Lecture 3: Data engineering [Draft]

CS 329S: Machine Learning Systems Design (cs329s.stanford.edu)

Prepared by Chip Huyen & the CS 329S course staff
Reviewed by Luke Metz
Errata and feedback: please send to chip@huyenchip.com

Note:

1. See the course overview and prerequisites on the lecture slides.
2. The course, including lecture slides and notes, is a work in progress. This is the first time the course is offered and the subject of ML systems design is fairly new, so we (Chip + the course staff) are all learning too. We appreciate your:
    a. **enthusiasm** for trying out new things
    b. **patience** bearing with things that don't quite work
    c. **feedback** to improve the course.

Table of contents

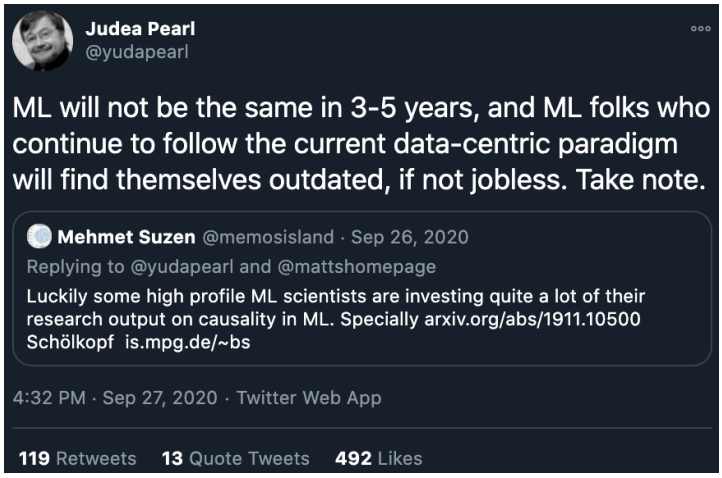
Mind vs. data

Progress in the last decade shows that the success of an ML system depends largely on the data it was trained on. Instead of focusing on improving ML algorithms, most companies focus on managing and improving their data[1].

Despite the success of models using massive amounts of data, many are skeptical of the emphasis on data as the way forward. In the last three years, at every academic conference I attended, there were always some debates among famous academics on the power of mind (inductive biases such as intelligent architectural designs) vs. data.

In theory, you can both pursue intelligent design and leverage computation, but spending time on one often takes time away from another[2].

On the mind over data camp, there's Dr. Judea Pearl, a Turing Award winner best known for his work on causal inference and Bayesian networks. The introduction to his book, "The book of why", is entitled "Mind over data," in which he emphasizes: "*Data is profoundly dumb*." He also went on Twitter to warn all data-centric ML people that they might be out of job in 3-5 years.



[3]

There's also a milder opinion from Dr. Chris Manning, who's a professor at Stanford and who's also a great person. He argued that huge computation and a massive amount of data with a simple learning device create incredibly bad learners. Structure allows us to design systems that can learn more from less data[4].

---

[1] More data usually beats better algorithms (Anand Rajaraman, Datawocky 2008)
[2] The Bitter Lesson (Richard Sutton, 2019)
[3] https://twitter.com/yudapearl/status/1310316514537385984
[4] Deep Learning and Innate Priors (Chris Manning vs. Yann LeCun debate).

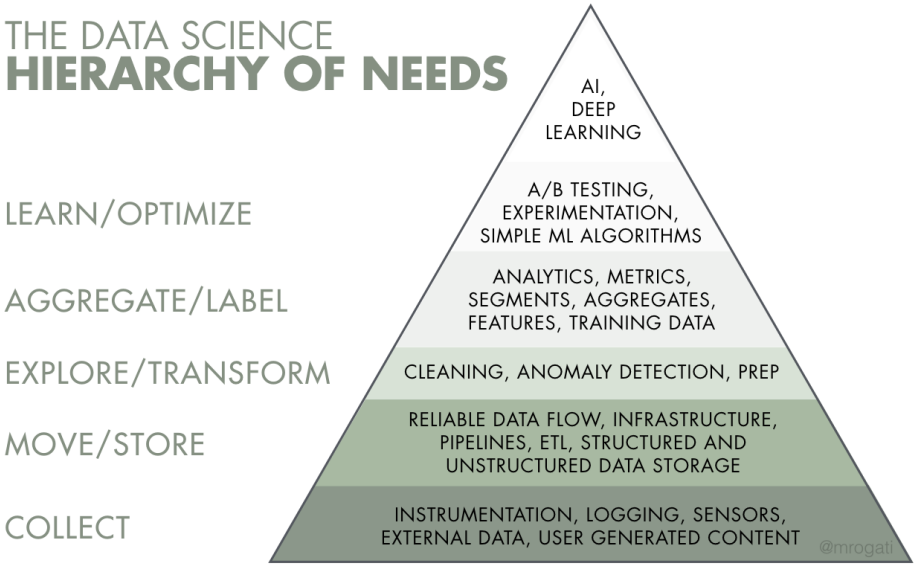
Many people in ML today are on the data over mind camp. Richard Sutton wrote a great blog post in which he claimed that:

> *"The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin. ... Seeking an improvement that makes a difference in the shorter term, researchers seek to leverage their human knowledge of the domain, but the only thing that matters in the long run is the leveraging of computation."*

When asked how Google search was doing so well, Peter Norvig, Google's Director of Search, responded: "*We don't have better algorithms. We just have more data.*"[5]

The debate isn't about whether *finite* data is necessary, but whether it's sufficient. The term *finite* here is important, because if we had infinite data, we can just look up the answer. Having a lot of data is different from having infinite data.

Regardless of which camp is right, data is important. Dr. Monica Rogati argued in "The data science hierarchy of needs[6]" that data lies at the foundation of data science. Without data, there's no data science.

