

# Data Engineering Fundamentals

---

The rise of ML in recent years is tightly coupled with the rise of big data. Large data systems, even without ML, are complex. If you haven't spent years and years working with them, it's easy to get lost in acronyms. There are many challenges and possible solutions that these systems generate. Industry standards, if there are any, evolve quickly as new tools come out and the needs of the industry expand, creating a dynamic and ever-changing environment. If you look into the data stack for different tech companies, it might seem like each is doing its own thing.

In this chapter, we'll cover the basics of data engineering that will, hopefully, give you a steady piece of land to stand on as you explore the landscape for your own needs. We'll start with different sources of data that you might work with in a typical ML project. We'll continue to discuss the formats in which data can be stored. Storing data is only interesting if you intend on retrieving that data later. To retrieve stored data, it's important to know not only how it's formatted but also how it's structured. Data models define how the data stored in a particular data format is structured.

If data models describe the data in the real world, databases specify how the data should be stored on machines. We'll continue to discuss data storage engines, also known as databases, for the two major types of processing: transactional and analytical.

When working with data in production, you usually work with data across multiple processes and services. For example, you might have a feature engineering service that computes features from raw data, and a prediction service to generate predictions based on computed features. This means that you'll have to pass computed features from the feature engineering service to the prediction service. In the following section of the chapter, we'll discuss different modes of data passing across processes.

During the discussion of different modes of data passing, we'll learn about two distinct types of data: historical data in data storage engines, and streaming data in real-time transports. These two different types of data require different processing paradigms, which we'll discuss in the section “[Batch Processing Versus Stream Processing](#)” on page 78.

Knowing how to collect, process, store, retrieve, and process an increasingly growing amount of data is essential to people who want to build ML systems in production. If you're already familiar with data systems, you might want to move directly to [Chapter 4](#) to learn more about how to sample and generate labels to create training data. If you want to learn more about data engineering from a systems perspective, I recommend Martin Kleppmann's excellent book *Designing Data-Intensive Applications* (O'Reilly, 2017).

## Data Sources

An ML system can work with data from many 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. This section aims to give a quick overview of different data sources to those unfamiliar with data in production. If you've already worked with ML in production for a while, feel free to skip this section.

One source is *user input data*, data explicitly input by users. User input can be text, images, videos, uploaded files, etc. If it's even remotely possible for users to input wrong data, they are going to do it. As a result, user input data can be easily malformed. Text might be too long or too short. Where numerical values are expected, users might accidentally enter text. If you let users upload files, they might upload files in the wrong formats. User input data requires more heavy-duty checking and processing.

On top of that, users also have little patience. In most cases, when we input data, we expect to get results back immediately. Therefore, user input data tends to require fast processing.

Another source is *system-generated data*. This is the data generated by different components of your systems, which include various types of logs and system outputs such as model predictions.

Logs can record the state and significant events of the system, such as memory usage, number of instances, services called, packages used, etc. They can record the results of different jobs, including large batch jobs for data processing and model training. These types of logs provide visibility into how the system is doing. The main purpose of this visibility is for debugging and potentially improving the application. Most of the time, you don't have to look at these types of logs, but they are essential when something is on fire.

Because logs are system generated, they are much less likely to be malformed the way user input data is. Overall, logs don't need to be processed as soon as they arrive, the way you would want to process user input data. For many use cases, it's acceptable to process logs periodically, such as hourly or even daily. However, you might still want to process your logs fast to be able to detect and be notified whenever something interesting happens.<sup>1</sup>

Because debugging ML systems is hard, it's a common practice to log everything you can. This means that your volume of logs can grow very, very quickly. This leads to two problems. The first is that it can be hard to know where to look because signals are lost in the noise. There have been many services that process and analyze logs, such as Logstash, Datadog, Logz.io, etc. Many of them use ML models to help you process and make sense of your massive number of logs.

The second problem is how to store a rapidly growing number of logs. Luckily, in most cases, you only have to store logs for as long as they are useful and can discard them when they are no longer relevant for you to debug your current system. If you don't have to access your logs frequently, they can also be stored in low-access storage that costs much less than higher-frequency-access storage.<sup>2</sup>

The system also generates data to record users' behaviors, such as clicking, choosing a suggestion, scrolling, zooming, ignoring a pop-up, or spending an unusual amount of time on certain pages. Even though this is system-generated data, it's still considered part of user data and might be subject to privacy regulations.<sup>3</sup>

---

1 "Interesting" in production usually means catastrophic, such as a crash or when your cloud bill hits an astronomical amount.

2 As of November 2021, AWS S3 Standard, the storage option that allows you to access your data with the latency of milliseconds, costs about five times more per GB than S3 Glacier, the storage option that allows you to retrieve your data with a latency from between 1 minute to 12 hours.

3 An ML engineer once mentioned to me that his team only used users' historical product browsing and purchases to make recommendations on what they might like to see next. I responded: "So you don't use personal data at all?" He looked at me, confused. "If you meant demographic data like users' age, location, then no, we don't. But I'd say that a person's browsing and purchasing activities are extremely personal."

There are also *internal databases*, generated by various services and enterprise applications in a company. These databases manage their assets such as inventory, customer relationship, users, and more. This kind of data can be used by ML models directly or by various components of an ML system. For example, when users enter a search query on Amazon, one or more ML models process that query to detect its intention—if someone types in “frozen,” are they looking for frozen foods or Disney’s *Frozen* franchise?—then Amazon needs to check its internal databases for the availability of these products before ranking them and showing them to users.

Then there’s the wonderfully weird 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 that they make available to you, though you’ll probably have to pay for it. Third-party data companies collect data on the public who aren’t their direct customers.

The rise of the internet and smartphones has made it much easier for all types of data to be collected. It used to be especially easy with smartphones since each phone used to have a unique advertiser ID—iPhones with Apple’s Identifier for Advertisers (IDFA) and Android phones with their Android Advertising ID (AAID)—which acted as a unique ID to aggregate all activities on a phone. Data from apps, websites, check-in services, etc. are collected and (hopefully) anonymized to generate activity history for each person.

Data of all kinds can be bought, such as social media activities, purchase history, web browsing habits, car rentals, and political leaning for different demographic groups getting as granular as 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. This data can be especially helpful for systems such as recommender systems to generate results relevant to users’ interests. Third-party data is usually sold after being cleaned and processed by vendors.

However, as users demand more data privacy, companies have been taking steps to curb the usage of advertiser IDs. In early 2021, Apple made their IDFA opt-in. This change has reduced significantly the amount of third-party data available on iPhones, forcing many companies to focus more on first-party data.<sup>4</sup> To fight back this change, advertisers have been investing in workarounds. For example, the China Advertising Association, a state-supported trade association for China’s advertising industry, invested in a device fingerprinting system called CAID that allowed apps like TikTok and Tencent to keep tracking iPhone users.<sup>5</sup>

---

4 John Koetsier, “Apple Just Crippled IDFA, Sending an \$80 Billion Industry Into Upheaval,” *Forbes*, June 24, 2020, <https://oreil.ly/rqPX9>.

5 Patrick McGee and Yuan Yang, “TikTok Wants to Keep Tracking iPhone Users with State-Backed Workaround,” *Ars Technica*, March 16, 2021, <https://oreil.ly/54pkg>.

# Data Formats

Once you have data, you might want to store it (or “persist” it, in technical terms). Since your data comes from multiple sources with different access patterns,<sup>6</sup> storing your data isn’t always straightforward and, for some cases, can be costly. It’s important to think about how the data will be used in the future so that the format you use will make sense. Here are some of the questions you might want to consider:

- How do I store multimodal data, e.g., a sample that might contain both images and texts?
- Where do I store my data so that it’s cheap and still fast to access?
- How do I store complex models so that they can be loaded and run correctly on different hardware?

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. [Table 3-1](#) consists of just a few of the common formats that you might encounter in your work. For a more comprehensive list, check out the wonderful Wikipedia page [“Comparison of Data-Serialization Formats”](#).

Table 3-1. Common data formats and where they are used

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	Binary	No	Python, PyTorch serialization

We’ll go over a few of these formats, starting with JSON. We’ll also go over the two formats that are common and represent two distinct paradigms: CSV and Parquet.

<sup>6</sup> “Access pattern” means the pattern in which a system or program reads or writes data.

## JSON

JSON, JavaScript Object Notation, is everywhere. Even though it was derived from JavaScript, it's language-independent—most modern programming languages 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. For example, your data can be stored in a structured format like the following:

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

The same data can also be stored in an unstructured blob of text like the following:

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

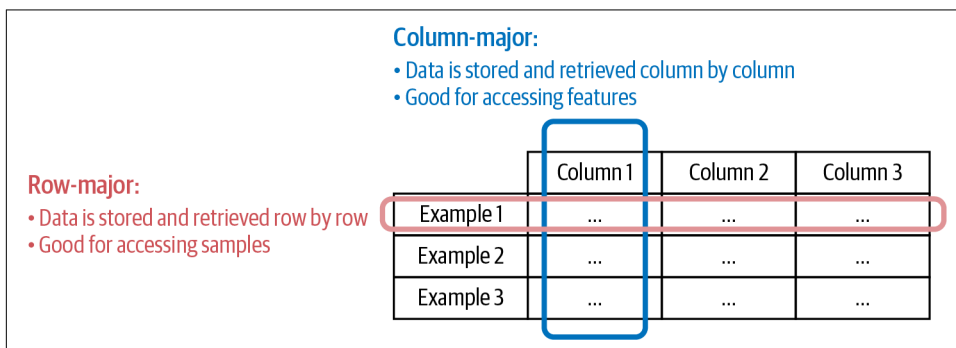
Because JSON is ubiquitous, the pain it causes can also be felt everywhere. Once you've committed the data in your JSON files to a schema, it's pretty painful to retrospectively go back to change the schema. JSON files are text files, which means they take up a lot of space, as we'll see in the section [“Text Versus Binary Format” on page 57](#).

## Row-Major Versus Column-Major Format

The two formats that are common and represent two distinct paradigms are CSV and Parquet. CSV (comma-separated values) is 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.

Because modern computers process sequential data more efficiently than nonsequential data, if a table is row-major, accessing its rows will be faster than accessing its columns in expectation. This means that for row-major formats, accessing data by rows is expected to be faster than accessing data by columns.

Imagine we have a dataset of 1,000 examples, and each example has 10 features. If we consider each example as a row and each feature as a column, as is often the case in ML, then the row-major formats like CSV are better for accessing examples, e.g., accessing all the examples collected today. Column-major formats like Parquet are better for accessing features, e.g., accessing the timestamps of all your examples. See [Figure 3-1](#).



*Figure 3-1. Row-major versus column-major formats*

Column-major formats allow flexible column-based reads, especially if your data is large with thousands, if not millions, of features. Consider if you have data about ride-sharing transactions that has 1,000 features but you only want 4 features: time, location, distance, price. With column-major formats, you can read the four columns corresponding to these four features directly. However, with row-major formats, if you don't know the sizes of the rows, you will have to read in all columns then filter down to these four 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 caching.

Row-major formats allow faster data writes. Consider the situation when you have to keep adding new individual examples to your data. For each individual example, it'd be much faster to write it to a file where your data is already in a row-major format.

Overall, row-major formats are better when you have to do a lot of writes, whereas column-major ones are better when you have to do a lot of column-based reads.

## NumPy Versus 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 column-major. A DataFrame is a two-dimensional table with rows and columns.

