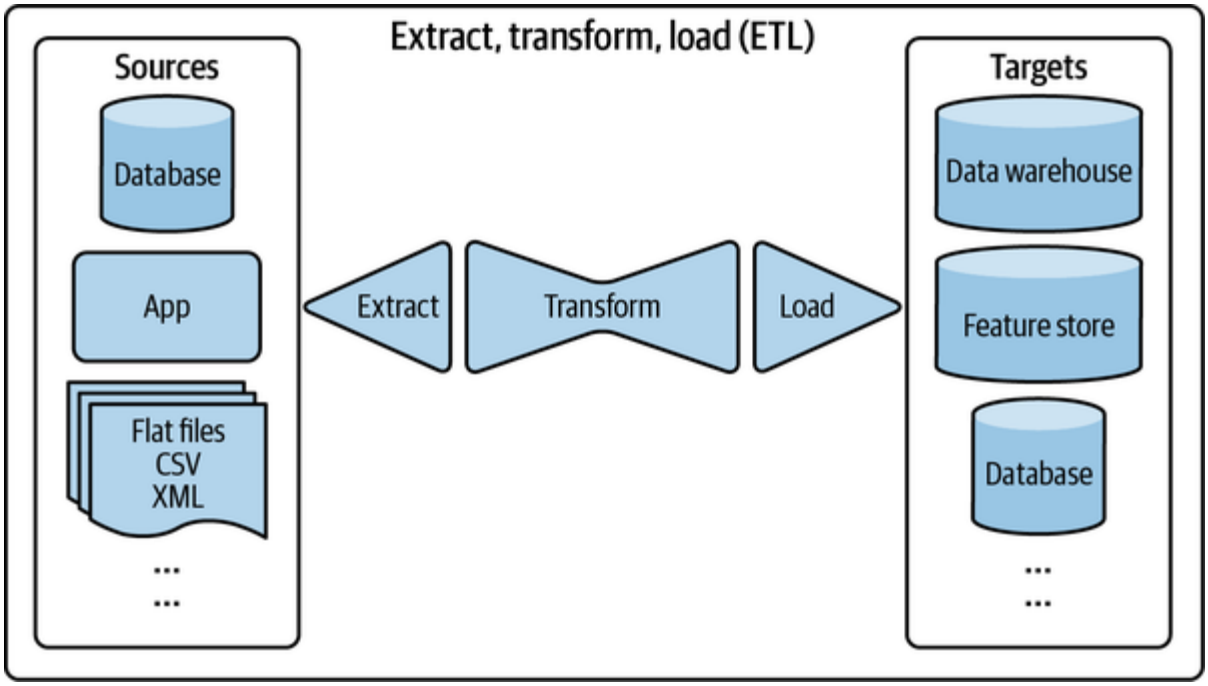
ETL: Extract, Transform, and Load

ETL refers to the general purpose processing and aggregating of data into the shape and format that you want.

Extract is extracting the data you want from your data sources. Some of them will be corrupted or malformatted. In this phase, you need to validate your data and reject the data that doesn't meet your requirements. This can save you a lot of time downstream.

Transform is where most of the data processing is done. You might want to join data from multiple sources and clean it. You might want to standardize, dedupe, sort, aggregate, derive new features, etc.

Load is deciding how and how often to load your transformed data into the target destination, which can be a file, a database, or a date warehouse.



As companies weight the pros and cons of storing structured data vs storing unstructured data, vendors evolve to offer hybrid solutions that combine the flexibility of data lakes and the data management aspect of data warehouses. Databricks and Snowflake both provide data lakehouse solutions.

Modes of Dataflow

In production, you often have multiple processes. How do we pass data between different processes that don't share memory? When data is passed from one process to another, we say that data flows from one process to another, which gives us a dataflow. There are three main modes:

- *Data passing through databases.* The easiest way. To pass data from process A to process B, process A can write that data into a database and process B can read from that database. This mode doesn't always work. Firstly, this requires that both processes be able to access the same database, which can be unfeasible if the processes are run by two different companies. Second, read/write from databases can be slow, making it unfeasible for applications with strict latency requirements (any consumer-facing application).
- *Data passing through services.* Send data directly through a network that connects these two processes. Process A can send a request to process B specifying the data it needs and B returns the requested data through the same network. This is request-driven. Two services in communication with each other can be run by different companies in different applications. They can also be parts of the same application.
- *Data passing through real-time transport.*

Batch Processing vs Stream Processing

Once your data arrives in data storage engines like databases, data lakes, or data warehouses, it becomes historical data. This is opposed to streaming data (data that is still steaming in). Historical data is often processed in batch jobs – jobs that are kicked off periodically. This is known as *batch processing*. *Stream processing* refers to doing computation on streaming data. Computation on streaming data can also be kicked off periodically, but the periods are much shorter than for batch jobs (e.g. every five minutes instead of every day).

Because batch processing happens much less frequently, it is usually done to compute features that change less often, such as drivers' ratings. Batch features – features extracted through batch processing – are also known as *static features*. Stream processing is used to compute features that change quickly, such as how many drivers are available right now. Features about the current state of the system like this are important to make the optimal price predictions. Streaming features are also known as *dynamic features*.

For many problems, you need both types of features. You need infrastructure that allows you to process streaming data as well as batch data and join them together to feed into your ML models. To do computation on data streams, you need a stream computation engine (the way Spark is a batch computation engine). The number of stream features used in an application such as fraud detection can be in the thousands. The stream feature extraction logic can require complex queries with join and aggregation along different dimensions. To extract these features requires efficient processing engines.

4. Training Data

Data is messy, complex, and possibly treacherous. If not handled properly, it can sink your entire ML operation. Training data encompasses all the data used in the developing phase of ML models, including the different splits used for training, validation, and testing.

Sampling

An integral part of the ML workflow. One samples from all possible real-world data to create training data; one samples from a dataset to create splits for training and testing; one even samples from all possible events that happen within your ML system for monitoring purposes.

Sampling is obviously necessary – we do not have access to all possible data in the world, so our subset of data is created by one sampling method or another. Another case is when it's infeasible to process all the data you have access to – it requires too much time or resources – so you sample the data to create a subset that is feasible to process. Lastly, it is also done because you can accomplish a task faster. When considering a new model, you do a quick experiment with a small subset of data to see if the new model is promising first before training the model on all your data.

Understanding different sampling methods and how to use them in our workflow can help us avoid potential sampling biases and also choose the methods that improve the efficiency of the data we sample.

Simple Random Sampling

Here, you give all samples in the population equal probabilities of being selected. E.g. if you randomly select 10% of the population, everyone has a 10% chance of being selected. This method is easy to implement. However, rare categories of data might not appear in your selection. Models trained on majority categories will perform poorly when they encounter a rare class not seen during training.

Stratified Sampling

To avoid this drawback, you can first divide your population into the groups you care about and sample from each group separately. This way, no matter how rare the minority class is, you'll ensure that samples from it will be included in the selection. Each group is called a stratum, hence the name stratified sampling. A drawback here is that it's not always possible to divide samples into groups. Alternatively, one sample might belong to multiple groups (in multilabel tasks).

Weighted Sampling

Each sample is given a weight, which determines the probability of it being selected. E.g. classes A, B, and C, might have probabilities of being selected at 0.5, 0.3, and 0.2. This method can leverage domain expertise. More recent data might be more valuable, hence you'd want it to have a higher chance of being selected. This also helps when the data you have comes from a different distribution compared to the true data. E.g. if in your data, red samples account for 25% and blue samples for 75%, but you know in the real world red and blue have equal probabilities to happen, you can give red samples weights three times higher than blue samples.

Reservoir Sampling

Especially useful when dealing with streaming data, which is usually what you have in production. Imagine you have an incoming stream of tweets and you want to sample a certain number, k, of tweets to do analysis or train a model on. You don't know how many tweets there are, but you know you can't fit them all in memory, which means you don't know in advance the probability at which a tweet should be selected. You want to ensure that each tweet has an equal probability of being selected, and that you can stop the algorithm at any time with the tweets sampled with the correct probability.

Importance Sampling

A very useful method. It allows us to sample from a distribution when we only have access to another distribution. If you have to sample *x* from a distribution *P(x),* but this is really expensive or infeasible to sample from. However, you have a distribution *Q(x)* that is a lot easier to sample from. You do this and weigh this sample by $\frac{P(x)}{Q(x)}$. Q(x) is the proposal

distribution or the importance distribution. It can be any distribution as long as $Q(x) > 0$ whenever $P(x) \neq 0$. This might be used in reinforcement learning.

Labelling

Data Lineage

Indiscriminately using data from multiple sources without examining their quality can cause your model to fail mysteriously. It's good practice to keep track of the origin of each of your data samples as well as its labels. This is known as *data lineage.* It helps you flag both potential biases in your data and debug your models.

Handling the lack of labels

Method	How	Ground truths required?
Weak supervision	Leverages (often noisy) heuristics to generate labels	No, but a small number of labels are recommended to guide the development of heuristics
Semi-supervision	Leverages structural assumptions to generate labels	Yes, a small number of initial labels as seeds to generate more labels
Transfer learning	Leverages models pretrained on another task for your new task	No for zero-shot learning Yes for fine-tuning, though the number of ground truths required is often much smaller than what would be needed if you train the model from scratch
Active learning	Labels data samples that are most useful to your model	Yes

Weak Supervision

The insight here is that people rely on heuristics, which can be developed with subject matter expertise, to label data. E.g. a doctor might use the following to determine whether a patient's case should be prioritized as emergent: "If the nurse's note mentions a serious condition, the patient's case should be given priority consideration."

A labelling function can encode these heuristics. Can be used with a keyword, such as "pneumonia", regular expressions or a database lookup (if a disease is found in a list of diseases). After you've written LFs, you can apply them to the samples you want to label.

Because LFs encode heuristics, and heuristics are noisy, labels produced by LFs are noisy. Weak supervision can be useful when your data has strict privacy requirements. You only need a small, cleared subset of data to write LFs, which can be applied to the rest of your data without anyone looking at it. With LFs, subject matter expertise can be versioned, reused, and shared.

Semi-supervision

This leverages structural assumptions to generate new labels based on a small set of initial labels. A classic method is self-training. You start by training a model on your existed set of labelled data and use this model to make predictions for unlabeled samples. Thus you can expand your training set and repeat.

Another method assumes that data samples that share similar characteristics share the same levels. For example, grouping Twitter hashtags together, as they should be about the same topic. Other ways include a clustering method or k-nearest neighbours.

Semi-supervision is most useful when the number of training labels is limited. How much of this limited data should be used to evaluate multiple candidate models to select the best one? Some companies use a large evaluation set to select the best, then continue training the champion model on the evaluation set.

Transfer Learning

This refers to a family of methods where a model developed for a task is reused as the starting point for a model on a second task. First, the base model is trained for a base task, which is usually a task that has cheap and abundant training data. The trained model can then be used for the task you're interested in. Sometimes you can use the base model directly, but often you will need to fine-tune the base model.

Transfer learning is appealing for tasks that don't have a lot of labelled data. Even for tasks with lots of labelled data, using a pretrained model as a starting point can often boost the performance significantly compared to training from scratch. A nontrivial portion of ML models in production today are the results of transfer learning, like BERT or GPT-3. Transfer learning also lowers the entry barriers into ML, as it helps reduce the up-front cost needed for labelling data to build ML applications.