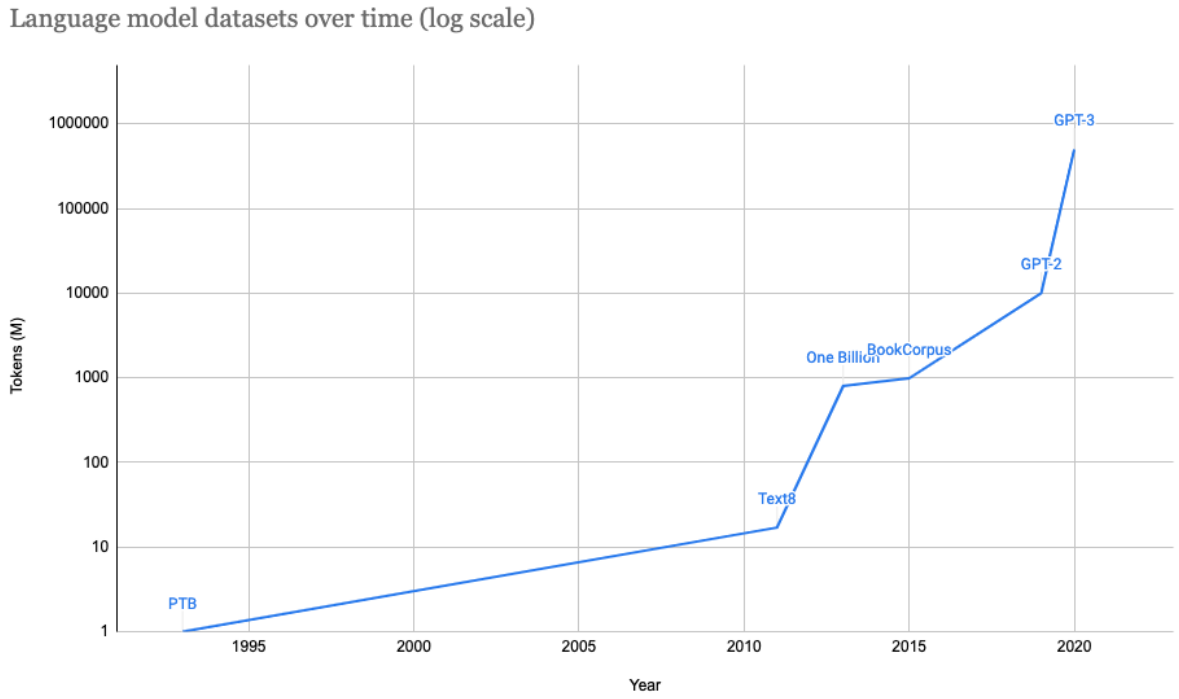


Models are getting bigger and using more data. Back in 2013, people were getting excited when the One Billion Words Benchmark for Language Modeling was released, which contains 0.8

[5] The Unreasonable Effectiveness of Data
[6] The AI Hierarchy of Needs (Monica Rogati, 2017)

billion tokens[7]. Six years later, OpenAI's GPT-2 used a dataset of 10 billion tokens. And another year later, GPT-3 used 500 billion tokens.

Dataset	Year	Tokens (M)
Penn Treebank	1993	1
Text8	2011	17
One Billion	2013	800
BookCorpus	2015	985
GPT-2 (OpenAI)	2019	10,000
GPT-3 (OpenAI)	2020	500,000



[7] https://opensource.google/projects/lm-benchmark

Data engineering 101

Data systems, in and of themselves, are beasts. If you haven't spent years and years digging through literature, it's very easy to get lost in acronyms. There are many challenges and possible solutions—if you look into the data stack for different tech companies it seems like each is doing their own thing.

In this lecture, we'll cover the basics of data engineering. What we cover is very, very basic. If you haven't already, we highly recommend that you take a database class.

Data sources

An ML system works with data from many different sources. One source is **user-generated data** which includes inputs (e.g. phrases to be translated into Google Translate) and clicks (e.g. booking a trip, clicking on or ignoring a suggestion, scrolling). User-generated data can be passive, e.g. user ignoring a popup, spending x seconds on page. Users tend to have little patience, so in general, user-generated data requires fast processing.

Another source **system-generated data** (sometimes called machine-generated data) such as logs, metadata, predictions made by models. Logs are generated to record the state of the system and significant events in the system for bookkeeping and debugging. They can be generated periodically and/or whenever something interesting happens.

Logs provide visibility into how the application is doing, and the main purpose of this visibility is for debugging and possibly improving the application. If you want to be alerted as soon as something abnormal happens on your system, logs should be processed as soon as they're generated.

There's also **enterprise applications data**. A company might use various enterprise applications to manage their assets such as inventory, customer relationship, users. These applications generate data that can be used for ML models. This data can be very large and need to be updated frequently.

Then there's the wonderfully weird and creepy world of **third-party data**. First-party data is the data that your company already collects about your users or customers. Second-party data is the data collected by another company on their own customers. Third-party data companies collect data on the general public who aren't their customers.

The rise of the Internet and smartphones has made it much easier for all types of data to be collected. It's especially easy with smartphones since each phone has a Mobile Advertiser ID, which acts as a unique ID to aggregate all activities on a phone. Data from apps, websites,

check-in services, etc. is collected and (hopefully) anonymized to generate activity history for each person.

You can buy all types of data (e.g. social media activities, purchase history, web browsing habits, car rentals, political leaning) for different demographic groups (e.g. men, age 25-34, working in tech, living in the Bay Area). From this data, you can infer information such as people who like brand A also like brand B.

Third-party data is usually sold as structured data after being cleaned and processed by vendors.

Data formats

Once you have data, you might want to store it. How to store multi-modal data -- e.g. when each sample might contain both images and texts? If you've trained an ML model, how to store it so it can be loaded correctly?

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[8]. The table below consists of just a few of the common formats that you might work with.

Format	**Binary/Text**	**Human-readable?**	**Example use cases**
JSON	Text	Yes	Everywhere
CSV	Text	Yes	Everywhere
Parquet	Binary	No	Hadoop, Amazon Redshift
Avro	Binary primary	No	Hadoop
Protobuf	Binary primary	No	Google, TensorFlow (TFRecord)
Pickle	Text, binary	No	Python, PyTorch serialization

JSON

The format that is ubiquitous everywhere is probably JSON. It's human-readable, language-independent (many programming languages support it), and versatile. Its key-value

[8] Comparison of data-serialization formats (Wikipedia).

pair paradigm allows you to make your data as structured as you want. For example, you can have your data like this:

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

Or you can have it as a blob of text like this:

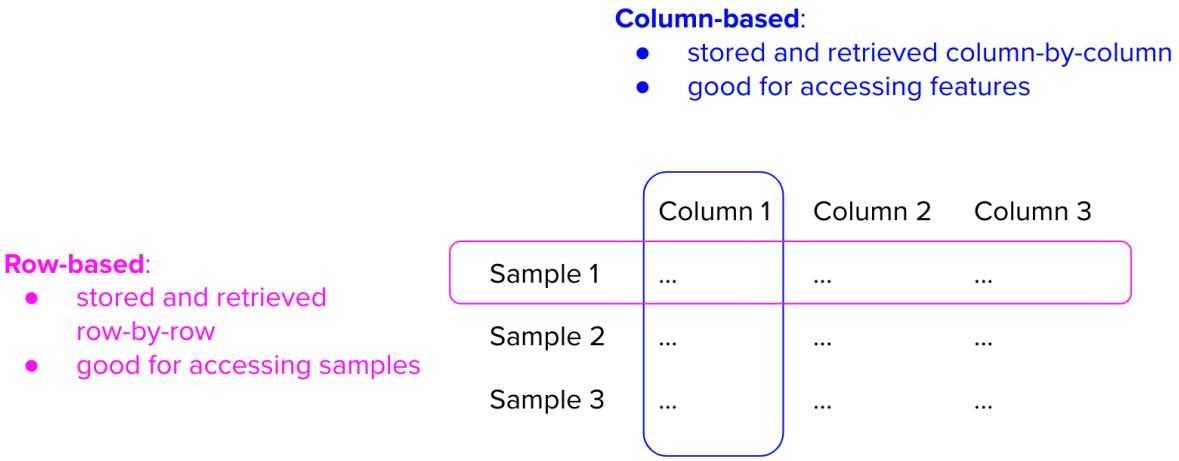
```
{
  "text": "Boatie McBoatFace, aged 12, is vibing, at 12 Ocean Drive,  Port Royal, 10021-3100"
}
```

Row-based vs. column-based

There are two formats that I want to go over in detail: CSV and Parquet. CSV is row-based -- data is stored and retrieved row-by-row. Parquet is column-based -- data is stored and retrieved column by column.