In NumPy, the major order can be specified. When an 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.

In the left panel of [Figure 3-2](#), you can see that accessing a DataFrame by row is so much slower than accessing the same DataFrame by column. If you convert this same DataFrame to a NumPy ndarray, accessing a row becomes much faster, as you can see in the right panel of the figure.<sup>7</sup>

```
# 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

```
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

Figure 3-2. (Left) Iterating a pandas DataFrame by column takes 0.07 seconds but iterating the same DataFrame by row takes 2.41 seconds. (Right) When you convert the same DataFrame into a NumPy ndarray, accessing its rows becomes much faster.

<sup>7</sup> For more pandas quirks, check out my [Just pandas Things](#) GitHub repository.





I use CSV as an example of the row-major format because it's popular and generally recognizable by everyone I've talked to in tech. However, some of the early reviewers of this book pointed out that they believe CSV to be a horrible data format. It serializes nontext characters poorly. For example, when you write float values to a CSV file, some precision might be lost—0.12345678901232323 could be arbitrarily rounded up as “0.12345678901”—as complained about in a [Stack Overflow thread](#) and [Microsoft Community thread](#). People on [Hacker News](#) have passionately argued against using CSV.

## Text Versus Binary Format

CSV and JSON are text files, whereas Parquet files are binary files. Text files are files that are in plain text, which usually means they are human-readable. Binary files are the catchall that refers to all nontext files. As the name suggests, binary files are typically files that contain only 0s and 1s, and are meant to be read or used by programs that know how to interpret the raw bytes. A program has to know exactly how the data inside the binary file is laid out to make use of the file. If you open text files in your text editor (e.g., VS Code, Notepad), you'll be able to read the texts in them. If you open a binary file in your text editor, you'll see blocks of numbers, likely in hexadecimal values, for corresponding bytes of the file.

Binary files are more compact. Here's a simple example to show how binary files can save space compared to text files. Consider that 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 only 32 bits or 4 bytes.

As an illustration, I use *interviews.csv*, which is a CSV file (text format) of 17,654 rows and 10 columns. When I converted it to a binary format (Parquet), the file size went from 14 MB to 6 MB, as shown in [Figure 3-3](#).

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.”<sup>8</sup>

---

<sup>8</sup> “Announcing Amazon Redshift Data Lake Export: Share Data in Apache Parquet Format,” Amazon AWS, December 3, 2019, <https://oreil.ly/iLD66>.

```
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
```

Figure 3-3. When stored in CSV format, my interview file is 14 MB. But when stored in Parquet, the same file is 6 MB.

## Data Models

Data models describe how data is represented. Consider cars in the real world. In a database, a car can be described using its make, its model, its year, its color, and its price. 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 way you represent cars in the first data model makes it easier for people looking to buy cars, whereas the second data model makes it easier for police officers to track down criminals.

In this section, we'll study two types of models that seem opposite to each other but are actually converging: relational models and NoSQL models. We'll go over examples to show the types of problems each model is suited for.

## Relational Model

Relational models are among the most persistent ideas in computer science. Invented by Edgar F. Codd in 1970,<sup>9</sup> the relational model is still going strong today, even getting more popular. The idea is simple but powerful. 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,<sup>10</sup> as shown in [Figure 3-4](#). Relations are unordered. You can shuffle the order of the rows or the order of 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.

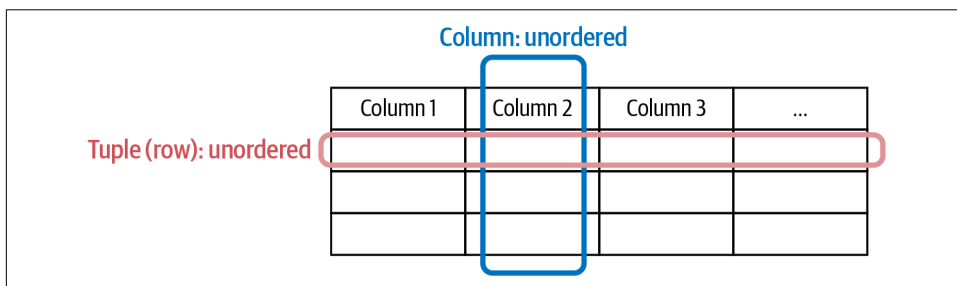


Figure 3-4. In a relation, the order of neither the rows nor the columns matters

It's often desirable for relations to be normalized. Data normalization can follow normal forms such as the first normal form (1NF), second normal form (2NF), etc., and readers interested can read more about it on [Wikipedia](#). In this book, we'll go through an example to show how normalization works and how it can reduce data redundancy and improve data integrity.

Consider the relation *Book* shown in [Table 3-2](#). There are a lot of duplicates in this data. For example, rows 1 and 2 are nearly identical, except for format and price. If the publisher information changes—for example, its name changes from “Banana

<sup>9</sup> Edgar F. Codd, “A Relational Model of Data for Large Shared Data Banks,” *Communications of the ACM* 13, no. 6 (June 1970): 377–87.

<sup>10</sup> For detail-oriented readers, not all tables are relations.

Press” to “Pineapple Press”—or its country changes, we’ll have to update rows 1, 2, and 4. If we separate publisher information into its own table, as shown in Tables 3-3 and 3-4, when a publisher’s information changes, we only have to update the Publisher relation.<sup>11</sup> This practice allows us to standardize spelling of the same value across different columns. It also makes it easier to make changes to these values, either because these values change or when you want to translate them into different languages.

Table 3-2. Initial Book relation

Title	Author	Format	Publisher	Country	Price
Harry Potter	J.K. Rowling	Paperback	Banana Press	UK	\$20
Harry Potter	J.K. Rowling	E-book	Banana Press	UK	\$10
Sherlock Holmes	Conan Doyle	Paperback	Guava Press	US	\$30
The Hobbit	J.R.R. Tolkien	Paperback	Banana Press	UK	\$30
Sherlock Holmes	Conan Doyle	Paperback	Guava Press	US	\$15

Table 3-3. Updated Book relation

Title	Author	Format	Publisher ID	Price
Harry Potter	J.K. Rowling	Paperback	1	\$20
Harry Potter	J.K. Rowling	E-book	1	\$10
Sherlock Holmes	Conan Doyle	Paperback	2	\$30
The Hobbit	J.R.R. Tolkien	Paperback	1	\$30
Sherlock Holmes	Conan Doyle	Paperback	2	\$15

Table 3-4. Publisher relation

Publisher ID	Publisher	Country
1	Banana Press	UK
2	Guava Press	US

One major downside of normalization is that your data is now spread across multiple relations. You can join the data from different relations back together, but joining can be expensive for large tables.

Databases built around the relational data model are relational databases. Once you’ve put data in your databases, you’ll want a way to retrieve it. The language that you can use to specify the data that you want from a database is called a *query language*. The most popular query language for relational databases today is SQL. Even though inspired by the relational model, the data model behind SQL has deviated from the original **relational model**. For example, SQL tables can contain

<sup>11</sup> You can further normalize the Book relation, such as separating format into a separate relation.

row duplicates, whereas true relations can't contain duplicates. However, this subtle difference has been safely ignored by most people.

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 the imperative paradigm, you specify the steps needed for an action and the computer executes these steps to return the outputs. In the declarative paradigm, you specify the outputs you want, and the computer figures out the steps needed to get you the queried outputs.

With an SQL database, you specify the pattern of data you want—the tables you want the data from, the conditions the results must meet, the basic data transformations such as join, sort, group, aggregate, etc.—but not how to retrieve the data. It is up to the database system to decide how to break the query into different parts, what methods to use to execute each part of the query, and the order in which different parts of the query should be executed.

With certain added features, SQL can be **Turing-complete**, which means that, in theory, SQL can be used to solve any computation problem (without making any guarantee about the time or memory required). However, in practice, it's not always easy to write a query to solve a specific task, and it's not always feasible or tractable to execute a query. Anyone working with SQL databases might have nightmarish memories of painfully long SQL queries that are impossible to understand and nobody dares to touch for fear that things might break.<sup>12</sup>

Figuring out how to execute an arbitrary query is the hard part, which is the job of query optimizers. A query optimizer examines all possible ways to execute a query and finds the fastest way to do so.<sup>13</sup> It's possible to use ML to improve query optimizers based on learning from incoming queries.<sup>14</sup> Query optimization is one of the most challenging problems in database systems, and normalization means that data is spread out on multiple relations, which makes joining it together even harder. Even though developing a query optimizer is hard, the good news is that you generally only need one query optimizer and all your applications can leverage it.

---

12 Greg Kemnitz, a coauthor of the original Postgres paper, [shared on Quora](#) that he once wrote a reporting SQL query that was 700 lines long and visited 27 different tables in lookups or joins. The query had about 1,000 lines of comments to help him remember what he was doing. It took him three days to compose, debug, and tune.

13 Yannis E. Ioannidis, "Query Optimization," *ACM Computing Surveys* (CSUR) 28, no. 1 (1996): 121–23, <https://oreil.ly/omXMg>

14 Ryan Marcus et al., "Neo: A Learned Query Optimizer," *arXiv preprint arXiv:1904.03711* (2019), <https://oreil.ly/wHy6p>.

## From Declarative Data Systems to Declarative ML Systems

Possibly inspired by the success of declarative data systems, many people have looked forward to declarative ML.<sup>15</sup> With a declarative ML system, users only need to declare the features' schema and the task, and the system will figure out the best model to perform that task with the given features. Users won't have to write code to construct, train, and tune models. Popular frameworks for declarative ML are **Ludwig**, developed at Uber, and **H2O AutoML**. In Ludwig, users can specify the model structure—such as the number of fully connected layers and the number of hidden units—on top of the features' schema and output. In H2O AutoML, you don't need to specify the model structure or hyperparameters. It experiments with multiple model architectures and picks out the best model given the features and the task.

Here is an example to show how H2O AutoML works. You give the system your data (inputs and outputs) and specify the number of models you want to experiment. It'll experiment with that number of models and show you the best-performing model:

```
# Identify predictors and response
x = train.columns
y = "response"
x.remove(y)

# For binary classification, response should be a factor
train[y] = train[y].asfactor()
test[y] = test[y].asfactor()

# Run AutoML for 20 base models
aml = H2OAutoML(max_models=20, seed=1)
aml.train(x=x, y=y, training_frame=train)

# Show the best-performing models on the AutoML Leaderboard
lb = aml.leaderboard

# Get the best-performing model
aml.leader
```

While declarative ML can be useful in many cases, it leaves unanswered the biggest challenges with ML in production. Declarative ML systems today abstract away the model development part, and as we'll cover in the next six chapters, with models being increasingly commoditized, model development is often the easier part. The hard part lies in feature engineering, data processing, model evaluation, data shift detection, continual learning, and so on.

---

<sup>15</sup> Matthias Boehm, Alexandre V. Evfimievski, Niketan Pansare, and Berthold Reinwald, "Declarative Machine Learning—A Classification of Basic Properties and Types," *arXiv*, May 19, 2016, <https://oreil.ly/OvW07>.

# NoSQL

The relational data model has been able to generalize to a lot of use cases, from ecommerce to finance to social networks. However, for certain use cases, this model can be restrictive. For example, it demands that your data follows a strict schema, and schema management is painful. In a survey by Couchbase in 2014, frustration with schema management was the #1 reason for the adoption of their nonrelational database.<sup>16</sup> It can also be difficult to write and execute SQL queries for specialized applications.