Active Learning

A method for improving the efficiency of data labels. The hope here is that ML models can achieve greater accuracy with fewer training labels if they can choose with data samples to learn from.

Instead of randomly labelling data samples, you label the samples that are most helpful to your models according to some metrics or heuristics. The most straight-forward metrics is uncertainty measurement – label the examples that your model is the least certain about, hoping that they will help your model learn the decision boundary better.

Class Imbalance

Where there is a substantial difference in the number of samples in each class of the training data. It can also happen in regression tasks – consider the task of estimating health-care bills. These are highly skewed – the median bill is low, but the 95^{th} percentile bill is astronomical. When predicting hospital bills, it may be more important to predict the latter rather than the former. Therefore, we might have to train the model to be better at predicting the 95^{th} percentile bills, even if it reduces the overall metrics.

Challenges of Class Imbalance

The first reason is that class imbalance often means there's insufficient signal for your model to learn to detect the minority classes. The second reason is that it is easier for your model to get stuck in a nonoptimal solution by exploiting a simple heuristic instead of learning anything useful about the underlying pattern of the data. The third reason is that it leads to asymmetric costs of error – the cost of a wrong prediction on a sample of the rare class might be much higher than a wrong prediction on a sample of the majority class.

Class imbalance is the norm. In real-world settings, rare events are often more interesting (or dangerous) than regular events. The classical example is fraud detection. Most credit card transactions are not fraudulent. 7 cents of every $100 is fraudulent. Another is churn prediction. The majority of your customers are probably not planning on cancelling their subscription, but your business has more to worry about the ones who will.

Handling Class Imbalance

Sensitivity to imbalance increases with the complexity of the problem, and noncomplex, linearly separable problems are unaffected by all levels of class imbalance. Class imbalance in binary classification problems is a much easier problem than in a multiclass classification problem.

A good model should learn to model that imbalance. However, developing a model good enough for that can be challenging, so we still have to rely on special training techniques. Three approaches:

Using the right evaluation metrics. Wrong metrics will give you the wrong idea of how your models are doing and thus won't help you develop or choose models good enough for your task. Accuracy is insufficient here as the performance of your model on the majority class will dominate this metric. F1, precision, and recall are metrics that measure your model's performance with respect to the positive class in binary classification problems. They are asymmetric metrics, which means that their values change depending on which class is considered the positive class.

Many classification problems can be modelled as regression problems. Your model can output a probability and based on that probability, you classify the sample. This means you can tune the threshold to increase the *true positive rate (recall)* while decreasing the *false positive rate* and vice versa. We can plot these against each other for various different thresholds. This plot is known as the ROC curve (receiver operating characteristics). This curves shows you how your model's performance changes depending on the threshold and helps you choose the threshold that works best for you. The area under the curve (AUC) should be maximized.

Since the ROC curve focuses only on the positive class and doesn't show you how well your model does on the negative class, some suggest plotting precision against recall instead. This gives a more informative picture of an algorithm's performance on tasks with heavy class imbalance.

Data-level methods: Resampling. Data-level methods modify the distribution of the training data to reduce the level of imbalance to make it easier for the model to learn. Resampling is one technique. This includes oversampling, adding more instances from the minority classes, and undersampling, removing instances of the majority classes. The easiest way to do so is to randomly remove instances of the majority class. The easiest way to oversample is to randomly make copies of the minority class until you have a ratio you're happy with.

When you resample your training data, never evaluate your model on resampled data, since it will cause your model to overfit to that resampled distribution. Undersampling runs the risk of losing important data from removing data. Oversampling runs the risk of overfitting on training data. Many sampling techniques have been developed to mitigate this, two of which are two-phase learning and dynamic sampling.

Algorithm-level methods. These keep the training data distribution intact but alter the algorithm to make it more robust to class imbalance. Often this involves adjusting the loss function. By giving training instances we care about higher weight, we can make the model focus more on learning these instances.

Data Augmentation

A family of techniques used to increase the amount of training data. Useful even when we have a lot of data – they can make our models more robust to noise. It is a standard step in computer vision tasks. Three ways:

- *Simple level-preserving transformations.* Randomly modify an image while preserving its label, e.g. by cropping, flipping, rotating, inverting, erasing a part of the image, etc. A rotated image of a dog is still a dog. In NLP, you can randomly replace a word with a similar word, assuming that this wouldn't change the meaning or the sentiment of the sentence, i.e. using 'glad' instead of 'happy'.
- *Perturbation.* Also a label-preserving operation. Adding a small amount of noise to an image can cause a neural network to misclassify it. Using deceptive data to trick a neural network into making wrong predictions is called adversarial attacks. Adding noise to samples is a common technique to create adversarial samples. This can help models recognize weak spots in their learned decision boundary and improve their performance.
- *Data synthesis.* In computer vision, one way to synthesize new data is to combine existing examples with discrete labels to generate continuous labels.

5. Feature Engineering

The promise of deep learning is that we won't have to handcraft features. For this reason, deep learning is sometimes called feature learning. Many features can automatically be learned and extracted by algorithms. However, deep learning isn't always the solution. For other applications, feature engineering is still an important task.

Common Feature Engineering Operations

Handling Missing Values

Not all types of missing values are equal.

- *Missing not at random (MNAR)*: a value is missing for reasons relating to the true value itself. Some respondents may choose not to disclose their income, especially if their income is higher.
- *Missing at random (MAR)*: a value is missing not due to the value itself, but due to another observed variable. Age might be missing more often for women rather than men, because women (in this survey) don't like disclosing their age.
- *Missing completely at random (MCAR)*: when there's no pattern in when the value is missing. Values in the employment column might be missing not because of the job itself or any other variable; people just forget to fill in that value sometimes for no particular reason. This type of missing is more rare as there are usually reasons for missing values, which should be investigated.

Possible solutions:

- *Deletion*: if a variable has too many missing values, it might be best to remove that variable. The drawback of this approach is you might remove important information and reduce the accuracy of your model. Another solution is row

deletion, if a sample is missing too many values. This method can work when MCAR is at play and the number of missing values is small (say less than 0.1%. However if MNAR then you may be deleting important information. E.g. missing income might mean higher income and thus samples missing these values are still sending a signal. Furthermore, removing rows of data can create biases in your model, especially if MAR. From previous example, if you delete people missing ages, you could be deleting more women than men and the model will suffer with regards to women.

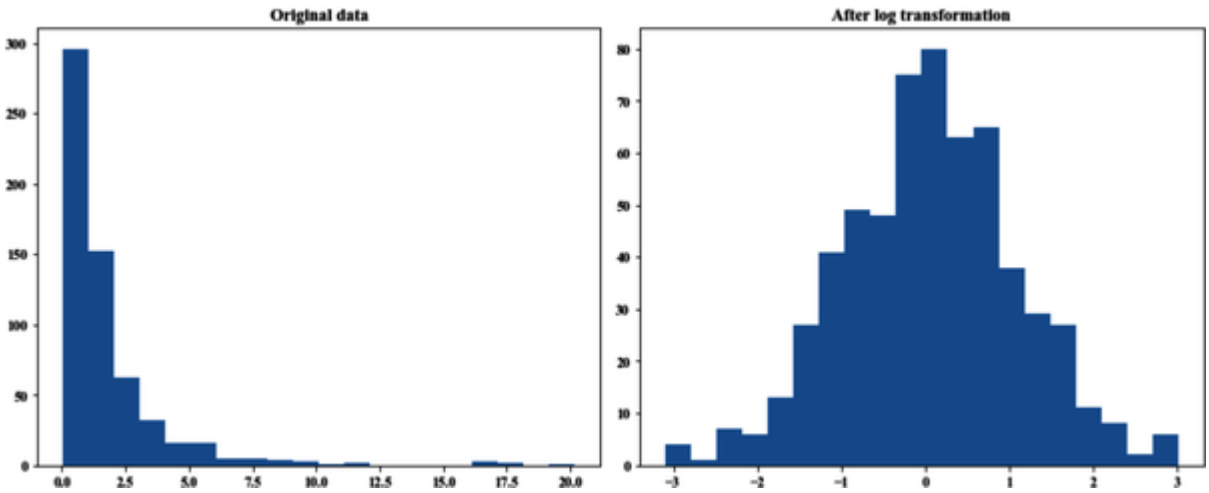
- *Imputation*: fill them with certain values. Common practices including filling them with the mean, median, or mode – all reasonable. You want to avoid filling them with possible values, such as missing number of children with 0, as 0 is a possible value for the number of children. It makes it harder to distinguish between people whose information is missing and people who don't have children. With imputation you risk injecting your own bias into and adding noise to your data, or worse, data leakage.

Scaling

Different features have different ranges – think of age and income. An ML model won't understand these ranges represent different things but both as numbers. Since 150,000 is much bigger than 40, it may give it more importance, regardless of which variable is actually more useful for generating predictions.

Before inputting features into models, it's important to scale them to be similar ranges. Neglecting to do so can cause your model to make gibberish predictions. Usually, this is done to get numerical features in the range [0,1] or [-1,1].

In practice, ML models tend to struggle with features that follow a skewed distribution. To help mitigate the skewness, a technique commonly used is *log transformation*: apply the log function to your feature. Be wary of the analysis performed on log-transformed data instead of the original data. A log transformation:



Scaling is a common source of data leakage. Secondly, it often requires global statistics – you have to look at the entire or a subset of training data to perform it. If new data has changed significantly compared to the training, these statistics won't be useful so it is important to retrain your model often to account for these changes.

Encoding Categorical Features

Feature Crossing

A technique to combine two or more features to generate new features. This is useful to model nonlinear relationships between features. This is essential for models that can't learn nonlinear relationships, such as linear and logistic regression, and tree-based models. It's less important in neural networks but can still be useful to help them converge faster. Feature crossing can quickly create too many features (for 100 features, 100x100 = 10,000 possible values). Another caveat is that overfitting becomes more likely with more features.

Data Leakage

This refers to the phenomenon when a form of the label "leaks" into the set of features used for making predictions, and this same information is not available during inference. It is challenging because often the leakage is nonobvious. The model can fail even after extensive evaluation and testing.

Example: suppose you want to build an ML model to predict whether a CT scan of a lung shows signs of cancer. You obtained the data from hospital A, removed the doctors' diagnosis from the data, and trained your model. It does well on the data from hospital A, but poorly on the data from hospital B. You learn that hospital A, when doctors thing a patient has lung cancer, send that patient to a more advanced scan machine which outputs slightly different CT scan images. Your model learned to rely on the info on the scan machine used to make predictions. Hospital B sends the patients to different CT scan machines at random, so your model has no information to rely on. We say that labels are leaked into the features during training.

Common Causes for Data Leakage

- *Splitting time-correlated data randomly instead of by time.* In many cases, data is time-correlated, which means that the time the data is generated affects its label distribution. E.g. stock prices – if 90% of tech stocks go down today, it's likely the other 10% go down too. When building models to predict the future stock prices, you want to split your training data by time, such as training the model on data from the first six days and evaluating it on data from the seventh day. If you randomly split your data, prices from the seventh day will be included in your train split and leak into your model the condition of the market on that day.
- *Scaling before splitting*. A common mistake is to use the entire training data to generate global statistics before splitting it into different splits, leaking the mean and variance of the test samples into the training process, allowing a model to adjust its

predictions for the test samples. This information isn't available in production, so the model's performance will likely degrade. To avoid this, always split your data before scaling, then use the statistics from the train split to scale all the splits. Some suggest splitting our data before EDA and data processing so that we don't accidentally gain information about the test split.