If we consider each sample as a row and each feature as a column:

- The row-based format is better for accessing samples.
    - E.g. accessing a ride-share transaction with all information about that transaction such as time, location, distance, price).
- The column-based format is better for accessing features.
    - E.g. accessing the prices of all the ride-share transactions.



Column-major means consecutive elements in a column are stored next to each other in memory. Row-major means the same but for elements in a row. Because modern computers process sequential data more efficiently than non-sequential data, if a table is row-major, accessing its rows will be much faster than accessing its columns.

Columnar format allows flexible data access. For example, your ride-share data has 1000 columns with all kinds of log/metadata information but you only want data from the 4 columns: time, location, distance, price. With columnar format, you can just read these 4 columns. However, with row-based format, if you don't know the sizes of the rows, you have to read in all columns before filtering down to these 4 columns. Even if you know the sizes of the rows, it can still be slow as you'll have to jump around the memory, unable to take advantage of any caching.

However, the columnar format isn't better than the row-based format for every occasion. It's good for use cases that require a lot of reads with almost no writes. For situations with a lot of writes, e.g. you have many transactions coming continuously and need to write them to file, writing to a row-based format will be much faster than writing to Parquet.

Slightly related: NumPy vs. Pandas

One subtle point that a lot of people don't pay attention to (which leads to misuses of Pandas) is that this library is built around **the columnar format**.

Pandas is built around DataFrame, a concept inspired by R's Data Frame, which is columnar. A DataFrame is a two-dimensional table with rows and columns.

In NumPy, the major order can be specified. When a ndarray is created, it's row-major by default if you don't specify the order. People coming to pandas from NumPy tend to treat DataFrame the way they would ndarray, e.g. trying to access data by rows, and find DataFrame slow.

For example, in this example below, accessing a row of a DataFrame is so much slower than accessing a column.

```
# Iterating pandas DataFrame by column
start = time.time()
for col in df.columns:
    for item in df[col]:
        pass
print(time.time() - start, "seconds")
```

```
0.06656503677368164 seconds
```

```
# Iterating pandas DataFrame by row
n_rows = len(df)
start = time.time()
for i in range(n_rows):
    for item in df.iloc[i]:
        pass
print(time.time() - start, "seconds")
```

```
2.4123919010162354 seconds
```

However, if you convert your DataFrame to NumPy ndarray, accessing a row becomes much faster.[9]

```
df_np = df.to_numpy()
n_rows, n_cols = df_np.shape
```

```
# Iterating NumPy ndarray by column
start = time.time()
for j in range(n_cols):
    for item in df_np[:, j]:
        pass
print(time.time() - start, "seconds")
```

```
0.005830049514770508 seconds
```

```
# Iterating NumPy ndarray by row
start = time.time()
for i in range(n_rows):
    for item in df_np[i]:
        pass
print(time.time() - start, "seconds")
```

```
0.019572019577026367 seconds
```

[9] For more Pandas quirks, check out just-pandas-things.

Text vs. binary format

CSV and JSON are text files whereas Parquet are binary files. Text files are files that are in plain texts, which usually mean they are human-readable. Binary files mean files that are in 0's and 1's, but the term "binary file" is often used as a term meaning "non-text file". If you open text files in your text editors (e.g. VSCode, Notepad), you'll be able to read the texts in them. If you open binary files in your text editors, you'll see blocks of numbers or weird characters.

Binary files are more compact. Consider you want to store the number "1000000". If you store it in a text file, it'll require 7 characters, and if each character is 1 byte, it'll require 7 bytes. If you store it in a binary file as int32, it'll take 32 bits, or 4 bytes.

For example, by converting my `interviews.csv` from text file (CSV) to binary file (Parquet), the file size went from 14MB to 6MB.

```
In [2]: df = pd.read_csv("data/interviews.csv")
        df.info()

        <class 'pandas.core.frame.DataFrame'>
        RangeIndex: 17654 entries, 0 to 17653
        Data columns (total 10 columns):
         #   Column      Non-Null Count  Dtype
        ---  ------      --------------  -----
         0   Company     17654 non-null  object
         1   Title       17654 non-null  object
         2   Job         17654 non-null  object
         3   Level       17654 non-null  object
         4   Date        17652 non-null  object
         5   Upvotes     17654 non-null  int64
         6   Offer       17654 non-null  object
         7   Experience  16365 non-null  float64
         8   Difficulty  16376 non-null  object
         9   Review      17654 non-null  object
        dtypes: float64(1), int64(1), object(8)
        memory usage: 1.3+ MB

In [3]: Path("data/interviews.csv").stat().st_size

Out[3]: 14200063

In [4]: df.to_parquet("data/interviews.parquet")
        Path("data/interviews.parquet").stat().st_size

Out[4]: 6211862
```

AWS recommends using the Parquet format because "*the Parquet format is up to 2x faster to unload and consumes up to 6x less storage in Amazon S3, compared to text formats*."[10]

[10] Announcing Amazon Redshift data lake export: share data in Apache Parquet format.

OLTP (OnLine Transaction Processing) vs. OLAP (OnLine Analytical Processing)

Row-based format is good for processing a lot of incoming transactions, whereas columnar format is good for pulling columns for data analysis. This leads to our next topic: **OnLine Transaction Processing** vs. **OnLine Analytical Processing.**

If your application handles a large amount of short transactions—e.g. ordering food, ordering rideshares, online shopping, money transferring—you might want to use an online transaction processing (OLTP) database to administer your transactions. You want transactions to be processed fast, e.g. in the order of milliseconds.

On top of that, OLTP databases also require:

- **I**solation controls guarantee that two transactions happen at the same time as if they were isolated. Two users accessing the same data won't change it at the same time. For example, you don't want two users to book the same driver at the same time.
- **A**tomicity controls guarantee that all the steps in a transaction are completed successfully as a group. If any steps between the transaction fail, all other steps must fail also. For example, if a user's payment fails, you don't want to still assign a driver to that user.

Atomicity and isolation are part of ACID (Atomicity, Consistency, Isolation, Durability), a standard set of properties that guarantee database transactions are processed reliably. For more details on ACID, take a database class!

OLTP processes online transactions, so most of the operations they do will be inserting, deleting, and updating an existing transaction.