The latest movement against the relational data model is NoSQL. Originally started as a hashtag for a meetup to discuss nonrelational databases, NoSQL has been retroactively reinterpreted as Not Only SQL,<sup>17</sup> as many NoSQL data systems also support relational models. Two major types of nonrelational models are the document model and the graph model. The document model targets use cases where data comes in self-contained documents and relationships between one document and another are rare. The graph model goes in the opposite direction, targeting use cases where relationships between data items are common and important. We'll examine each of these two models, starting with the document model.

## Document model

The document model 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 could be considered analogous to a table in a relational database, and a document analogous to a row. In fact, you can convert a relation into a collection of documents that way. For example, you can convert the book data in Tables 3-3 and 3-4 into three JSON documents as shown in Examples 3-1, 3-2, and 3-3. However, a collection of documents is much more flexible than a table. All rows in a table must follow the same schema (e.g., have the same sequence of columns), while documents in the same collection can have completely different schemas.

---

16 James Phillips, “Surprises in Our NoSQL Adoption Survey,” *Couchbase*, December 16, 2014, <https://oreil.ly/ueyEX>.

17 Martin Kleppmann, *Designing Data-Intensive Applications* (Sebastopol, CA: O'Reilly, 2017).

*Example 3-1. Document 1: harry\_potter.json*

```
{
  "Title": "Harry Potter",
  "Author": "J .K. Rowling",
  "Publisher": "Banana Press",
  "Country": "UK",
  "Sold as": [
    {"Format": "Paperback", "Price": "$20"},
    {"Format": "E-book", "Price": "$10"}
  ]
}
```

*Example 3-2. Document 2: sherlock\_holmes.json*

```
{
  "Title": "Sherlock Holmes",
  "Author": "Conan Doyle",
  "Publisher": "Guava Press",
  "Country": "US",
  "Sold as": [
    {"Format": "Paperback", "Price": "$30"},
    {"Format": "E-book", "Price": "$15"}
  ]
}
```

*Example 3-3. Document 3: the\_hobbit.json*

```
{
  "Title": "The Hobbit",
  "Author": "J.R.R. Tolkien",
  "Publisher": "Banana Press",
  "Country": "UK",
  "Sold as": [
    {"Format": "Paperback", "Price": "$30"},
  ]
}
```

Because the document model doesn't enforce a schema, it's often referred to as schemaless. This is misleading because, as discussed previously, data stored in documents will be read later. The application that reads the documents usually 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.

The document model has better locality than the relational model. Consider the book data example in Tables 3-3 and 3-4 where the information about a book is spread across both the Book table and the Publisher table (and potentially also the Format table). To retrieve information about a book, you'll have to query multiple tables.



In the document model, all information about a book can be stored in a document, making it much easier to retrieve.

However, compared to the relational model, it's harder and less efficient to execute joins across documents compared to across tables. For example, if you want to find all books whose prices are below \$25, you'll have to read all documents, extract the prices, compare them to \$25, and return all the documents containing the books with prices below \$25.

Because of the different strengths of the document and relational data models, it's common to use both models for different tasks in the same database systems. More and more database systems, such as PostgreSQL and MySQL, support them both.

## Graph model

The graph model is built around the concept of a “graph.” A graph consists of nodes and edges, where the edges represent the relationships between the nodes. A database that uses graph structures to store its data is called a graph database. If in document databases, the content of each document is the priority, then in graph databases, the relationships between data items are the priority.

Because the relationships are modeled explicitly in graph models, it's faster to retrieve data based on relationships. Consider an example of a graph database in [Figure 3-5](#). The data from this example could potentially come from a simple social network. In this graph, nodes can be of different data types: person, city, country, company, etc.

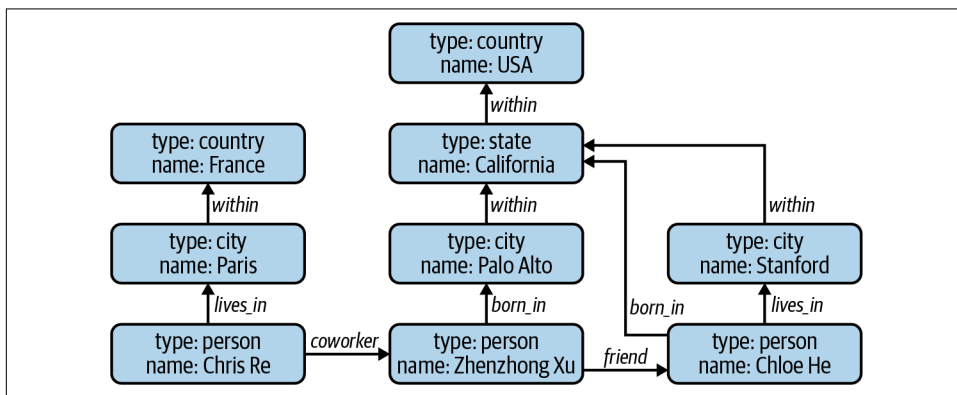


Figure 3-5. An example of a simple graph database

Imagine you want to find everyone who was born in the USA. Given this graph, you can start from the node USA and traverse the graph following the edges “within” and “born\_in” to find all the nodes of the type “person.” Now, imagine that instead of using the graph model to represent this data, we use the relational model. There'd be no easy way to write an SQL query to find everyone who was born in the USA,

especially given that there are an unknown number of hops between *country* and *person*—there are three hops between Zhenzhong Xu and USA while there are only two hops between Chloe He and USA. Similarly, there'd be no easy way for this type of query with a document database.

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

## Structured Versus Unstructured Data

Structured data follows a predefined data model, also known as a data schema. For example, the data model might specify that each data item consists of two values: the first value, “name,” is a string of at most 50 characters, and the second value, “age,” is an 8-bit integer in the range between 0 and 200. The predefined structure makes your data easier to analyze. If you want to know the average age of people in the database, all you have to do is to extract all the age values and average them out.

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, often causing mysterious bugs in the process. For example, you've never kept your users' email addresses before but now you do, so you have to retrospectively update email information to all previous users. One of the strangest bugs one of my colleagues encountered was when they could no longer use users' ages with their transactions, and their data schema replaced all the null ages with 0, and their ML model thought the transactions were made by people 0 years old.<sup>18</sup>

Because business requirements change over time, committing to a predefined data schema can become too restricting. Or you might have data from multiple data sources that are beyond your control, and it's impossible to make them follow the same schema. This is where unstructured data becomes appealing. Unstructured data doesn't adhere to a predefined data schema. It's usually text but can also be numbers, dates, images, audio, etc. For example, a text file of logs generated by your ML model is unstructured data.

Even though unstructured data doesn't adhere to a schema, it might still contain intrinsic patterns that help you extract structures. For example, the following text is unstructured, but you can notice the pattern that each line contains two values separated by a comma, the first value is textual, and the second value is numerical. However, there is no guarantee that all lines must follow this format. You can add a new line to that text even if that line doesn't follow this format.

---

<sup>18</sup> In this specific example, replacing the null age values with -1 solved the problem.

Lisa, 43  
Jack, 23  
Huyen, 59

Unstructured data also allows for more flexible storage options. For example, if your storage follows a schema, you can only store data following that schema. But if your storage doesn't follow a schema, you can store any type of data. You can convert all your data, regardless of types and formats, into bytestrings and store them together.

A repository for storing structured data is called a data warehouse. A repository for storing unstructured data is called a data lake. Data lakes are usually used to store raw data before processing. Data warehouses are used to store data that has been processed into formats ready to be used. [Table 3-5](#) shows a summary of the key differences between structured and unstructured data.

*Table 3-5. The key differences between structured and unstructured data*

Structured data	Unstructured data
Schema clearly defined	Data doesn't have to follow a schema
Easy to search and analyze	Fast arrival
Can only handle data with a specific schema	Can handle data from any source
Schema changes will cause a lot of troubles	No need to worry about schema changes (yet), as the worry is shifted to the downstream applications that use this data
Stored in data warehouses	Stored in data lakes

## Data Storage Engines and Processing

Data formats and data models specify the interface for how users can store and retrieve data. Storage engines, also known as databases, are the implementation of how data is stored and retrieved on machines. It's useful to understand different types of databases as your team or your adjacent team might need to select a database appropriate for your application.

Typically, there are two types of workloads that databases are optimized for, transactional processing and analytical processing, and there's a big difference between them, which we'll cover in this section. We will then cover the basics of the ETL (extract, transform, load) process that you will inevitably encounter when building an ML system in production.

### Transactional and Analytical Processing

Traditionally, a transaction refers to the action of buying or selling something. 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, watching a YouTube video, and so on. Even though these different 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.<sup>19</sup> This type of processing is known as *online transaction processing* (OLTP).

Because these transactions often involve users, they need to be processed fast (low latency) so that they don't keep users waiting. The processing method needs to have high availability—that is, the processing system needs to be available any time a user wants to make a transaction. If your system can't process a transaction, that transaction won't go through.

Transactional databases are designed to process online transactions and satisfy the low latency, high availability requirements. When people hear transactional databases, they usually think of ACID (atomicity, consistency, isolation, durability). Here are their definitions for those needing a quick reminder:

#### *Atomicity*

To guarantee that all the steps in a transaction are completed successfully as a group. If any step in the transaction fails, 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.

#### *Consistency*

To guarantee that all the transactions coming through must follow predefined rules. For example, 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. 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.

#### *Durability*

To guarantee that once a transaction has been committed, it will remain committed even in the case of a system failure. For example, after you've ordered a ride and your phone dies, you still want your ride to come.

However, transactional databases don't necessarily need to be ACID, and some developers find ACID to be too restrictive. According to Martin Kleppmann, "systems that do not meet the ACID criteria are sometimes called BASE, which stands for *Basically Available, Soft state, and Eventual consistency*. This is even more vague than the definition of ACID."<sup>20</sup>

---

<sup>19</sup> This paragraph, as well as many parts of this chapter, is inspired by Martin Kleppmann's *Designing Data-Intensive Applications*.

<sup>20</sup> Kleppmann, *Designing Data-Intensive Applications*.

Because each transaction is often processed as a unit separately from other transactions, transactional databases are often row-major. This also means that transactional databases might not be efficient for questions such as “What’s the average price for all the rides in September in San Francisco?” This kind of analytical question requires aggregating data in columns across multiple rows of data. Analytical databases are designed for this purpose. They are efficient with queries that allow you to look at data from different viewpoints. We call this type of processing *online analytical processing* (OLAP).

However, both the terms OLTP and OLAP have become outdated, as shown in [Figure 3-6](#), for three reasons. First, the separation of transactional and analytical databases was due to limitations of technology—it was hard to have databases that could handle both transactional and analytical queries efficiently. However, this separation is being closed. Today, we have transactional databases that can handle analytical queries, such as [CockroachDB](#). We also have analytical databases that can handle transactional queries, such as [Apache Iceberg](#) and [DuckDB](#).