- *Filling in missing data with statistics from the test split*. Leakage might occur when imputing missing values with the mean/median if it is calculated using the entire data instead of just the train split. Same solution as before, use only statistics from the train split to fill in missing values in all the splits.
- *Poor handling of data duplication before splitting*. If there are duplicate samples, make sure to remove them otherwise they may appear in both train and test splits. This often happens from data collection or merging of different data sources. Check for duplicates before and after splitting. If oversampling your data, do it after splitting.

Detecting Data Leakage

Can happen during many steps, from generating, collecting, sampling, splitting, and processing data to feature engineering. It is important to monitor for data leakage during the entire lifecycle of an ML project.

Measure the predictive power of each feature or a set of features with respect to the target variable. If a feature has unusually high correlation, investigate how this feature is generated and whether the correlation makes sense. It is possible that two features independently don't contain leakage, but two features together can contain leakage. E.g. if predicting how long an employee will stay at a company, the start and end dates separately don't tell us much, but together will give us that information.

Likewise, if removing a feature causes the model's performance to deteriorate significantly, investigate why that feature is so important. Do ablation studies with a subset of features that you suspect the most. Domain expertise is important here.

Finally, be careful how you use the test split. It should only be used to report the model's final performance. If used for ideas for new features or to tune hyperparameters, you risk leaking information from the future into your training process.

Engineering Good Features

Generally, adding more features leads to better model performance. However, having too many can be bad both during training and serving your model:

- The more features you have, the more opportunities there are for data leakage.
- Too many features can cause overfitting.
- Requires increased memory capacity to serve a model, which gets expensive.
- It can increase inference latency when doing online prediction, especially if you need to extract these features from raw data for predictions online.

- Useless features become technical debts. Whenever your data pipeline changes, all the affected features need to be adjusted accordingly.

In theory, if a feature doesn't help a model make good predictions, regularization techniques should reduce that feature's weight to 0. However, in practice, it might help models learn faster if the features that are not useful are removed. There are two factors to consider when evaluating whether a feature is good for a model.

Feature Importance

With tree-based methods, it is easy to use the built-in feature importance method. For more model-agnostic methods, you might want to look into SHAP (SHapley Additive exPlanations). A feature's importance to a model is measured by how much that model's performance deteriorates if that feature is removed from the model. SHAP is great because it not only measures a feature's importance to an entire model, it also measures each feature's contribution to a model's specific prediction. Not only good for choosing the right features, feature importance techniques are great for interpretability: you understand how your models work under the hood.

Feature Generalization

Features should generalize to unseen data. Not all features generalize equally. This is less scientific than feature importance and it requires intuition, domain knowledge, and statistical intuition. You should consider feature coverage and the distribution of feature values.

Coverage is the percentage of the samples that has values for this feature in the data – so the fewer values that are missing, the higher the coverage. Generally, if a feature appears in only a very small % of the data, it is not going to be very generalizable. However, in cases where MNAR, this feature could be a strong indicator. E.g. if a feature appears in only 1% of your data, but 99% of the examples with this feature have positive labels, this feature is useful and you should use it.

Coverage of a feature can differ wildly between different slices of data and even in the same slice of data over time. If the coverage of a feature differs a lot between the train and test split, this is an indication that your train and test splits don't come from the same distribution. You might want to investigate whether the way you split your data makes sense and whether this feature is a cause for data leakage. For the feature values that are present, you might want to look into their distribution.

6. Model Development and Offline Evaluation

Model development is an iterative process. After each iteration, you want to compare your model's performance against its performance in previous iterations and evaluate how suitable this iteration is for production.

Evaluating ML Models

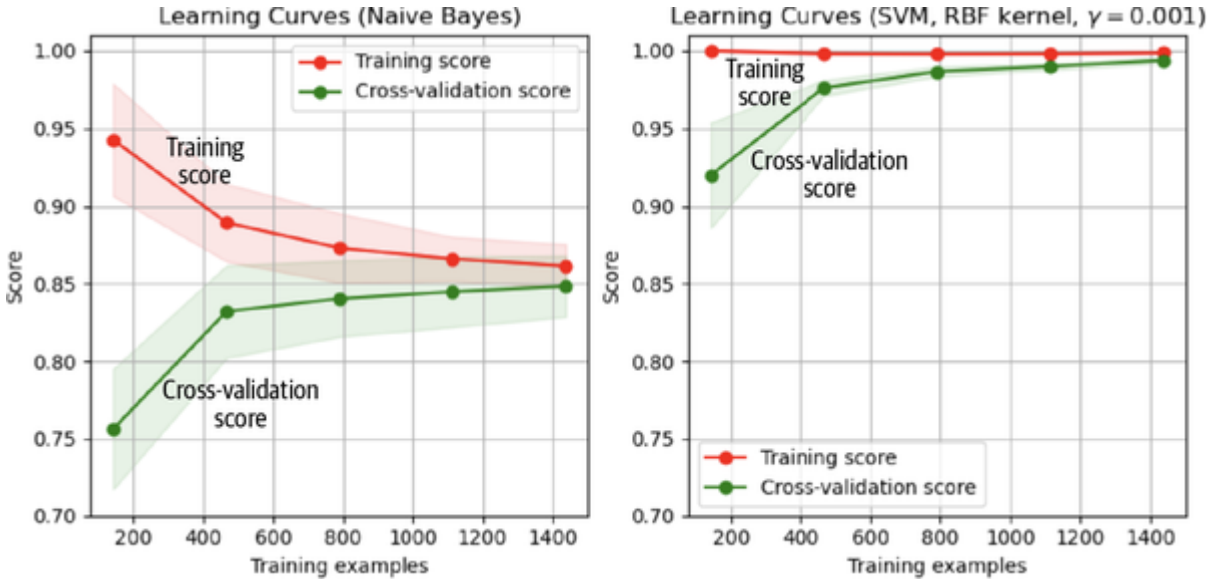
When selecting a model for your problem, you don't choose from every possible model out there, but usually focus on a set of models suitable for your problem. If your client wants you to build a system to detect fraudulent transactions, you know this is an anomaly detection problem, and common algorithms for this includes k-nearest neighbours, clustering, and neural networks. Knowledge of common ML tasks and the typical approaches to solve them is essential in this process.

When considering what model to use, it's important to consider not only the model's performance, measured by metrics such as accuracy, F1 score, and log loss, but also its other properties, such as how much data, compute, and time it needs to train, what's its inference latency, and interpretability. A logistic regression model might have lower accuracy than a neural network, but it requires less labelled data to start, it's faster to train, deploy, and it's more interpretable.

Tips for model selection

- *Avoid the state-of-the-art trap*. Researchers often only evaluate models in academic settings, which means that a model being state of the art means that it performs better than existing models on some static datasets. It doesn't mean the model will be fast or cheap enough for you to implement. It doesn't even mean that this model will perform better than other models on your data. Stay up to date with new technologies, evaluate whether they could help your business. However if a simpler solution solves your business problem, use the simpler solution.
- *Start with the simplest models*. Simple is better than complex. Simplicity serves three purposes. First, simpler models are easier to deploy, and deploying your model early allows you to validate that your prediction pipeline is consistent with your training pipeline. Second, starting with something simple and adding more complex components step-by-step makes it easier to understand your model and debug it. Third, the simplest model serves as a baseline to which you can compare your more complex models. Some simple models are already very complex, such as pretrained BERT models for NLP problems. This model might be low effort to start with, but high effort to improve upon. A simpler model will provide you with plenty of room to improve upon.
- *Avoid human biases in selecting models*. There are a lot of biases in evaluating models. Part of the process of evaluating an ML architecture is to experiment with different features and sets of hyperparameters to find the best model of that architecture. If an engineer is more excited about an architecture, they will likely spend more time experimenting with it, resulting in a better-performing model. When comparing different architectures, compare them under comparable setups. If you run 100 experiments for one, do so for the other.
- *Evaluate good performance now versus good performance later*. The best model now does not always mean the best model in the future. A model might work now with little data, however if you add plenty of data, a new architecture might be better. One way to estimate how you're model's performance might change with more data is to use learning curves. A learning curve of a model is a plot of its performance (e.g.

training loss, training accuracy, validation accuracy) against the number of training samples it uses. Learning curves can give you a sense of whether you can expect any performance gain at all from more training data. While evaluating models, you want to take into account their potential for improvements in the future, and how easy it is to achieve those improvements (neural networks perform better with more training data, but might be difficult to implement).



- *Evaluate trade-offs*. There are many. One common one is the false positives/false negatives trade-off, where reducing one might increase the number of the other. Business use case is important here. In a medical setting, false negatives are more dangerous than false positives. In security, the opposite might be true. Another trade-off is that between compute requirement and accuracy – a more complex model might deliver higher accuracy, but require a GPU. There is also the interpretability and performance trade-off, which is important in insurance.
- *Understand your model's assumptions*. "All models are wrong, but some are useful". The real world is complex, and models can only approximate using assumptions. Every model comes with its own assumptions. Understanding what assumptions a model makes and whether our data satisfies those assumptions can help you evaluate which model works best for your use case. Examples: *Prediction assumption* – every model makes the assumption that it's possible to predict Y based on X; *IID* – many models assume that examples are independent and identically distributed (meaning all examples are independently drawn from the same joint distribution); *Normally distributed* – many methods assume the data is normally distributed; *Boundaries* – a linear classifier assumes that decision boundaries are linear.

Ensembles

When considering an ML solution, you often start with a system that contains just one model. After developing this model, you think about how to improve its performance. One method that consistently boosts performance 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*. Ensemble methods often win Kaggle competitions, but are less favoured in production because they are harder to deploy and maintain.

Why does it work so well? Imagine you have three 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 three classifiers are not correlated, taking the majority vote of these three classifiers bumps accuracy up to 78.4%.

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

When creating an ensemble, the less correlation among the base learners the better. Therefore, it's common to choose different types of models for an ensemble, e.g. one gradient-boosted tree, one neural network, and one transformer model. There are three ways to create an ensemble: bagging, boosting, and stacking.

Bagging

Bootstrap aggregating is designed to help improve both the training stability and accuracy of ML algorithms. It reduces variance and helps to avoid overfitting. Given a dataset, instead of training one classifier on the entire dataset, you sample with replacement to create different datasets, called bootstraps, and train a classification or regression model on each of these bootstraps. Sampling with replacement ensures that each bootstrap is created independently from its peers. A random forest is an example of bagging – it is a collection of decision trees constructed by both bagging and feature randomness, where each tree can pick only from a random subset of features to use.

Boosting

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 weighed differently among iterations. As a result, future weak learners focus more on the examples that previous weak learners misclassified. The final strong classifier is a weighted combination of the existing classifiers – classifiers with smaller training errors have higher weights.

Stacking

This means you train base learners from the training data, then create a meta-learner that combines the outputs of the base learners to output final predictions. The meta-learner can be as simple as a heuristic: the majority vote for classification tasks or the average vote for regression tasks, from all base learners. It can also be another model, such as a logistic or linear regression model.