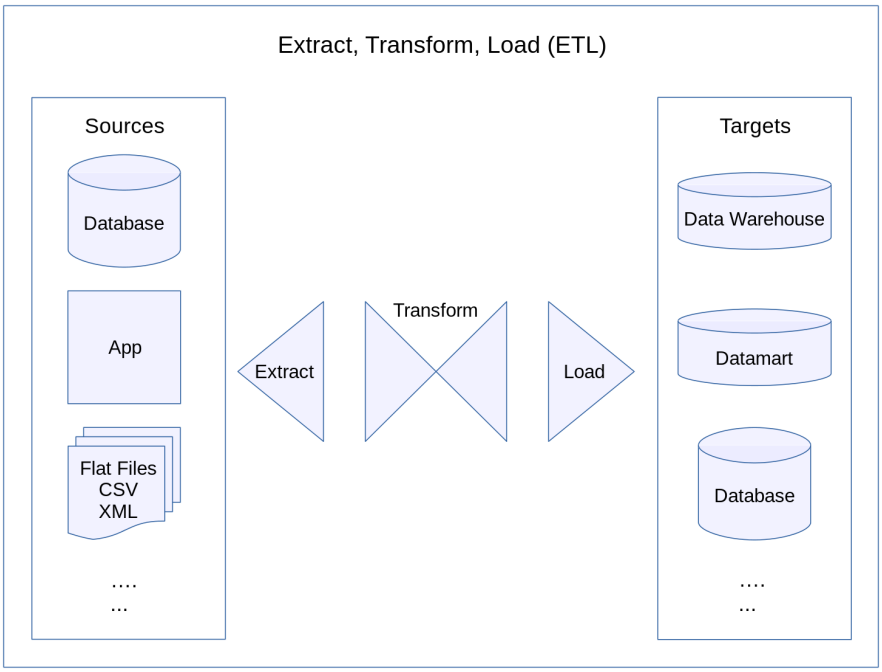
However, OLTP databases aren't efficient for operations that **aggregate** data in columns across multiple rows, e.g. finding the answer to the question "What's the average price for all the rides in September in San Francisco?" For these types of questions, online analytical processing (OLAP) databases would work much better. OLAP databases are designed for queries that allow you to look at data from different viewpoints.

ETL: Extract, Transform, Load

OLTP databases can be processed and aggregated to generate OLAP databases through a process called ETL (extract, transform, load).

- Extract: extracting data from data sources.
- Transform: processing data into the target format.
- Load: loading it into the target destination e.g. a file or a data warehouse.

Transform is the meaty part of the process. This is where you clean and validate your data. You can join data from multiple sources. You can split them into different portions. You can apply operations such as transposing, deduplicating, aggregating, etc.



The idea of ETL sounds simple but powerful, and it's the underlying structure of the data layer at many organizations.

Structured vs. unstructured data

Structured vs. unstructured data is pretty self-explanatory so we'll go over this quickly. "Structured" means that the data's structure is clearly defined, aka following a schema. This makes it easier to search and analyze data.

However, the problems with structured data is that you have to structure it, which means you have to commit to a schema. If your schema changes, you'll have to retrospectively update all your data. For example, you've never tracked transactions' locations before but now you do, so you have to retrospectively update location information to all previous transactions.

Once you've committed to a data schema in a storage, you can only store data that follows the schema in that storage. However, if you keep that data storage unstructured, you can store any type of data (e.g. just convert all your data to bytestrings). A repository for storing structured data (data that has already been processed) is called a data warehouse. A repository for storing unstructured data (also called raw data) is called a data lake.

ETL to ELT

In the early days of big data, when the Internet became ubiquitous and hardware became more powerful, it became easier to collect data and the amount of data grew rapidly. Not only that, the nature of data also changed. Data schema evolved and the number of possible data sources expanded.

Finding it difficult to keep data structured, some companies had this idea: "Why not just store all data in a data lake so we don't have to deal with schema changes? Whichever application needs data can just pull out data from there and process it." This process of loading data into storage first then processing it later is sometimes called ELT (extract, load, transform).

However, as data keeps on growing, it becomes infeasible to just store everything and search in massive data lakes for the piece of data that you want. At the same time, as companies switch to running applications on the cloud and infrastructures become standardized, data structures also become standardized, which makes some companies switch back to storing data in a structured way.

Structured	**Unstructured**
Schema clearly defined	Whatever
Easy to search and analyze	Fast arrival (e.g. no need to clean up first)
Can only handle data with specific schema	Can handle data from any source
Schema changes will cause a lot of trouble	No need to worry about schema changes
Data warehouse	Data lake

Batch processing vs. stream processing

Most ETL jobs are batch jobs -- data is processed in batches. To process data and return results as soon as each sample arrives is called stream processing. Stream processing enables online prediction as we learned in the previous lesson.

During the traditional ML model development, the data we work with is historical data -- data that already exists in some storage formats (e.g. files, databases). We want to train our model on many samples, and one way to fasten the training process is to train on multiple samples at the same time. We care less about how long it takes for the model to process a single sample, and more about how long it takes for the model to process all samples.

During inference, if your application is user-facing, you care about how long it takes for your system to process each individual sample. To be able to do online prediction with this latency constraint, your system needs two components:

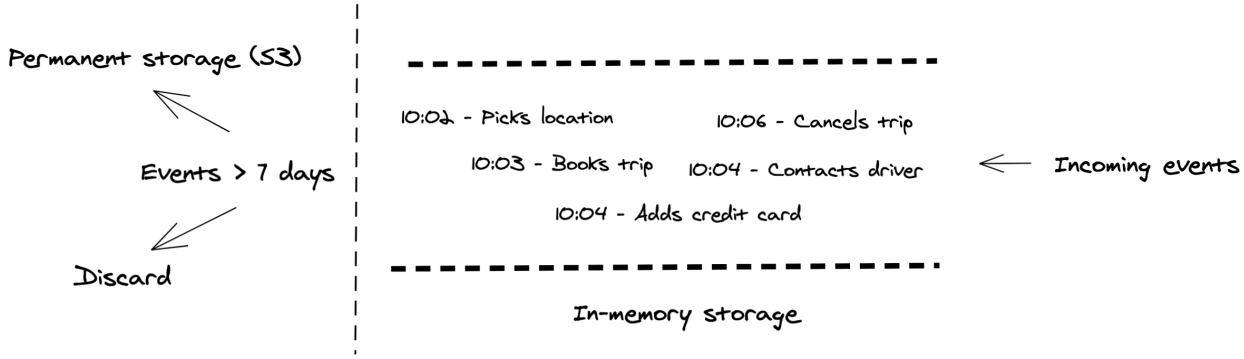
- **Fast inference**: model that can make predictions in the order of milliseconds.
- **Real-time pipeline**: a pipeline that can process data, input it into model, and return a prediction in real-time.

We'll cover fast inference in a future lecture. In this lecture, we'll cover the real-time pipeline. A popular real-time pipeline is the one that leverages **stream processing**.

Suppose you have a ride sharing app and want to detect fraudulent transactions e.g. payments using stolen credit cards. When the true credit owner discovers unauthorized payments, they'll dispute with their bank and you'll have to refund the charges. To maximize profits, fraudsters might call multiple rides either in succession or from multiple accounts. In 2019, merchants estimate fraudulent transactions account for an average of 27% of their annual online sales[11]. The longer it takes for you to detect the stolen credit card, the more money you'll lose.