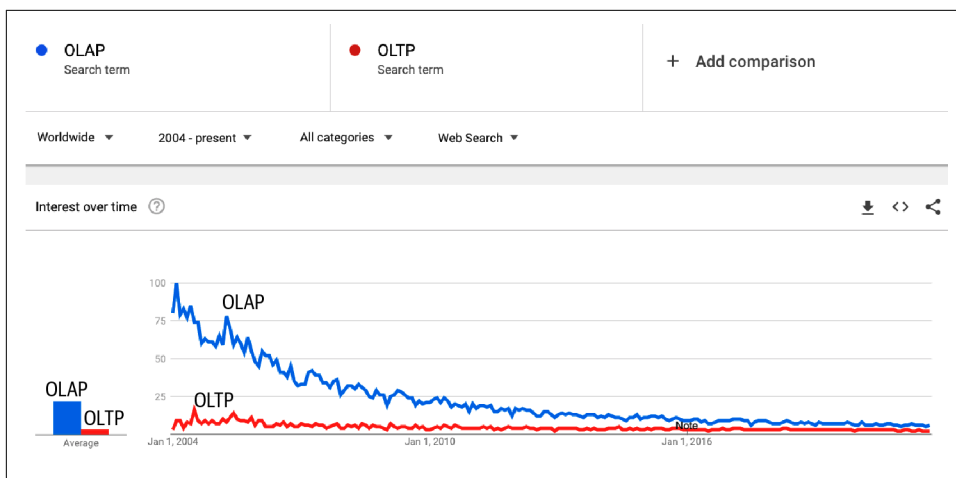


Figure 3-6. OLAP and OLTP are outdated terms, as of 2021, according to [Google Trends](#)

Second, in the traditional OLTP or OLAP paradigms, storage and processing are tightly coupled—how data is stored is also how data is processed. This may result in the same data being stored in multiple databases and using different processing engines to solve different types of queries. An interesting paradigm in the last decade has been to decouple storage from processing (also known as compute), as adopted by many data vendors including Google’s BigQuery, Snowflake, IBM,

and Teradata.<sup>21</sup> In this paradigm, the data can be stored in the same place, with a processing layer on top that can be optimized for different types of queries.

Third, “online” has become an overloaded term that can mean many different things. Online used to just mean “connected to the internet.” Then, it grew to also mean “in production”—we say a feature is online after that feature has been deployed in production.

In the data world today, *online* might refer to the speed at which your data is processed and made available: online, nearline, or offline. According to Wikipedia, online processing means data is immediately available for input/output. Nearline, which is short for near-online, means data is not immediately available but can be made online quickly without human intervention. *Offline* means data is not immediately available and requires some human intervention to become online.<sup>22</sup>

## ETL: Extract, Transform, and Load

In the early days of the relational data model, data was mostly structured. When data is *extracted* from different sources, it’s first *transformed* into the desired format before being *loaded* into the target destination such as a database or a data warehouse. This process is called *ETL*, which stands for extract, transform, and load.

Even before ML, ETL was all the rage in the data world, and it’s still relevant today for ML applications. ETL refers to the general purpose processing and aggregating of data into the shape and the format that you want.

Extract is extracting the data you want from all your data sources. Some of them will be corrupted or malformed. In the extracting phase, you need to validate your data and reject the data that doesn’t meet your requirements. For rejected data, you might have to notify the sources. Since this is the first step of the process, doing it correctly can save you a lot of time downstream.

Transform is the meaty part of the process, 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 the value ranges (e.g., one data source might use “Male” and “Female” for genders, but another uses “M” and “F” or “1” and “2”). You can apply operations such as transposing, deduplicating, sorting, aggregating, deriving new features, more data validating, etc.

---

21 Tino Tereshko, “Separation of Storage and Compute in BigQuery,” Google Cloud blog, November 29, 2017, <https://oreil.ly/utf7z>; Suresh H., “Snowflake Architecture and Key Concepts: A Comprehensive Guide,” Hevo blog, January 18, 2019, <https://oreil.ly/GyvKl>; Preetam Kumar, “Cutting the Cord: Separating Data from Compute in Your Data Lake with Object Storage,” IBM blog, September 21, 2017, <https://oreil.ly/Nd3xD>; “The Power of Separating Cloud Compute and Cloud Storage,” Teradata, last accessed April 2022, <https://oreil.ly/f82gP>.

22 Wikipedia, s.v. “Nearline storage,” last accessed April 2022, <https://oreil.ly/OCmiB>.

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 data warehouse.

The idea of ETL sounds simple but powerful, and it's the underlying structure of the data layer at many organizations. An overview of the ETL process is shown in Figure 3-7.

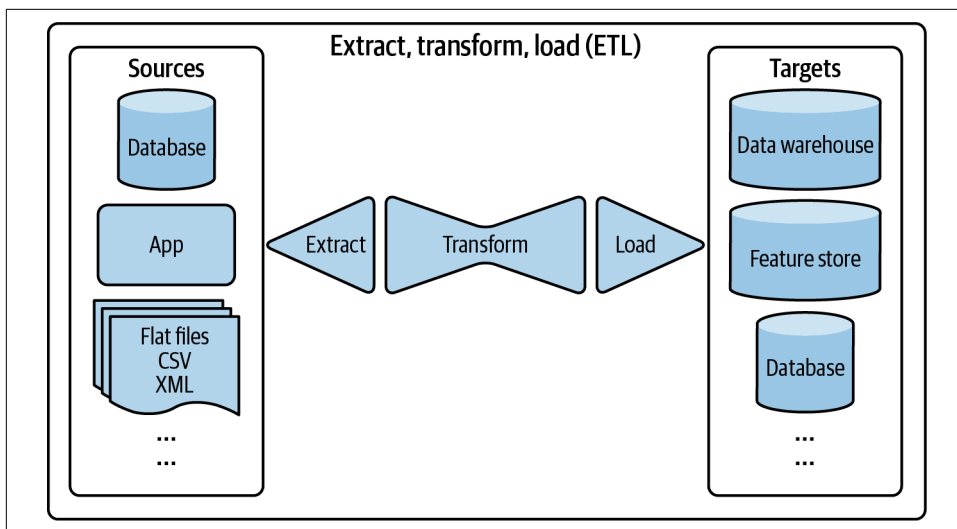


Figure 3-7. An overview of the ETL process

When the internet first became ubiquitous and hardware had just become so much more powerful, collecting data suddenly became so much easier. The amount of data grew rapidly. Not only that, but the nature of data also changed. The number of data sources expanded, and data schemas evolved.

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 raw 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). This paradigm allows for the fast arrival of data since there’s little processing needed before data is stored.

However, as data keeps on growing, this idea becomes less attractive. It’s inefficient to search through a massive amount of raw data for the data that you want.<sup>23</sup> At the same time, as companies switch to running applications on the cloud

<sup>23</sup> In the first draft of this book, I had cost as a reason why you shouldn’t store everything. However, as of today, storage has become so cheap that the storage cost is rarely a problem.

and infrastructures become standardized, data structures also become standardized. Committing data to a predefined schema becomes more feasible.

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

## Modes of Dataflow

In this chapter, we've been discussing data formats, data models, data storage, and processing for data used within the context of a single process. Most of the time, in production, you don't have a single process but multiple. A question arises: 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 the data flows from one process to another, which gives us a dataflow. There are three main modes of dataflow:

- Data passing through databases
- Data passing through services using requests such as the requests provided by REST and RPC APIs (e.g., POST/GET requests)
- Data passing through a real-time transport like Apache Kafka and Amazon Kinesis

We'll go over each of them in this section.

### Data Passing Through Databases

The easiest way to pass data between two processes is through databases, which we've discussed in the section [“Data Storage Engines and Processing” on page 67](#). For example, to pass data from process A to process B, process A can write that data into a database, and process B simply reads from that database.

This mode, however, doesn't always work because of two reasons. First, it requires that both processes must be able to access the same database. This might be infeasible, especially if the two processes are run by two different companies.

Second, it requires both processes to access data from databases, and read/write from databases can be slow, making it unsuitable for applications with strict latency requirements—e.g., almost all consumer-facing applications.



## Data Passing Through Services

One way to pass data between two processes is to send data directly through a network that connects these two processes. To pass data from process B to process A, process A first sends a request to process B that specifies the data A needs, and B returns the requested data through the same network. Because processes communicate through requests, we say that this is *request-driven*.

This mode of data passing is tightly coupled with the service-oriented architecture. A service is a process that can be accessed remotely, e.g., through a network. In this example, B is exposed to A as a service that A can send requests to. For B to be able to request data from A, A will also need to be exposed to B as a service.

Two services in communication with each other can be run by different companies in different applications. For example, a service might be run by a stock exchange that keeps track of the current stock prices. Another service might be run by an investment firm that requests the current stock prices and uses them to predict future stock prices.

Two services in communication with each other can also be parts of the same application. Structuring different components of your application as separate services allows each component to be developed, tested, and maintained independently of one another. Structuring an application as separate services gives you a microservice architecture.

To put the microservice architecture in the context of ML systems, imagine you're an ML engineer working on the price optimization problem for a company that owns a ride-sharing application like Lyft. In reality, Lyft has **hundreds of services** in its microservice architecture, but for the sake of simplicity, let's consider only three services:

### *Driver management service*

Predicts how many drivers will be available in the next minute in a given area.

### *Ride management service*

Predicts how many rides will be requested in the next minute in a given area.

### *Price optimization service*

Predicts the optimal price for each ride. The price for a ride should be low enough for riders to be willing to pay, yet high enough for drivers to be willing to drive and for the company to make a profit.

Because the price depends on supply (the available drivers) and demand (the requested rides), the price optimization service needs data from both the driver management and ride management services. Each time a user requests a ride, the price optimization service requests the predicted number of rides and predicted number of drivers to predict the optimal price for this ride.<sup>24</sup>

The most popular styles of requests used for passing data through networks are REST (representational state transfer) and RPC (remote procedure call). Their detailed analysis is beyond the scope of this book, but one major difference is that REST was designed for requests over networks, whereas RPC “tries to make a request to a remote network service look the same as calling a function or method in your programming language.” Because of this, “REST seems to be the predominant style for public APIs. The main focus of RPC frameworks is on requests between services owned by the same organization, typically within the same data center.”<sup>25</sup>

Implementations of a REST architecture are said to be RESTful. Even though many people think of REST as HTTP, REST doesn’t exactly mean HTTP because HTTP is just an implementation of REST.<sup>26</sup>

## Data Passing Through Real-Time Transport

To understand the motivation for real-time transports, let’s go back to the preceding example of the ride-sharing app with three simple services: driver management, ride management, and price optimization. In the last section, we discussed how the price optimization service needs data from the ride and driver management services to predict the optimal price for each ride.

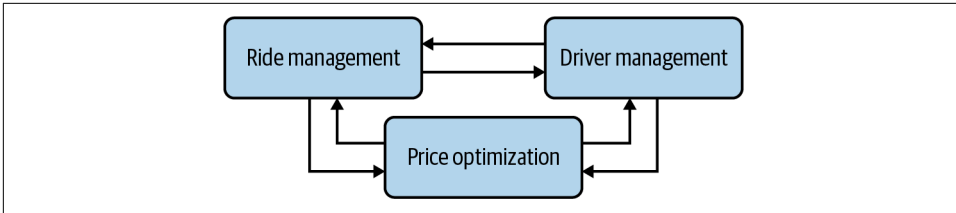
Now, imagine that the driver management service also needs to know the number of rides from the ride management service to know how many drivers to mobilize. It also wants to know the predicted prices from the price optimization service to use them as incentives for potential drivers (e.g., if you get on the road now you can get a 2x surge charge). Similarly, the ride management service might also want data from the driver management and price optimization services. If we pass data through services as discussed in the previous section, each of these services needs to send requests to the other two services, as shown in **Figure 3-8**.