Experiment Tracking and Versioning

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

The process of tracking the progress and results of an experiment is called *experiment tracking*. The process of logging all the details of an experiment for the purpose of possibly recreating it later or comparing it with other experiments is called *versioning.*

Experiment Tracking

Many problems arise during the training process, including loss not decreasing, overfitting, underfitting, 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. Things you may want to track for each experiment:

- The *loss curve* corresponding to the train split and each of the eval splits.
- The *model performance metrics* that you care about on all non-test splits, such as accuracy, F1, perplexity.
- The log of *corresponding sample, prediction, and ground truth label*. This comes in handy for ad hoc analytics and sanity check.
- The *speed* of your model, evaluated by the number of steps per second.
- *System performance metrics* such as memory usage and CPU/GPU utilization. They're important to identify bottlenecks and avoid wasting system resources.

In theory, it's not a bad idea to track everything you can, even if you don't look at most of the results. When something does happen, one or more of them might give you clues to understand or debug your model. 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. Third-party experiment tracking tools can give you nice dashboards to visualize the experiments.

Versioning

ML systems are part code, part data, so you need to not only version your code but your data as well. Code versioning is an industry standard, but many don't version their data. It is challenging to do so because data is often much larger than code, so the same strategy used for versioning code cannot be used for versioning data.

Code versioning is done by keeping track of all the changes made to a codebase, with each change being measured by line-by-line comparison (this is too difficult to do for data). Code versioning allows users to revert to a previous version of the codebase by keeping copies of all the old files. A dataset might be too large that duplicating it multiple times might be unfeasible. Code versioning allows 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.

There is also confusion into what constitutes a diff when we version data – does this mean changes in the content of any file in the data repository or only when a file is removed/added? Another confusion is in how to resolve merge conflicts. Furthermore, if you use user data to train your model, regulations like GDPR might make versioning this data complicated.

Debugging ML Models

Things that can cause an ML model to fail**:**

- *Theoretical constraints*. A 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 those assumptions. E.g. using a linear model for the data whose decision boundaries aren't linear.
- *Poor implementation of model*. The model might be a good fit for the data, but the bugs are in the implementation of the model. The more components a model has, the more things that can go wrong, and the harder it is to figure out which has gone wrong.
- *Poor choice of hyperparameters.*
- *Data problems*. Something went wrong in data collection and pre-processing, such as data samples and labels being incorrectly paired, noisy labels, features normalized incorrectly, etc.
- *Poor choice of features.* Too many features can cause your model to overfit or cause data leakage. Too few might lack predictive power.

Debugging should be both preventative and curative. You should have healthy practices to minimize the opportunities for bugs to proliferate as well as a procedure for detecting, locating, and fixing bugs. There is still no scientific approach to debugging in ML, but there are techniques:

- *Start simple and gradually add more components.* Start with the simplest model and add components to see if it helps or hurts the performance. A complex model is hard to debug because a problem could have been caused by one of many components.

- *Overfit a single batch.* After an 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%. If your model can't overfit a small amount of data, there might be something wrong with your implementation.
- *Set a random seed.* Many factors contribute to the randomness of your model: weight initialization, dropout, data shuffling. Randomness makes it hard to compare results across different experiments – a change in performance could be due to a change in the model or a different random seed. Setting a random seed ensures consistency between different runs. It allows you to reproduce errors and others to reproduce your results.

Distributed Training

As models get bigger and more resource-intensive, companies care more about training at scale. When your data doesn't fit into memory, your algorithms for pre-processing (normalizing, zero-centering, etc.), shuffling and batching data will need to run out of core and in parallel. With one machine and only a few samples at a time, this leads to instability for gradient descent-based optimization.

Data parallelism

It is now the norm to train ML models on multiple machines. With data parallelism, you split your data on multiple machines, train your model on all of them, and accumulate gradients.

Model parallelism

With data parallelism, each worker has its own copy of the whole model and does all the computation necessary for its copy of the model. Model parallelism is when different components of the model are trained on different machines.

AutoML

This refers to automating the process of finding ML algorithms to solve real-world problems. The most popular form in production is hyperparameter tuning. A hyperparameter is a parameter supplied by users whose value is used to control the learning process, e.g. the learning rate, batch size, number of layers, number of hidden units, optimizer, etc. With different sets of hyperparameters the same model can give drastically different performances on the same dataset. The goal of hyperparameter tuning is to find the optimal set of hyperparameters for a given model within a search space – the performance of each set evaluated on a validation set.

Popular ML frameworks come either with built-in utilities or have third-party utilities for tuning – for example scikit-learn with auto-sklearn and Tensorflow with Keras Tuner. Other popular methods include random search, grid search, and Bayesian optimization.

Model Offline Evaluation

Lacking a clear understanding of how to evaluate your ML systems might make it impossible to find the best solution for your need. Ideally, the evaluation methods should be the same during both development and production. In many cases, this is impossible because during development you have ground truth labels, which you don't during production. For certain tasks, it is possible to infer labels in production based on users' feedback. For example, for a recommendation task, one can infer a good recommendation by whether users click on it.

Baseline

Evaluation metrics, by themselves, mean little. Accuracy of 90% is as good as random guessing if the positive class accounts for 90% of the labels. It's essential to know the baseline you're evaluating your model against. Useful baselines:

- *Random baseline*. If our model just predicts at random, what's the expected performance?
- *Simple heuristic*. If you make predictions based on simple heuristics, what performance would you expect? E.g. if you rank items in reverse chronological order, showing the latest items first, would users stay on the app longer?
- *Zero rule baseline*. When your model always predicts the most common class. If this can predict something 70% of the time, your model has to outperform this significantly to justify the added complexity.
- *Human baseline*. The goal of ML is often to automate what would have otherwise been done by humans, so how does it perform against human experts? One example is for self-driving cars.
- *Existing solutions*. How does your new model perform against past ones? Even if inferior in performance, is it easier or cheaper to use?

Evaluation Methods

In academic settings, we focus on performance metrics. In production, we also want our models to be robust, fair, calibrated, and overall make sense. Methods that help:

Perturbation Tests

Ideally, the inputs used to develop your models should be similar to the inputs your model will have to work with in production – though not always the case. Often, production data is noisier. To get a sense of how 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. You might want to choice the model that works best on the perturbed data instead of on the clean data. The more sensitive your model is to noise, the harder it will be to maintain it.

Invariance Tests

Certain changes to the inputs shouldn't lead to changes in the output. For example, changes to race information shouldn't affect the mortgage outcome. If there are biases in the model,

one solution is to keep the inputs the same but change the sensitive information to see if the outputs change. Better, remove the sensitive information from the features used to train the model in the first place.

Model Calibration

To measure a model's calibration, a simple method is counting: you count the number of times your model outputs the probability X and the frequency Y of that prediction coming true, and plot X against Y. The graph for a perfectly calibrated model will have X equal Y at all data points. This is one of the most important properties of any predictive system.

Confidence Measurement

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 cause annoyance and make users lose trust in the system. If you only want to show the predictions that your model is certain about, how do you measure that certainty? What is the threshold? What do you do with predictions below that threshold – discard them, loop in humans, or ask more info from users?

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

Slice-based evaluation

Slicing means to separate your data into subsets and look at your model's performance on each subset separately. Some get too focused on overall F1 or accuracy on the entire data, but not focusing enough on slice-based metrics. This leads to two problems.

Firstly, the model will perform differently on different slices of data where it should perform the same. The focus on overall performance can be harmful for some demographics. Another problem is that when the model performs the same on different slices of data where it should perform differently.

To track your model's performance on critical slices, you first need to know what your critical slices are. This is more an art than science, requiring intensive data exploration and analysis. Three approaches:

- *Heuristics-based.* Slice your data using domain knowledge.
- *Error analysis.* Manually go through misclassified examples and find patterns among them.
- *Slice finder.* Algorithms such as beam search, clustering, or decision.

7. Model Deployment and Prediction Service

"Deploy" is a loose term that generally means making your model running and accessible. During model development, your model usually runs in a development environment. To be

deployed, your model will have to leave the development environment. This can be to a staging area for testing or to a production environment to be used by your end users.

If you want to deploy a model for your friends to play with, all you have to do is wrap your predict function in a POST request endpoint using Flask or FastAPI, put the dependencies this predict function needs to run in a container, and push your model and its associated container to a cloud service to expose the endpoint. You can have functional deployment in an hour. The hard parts include making your model available to millions of users with a latency of milliseconds, setting up the infrastructure so that the right person can immediately be notified when something goes wrong, figuring out what went wrong, and seamlessly deploying the updates to fix what's wrong.

ML Deployment Myths

1. *You only deploy one or two ML models at a time.* In reality, companies have many ML models. Uber will need a model to predict each of the following elements: ride demand, driver availability, estimated time of arrival, dynamic pricing, fraudulent transaction, customer churn, and more. Additionally, the app runs in multiple countries, so you need models that generalize across different user-profiles, cultures and languages. Uber has thousands of models in production.
2. *If we don't do anything, model performance remains the same.* Software ages poorly. ML is the same. On top of that, ML systems suffer from data distribution shifts, when the data distribution your model encounters in production is different from the data distribution it was trained on. An ML model performs best right after training and degrades over time.
3. *You won't need to update your models much.* "How often can I update my models?" should be the question you ask yourself. In 2015, Etsy deployed 50 times/day, Netflix thousands of times per day, AWS every 11.7 seconds. TikTok might update their models every 10 minutes.

Batch Prediction vs Online Prediction

How will the system generate and serve its predictions to end users: online or batch. There are three main modes of prediction:

- Batch prediction, which uses only batch features.
- Online prediction that uses only batch features (e.g. precomputed embeddings).
- Online prediction that uses both batch and streaming features, i.e. streaming prediction.

Online prediction is when predictions are generated and returned as soon as requests for these predictions arrive. For example, entering an English sentence into Google Translate and getting back its French translation immediately. This is also known as *on-demand prediction*. Traditionally, requests are sent to the prediction service via RESTful APIs, e.g. HTTP requests. When prediction requests are sent via HTTP requests, online prediction is also known as *synchronous prediction*: predictions generated in sync with requests.

Batch prediction is when predictions are generated periodically or whenever triggered. The predictions are stored somewhere, such as in SQL tables, and retrieved as needed. Netflix might generate movie recommendations for all users every 4 hours, and the precomputed recommendations are fetched and shown to users when they log on. Batch predictions are also known as *asynchronous prediction*.

They don't have to be mutually exclusive. One hybrid solution is that you precompute predictions for popular queries, then generate predictions online for less popular queries. UberEats will use batch prediction to generate restaurant recommendations. However, when you click on a restaurant, food item recommendations are generated using online prediction.

A problem with online prediction is that your model might take too long to generate predictions. With batch prediction, you can generate predictions for multiple inputs at once, leveraging distributed techniques to process a high volume of samples efficiently. Batch prediction is good for when you want to generate a lot of predictions and don't need the results immediately. You don't even have to use all the predictions: you can make predictions for all customers on how likely they are to buy a new product, and reach out to the top 10%.

A problem with batch prediction is that it makes your model less responsive to users' change preferences. Another problem is that you need to know what requests to generate predictions for in advance. There are some unpredictable queries – such as translation – that might be impossible to anticipate. For this you need online prediction. Batch prediction could be annoying in recommender systems – which could affect user engagement and retention. In other cases, it would be catastrophic, such as high-frequency trading, autonomous vehicles, voice assistants, and fraud detection.

Batch prediction is a workaround for when online prediction isn't cheap or fast enough. As hardware becomes more customized and powerful, online prediction might become the default. To overcome the latency challenge of online prediction, two components are required:

- A (near) real-time pipeline that can work with incoming data, extract streaming features, input them into a model, and return a prediction in near real time. A streaming pipeline with real-time transport and a stream computation engine can help with that.
- A model that can generate predictions at a speed acceptable to its end users. This usually means milliseconds.