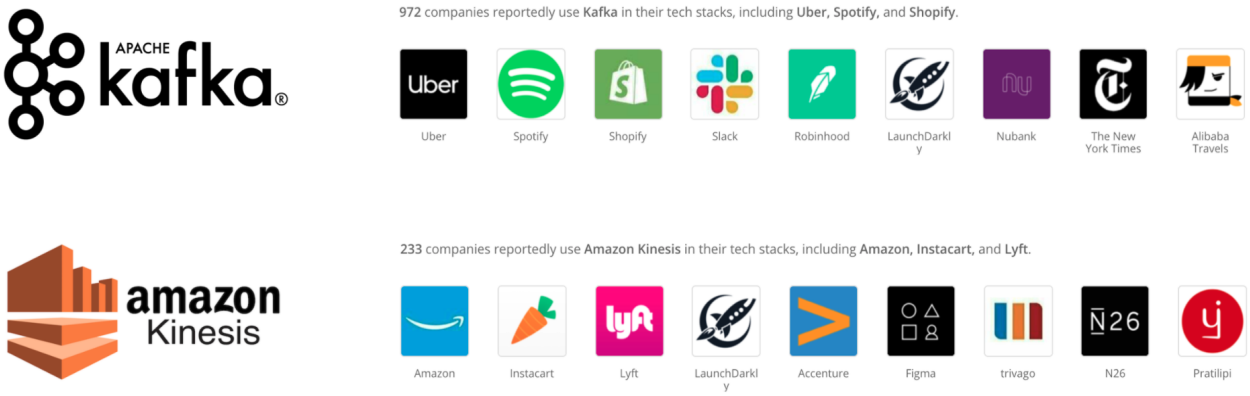
To detect whether a transaction is fraudulent, looking at that transaction alone isn't enough. You need to at least look into the **recent** history of the user involved in that transaction, their **recent** trips and activities in-app, the credit card's **recent** transactions, and other transactions happening around the same time.

To quickly access these types of information, you want to keep as much of them in-memory as possible. Every time an event you care about happens -- a user choosing a location, booking a trip, contacting a driver, canceling a trip, adding a credit card, removing a credit card, etc. -- information about that event goes into your in-memory storage. It stays there for as long as they are useful (usually in order of days) then either goes into permanent storage (e.g. S3) or is discarded.



[11] American Express Insights 2019 Digital Payments Survey. 2019.

The most common tool for this is Apache Kafka, with alternatives such as Amazon Kinesis. Kafka is a **stream storage**: it stores data as it streams.



Companies that use Apache Kafka and Amazon Kinesis by Stackshare

Once you've had a way to manage streaming data, you want to extract features to input into your ML models. Features from dynamic data are called dynamic features. They are based on what's happening right now -- what you're watching, what you've just liked, etc. Knowing a user's interests right now will allow your systems to make recommendations much more relevant to them.

On top of features from streaming data, you might also need features from static data (when was this account created, what's the user's rating, etc.). You need a tool that allows you to process streaming data as well as static data and join them together from various data sources.

People generally use "batch processing" to refer to static data processing because you can process them in batches. This is opposed to "stream processing", which processes each event as it arrives. Batch processing is efficient -- you can leverage tools like MapReduce to process large amounts of data. Stream processing is low latency because you can process each piece of data as soon as it comes.

Processing stream data is more difficult because the data amount is unbounded and the data comes in at variable rates and speeds. It's easier to make a stream processor do batch processing than making a batch processor do stream processing.

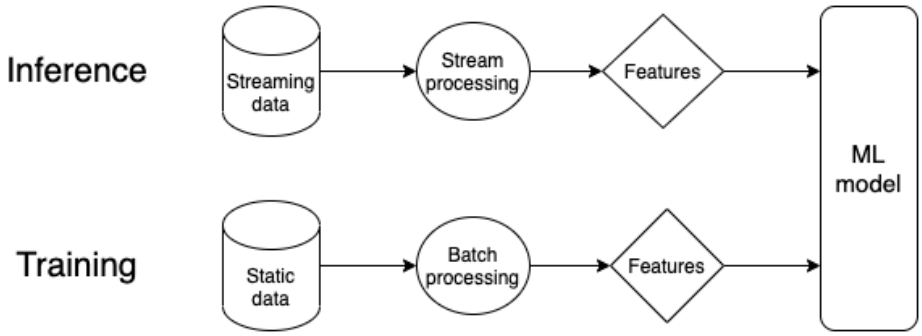
Apache Kafka has some capacity for stream processing and some companies use this capacity on top of their Kafka stream storage, but Kafka stream processing is limited in its ability to deal with various data sources. There have been efforts to extend SQL, the popular query language

intended for static data tables, to handle data streams[12][13]. However, the most popular tool for stream processing is Apache Flink, with native support for batch processing.

In the early days of machine learning production, many companies built their ML systems on top of their existing MapReduce/Spark/Hadoop data pipeline. When these companies want to do real-time inference, they need to build a separate pipeline for streaming data.

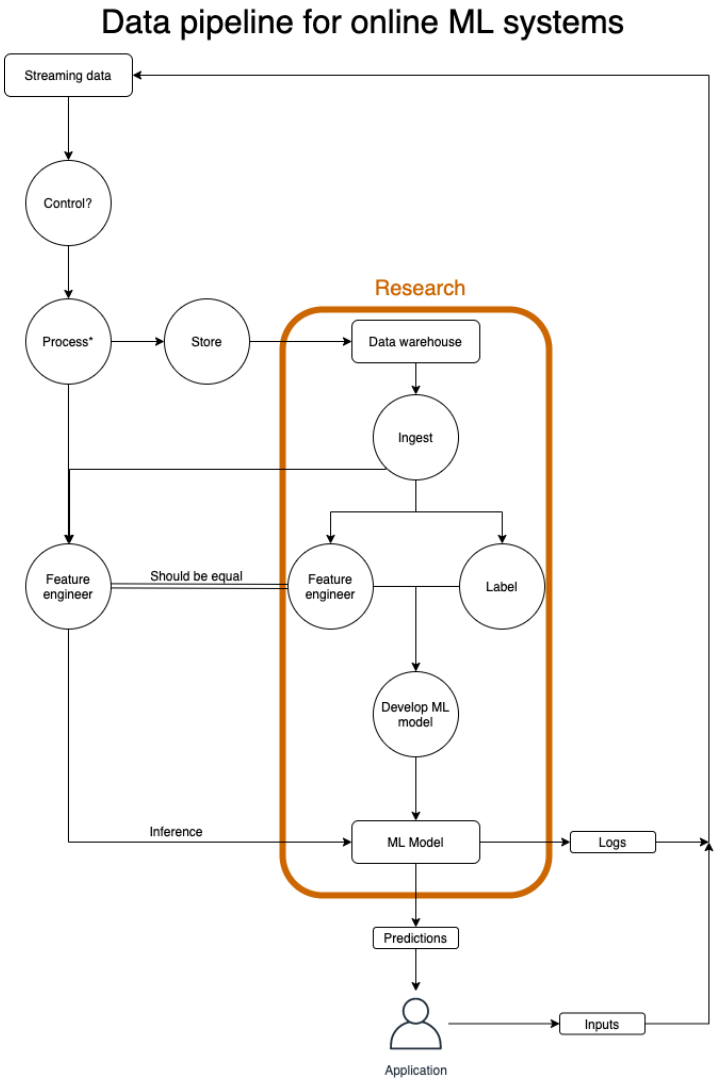
⚠ **Having two different pipelines to process your data is a common cause for bugs in ML production** ⚠

For example, the changes in one pipeline aren't correctly replicated in the other leading to two pipelines extracting two different sets of features. This is especially common if the two pipelines are maintained by two different teams, e.g. the development team maintains the batch pipeline for training while the deployment team maintains the stream pipeline for inference.