---

<sup>24</sup> In practice, the price optimization might not have to request the predicted number of rides/drivers every time it has to make a price prediction. It’s a common practice to use the cached predicted number of rides/drivers and request new predictions every minute or so.

<sup>25</sup> Kleppmann, *Designing Data-Intensive Applications*.

<sup>26</sup> Tyson Trautmann, “Debunking the Myths of RPC and REST,” *Ethereal Bits*, December 4, 2012 (accessed via the Internet Archive), <https://oreil.ly/4sUrL>.



*Figure 3-8. In the request-driven architecture, each service needs to send requests to two other services*

With only three services, data passing is already getting complicated. Imagine having hundreds, if not thousands of services like what major internet companies have. Interservice data passing can blow up and become a bottleneck, slowing down the entire system.

Request-driven data passing is synchronous: the target service has to listen to the request for the request to go through. If the price optimization service requests data from the driver management service and the driver management service is down, the price optimization service will keep resending the request until it times out. And if the price optimization service is down before it receives a response, the response will be lost. A service that is down can cause all services that require data from it to be down.

What if there's a broker that coordinates data passing among services? Instead of having services request data directly from each other and creating a web of complex interservice data passing, each service only has to communicate with the broker, as shown in [Figure 3-9](#). For example, instead of having other services request the driver management services for the predicted number of drivers for the next minute, what if whenever the driver management service makes a prediction, this prediction is broadcast to a broker? Whichever service wants data from the driver management service can check that broker for the most recent predicted number of drivers. Similarly, whenever the price optimization service makes a prediction about the surge charge for the next minute, this prediction is broadcast to the broker.

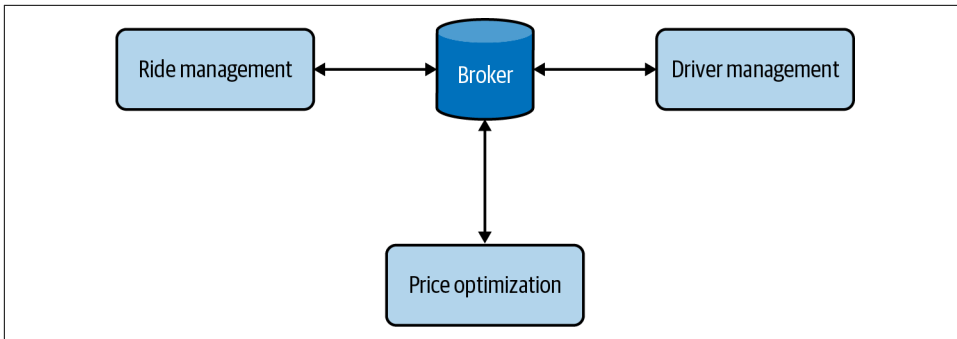


Figure 3-9. With a broker, a service only has to communicate with the broker instead of with other services

Technically, a database can be a broker—each service can write data to a database and other services that need the data can read from that database. However, as mentioned in the section [“Data Passing Through Databases” on page 72](#), reading and writing from databases are too slow for applications with strict latency requirements. Instead of using databases to broker data, we use in-memory storage to broker data. Real-time transports can be thought of as in-memory storage for data passing among services.

A piece of data broadcast to a real-time transport is called an event. This architecture is, therefore, also called *event-driven*. A real-time transport is sometimes called an event bus.

Request-driven architecture works well for systems that rely more on logic than on data. Event-driven architecture works better for systems that are data-heavy.

The two most common types of real-time transports are pubsub, which is short for publish-subscribe, and message queue. In the pubsub model, any service can publish to different topics in a real-time transport, and any service that subscribes to a topic can read all the events in that topic. The services that produce data don’t care about what services consume their data. Pubsub solutions often have a retention policy—data will be retained in the real-time transport for a certain period of time (e.g., seven days) before being deleted or moved to a permanent storage (like Amazon S3). See [Figure 3-10](#).

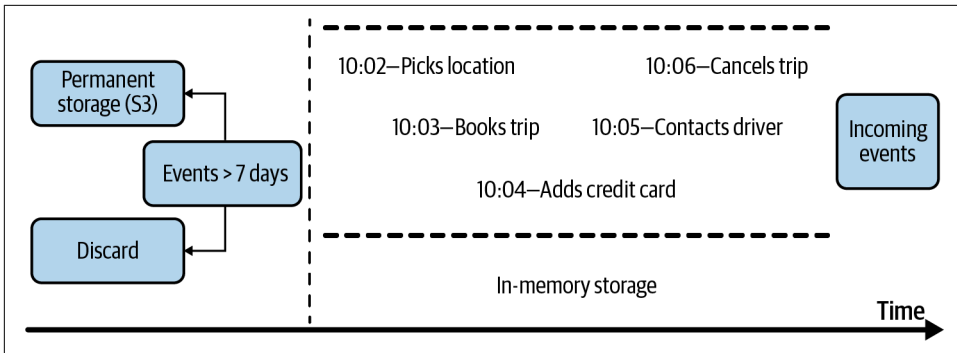


Figure 3-10. Incoming events are stored in in-memory storage before being discarded or moved to more permanent storage

In a message queue model, an event often has intended consumers (an event with intended consumers is called a message), and the message queue is responsible for getting the message to the right consumers.

Examples of pubsub solutions are Apache Kafka and Amazon Kinesis.<sup>27</sup> Examples of message queues are Apache RocketMQ and RabbitMQ. Both paradigms have gained a lot of traction in the last few years. Figure 3-11 shows some of the companies that use Apache Kafka and RabbitMQ.

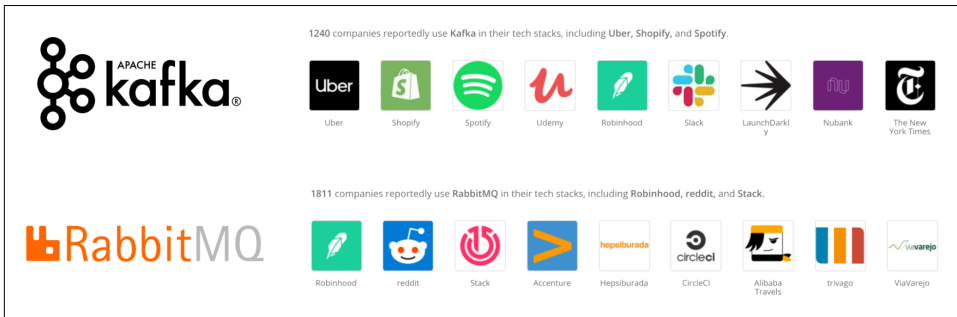


Figure 3-11. Companies that use Apache Kafka and RabbitMQ. Source: Screenshot from *Stackshare*

<sup>27</sup> If you want to learn more about how Apache Kafka works, Mitch Seymour has a great [animation](#) to explain it using otters!

# Batch Processing Versus 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 streaming in). Historical data is often processed in batch jobs—jobs that are kicked off periodically. For example, once a day, you might want to kick off a batch job to compute the average surge charge for all the rides in the last day.

When data is processed in batch jobs, we refer to it as *batch processing*. Batch processing has been a research subject for many decades, and companies have come up with distributed systems like MapReduce and Spark to process batch data efficiently.

When you have data in real-time transports like Apache Kafka and Amazon Kinesis, we say that you have streaming data. *Stream processing* refers to doing computation on streaming data. Computation on streaming data can also be kicked off periodically, but the periods are usually much shorter than the periods for batch jobs (e.g., every five minutes instead of every day). Computation on streaming data can also be kicked off whenever the need arises. For example, whenever a user requests a ride, you process your data stream to see what drivers are currently available.

Stream processing, when done right, can give low latency because you can process data as soon as data is generated, without having to first write it into databases. Many people believe that stream processing is less efficient than batch processing because you can't leverage tools like MapReduce or Spark. This is not always the case, for two reasons. First, streaming technologies like Apache Flink are proven to be highly scalable and fully distributed, which means they can do computation in parallel. Second, the strength of stream processing is in stateful computation. Consider the case where you want to process user engagement during a 30-day trial. If you kick off this batch job every day, you'll have to do computation over the last 30 days every day. With stream processing, it's possible to continue computing only the new data each day and joining the new data computation with the older data computation, preventing redundancy.

Because batch processing happens much less frequently than stream processing, in ML, batch processing is usually used to compute features that change less often, such as drivers' ratings (if a driver has had hundreds of rides, their rating is less likely to change significantly from one day to the next). *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, how many rides have been requested in the last minute, how many rides will be finished in the next two minutes, the median price of the last 10 rides in this area, etc. Features about the current state of the system like these are important to make the optimal price predictions. *Streaming features*—features extracted through stream processing—are also known as *dynamic features*.

For many problems, you need not only batch features or streaming features, but both. 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. We'll discuss more on how batch features and streaming features can be used together to generate predictions in [Chapter 7](#).

To do computation on data streams, you need a stream computation engine (the way Spark and MapReduce are batch computation engines). For simple streaming computation, you might be able to get away with the built-in stream computation capacity of real-time transports like Apache Kafka, but Kafka stream processing is limited in its ability to deal with various data sources.

For ML systems that leverage streaming features, the streaming computation is rarely simple. The number of stream features used in an application such as fraud detection and credit scoring can be in the hundreds, if not thousands. The stream feature extraction logic can require complex queries with join and aggregation along different dimensions. To extract these features requires efficient stream processing engines. For this purpose, you might want to look into tools like Apache Flink, KSQL, and Spark Streaming. Of these three engines, Apache Flink and KSQL are more recognized in the industry and provide a nice SQL abstraction for data scientists.

Stream processing 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 to make a batch processor do stream processing. Apache Flink's core maintainers have been arguing for years that batch processing is a special case of stream processing.<sup>28</sup>

## Summary

This chapter is built on the foundations established in [Chapter 2](#) around the importance of data in developing ML systems. In this chapter, we learned it's important to choose the right format to store our data to make it easier to use the data in the future. We discussed different data formats and the pros and cons of row-major versus column-major formats as well as text versus binary formats.

We continued to cover three major data models: relational, document, and graph. Even though the relational model is the most well known given the popularity of SQL, all three models are widely used today, and each is good for a certain set of tasks.

When talking about the relational model compared to the document model, many people think of the former as structured and the latter as unstructured. The division

---

28 Kostas Tzoumas, "Batch Is a Special Case of Streaming," *Ververica*, September 15, 2015, <https://oreil.ly/Ic1l2>.

between structured and unstructured data is quite fluid—the main question is who has to shoulder the responsibility of assuming the structure of data. Structured data means that the code that writes the data has to assume the structure. Unstructured data means that the code that reads the data has to assume the structure.

We continued the chapter with data storage engines and processing. We studied databases optimized for two distinct types of data processing: transactional processing and analytical processing. We studied data storage engines and processing together because traditionally storage is coupled with processing: transactional databases for transactional processing and analytical databases for analytical processing. However, in recent years, many vendors have worked on decoupling storage and processing. Today, we have transactional databases that can handle analytical queries and analytical databases that can handle transactional queries.

When discussing data formats, data models, data storage engines, and processing, data is assumed to be within a process. However, while working in production, you'll likely work with multiple processes, and you'll likely need to transfer data between them. We discussed three modes of data passing. The simplest mode is passing through databases. The most popular mode of data passing for processes is data passing through services. In this mode, a process is exposed as a service that another process can send requests for data. This mode of data passing is tightly coupled with microservice architectures, where each component of an application is set up as a service.