Model Compression

Having a real-time pipeline isn't enough for online prediction. It also require fast inference. Inference is the process of generating predictions. If the model you want to deploy takes too long to generate predictions, there are three main approaches to reduce its inference latency: make it do inference faster, make the model smaller, or make the hardware it's deployed on run faster.

The process of making a model smaller is called *model compression,* and the process to make it do inference faster is called *inference optimization.* There are four main techniques for the former.

Low-Rank Factorization

To replace high-dimensional tensors with lower-dimensional tensors. This reduces the number of parameters and increases speed.

Knowledge Distillation

A small model (student) is trained to mimic a larger model or ensemble (teacher). The smaller model is what you'll deploy. This can work regardless of architectural differences between the teacher and student. The disadvantage is that it's highly dependent on the availability of a teacher network.

Pruning

Originally used for decision trees. Can also be done for neural networks. One way is to remove entire nodes, which means changing its architecture and reducing its number of parameters. The more common meaning is to find parameters least useful to predictions and set them to 0. The total number of parameters are not reduced, only the number of nonzero parameters, so the architecture remains the same. This helps make the network more sparse, requiring less storage space and improving computational performance of inference without compromising overall accuracy.

Quantization

The most general and commonly used method. This reduces a model's size by using fewer bits to represent its parameters. It not only reduces memory footprint but also improves the computation speed.

8. Data Distribution Shifts and Monitoring

Deploying a model isn't the end of the process. A model's performance degrades over time in production. We have to continually monitor the model's performance to detect issues as well as deploy updates to fix these issues.

Causes of ML System Failures

A failure happens when one or more expectations of the system is violated. For an ML system we care about both its operational metrics and its ML performance metrics, i.e. we care about its latency and throughput, but also it retains a high-level of accuracy.

Operational expectation violations are easier to detect as they're usually accompanied by an operational breakage such as a timeout, a 404 error on a webpage, an out-of-memory error, etc. ML performance expectation violations are harder to detect as doing so requires measuring and monitoring the models' performance in production. Often, ML systems fail silently. To effectively detect and fix ML system failures in production, it's useful to

understand why a model, after proving to work well during development, would fail in production.

Software System Failures

Failures that would have happened to non-ML systems.

- *Dependency failure.* A software package or codebase that your system depends on breaks, which leads your system to break.
- *Deployment failure.* Such as when you accidently deploy the binaries of an older version of your model, instead of the current version.
- *Hardware failures.* For example, the CPUs you use overheat and break down.
- *Crashing.* A hosted service is down, so your system is down.

Most often, systems break down due to causes not directly related to ML. Addressing software system failures requires not ML skills, but traditional software engineering skills.

ML-Specific Failures

Examples include data collection and processing problems, poor hyperparameters, changes in the training pipeline not correctly replicated in the inference pipeline and vice versa, data distribution shifts, edge cases, and degenerate feedback loops. They can be hard to detect and fix.

- *Production data different from training data.* When we say a model learns from the training data, it means that the model learns the underlying distribution of the training data with the goal of leveraging this learned distribution to generate accurate predictions for unseen data. It is essential for the training and unseen data to come from a similar distribution. This is unlikely to be the case in the real world, as it is constantly changing. The divergence leads to a common failure mode known as the *train-serving skew*: a model that does great in development but performs poorly when deployed. Data shifts happen all the time (not only for unusual events – COVID). They can happen from sudden events (competitors change prices, a celeb promotes your product), gradually (social norms, cultures, trends, industries change over time), or due to seasonal variations. Due to the complexity of ML systems and poor practices in deploying them, a large % of what might look like data shifts on monitoring dashboards are caused by internal errors (bugs in the data pipeline, missing values incorrectly imputed, etc.)
- *Edge cases.* These are the data samples so extreme that they cause the model to make catastrophic mistakes. Difference between outliers and edge cases? Outliers refer to data: an example that differs significantly from other examples. Edge cases refer to performance: an example where a model performs significantly worse than other examples. An outlier can be an edge case, but not all outliers are edge cases.
- *Degenerate feedback loops.* This can happen when the predictions themselves influence the feedback, which in turn influences the next iteration of the model. More formally: “a degenerate feedback loop is created when a system’s outputs are used to generate the system’s future inputs, which in turn influence the system’s

future outputs." Especially common in tasks with natural labels from users, such as recommender systems. E.g. you build a system to recommend songs to users. The songs that are ranked high by the system are shown first to users. Because they are shown first, users click on them more, which makes the system more confident that these recommendations are good, etc. This is why popular films, books, etc. keep getting more popular, which makes it hard for new items to break into popular lists. “Exposure bias”, “popularity bias”, “echo chambers”.

Detecting degenerate feedback loops

For recommender system, it is possible to detect this by measuring the popularity diversity of a system’s outputs even when the system is offline. This can be measured based on how many times it has been interacted with (seen, liked, bought) in the past. The popularity of all items will likely follow a long-tail distribution: a small number of items interacted with a lot, most items rarely interacted with. Various metrics such as aggregate diversity can help you measure the diversity of the outputs. Low scores mean that the outputs of your system are homogenous, which might be caused by popularity bias.

Correcting degenerate feedback loops

Introducing randomization in the predictions can reduce their homogeneity. Don’t only show users the items the system ranks highly for them, also show them random items and use their feedback to determine the true quality of these items, done at TikTok. Randomization can improve diversity, at the cost of user experience. An intelligent exploration strategy can find the appropriate trade-off between the two.

Data Distribution Shifts

This refers to the phenomenon in supervised learning when the data a model works with changes over time, which causes this model’s predictions to become less accurate as time passes. The distribution of the data the model is trained on is called the *source distribution*. The distribution of the data the model runs inference on is called the *target distribution*.

Types of Data Distribution Shifts

The inputs to a model are X, its outputs Y. In supervised learning, the training data can be viewed as a set of samples from the joint distribution P(X,Y), and then ML usually models P(Y|X). This joint distribution can be decomposed in two ways:

- P(X,Y) = P(Y|X)P(X)
- P(X,Y) = P(X|Y)P(Y)

P(Y|X) denotes the conditional probability of an output given an input. P(X) denotes the probability density of the input, etc.

- *Covariate shift:* When P(X) changes but P(Y|X) remains the same. A covariate is an independent variable (feature) that can influence the output of a given statistical trial but which is not of direct interest. An example of when this would occur is for the task of detecting breast cancer. The risk is higher for woman over 40, so age is an

input. You might have more women over 40 in your training data than in your inference data, so the input distributions differ between the two. However, for an example with a given age, such as above 40, the probability that this example has breast cancer is constant, thus P(Y|X) is the same. Covariate shifts can happen due to biases in the data selection process – selection bias problem closely associated. Can also happen when training data is artificially altered to make it easier for your model to learn (when you oversample or use active learning).

- *Label shift:* When P(Y) changes but P(Y|X) remains the same. Also known as prior shift. The output distribution changes but, for a given output, the input distribution stays the same. In preceding example, because there are more women over 40 in our training data than in our inference data, the % of positive labels is higher during training. However, if you randomly select person A from training data and person B from test data, both with breast cancer, they have the same probability of being over 40. So, P(X|Y) is the same. Not all covariate shifts result in label shifts. If there is now a preventative drug that every woman takes that helps reduce their chance of getting cancer. P(Y|X) reduces for women of all ages, so no longer a case of covariate shift. However, given a person with cancer, the age distribution remains the same, so this is still a case of label shift.
- *Concept drift:* When P(Y|X) changes but P(X) remains the same. AKA posterior shift. Same input, different output. Could happen due to events like COVID. Often just cyclic or seasonal. Flight ticket prices rise during holiday seasons. Companies might have different models to deal with different times of the year.

Detecting Data Distribution Shifts

These shifts are only a problem if they cause your model's performance to degrade. Firstly, we can monitor our model's accuracy-related metrics to see whether things have changed. If these are unavailable or too delayed to be useful, we can monitor other distributions of interest instead, such as the input distribution P(X), the label distribution P(Y), and the conditional distributions P(Y|X) and P(X|Y).

Statistical Methods

We can compare statistics such as min, max, mean, median, variance, quantiles, skewness, kurtosis, etc. A more sophisticated solution is to use a two-sample hypothesis test, to determine whether the difference between two populations (sets of data) is statistically significant. If it is, then the probability that the difference is a random fluctuation due to sampling variability is very low and therefore, the two populations comes from distinct distributions. A caveat is that this does not mean it is practically important. If the difference is from a small sample, then it is a serious difference. However, if it takes a very large number of samples to detect, then the different is probably not worth worrying about.

One basic two-sample test is the Kolmogorov-Smirnov test. Only works on low-dimensional data so you should reduce the dimensionality of your data before performing the test.

Addressing Data Distribution Shifts

This depends on how sophisticated your ML infrastructure setup is. To make a model work with a new distribution in production, there are three main approaches:

- *Train models using massive datasets.* The hope here is that if the training dataset is large enough, the model will be able to learn such a comprehensive distribution that whatever data points the model will encounter in production will likely come from this distribution.
- *Adapt a trained model to a target distribution without requiring new labels.* This is very underexplored and not used widely in industry.
- *Retrain your model using the labelled data from the target distribution.* Currently what is being done in industry. Two questions – whether to train your model from scratch or continue training it from the last checkpoint. Second, what data to use – from the last 24 hours, week, 6 months, or from the point when the data has started to drift.

If you consider your distribution to be a domain, then the question of how to adapt your model to new distributions is similar to how to adapt your model to different domains. Similarly, if you consider learning a joint distribution P(X,Y) as a task, then adapting a model trained on one joint distribution for another joint distribution can be framed as a form of transfer learning. The difference is that with transfer learning you don't retrain the base model from scratch for the second task. To adapt your model to a new distribution, you might need to do so.

Addressing these shifts doesn't have to start after the shifts have happened. It is possible to design your system to make it more robust to shifts. A system uses multiple features and different features shift at different rates. When choosing features for your models, you might want to consider the trade-off between the performance and the stability of a feature. A feature may be very good for accuracy but deteriorate quickly, forcing you to train your model more often.

Monitoring and Observability

Monitoring refers to the act of tracking, measuring, and logging different metrics that can help us determine when something goes wrong. Observability means setting up our system in a way that gives us visibility into our system to help us investigate what went wrong. The process of setting up our system in this way is also called "instrumentation". Examples include adding timers to your functions, counting NaNs in your features, tracking how inputs are transformed through your systems, logging unusual events, etc. Observability is a part of monitoring. Without some level of observability, monitoring is impossible.

Monitoring is all about metrics. The first class you need are operational metrics, those designed to convey the health of your systems. General divided into three levels:

- The network the system is run on.
- The machine the system is run on.

- The application that the system runs.

Examples of these metrics include latency; throughput; the number of prediction requests your models receives in the last minute, hour, day; the percentage of requests that return with a 2xx code; CPU/GPU utilization; memory utilization, etc. No matter how good your ML model is, if the system is down, no one is benefitting from it.

ML-Specific Metrics

Generally four artifacts to monitor: a model's accuracy-related metrics, predictions, features, and raw inputs. These are generated at four different stages of an ML system pipeline.