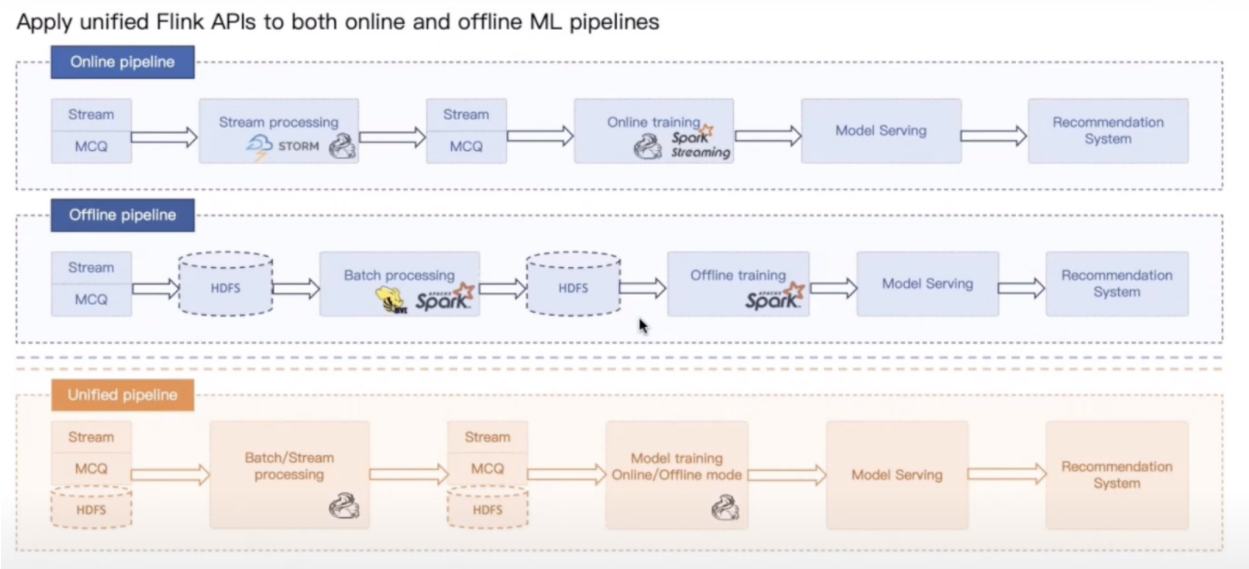
Here is a more detailed but also more complex feature of the data pipeline for ML systems that do online prediction. The part in the orange retangular is what people are often exposed to in an academic environment.

[12] http://cs.brown.edu/~ugur/streamsql.pdf
[13] https://en.wikipedia.org/wiki/StreamSQL



Companies including Uber and Weibo have made major infrastructure overhaul to unify their batch and stream processing pipelines with Flink[14][15].

[14] https://www.infoq.com/presentations/sql-streaming-apache-flink/
[15] https://www.youtube.com/watch?v=WQ520rWgd9A&ab_channel=FlinkForward



🌳 **Resources** 🌳

Here are a few resources if you want to learn more about how companies manage their data pipeline.

- Uber's Big Data Platform: 100+ Petabytes with Minute Latency (Reza Shiftehfar, Uber Engineering blog 2018)
- Keystone Real-time Stream Processing Platform (Zhenzhong Xu, Netflix Technology Blog 2018)
- A Beginner's Guide to Data Engineering (Robert Chang, 2018)

Creating training datasets

Once we've collected and transformed our data, we might use it to create training datasets for our ML models. A word of caution before we move forward. Use data but don't trust data too much. It's full of potentials for biases, which can come from any of the following sources:

- sampling/selection biases
- under/over-representation of subgroups
- human biases embedded in historical data
- labeling biases
- & more

Labeling

Despite the promise of unsupervised ML, most ML models today still need labels to learn. The performance of an ML model depends heavily on the quality and quantity of labels it's trained on.

The challenges of hand labels

Acquiring hand labels for your data is challenging.
First, **hand-labeling data can be expensive, especially if subject matter expertise is required**. For example, to classify whether a comment is spam, you might be able to quickly find 200 annotators on a crowdsourcing platform. However, if you want to label chest X-rays, you'd need to find board-certified radiologists, whose time can be limited and very expensive.

Second, **hand labeling means that someone has to look at your data, which isn't always possible if your data has strict privacy requirements**. For example, you can't just ship your patient's medical records or your company's confidential financial information to anyone. In many cases, the data isn't allowed to leave your organization, which means the annotators have to be brought on premise.

Third, **hand labeling is slow as it's on a linear scale**. While the more data you label, the faster your speed tends to be, the improvement isn't in orders of magnitude. Labeling 1000 samples takes approximately 10 times longer than labeling 100 samples.

Last but not least, **hand labels aren't adaptive**. If the task changes or data changes, you'll have to relabel your data. For example, you've been using two classes NEGATIVE and POSITIVE for your sentiment analysis task, but now you want to use three classes NEGATIVE, POSITIVE, and ANGRY (since you want to attend to messages with angry sentiment faster), you will need to look at your data again to see which existing NEGATIVE labels should be changed to ANGRY.

Label multiplicity

To obtain a large quantity of labels, companies often have to use data from multiple sources and rely on multiple annotators. These different data sources and annotators might have different levels of accuracy. Using them indiscriminately without examining them can cause your model to fail mysteriously.

⚠ More data isn't always better ⚠

Consider a case when you've trained a moderately good model with 100K data samples. Your ML engineers are confident that more data will improve the model performance, so you spend a lot of money to hire annotators to label 1 million data samples.

However, the model performance actually decreases after being trained on the new data. The reason is that the new million samples were crowdsourced to annotators who labeled data with much less accuracy than the original data. It can be especially difficult to remedy this if you've already mixed your data and can't differentiate new data from old data.

Another problem with multiple annotators with different levels of expertise is label ambiguity. Consider this simple task of entity recognition. You give three annotators the following sample to annotate, and receive back three different solutions.

Darth Sidious, known simply as the Emperor, was a Dark Lord of the Sith who reigned over the galaxy as Galactic Emperor of the First Galactic Empire.