A mode of data passing that has become increasingly popular over the last decade is data passing through a real-time transport like Apache Kafka and RabbitMQ. This mode of data passing is somewhere between passing through databases and passing through services: it allows for asynchronous data passing with reasonably low latency.

As data in real-time transports have different properties from data in databases, they require different processing techniques, as discussed in the section [“Batch Processing Versus Stream Processing” on page 78](#). Data in databases is often processed in batch jobs and produces static features, whereas data in real-time transports is often processed using stream computation engines and produces dynamic features. Some people argue that batch processing is a special case of stream processing, and stream computation engines can be used to unify both processing pipelines.

Once we have our data systems figured out, we can collect data and create training data, which will be the focus of the next chapter.



# Feature Engineering

In 2014, the paper “[Practical Lessons from Predicting Clicks on Ads at Facebook](#)” claimed that having the right features is the most important thing in developing their ML models. Since then, many of the companies that I’ve worked with have discovered time and time again that once they have a workable model, having the right features tends to give them the biggest performance boost compared to clever algorithmic techniques such as hyperparameter tuning. State-of-the-art model architectures can still perform poorly if they don’t use a good set of features.

Due to its importance, a large part of many ML engineering and data science jobs is to come up with new useful features. In this chapter, we will go over common techniques and important considerations with respect to feature engineering. We will dedicate a section to go into detail about a subtle yet disastrous problem that has derailed many ML systems in production: data leakage and how to detect and avoid it.

We will end the chapter discussing how to engineer good features, taking into account both the feature importance and feature generalization. Talking about feature engineering, some people might think of feature stores. Since feature stores are closer to infrastructure to support multiple ML applications, we’ll cover feature stores in [Chapter 10](#).

# Learned Features Versus Engineered Features

When I cover this topic in class, my students frequently ask: “Why do we have to worry about feature engineering? Doesn’t deep learning promise us that we no longer have to engineer features?”

They are right. The promise of deep learning is that we won’t have to handcraft features. For this reason, deep learning is sometimes called feature learning.<sup>1</sup> Many features can be automatically learned and extracted by algorithms. However, we’re still far from the point where all features can be automated. This is not to mention that, as of this writing, the majority of ML applications in production aren’t deep learning. Let’s go over an example to understand what features can be automatically extracted and what features still need to be handcrafted.

Imagine that you want to build a sentiment analysis classifier to classify whether a comment is spam or not. Before deep learning, when given a piece of text, you would have to manually apply classical text processing techniques such as lemmatization, expanding contractions, removing punctuation, and lowercasing everything. After that, you might want to split your text into  $n$ -grams with  $n$  values of your choice.

For those unfamiliar, an  $n$ -gram is a contiguous sequence of  $n$  items from a given sample of text. The items can be phonemes, syllables, letters, or words. For example, given the post “I like food,” its word-level 1-grams are [“I”, “like”, “food”] and its word-level 2-grams are [“I like”, “like food”]. This sentence’s set of  $n$ -gram features, if we want  $n$  to be 1 and 2, is: [“I”, “like”, “food”, “I like”, “like food”].

Figure 5-1 shows an example of classical text processing techniques you can use to handcraft  $n$ -gram features for your text.

---

<sup>1</sup> Loris Nanni, Stefano Ghidoni, and Sheryl Brahnam, “Handcrafted vs. Non-handcrafted Features for Computer Vision Classification,” *Pattern Recognition* 71 (November 2017): 158–72, <https://oreil.ly/CGfYQ>; Wikipedia, s.v. “Feature learning,” <https://oreil.ly/fjmwN>.



Figure 5-1. An example of techniques that you can use to handcraft n-gram features for your text

Once you’ve generated n-grams for your training data, you can create a vocabulary that maps each n-gram to an index. Then you can convert each post into a vector based on its n-grams’ indices. For example, if we have a vocabulary of seven n-grams as shown in Table 5-1, each post can be a vector of seven elements. Each element corresponds to the number of times the n-gram at that index appears in the post. “I like food” will be encoded as the vector [1, 1, 0, 1, 1, 0, 1]. This vector can then be used as an input into an ML model.

Table 5-1. Example of a 1-gram and 2-gram vocabulary

I	like	good	food	I like	good food	like food
0	1	2	3	4	5	6

Feature engineering requires knowledge of domain-specific techniques—in this case, the domain is natural language processing (NLP) and the native language of the text. It tends to be an iterative process, which can be brittle. When I followed this method for one of my early NLP projects, I kept having to restart my process either because I had forgotten to apply one technique or because one technique I used turned out to be working poorly and I had to undo it.

However, much of this pain has been alleviated since the rise of deep learning. Instead of having to worry about lemmatization, punctuation, or stopword removal, you can just split your raw text into words (i.e., tokenization), create a vocabulary out of those words, and convert each of your words into one-shot vectors using this vocabulary. Your model will hopefully learn to extract useful features from this. In this new method, much of feature engineering for text has been automated. Similar progress has been made for images too. Instead of having to manually extract features from raw images and input those features into your ML models, you can just input raw images directly into your deep learning models.

However, an ML system will likely need data beyond just text and images. For example, when detecting whether a comment is spam or not, on top of the text in the comment itself, you might want to use other information about:

*The comment*

How many upvotes/downvotes does it have?



*The user who posted this comment*

When was this account created, how often do they post, and how many upvotes/downvotes do they have?



*The thread in which the comment was posted*

How many views does it have? Popular threads tend to attract more spam.

There are many possible features to use in your model. Some of them are shown in [Figure 5-2](#). The process of choosing what information to use and how to extract this information into a format usable by your ML models is feature engineering. For complex tasks such as recommending videos for users to watch next on TikTok, the number of features used can go up to millions. For domain-specific tasks such as predicting whether a transaction is fraudulent, you might need subject matter expertise with banking and frauds to be able to come up with useful features.

Comment ID	Time	User	Text	# 	# 	Link	# img	Thread ID	Reply to	# replies	...
93880839	2020-10-30 T 10:45 UTC	gitrekt	Your mom is a nice lady.	1	0	0	0	2332332	n0tab0t	1	...

User ID	Created	User	Subs	# 	# 	# replies	Karma	# threads	Verified email	Awards	...
4402903	2015-01-57 T 3:09 PST	gitrekt	[r/ml, r/memes, r/socialist]	15	90	28	304	776	No		...



Thread ID	Time	User	Text	# 	# 	Link	# img	# replies	# views	Awards	...
93883208	2020-10-30 T 2:45 PST	doge	Human is temporary, AGI is forever	120	50	1	0	32	2405	1	...

Figure 5-2. Some of the possible features about a comment, a thread, or a user to be included in your model

## Common Feature Engineering Operations

Because of the importance and the ubiquity of feature engineering in ML projects, there have been many techniques developed to streamline the process. In this section, we will discuss several of the most important operations that you might want to consider while engineering features from your data. They include handling missing values, scaling, discretization, encoding categorical features, and generating the old-school but still very effective cross features as well as the newer and exciting positional features. This list is nowhere near being comprehensive, but it does comprise some of the most common and useful operations to give you a good starting point. Let's dive in!

### Handling Missing Values

One of the first things you might notice when dealing with data in production is that some values are missing. However, one thing that many ML engineers I've interviewed don't know is that not all types of missing values are equal.<sup>2</sup> To illustrate this point, consider the task of predicting whether someone is going to buy a house in the next 12 months. A portion of the data we have is in [Table 5-2](#).

<sup>2</sup> In my experience, how well a person handles missing values for a given dataset during interviews strongly correlates with how well they will do in their day-to-day jobs.

Table 5-2. Example data for predicting house buying in the next 12 months

ID	Age	Gender	Annual income	Marital status	Number of children	Job	Buy?
1		A	150,000		1	Engineer	No
2	27	B	50,000			Teacher	No
3		A	100,000	Married	2		Yes
4	40	B			2	Engineer	Yes
5	35	B		Single	0	Doctor	Yes
6		A	50,000		0	Teacher	No
7	33	B	60,000	Single		Teacher	No
8	20	B	10,000			Student	No

There are three types of missing values. The official names for these types are a little bit confusing, so we'll go into detailed examples to mitigate the confusion.

#### *Missing not at random (MNAR)*

This is when the reason a value is missing is because of the true value itself. In this example, we might notice that some respondents didn't disclose their income. Upon investigation it may turn out that the income of respondents who failed to report tends to be higher than that of those who did disclose. *The income values are missing for reasons related to the values themselves.*

#### *Missing at random (MAR)*

This is when the reason a value is missing is not due to the value itself, but due to another observed variable. In this example, we might notice that age values are often missing for respondents of the gender "A," which might be because the people of gender A in this survey don't like disclosing their age.

#### *Missing completely at random (MCAR)*

This is when *there's no pattern in when the value is missing*. In this example, we might think that the missing values for the column "Job" might be completely random, not because of the job itself and not because of any other variable. People just forget to fill in that value sometimes for no particular reason. However, this type of missing is very rare. There are usually reasons why certain values are missing, and you should investigate.

When encountering missing values, you can either fill in the missing values with certain values (imputation) or remove the missing values (deletion). We'll go over both.

## Deletion

When I ask candidates about how to handle missing values during interviews, many tend to prefer deletion, not because it's a better method, but because it's easier to do.

One way to delete is *column deletion*: if a variable has too many missing values, just remove that variable. For example, in the example above, over 50% of the values for the variable “Marital status” are missing, so you might be tempted to remove this variable from your model. The drawback of this approach is that you might remove important information and reduce the accuracy of your model. Marital status might be highly correlated to buying houses, as married couples are much more likely to be homeowners than single people.<sup>3</sup>

Another way to delete is *row deletion*: if a sample has missing value(s), just remove that sample. This method can work when the missing values are completely at random (MCAR) and the number of examples with missing values is small, such as less than 0.1%. You don't want to do row deletion if that means 10% of your data samples are removed.

However, removing rows of data can also remove important information that your model needs to make predictions, especially if the missing values are not at random (MNAR). For example, you don't want to remove samples of gender B respondents with missing income because the fact that income is missing is information itself (missing income might mean higher income, and thus, more correlated to buying a house) and can be used to make predictions.

On top of that, removing rows of data can create biases in your model, especially if the missing values are at random (MAR). For example, if you remove all examples missing age values in the data in [Table 5-2](#), you will remove all respondents with gender A from your data, and your model won't be able to make good predictions for respondents with gender A.

## Imputation

Even though deletion is tempting because it's easy to do, deleting data can lead to losing important information and introduce biases into your model. If you don't want to delete missing values, you will have to impute them, which means “fill them with certain values.” Deciding which “certain values” to use is the hard part.

---

<sup>3</sup> Rachel Bogardus Drew, “3 Facts About Marriage and Homeownership,” Joint Center for Housing Studies of Harvard University, December 17, 2014, <https://oreil.ly/MWxFp>.

One common practice is to fill in missing values with their defaults. For example, if the job is missing, you might fill it with an empty string “”. Another common practice is to fill in missing values with the mean, median, or mode (the most common value). For example, if the temperature value is missing for a data sample whose month value is July, it’s not a bad idea to fill it with the median temperature of July.

Both practices work well in many cases, but sometimes they can cause hair-pulling bugs. One time, in one of the projects I was helping with, we discovered that the model was spitting out garbage because the app’s frontend no longer asked users to enter their age, so age values were missing, and the model filled them with 0. But the model never saw the age value of 0 during training, so it couldn’t make reasonable predictions.