Monitoring accuracy-related metrics. If your system receives any type of user feedback for the predictions it makes – click, hide, purchase, upvote, favourite, bookmark, share, etc. – you should log and track it. Accuracy-related metrics are the most direct metrics to help you decide whether a model's performance has degraded.

Monitoring predictions. These are easy to visualize and their summary statistics are straight-forward to compute and interpret. You can monitor predictions for distribution shifts. You can monitor them for anything odd happening, such as predicting an unusual number of False in a row. Errors here can be detected quicker than accuracy-related metrics.

Monitoring features. Compared to raw input data, features are well structured following a predefined schema. The first step of monitoring is feature validation ensuring the features follow an expected schema. If these expectations are violated, there might be a shift in the underlying distribution. Check:

- If the min, max, or median values of a feature are within an acceptable range.
- If the values satisfy a regular expression format.
- If the values of a feature are always greater than the values of another.

Monitoring raw inputs. Before they are processed into features. This can be difficult if they come from multiple sources in different formats, following multiple structures.

Monitoring Toolbox

- *Logs.* Record events produced at runtime. An event can be anything of interest to the system developers. Examples include when a container starts, the amount of memory it takes, when a function is called, the input and output of that function, etc. Also crashes, error codes, etc. Dating app Badoo handles 20 billion events a day. The hard part might not be in detecting when something happened, but where the problem was, as systems contain so many different components. We want it to be easy to find the event later, this is called *distributed tracing*. We give each process a unique ID so that the error message contains that ID. Thus, we can search for log messages associated with it. We also want all metadata associated with it: the time when it happens, where it happens, etc.

- *Dashboards.* These visualize metrics. They also make monitoring accessible to non-engineers.
- *Alerts.* Alert the right people that something is wrong.

Observability

When something goes wrong within an observable system, we should be able to figure out what went wrong by looking at the system's logs and metrics. Observability is about instrumenting your system in a way to ensure that sufficient information about a system's runtime is collected and analyzed. It should be simple to find the errors.

In ML, this encompasses interpretability. Interpretability helps us understand how an ML model works and observability helps us understand how the entire ML system, which includes the model, works.

9. Continual Learning and Test in Production

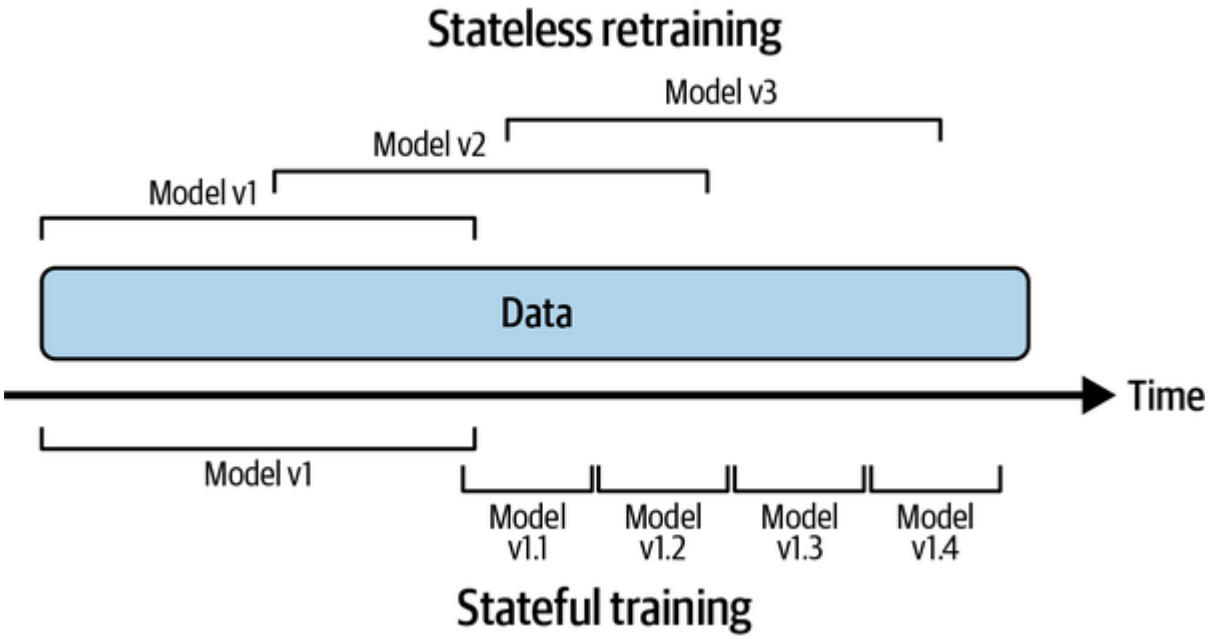
How do we adapt our models to data distribution shifts? By continually updating our ML models. Companies that employ continual learning in production update their models in micro-batches, e.g. they might update the existing model after every 512 examples.

The updated model shouldn't be deployed until it's been evaluated, so you shouldn't make changes to the existing model directly. You create a replica of the existing model and update this replica on new data, and only replace the existing model with the updated replica if the latter proves to be better. The existing model is called the *champion model* and the replica the *challenger.*

Many companies don't need to update their models too frequently because they don't have enough traffic for it to make sense. Furthermore, their models don't decay that fast.

Stateless Retraining vs Stateful Training

Continual learning isn't about the retraining frequency, but the manner in which the model is retrained. Most companies do stateless retraining – the model is trained from scratch each time. Continual learning means also allowing stateful training – the model continues training on new data. This is also known as fine-tuning or incremental learning.



Stateful training allows you to update your model with less data. Training a model from scratch requires a lot more data. Some companies have found that stateful training allows their models to converge faster and require much less compute power – 45x less.

With stateless retraining, a data sample might be reused during multiple training iterations of a model, which means that data needs to be stored. In the stateful training paradigm, each model update is trained only using the fresh data, so a date sample is only used once for training.

The training frequency is just a knob to twist. You can update your models are frequently as you want. Continual learning is about setting up infrastructure in a way that allows you to update the models whenever it is needed and deploy this update quickly.

- *Model iteration.* A new feature is added to an existing model architecture or the model architecture is changed.
- *Data iteration.* The model architecture and features remain the same, but you refresh this model with new data.

Stateful training is mostly applied for data iteration, since the former requires training the model from scratch.

Why Continual Learning?

The first use case is to combat data distribution shifts, especially when it happens suddenly. E.g. usually Wednesday nights are quiet in a neighbourhood, but this week there's an event. If you are Uber, your model needs to respond to the change in ride demand but increasing its price prediction and mobilizing more drivers to that neighbourhood. Otherwise, there will be negative user experience and lost revenue.

Another use case is to adapt to rare events. Amazon will not be able to collect enough historical data to predict how customers will behave throughout Black Friday. To improve performance, your model should learn throughout the data with fresh data.

A challenge that continual learning can help overcome is the 'continuous cold start' problem. The cold start problem arises when your model has to make predictions for a new user without any historical data. Continuous cold start is a generalization of this problem, as it can happen not just with new users but with existing users too. For example, when one switches from a laptop to a mobile phone and their behaviour is different on each platform. It can also happen when a user visits a service so infrequently that whatever historical data the service has about this user is outdated. If your model doesn't adapt quickly enough, it won't be able to make relevant recommendations to these users. If we could make our models adapt to each user within their visiting session, the models would be able to make accurate, relevant predictions to users even on their first visit. TikTok does this.

Continual Learning Challenges

Fresh data access challenge. If you want to update your model every hour, you need new data every hour. Many companies pull new training data from their data warehouses. The speed at which you pull data from these warehouses depends on the speed at which this data is deposited into your data warehouses. The speed can be slow, especially if data comes from multiple sources. An alternative is directly from real-time transports such as Kafka.

Evaluation challenge. Making sure the update is good enough to be deployed. The risks for big failures amplify with continual learning. The more frequently you update your models, the more opportunities there are for updates to fail. Evaluation takes time, which can be another bottleneck for model update frequency.

Algorithm challenge. This only affects matrix-based and tree-based models that want to be updated very fast (e.g. hourly). It is much easier to adapt models like neural networks than matrix-based or tree-based models to the continual learning paradigm.

Four Stages of Continual Learning

1. Manual, stateless retraining

Teams usually focus on developing new models, not updating existing ones. They only do so when performance has degraded so much, or when the team has time to update it. There may be no fixed schedule for how often to retrain every model, and they may be updated at different rates – this process is manual and ad hoc. Someone queries the data warehouse for new data, another cleans it and extracts features from it, another retrains that model from scratch on both the old and new data, someone else deploys it.

2. Automated retraining

After a few years, the team has managed to deploy models to solve most of the obvious problems. The priority is no longer to develop new models, but to maintain and improve existing ones. The ad hoc, manual process mentioned previously is too big a pain to

continue. The team writes a script to automatically execute all the retraining steps. This script is then run periodically using a batch process such as Spark. For most companies at this stage, the retraining frequency is set based on gut feeling. When creating scripts to automate this process, you need to take into account that different models might require different retraining schedules. The feasibility of this stage revolves around the feasibility of write a script to automate your workflow and configure your infrastructure to automatically:

Pull the data -> Downsampe/upsample if necessary -> Extract features -> Process and/or annotate labels to create training data -> Kick off the training process -> Evaluate the newly trained model -> Deploy it.

3. Automated, stateful training

In step above, you retrain your model from scratch, which can be costly. Here, you reconfigure your automatic updating script so that when the model update is kicked off, it first locates the previous checkpoint and loads it into memory before continuing training on this checkpoint. The main thing needed is a way to track your data and model lineage.

4. Continual Learning

In previous stage, the models are still updated based on a fixed schedule set out by developers. Finding the optimal schedule isn't straightforward and can be situation-dependent. Instead of relying on a fixed schedule, you might want your models to be automatically updated whenever data distributions shift and the model's performance plummets. The move from stage 3 to 4 is steep. You first need a mechanism to trigger model updates. This can be:

- *Time-based*. E.g. every 5 minutes
- *Performance-based*. E.g. whenever model performance plummets.
- *Volume-based*. E.g. whenever the total amount of labelled data increases by 5%.
- *Drift-based*. E.g. whenever a major data distribution shift is detected.

For this trigger mechanism to work, you need a solid monitoring solution. The hard part is not to detect the changes, but to determine which of these changes matter. If your monitoring solution gives a lot of false alerts, your model will be updated much more frequently than it needs to be.

How Often to Update Your Models

Before answering this question, you need to figure out how much gain your model will get from being updated with fresh data. The more gain it will get from fresher data, the more frequently it should be retrained.

One way to figure this out is by training your model on the data from different time windows in the past and evaluating it on the data from today to see how the performance changes. For example, you have data from the year 2020. To measure the value of data freshness, you can experiment with training model version A on data from January to June 2020, model version B on that from April to September 2020, etc. then test each of these

model versions on the data from December. The difference in the performance of these versions will give you a sense of the performance gain your model can get from fresher data. If the model trained from three months ago is much worse than that from last month, you shouldn't wait 3 months to retrain your model. At larger companies, this may need to be done in units of weeks, days, hours or minutes.

You should be doing both data and model iteration from time to time. However, the more resources you spend on one, the less you can spend on the other. If you find that iterating on your data doesn't give you much performance gain, then spend these resources on finding a better model. If finding a better model architecture requires 100x compute for training and gives you 1% performance, whereas updating the same model on data from the last 3 hours requires only 1x compute and gives the same gain, you're better off iterating on data.