Annotator	# entities	Annotation
1	3	[**Darth Sidious**], known simply as the Emperor, was a [**Dark Lord of the Sith**] who reigned over the galaxy as [**Galactic Emperor of the First Galactic Empire**]
2	6	[**Darth Sidious**], known simply as the [**Emperor]**, was a **[Dark Lord**] of the [**Sith**] who reigned over the galaxy as [**Galactic Emperor**] of the [**First Galactic Empire**].
3	4	[**Darth Sidious**], known simply as the [**Emperor]**, was a **[Dark Lord of the Sith**] who reigned over the galaxy as [**Galactic Emperor of the First Galactic Empire**].

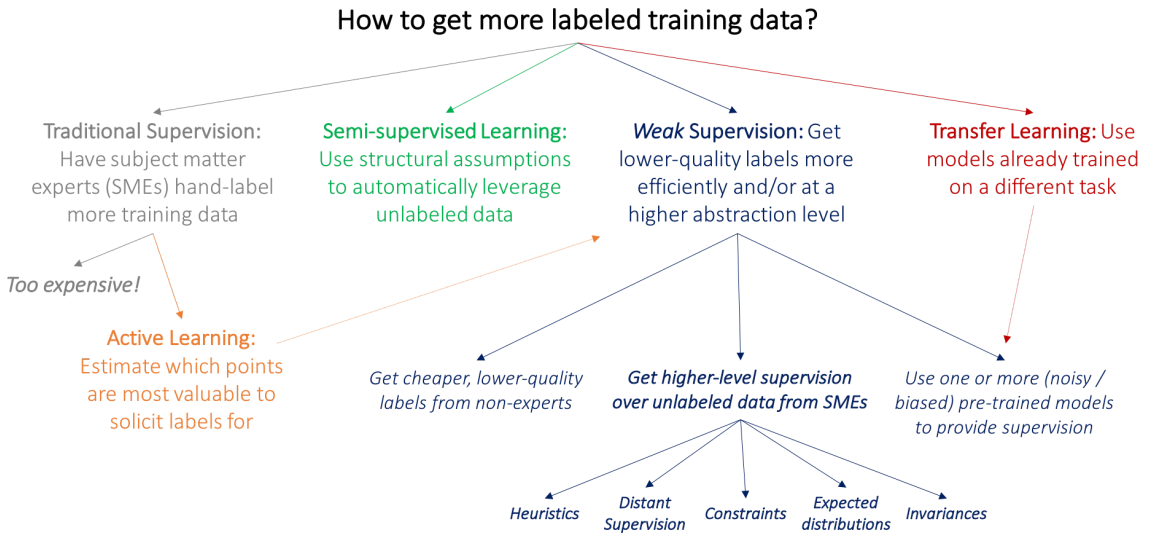
Annotating disagreement can be especially common among tasks that require a high level of domain expertise. One human-expert thinks the label should be A while another believes it should be B -- how do we resolve this conflict to obtain one single ground truth? If human experts can't agree on a label, then what does human-level performance even mean?

Solution to label multiplicity

To minimize the disagreement among annotators, it's important to have a clear problem definition. For example, in the entity recognition task above, some disagreement could have been eliminated if we clarify that in case of multiple possible entities, pick the entity that comprises the longest substring. This means **Galactic Emperor of the First Galactic Empire** instead of **Galactic Emperor** and **First Galactic Empire**. It's also important to have training with anticipated difficult cases to make sure that all annotators understand the rules.

On top of that, we need to always keep track of where each of our data samples comes from. For example, in the example above, you might see that your new trained model fails mostly on the recently acquired data samples, and when you look into the wrong predictions, you might see that it's because the recently acquired data samples have wrong labels.

How to deal with the lack of labels



16

Weak supervision

If hand labeling is so problematic, what if we don't use hand labels altogether? One approach that has gained popularity is weak supervision with tools such as Snorkel[17][18]. The core idea of Snorkel is labeling function: a function that encodes subject matter expertise. People often rely on heuristics to label data. For example, a doctor might use this heuristics to decide whether a patient's case should be prioritized as emergent.

If the nurse's note mentions serious conditions like pneumonia, the patient's case should be given priority consideration.

We can encode this as a function.

```
def labeling_function(note):
    if "pneumonia" in note:
        return "EMERGENT"
```

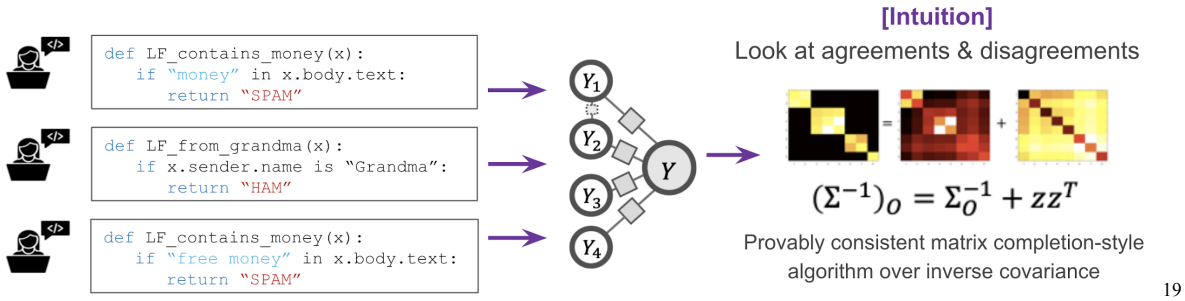
Labeling functions can encode many different types of heuristics, from keyword heuristic as above, to regular expressions, to database lookup (e.g. if the note contains the disease listed in the dangerous disease list), to other models (e.g. if an existing system classifies this as EMERGENT).

---

[16] Weak Supervision: A New Programming Paradigm for Machine Learning (Ratner et al., 2019)
[17] https://www.snorkel.org/
[18] Disclaimer: I was one of Snorkel AI's early engineers.

The labeling functions can then be applied to the data you want to label.

Labeling function is a simple but powerful concept. However, labeling functions are noisy. They can conflict with each other. They might only cover a proportion of your data (e.g. only 20% of your data is covered by at least one labeling function). It's important to combine, denoise, and reweight all labeling functions to get a set of most likely-to-be-correct labels.



19

In this way, subject matter expertise can be versioned, reused, and shared. If your data changes, you can just reapply labeling functions to all your data samples. If your data has privacy requirements, you only need to see a small, cleared subset of your data to write labeling functions, and apply the labeling functions to the rest of your data without looking at it.