In general, you want to avoid filling missing values with possible values, such as filling the missing number of children with 0—0 is a possible value for the number of children. It makes it hard to distinguish between people whose information is missing and people who don’t have children.

Multiple techniques might be used at the same time or in sequence to handle missing values for a particular set of data. Regardless of what techniques you use, one thing is certain: there is no perfect way to handle missing values. With deletion, you risk losing important information or accentuating biases. With imputation, you risk injecting your own bias into and adding noise to your data, or worse, data leakage. If you don’t know what data leakage is, don’t panic, we’ll cover it in the section “[Data Leakage](#)” on page 135.

## Scaling

Consider the task of predicting whether someone will buy a house in the next 12 months, and the data shown in [Table 5-2](#). The values of the variable Age in our data range from 20 to 40, whereas the values of the variable Annual Income range from 10,000 to 150,000. When we input these two variables into an ML model, it won’t understand that 150,000 and 40 represent different things. It will just see them both as numbers, and because the number 150,000 is much bigger than 40, it might 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. This process is called *feature scaling*. This is one of the simplest things you can do that often results in a performance boost for your model. Neglecting to do so can cause your model to make gibberish predictions, especially with classical algorithms like gradient-boosted trees and logistic regression.<sup>4</sup>

---

<sup>4</sup> Feature scaling once boosted my model’s performance by almost 10%.



An intuitive way to scale your features is to get them to be in the range  $[0, 1]$ . Given a variable  $x$ , its values can be rescaled to be in this range using the following formula:

$$x' = \frac{x - \min(x)}{\max(x) - \min(x)}$$

You can validate that if  $x$  is the maximum value, the scaled value  $x'$  will be 1. If  $x$  is the minimum value, the scaled value  $x'$  will be 0.

If you want your feature to be in an arbitrary range  $[a, b]$ —empirically, I find the range  $[-1, 1]$  to work better than the range  $[0, 1]$ —you can use the following formula:

$$x' = a + \frac{(x - \min(x))(b - a)}{\max(x) - \min(x)}$$

Scaling to an arbitrary range works well when you don't want to make any assumptions about your variables. If you think that your variables might follow a normal distribution, it might be helpful to normalize them so that they have zero mean and unit variance. This process is called *standardization*:

$$x' = \frac{x - \bar{x}}{\sigma},$$

with  $\bar{x}$  being the mean of variable  $x$ , and  $\sigma$  being its standard deviation.

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. An example of how the log transformation can make your data less skewed is shown in **Figure 5-3**. While this technique can yield performance gain in many cases, it doesn't work for all cases, and you should be wary of the analysis performed on log-transformed data instead of the original data.<sup>5</sup>

---

<sup>5</sup> Changyong Feng, Hongyue Wang, Naiji Lu, Tian Chen, Hua He, Ying Lu, and Xin M. Tu, "Log-Transformation and Its Implications for Data Analysis," *Shanghai Archives of Psychiatry* 26, no. 2 (April 2014): 105–9, <https://oreil.ly/hHJjt>.

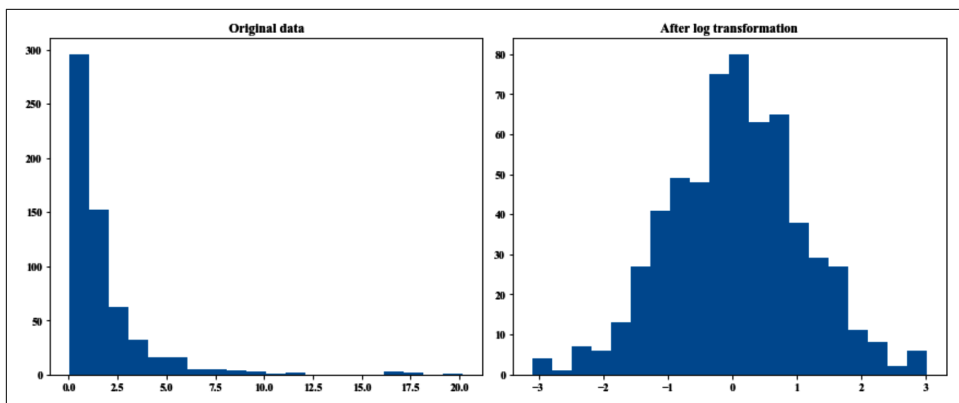


Figure 5-3. In many cases, the log transformation can help reduce the skewness of your data

There are two important things to note about scaling. One is that it's a common source of data leakage (this will be covered in greater detail in the section [“Data Leakage” on page 135](#)). Another is that it often requires global statistics—you have to look at the entire or a subset of training data to calculate its min, max, or mean. During inference, you reuse the statistics you had obtained during training to scale new data. If the new data has changed significantly compared to the training, these statistics won't be very useful. Therefore, it's important to retrain your model often to account for these changes.

## Discretization

This technique is included in this book for completeness, though in practice, I've rarely found discretization to help. Imagine that we've built a model with the data in [Table 5-2](#). During training, our model has seen the annual income values of “150,000,” “50,000,” “100,000,” and so on. During inference, our model encounters an example with an annual income of “9,000.50.”

Intuitively, we know that \$9,000.50 a year isn't much different from \$10,000/year, and we want our model to treat both of them the same way. But the model doesn't know that. Our model only knows that 9,000.50 is different from 10,000, and it will treat them differently.

Discretization is the process of turning a continuous feature into a discrete feature. This process is also known as quantization or binning. This is done by creating buckets for the given values. For annual income, you might want to group them into three buckets as follows:

- Lower income: less than \$35,000/year
- Middle income: between \$35,000 and \$100,000/year
- Upper income: more than \$100,000/year

Instead of having to learn an infinite number of possible incomes, our model can focus on learning only three categories, which is a much easier task to learn. This technique is supposed to be more helpful with limited training data.

Even though, by definition, discretization is meant for continuous features, it can be used for discrete features too. The age variable is discrete, but it might still be useful to group the values into buckets such as follows:

- Less than 18
- Between 18 and 22
- Between 22 and 30
- Between 30 and 40
- Between 40 and 65
- Over 65

The downside is that this categorization introduces discontinuities at the category boundaries—\$34,999 is now treated as completely different from \$35,000, which is treated the same as \$100,000. Choosing the boundaries of categories might not be all that easy. You can try to plot the histograms of the values and choose the boundaries that make sense. In general, common sense, basic quantiles, and sometimes subject matter expertise can help.

## Encoding Categorical Features

We've talked about how to turn continuous features into categorical features. In this section, we'll discuss how to best handle categorical features.

People who haven't worked with data in production tend to assume that categories are *static*, which means the categories don't change over time. This is true for many categories. For example, age brackets and income brackets are unlikely to change, and you know exactly how many categories there are in advance. Handling these categories is straightforward. You can just give each category a number and you're done.

However, in production, categories change. Imagine you're building a recommender system to predict what products users might want to buy from Amazon. One of the features you want to use is the product brand. When looking at Amazon's historical

data, you realize that there are a lot of brands. Even back in 2019, there were already over two million brands on Amazon!<sup>6</sup>

The number of brands is overwhelming, but you think: “I can still handle this.” You encode each brand as a number, so now you have two million numbers, from 0 to 1,999,999, corresponding to two million brands. Your model does spectacularly on the historical test set, and you get approval to test it on 1% of today’s traffic.

In production, your model crashes because it encounters a brand it hasn’t seen before and therefore can’t encode. New brands join Amazon all the time. To address this, you create a category UNKNOWN with the value of 2,000,000 to catch all the brands your model hasn’t seen during training.

Your model doesn’t crash anymore, but your sellers complain that their new brands are not getting any traffic. It’s because your model didn’t see the category UNKNOWN in the train set, so it just doesn’t recommend any product of the UNKNOWN brand. You fix this by encoding only the top 99% most popular brands and encode the bottom 1% brand as UNKNOWN. This way, at least your model knows how to deal with UNKNOWN brands.

Your model seems to work fine for about one hour, then the click-through rate on product recommendations plummets. Over the last hour, 20 new brands joined your site; some of them are new luxury brands, some of them are sketchy knockoff brands, some of them are established brands. However, your model treats them all the same way it treats unpopular brands in the training data.

This isn’t an extreme example that only happens if you work at Amazon. This problem happens quite a lot. For example, if you want to predict whether a comment is spam, you might want to use the account that posted this comment as a feature, and new accounts are being created all the time. The same goes for new product types, new website domains, new restaurants, new companies, new IP addresses, and so on. If you work with any of them, you’ll have to deal with this problem.

Finding a way to solve this problem turns out to be surprisingly difficult. You don’t want to put them into a set of buckets because it can be really hard—how would you even go about putting new user accounts into different groups?

One solution to this problem is the *hashing trick*, popularized by the package Vowpal Wabbit developed at Microsoft.<sup>7</sup> The gist of this trick is that you use a hash function to generate a hashed value of each category. The hashed value will become the index of that category. Because you can specify the hash space, you can fix the number of encoded values for a feature in advance, without having to know how many

---

<sup>6</sup> “Two Million Brands on Amazon,” *Marketplace Pulse*, June 11, 2019, <https://oreil.ly/zrqtd>.

<sup>7</sup> Wikipedia, s.v. “Feature hashing,” <https://oreil.ly/tINTc>.

categories there will be. For example, if you choose a hash space of 18 bits, which corresponds to  $2^{18} = 262,144$  possible hashed values, all the categories, even the ones that your model has never seen before, will be encoded by an index between 0 and 262,143.

One problem with hashed functions is collision: two categories being assigned the same index. However, with many hash functions, the collisions are random; new brands can share an index with any of the existing brands instead of always sharing an index with unpopular brands, which is what happens when we use the preceding UNKNOWN category. The impact of colliding hashed features is, fortunately, not that bad. In research done by Booking.com, even for 50% colliding features, the performance loss is less than 0.5%, as shown in Figure 5-4.<sup>8</sup>

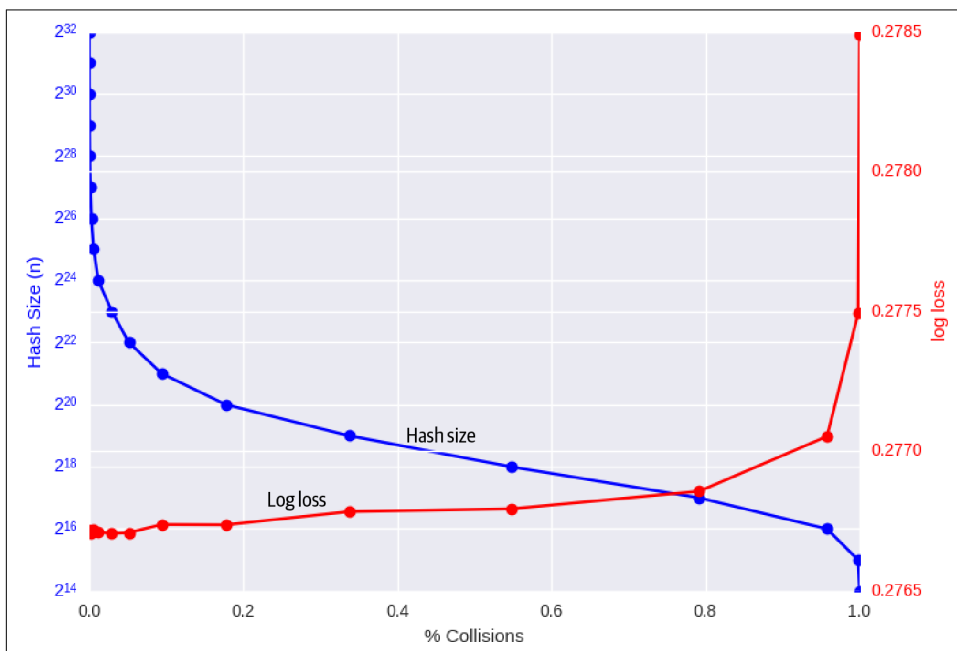


Figure 5-4. A 50% collision rate only causes the log loss to increase less than 0.5%.