Test in Production

Offline evaluation isn't enough. Train/test splits are static, and they have to be so, so that you have a trusted benchmark to compare multiple models. It's hard to compare the test results of two models if they are tested on different test sets. However, if you update the model to adapt to a new data distribution, you cannot just evaluate this new model on test splits from the old distribution. You'll need to test it on the most recent data that you have access to. The method of testing a predictive model on data from a specific period of time in the past is known as a *backtest.*

Backtests are not enough to replace static test splits. If something went wrong with your data pipeline and some data from the last hour is corrupted, evaluating your model solely on this recent data isn't sufficient. With backtests, you should still evaluate your model on a static test that you have extensively studied and mostly trust as a form of sanity check.

Because data distributions shift, the fact that a model does well on data from the last hour, doesn't mean it will continue doing well in the future. The only way to know whether a model will do well in production is to deploy it – test in production.

Shadow Deployment

The safest way to deploy your model or update:

1. Deploy the candidate model in parallel with the existing model.
2. For each incoming request, route it to both models to make predictions, but only serve the existing model's prediction to the user.
3. Log the predictions from the new model for analysis purposes.

Only when the new model's predictions are satisfactory do you replace the existing model with the new model. This technique is safe, but expensive – it doubles the number of predictions your system has to generate, which means doubling your compute inference cost.

A/B Testing

A way to compare two variants of an object, typically by testing responses to these two variants, and determining which of the two variants is more effective. In this case, the existing model is one variant and the candidate model is another. A/B testing is used to determine which model is better according to some predefined metrics.

Microsoft and Google conduct over 10,000 A/B tests annually. It works as follows:

1. Deploy the candidate model alongside the existing model.
2. A percentage of traffic is routed to the new model for predictions; the rest is routed to the existing model. There are cases where one model's predictions might affect another model's predictions – e.g. in ride-sharing's dynamic pricing, a model's predicted prices might influence the number of available drivers and riders, etc. In those cases, you might have to run your variants alternatively, e.g. serve model A one day and then serve model B the next day.
3. Monitor and analyze the predictions and user feedback from both models to determine whether the difference in the two models' performance is statistically significant. To measure this, one could use two-sample tests.

A/B testing consists of a randomized experiment: the traffic routed to each model has to be truly random. If not, the result will be invalid as we won't know if there are other factors at play determining the different results. Second, the A/B test should be run on a sufficient number of samples to gain enough confidence about the outcome.

It is possible to do A/B testing with more than two variants as you might have more than one candidate model.

Canary Release

A technique to reduce the risk of introduce a new software version in prediction by slowly rolling out the change to a small subset of users before rolling it out to the entire infrastructure and making it available to everybody. In ML deployment:

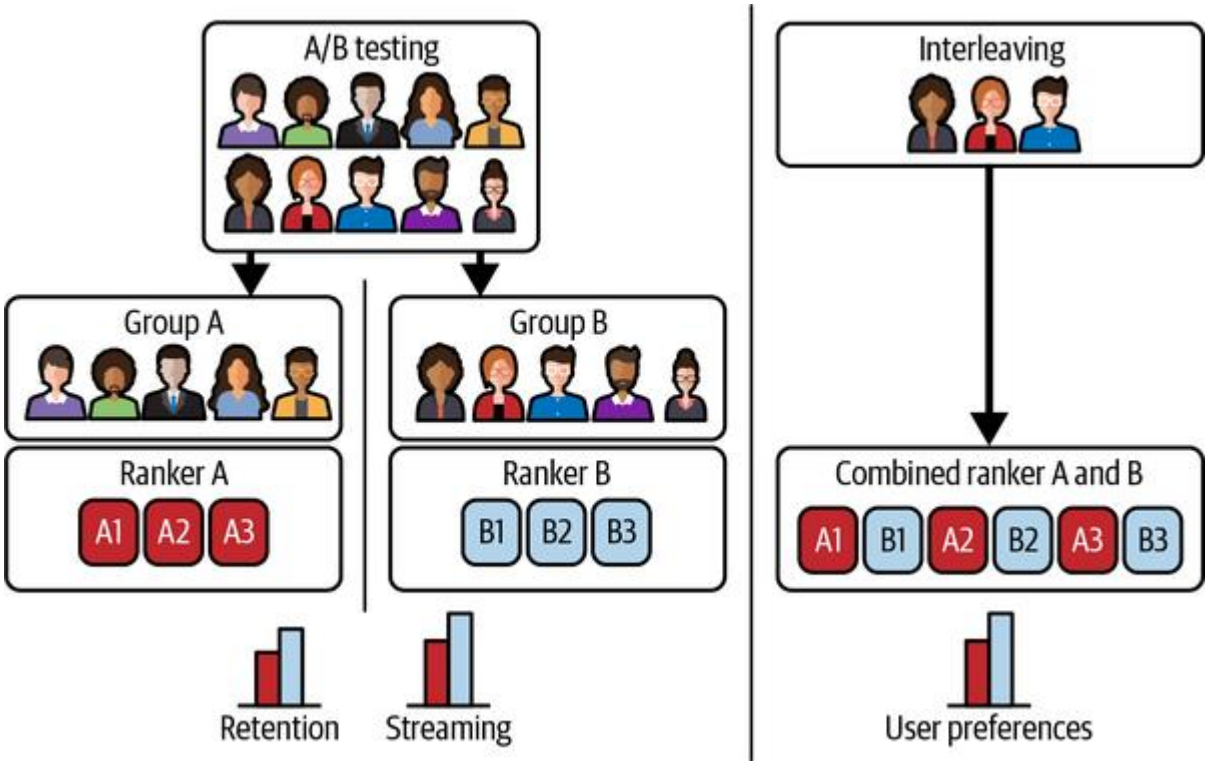
1. Deploy the candidate model alongside the existing model. The candidate model is the canary.
2. A portion of traffic is routed to the candidate model.
3. If its performance is satisfactory, increase the traffic to the candidate model. If not, abort the canary and route all traffic back to the existing model.
4. Stop when either the canary serves all the traffic (the candidate model has replaced the existing model) or when the canary is aborted.

It is a similar setup to A/B testing, but you can do canary analysis without A/B testing. You don't have to randomize the traffic route to each model. You could roll out the candidate model to a less critical market before rolling out to everybody.

Interleaving Experiments

Instead of exposing a user to recommendations from one model, you expose them to recommendations from both and see which model's recommendations they click on. Netflix

found that interleaving “reliably identifies the best algorithms with considerably smaller sample size compared to traditional A/B testing”.

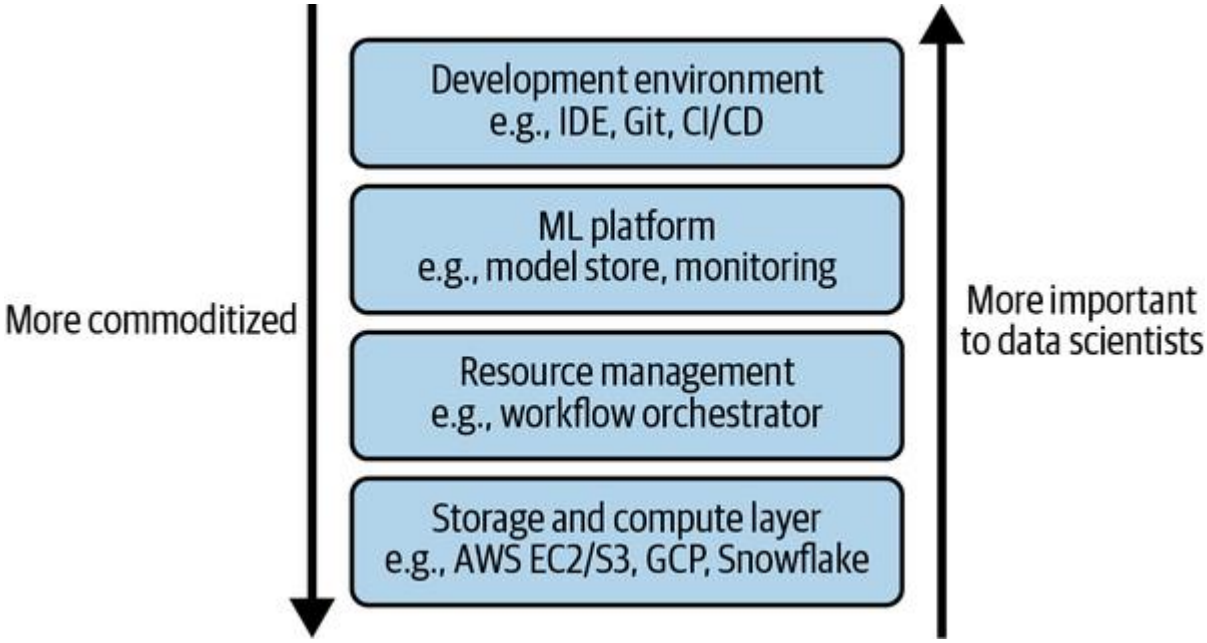


There’s no guarantee that interleaving will lead to better core metrics, because it is decided by user preferences. For interleaving to yield valid results, we must ensure that at any given position, a recommendation is equally likely to be generated by A or B.

10. Infrastructure and Tooling for MLOps

You might know the right things to do for your ML systems, but you need your infrastructure to be set up the right way. Infrastructure, when set up right, can help automate processes, reducing the need for specialized knowledge and engineering time. This, in turn, can speed up the development and delivery of ML applications, reduce the surface area for bugs, and enable new use cases. When set up wrong, infrastructure is painful to use and expensive to replace. The infrastructure required for you depends on the number of applications you develop and how specialized they are. Some companies will only need Jupyter and Pandas (no infra). Others, like Google and Uber will need their own highly specialized infrastructure. Companies in the middle of the spectrum will likely benefit from generalized ML infrastructure that is being increasingly standardized.

In the ML world, infrastructure is the set of fundamental facilities that support the development and maintenance of ML systems. What is considered “fundamental” varies from company to company. Four layers are shown below:



Data and compute are the essential resources needed for any ML project, and thus the storage and compute layers from the infrastructural foundation for any company that wants to apply ML. The dev environment is what data scientists have to interact with daily.

Storage and Compute

ML systems work with a lot of data, and this data needs to be stored somewhere. The *storage layer* is where data is collected and stored. At its simplest form, it can be a hard drive disk or solid state disk. The storage layer can be in one place, e.g. in Amazon S3 or in Snowflake, or spread out over multiple locations. It can be in a private data center or on the cloud, which is now the norm.

The *compute layer* refers to all the compute resources a company has access to and the mechanism to determine how these resources can be used. This determines the scalability of your workloads. It is the engine to execute your jobs. At its simplest form, it can be a single CPU or GPU core that does all your computation. Its most common form is cloud compute managed by a cloud provider such as AWS or GCP.

The compute layer can usually be sliced into smaller compute units to be used concurrently. E.g. a CPU core might support two concurrent threads, each thread is used as a compute unit to execute its own job. Or multiple CPU cores might be joined together to form a larger compute unit to execute a larger job. However, the compute layer doesn't always use threads or cores as compute units – they may use other units of computation. Spark uses "job" as its unit, Kubernetes "pod", etc.

To execute a job, you first need to load the required data into your compute unit's memory, then execute the required operations on that data. If the compute unit doesn't have enough memory to load this data, the operation will be impossible. Therefore, a compute unit is

mainly characterized by two metrics: how much memory it has and how fast it runs an operation.