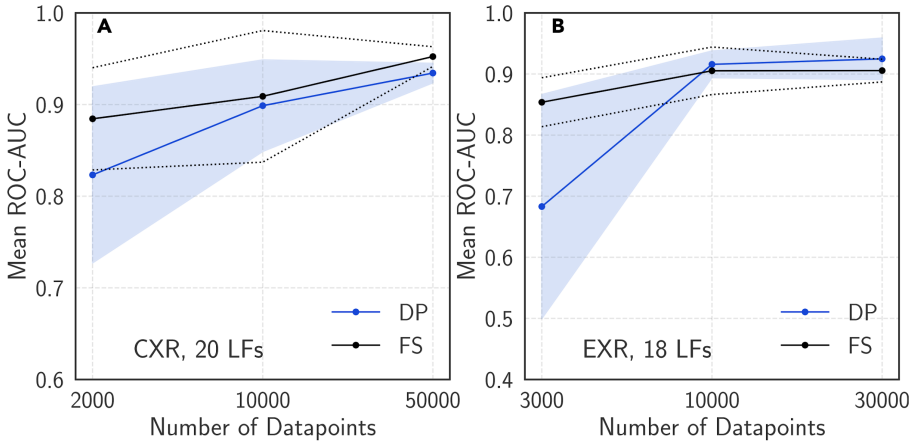
Hand labeling	**Programmatic labeling**
Expensive: esp. when subject matter expertise required	**Cost saving**: Expertise can be versioned, shared, reused across organization
Non-private: Need to ship data to human annotators	**Privacy**: Create LFs using a cleared data subsample then apply LFs to other data without looking at individual samples.
Slow: Time required scales linearly with # labels needed	**Fast**: Easily scale 1K -> 1M samples
Non-adaptive: Every change requires re-labeling the dataset	**Adaptive**: When changes happen, just reapply LFs!

In a case study with Stanford Medicine, models trained with weakly-supervised labels obtained by a single radiologist after 8 hours on writing labeling functions had comparable performance with models trained on data obtained through almost a year of hand labeling. Two notes about the experiment:

- The performance was improving with more unlabeled data without added labeling function.

[19] Snorkel: Rapid Training Data Creation with Weak Supervision (Ratner et al., 2017)

- They were able to reuse 6 labeling functions between the CXR (Chest X-Rays) task and EXR (Extremity X-Rays) task.
-



DP denotes model trained with programmatic labels, and FS is fully-supervised[20]

Weak supervision is only one of many ways to deal with the lack of data labels. Here are other methods that have gained popularity in the last few years.

Semi-supervised

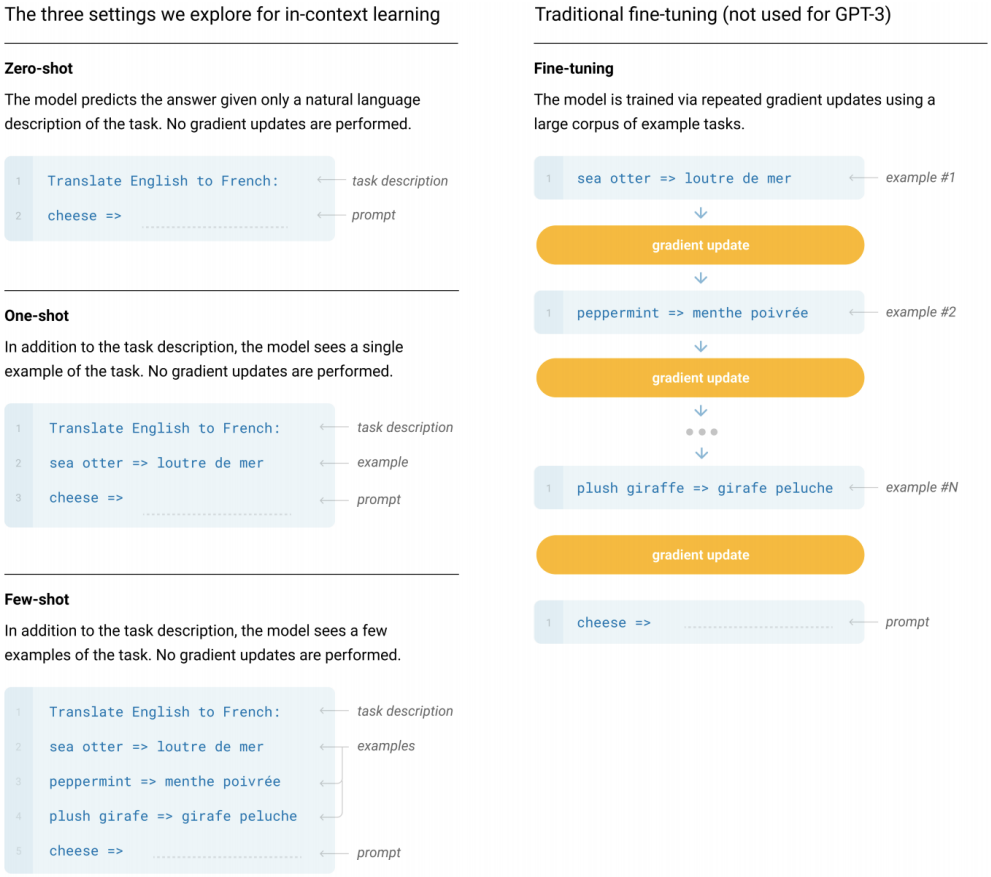
Start with a small set of labels and use structural assumptions to generate more labels. For example, for the task of classify the topic of Twitter hashtags, you can start by labeling the hashtag #AI as Computer Science. Assuming that hashtags that appear in the same tweet or profile are likely about the same topic, you can also label #ML, #BigData as Computer Science.

Transfer learning

Apply the model trained for one task to another task. It might or might not require fine-tuning (gradient updates). For example, you can fine-tune a model trained on language modeling to make it do sentiment analysis. One of the reasons for the recent excitement around large language models like BERT or GPT-3 is because you can fine-tune them to use on tasks with less labeled data.

---

[20] Cross-Modal Data Programming Enables Rapid Medical Machine Learning (Dunnmon et al., 2020)



21

Active learning

Active learning still requires hand labels, but instead of randomly labeling a subset of data, you label the samples that are most helpful to your model.

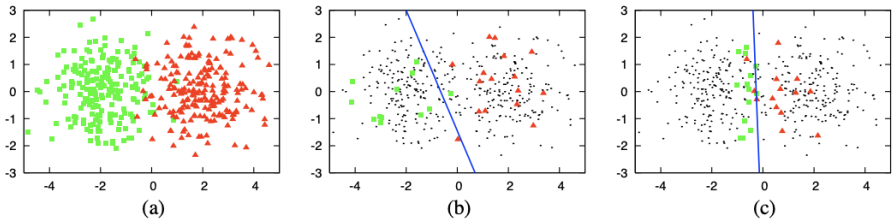


Figure 2: An illustrative example of pool-based active learning. (a) A toy data set of 400 instances, evenly sampled from two class Gaussians. The instances are represented as points in a 2D feature space. (b) A logistic regression model trained with 30 labeled instances randomly drawn from the problem domain. The line represents the decision boundary of the classifier (70% accuracy). (c) A logistic regression model trained with 30 actively queried instances using uncertainty sampling (90%).

22

[21] Language Models are Few-Shot Learners (OpenAI 2020)
[22] Active Learning Literature Survey (Burr Settles, 2010)

==========

To cite this lecture note, please use:

```
@booklet{cs329s_lectures,
  title  = {CS 329S: Machine learning systems design},
  author = {Chip Huyen},
  url    = {https://cs329s.stanford.edu},
  year   = {2021},
  note   = {Lecture notes}
}
```