The memory metric can be specified using units like GB. A compute unit with 8GB of memory can handle more data in memory than one with only 2GB, and it is generally more expensive. The operation speed is more contentious, but the most common metric is FLOPS – floating point operations per second. When evaluating performance, most just look at the number of cores a computer unit has, e.g. 4 CPU cores. When evaluating a new compute unit, it's important to evaluate how long it will take to do common workloads.

Public Cloud vs Private Data Centers

Like data storage, the compute layer is largely commoditized. Instead of setting up their own data centers for storage and compute, companies pay cloud providers like AWS and Azure for the exact amount of compute they use. Cloud compute makes it easy for companies to start building without worrying about the compute layer – good for variable-sized workloads. Leveraging the cloud tends to give companies higher returns than building their own storage and compute layers early on, though this becomes less defensible as a company grows.

Resource Management

In the pre-cloud world, storage and compute were finite. Resource management back then centered around how to make the most out of limited resources. In the cloud world where storage and compute resources are more elastic, the concern has shifted to how to use resource cost-effectively. Adding more resources to an application doesn't mean decreasing resources for other applications, it just makes it costlier. This needs to be justified by the return, e.g. extra revenue or saved engineering time.

Cron, Schedulers, and Orchestrators

Two key characteristics of ML workflows that influence their resource management: repetitiveness and dependencies. ML workloads are rarely one-time operations; you might train a model every week. This repetitive process can be schedule and orchestrated to run smoothly and cost-effectively using available resources.

Scheduling repetitive jobs to run at fixed times is what a *cron* does: run a script at a predetermined time and tell you whether the job succeeds or fails. It doesn't care about the dependencies between the jobs it runs.

Steps in an ML workflow might have complex dependency relationships with each other:

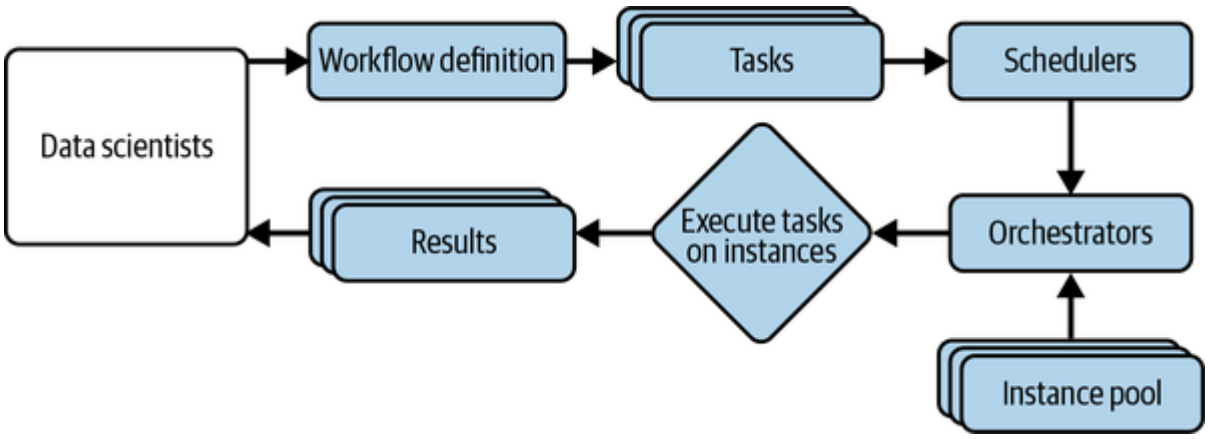
1. Pull last week's data from data warehouses.
2. Extract features from this pulled data.
3. Train two models, A and B, on the extracted features.
4. Compare A and B on the test set.
5. Deploy A if it is better, otherwise deploy B.

Each step depends on the success of the previous step. The order of execution and dependencies among these steps can be represented using a graph, specifically a DAG: *directed acyclic graph*. It has to be direct to express the dependencies among the steps. It can't contain cycles, otherwise the job will run on forever. *Schedulers* are cron programs that can handle dependencies. It takes in the DAG of a workflow and schedules each step accordingly. They allow you to specify what to do if a job fails or succeeds.

If schedulers are concerned with when to run jobs and what resources are needed to run those jobs, *orchestrators* are concerned with where to get those resources. Schedulers deal with job-type abstractions such as DAGs, priority queues, user-level quotas, etc. Orchestrators deal with lower-level abstractions like machines, instances, clusters, replication, etc. If the orchestrator notices that there are more jobs than the pool of available instances, it can increase the number of instances in the available instance pool. The most well-known of these is Kubernetes.

Data Science Workflow Management

Tools here manage workflows. You can specify your workflows as DAGs. Workflows can be defined using either code (Python) or configuration files (YAML). Each step in a workflow is called a task. Most workflow tools come with schedulers. Once a workflow is defined, the underlying scheduler works with an orchestrator to allocate resources to run the workflow:



Common tools for this are Airflow, Argo, Kubeflow and Metaflow.

ML Platform

As companies find uses for ML in more and more applications, there's more to be gained by leveraging the same set of tools for multiple applications instead of supporting a separate set of tools for each application. This shared set of tools for ML deployment makes up the ML platform. Components often found in ML platforms are model deployment, model store, and feature store. Evaluating a tool for each of these categories depends on your use case. Two aspects to consider:

- Whether the tools works with your cloud provider. Tools usually support integration with only a handful of cloud providers.

- Whether it's open source or a managed service. If it's open source, you can host it yourself. If it's a managed service, your models and likely some of your data will be on its service, which might not work for certain regulations. Some managed services work with virtual private clouds, which allows you to deploy your machines in your own cloud clusters, helping with compliance.

Model Deployment

A deployment service can help with both pushing your models and their dependencies to production and exposing your models as endpoints. Deployment is the most mature among all ML platform components and many tools exist for this. All major cloud providers offer tools for deployment: AWS with SageMaker, GCP with Vertex AI, Azure with Azure ML, etc.

When looking into a deployment tool, consider how easy it is to do both online prediction and batch prediction with the tool. It can be tricky. Some companies have separate deployment pipelines for online and batch prediction, e.g. Seldon for online and Databricks for batch. Furthermore, look into how easy it is to perform quality tests on the models before deployment.

Model Store

Not as simple as a place to store your models. A model can go wrong for many reasons, and it is difficult to fix it without knowing the specific cause. To help with debugging and maintenance, it's important to track as much info associated with a model as possible. A few necessary artifacts:

- *Model definition.* Needed to create the shape of the model, e.g. what loss function it uses, how many hidden layers, how many parameters in each layer, etc.
- *Model parameters.* The actual values of the parameters of your model. Combined with the model definition can be used to re-create a model that can be used to make predictions.
- *Featurize and prediction functions.* Given a prediction request, how do you extract features and input these features into the model to get back a prediction? The featurize and prediction functions provide the instruction to do so.
- *Dependencies.* Python version, Python packages – usually packaged together in a container.
- *Data.* The data used to train the model – perhaps pointers to the location, plus name/version.
- *Model generation code.* How model was created such as how it was trained, how splits were created, number of experiments run, range of hyperparameters considers, actual set of final hyperparameters used.
- *Experiment artifacts.* Generated during model development process, such as graphs for loss curve, or raw numbers for performance on test set.
- *Tags.* To help with model discovery and filtering, such as owner of model or task (business problem this solves, fraud detection).

Feature Store

This should help address three main problems:

- *Feature management.* Features used for one model can be used for another model. A feature store can help teams share and discover features. Here, a feature store can be considered a feature catalogue.
- *Feature computation.* A feature store can help with both performing feature computation, i.e. average age of drivers on the policy, and storing the results of this computation. Here, a feature store acts as a data warehouse.
- *Feature consistency.* Modern feature stores often unify the logic for both batch and streaming features, ensuring the consistency between features during training and inference.

Development Environment

This is where ML engineers write code, run experiments, and interact with the production environment where champion models are deployed and challenger models evaluated. The dev environment consists of an IDE (integrated development environment), versioning, and CI/CD. "If you only have time to set up one piece of infrastructure well, make it the development environment for data scientists."

The environment should be set up to contain all the tools that can make it easier for engineers to do their job. It should also consist of tools for versioning. Companies often use an ad hoc set of tools to version their ML workflows, such as Git to version control code, DVC to version data, MLflow to track artifacts of models when deploying them, etc. The environment should also be set up with a CI/CD test suite to test your code before pushing it to the staging or production environment. GitHub Actions is a tool to orchestrate this.

The IDE is the editor where you write your code, such as PyCharm or VS Code. Many people just use notebooks.

Standardizing Dev Environments

Standardized, if not company-wide, then at least team-wide. A popular option is to use a cloud dev environment with a local IDE. For example, using PyCharm installed on your computer and connecting the local IDE to the cloud environment using a secure protocol like Secure Shell (SSH). Moving from local dev environments to cloud dev environments has many benefits. Firstly, it makes IT support easier – having to support only one type of cloud instance rather than 1000 different local machines. Second, it's convenient for remote work – just SSH into your dev environment wherever you go from any computer. Third, cloud dev environments can help with security. Fourth, having the dev environment on the cloud reduces the gap between the dev environment and the production environment. If your production environment is in the cloud, it makes sense that your dev environment is too.

From Dev to Prod: Containers

During development, you usually just work with one machine because your workload doesn't fluctuate – your model doesn't change from serving only 1000 requests an hour to 1 million requests an hour. At production, service might be spread out on multiple instances.

This changes depending on incoming workloads, which can be unpredictable. Most cloud providers have taken care of the autoscaling part, though you have to worry about setting up new instances.

When you consistently work with the same instance, you can install dependencies once and use them whenever you use this instance. How do you re-create an environment on any new instance? Container technology – of which Docker is the most popular – is designed to answer this. With Docker, you create a Dockerfile with step-by-step instructions on how to re-create an environment in which your model can run: install this package, download this pretrained model, set environment variables, navigate into a folder, etc. These instructions allow hardware anywhere to run your code.

Two key concepts in Docker are image and container. Running all the instructions in a Dockerfile gives you a Docker image. If you run this image, you get back a Docker container. The Dockerfile is the recipe to construct a mold, which is a Docker image. From this mold, you can create multiple running instances; each is a Docker container.

You can build a Docker image from scratch or from another image. NVIDIA might provide a Docker image that contains TensorFlow and all necessary libraries to optimize TensorFlow for GPUs. If you want to build an application that runs TensorFlow on GPUs, it's not a bad idea to use this Docker image as your base and install dependencies specific to your application on top of this base image. You can find this images in a container registry such as Docker Hub or AWS ECR.

Different containers might be necessary when different steps in your pipeline have conflicting dependencies, such as your featurizer code requires NumPy 0.8, but your model requires NumPy 1.0. A tool to help you manage multiple containers is called container orchestration. However, each of your containers might run on its own host. Kubernetes is a tool for that. It creates a network for containers to communicate and share resources. It can help you spin up containers on more instances when you need more compute/memory as well as shutting down containers when you no longer need them.

11. The Human Side of ML

Chip Huyen
@chipro

Things I'd prioritize learning if I was to study to become a ML engineer again:

1. Version control
2. SQL + NoSQL
3. Python
4. Pandas/Dask
5. Data structures
6. Prob & stats
7. ML algos
8. Parallel computing
9. REST API
10. Kubernetes + Airflow
11. Unit/integration tests

6:30 AM · Oct 11, 2020 · Twitter Web App