Source: Lucas Bernardi

You can choose a hash space large enough to reduce the collision. You can also choose a hash function with properties that you want, such as a locality-sensitive hashing function where similar categories (such as websites with similar names) are hashed into values close to each other.

<sup>8</sup> Lucas Bernardi, “Don’t Be Tricked by the Hashing Trick,” Booking.com, January 10, 2018, <https://oreil.ly/VZmaY>.

Because it's a trick, it's often considered hacky by academics and excluded from ML curricula. But its wide adoption in the industry is a testimonial to how effective the trick is. It's essential to Vowpal Wabbit and it's part of the frameworks of scikit-learn, TensorFlow, and gensim. It can be especially useful in continual learning settings where your model learns from incoming examples in production. We'll cover continual learning in [Chapter 9](#).

## Feature Crossing

Feature crossing is the technique to combine two or more features to generate new features. This technique is useful to model the nonlinear relationships between features. For example, for the task of predicting whether someone will want to buy a house in the next 12 months, you suspect that there might be a nonlinear relationship between marital status and number of children, so you combine them to create a new feature “marriage and children” as in [Table 5-3](#).

*Table 5-3. Example of how two features can be combined to create a new feature*

Marriage	Single	Married	Single	Single	Married
Children	0	2	1	0	1
Marriage and children	Single, 0	Married, 2	Single, 1	Single, 0	Married, 1

Because feature crossing helps model nonlinear relationships between variables, it's essential for models that can't learn or are bad at learning nonlinear relationships, such as linear regression, logistic regression, and tree-based models. It's less important in neural networks, but it can still be useful because explicit feature crossing occasionally helps neural networks learn nonlinear relationships faster. DeepFM and xDeepFM are the family of models that have successfully leveraged explicit feature interactions for recommender systems and click-through-rate prediction.<sup>9</sup>

A caveat of feature crossing is that it can make your feature space blow up. Imagine feature A has 100 possible values and feature B has 100 possible features; crossing these two features will result in a feature with  $100 \times 100 = 10,000$  possible values. You will need a lot more data for models to learn all these possible values. Another caveat is that because feature crossing increases the number of features models use, it can make models overfit to the training data.

---

<sup>9</sup> Huifeng Guo, Ruiming Tang, Yunming Ye, Zhenguo Li, and Xiuqiang He, “DeepFM: A Factorization-Machine Based Neural Network for CTR Prediction,” *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI, 2017)*, <https://oreil.ly/1Vs3v>; Jianxun Lian, Xiaohuan Zhou, Fuzheng Zhang, Zhongxia Chen, Xing Xie, and Guangzhong Sun, “xDeepFM: Combining Explicit and Implicit Feature Interactions for Recommender Systems,” *arXiv*, 2018, <https://oreil.ly/WFmFt>.

## Discrete and Continuous Positional Embeddings

First introduced to the deep learning community in the paper “[Attention Is All You Need](#)” (Vaswani et al. 2017), positional embedding has become a standard data engineering technique for many applications in both computer vision and NLP. We’ll walk through an example to show why positional embedding is necessary and how to do it.

Consider the task of language modeling where you want to predict the next token (e.g., a word, character, or subword) based on the previous sequence of tokens. In practice, a sequence length can be up to 512, if not larger. However, for simplicity, let’s use words as our tokens and use the sequence length of 8. Given an arbitrary sequence of 8 words, such as “Sometimes all I really want to do is,” we want to predict the next word.

### Embeddings

An embedding is a vector that represents a piece of data. We call the set of all possible embeddings generated by the same algorithm for a type of data “an embedding space.” All embedding vectors in the same space are of the same size.

One of the most common uses of embeddings is word embeddings, where you can represent each word with a vector. However, embeddings for other types of data are increasingly popular. For example, ecommerce solutions like Criteo and Coveo have embeddings for products.<sup>10</sup> Pinterest has embeddings for images, graphs, queries, and even users.<sup>11</sup> Given that there are so many types of data with embeddings, there has been a lot of interest in creating universal embeddings for multimodal data.

If we use a recurrent neural network, it will process words in sequential order, which means the order of words is implicitly inputted. However, if we use a model like a transformer, words are processed in parallel, so words’ positions need to be explicitly inputted so that our model knows the order of these words (“a dog bites a child” is very different from “a child bites a dog”). We don’t want to input the absolute positions, 0, 1, 2, ..., 7, into our model because empirically, neural networks don’t work well with inputs that aren’t unit-variance (that’s why we scale our features, as discussed previously in the section “[Scaling](#)” on page 126).

---

10 Flavian Vasile, Elena Smirnova, and Alexis Conneau, “Meta-Prod2Vec—Product Embeddings Using Side-Information for Recommendation,” *arXiv*, July 25, 2016, <https://oreil.ly/KDaEd>; “Product Embeddings and Vectors,” Coveo, <https://oreil.ly/ShasY>.

11 Andrew Zhai, “Representation Learning for Recommender Systems,” August 15, 2021, <https://oreil.ly/OchiL>.

If we rescale the positions to between 0 and 1, so 0, 1, 2, ..., 7 become 0, 0.143, 0.286, ..., 1, the differences between the two positions will be too small for neural networks to learn to differentiate.

A way to handle position embeddings is to treat it the way we'd treat word embedding. With word embedding, we use an embedding matrix with the vocabulary size as its number of columns, and each column is the embedding for the word at the index of that column. With position embedding, the number of columns is the number of positions. In our case, since we only work with the previous sequence size of 8, the positions go from 0 to 7 (see [Figure 5-5](#)).

The embedding size for positions is usually the same as the embedding size for words so that they can be summed. For example, the embedding for the word “food” at position 0 is the sum of the embedding vector for the word “food” and the embedding vector for position 0. This is the way position embeddings are implemented in Hugging Face’s BERT as of August 2021. Because the embeddings change as the model weights get updated, we say that the position embeddings are learned.

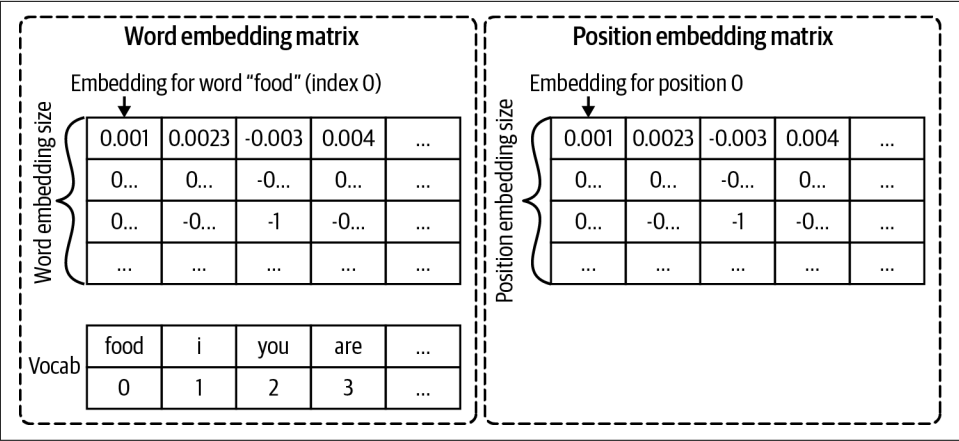


Figure 5-5. One way to embed positions is to treat them the way you’d treat word embeddings

Position embeddings can also be fixed. The embedding for each position is still a vector with  $S$  elements ( $S$  is the position embedding size), but each element is predefined using a function, usually sine and cosine. In [the original Transformer paper](#), if the element is at an even index, use sine. Else, use cosine. See [Figure 5-6](#).



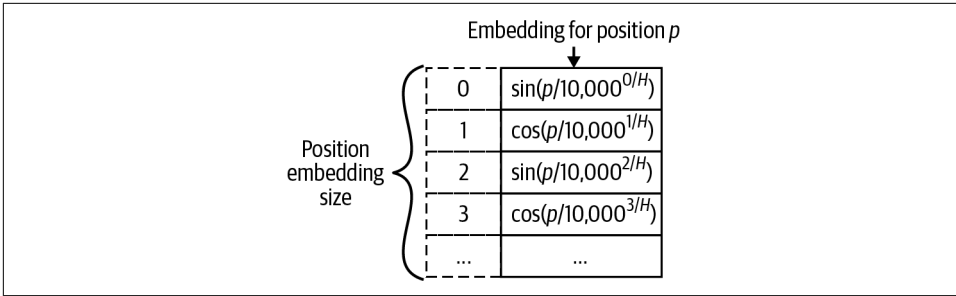


Figure 5-6. Example of fixed position embedding.  $H$  is the dimension of the outputs produced by the model.

Fixed positional embedding is a special case of what is known as Fourier features. If positions in positional embeddings are discrete, Fourier features can also be continuous. Consider the task involving representations of 3D objects, such as a teapot. Each position on the surface of the teapot is represented by a three-dimensional coordinate, which is continuous. When positions are continuous, it'd be very hard to build an embedding matrix with continuous column indices, but fixed position embeddings using sine and cosine functions still work.

The following is the generalized format for the embedding vector at coordinate  $v$ , also called the Fourier features of coordinate  $v$ . Fourier features have been shown to improve models' performance for tasks that take in coordinates (or positions) as inputs. If interested, you might want to read more about it in [“Fourier Features Let Networks Learn High Frequency Functions in Low Dimensional Domains”](#) (Tancik et al. 2020).

$$\gamma(v) = [a_1 \cos(2\pi b_1^T v), a_1 \sin(2\pi b_1^T v), \dots, a_m \cos(2\pi b_m^T v), a_m \sin(2\pi b_m^T v)]^T$$

## Data Leakage

In July 2021, *MIT Technology Review* ran a provocative article titled “Hundreds of AI Tools Have Been Built to Catch Covid. None of Them Helped.” These models were trained to predict COVID-19 risks from medical scans. The article listed multiple examples where ML models that performed well during evaluation failed to be usable in actual production settings.

In one example, researchers trained their model on a mix of scans taken when patients were lying down and standing up. “Because patients scanned while lying down were more likely to be seriously ill, the model learned to predict serious covid risk from a person's position.”

In some other cases, models were “found to be picking up on the text font that certain hospitals used to label the scans. As a result, fonts from hospitals with more serious caseloads became predictors of covid risk.”<sup>12</sup>

Both of these are examples of data leakage. *Data leakage* 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.

Data leakage is challenging because often the leakage is nonobvious. It’s dangerous because it can cause your models to fail in an unexpected and spectacular way, even after extensive evaluation and testing. Let’s go over another example to demonstrate what data leakage is.

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 did really well on the test data from hospital A, but poorly on the data from hospital B.

After extensive investigation, you learned that at hospital A, when doctors think that a patient has lung cancer, they send that patient to a more advanced scan machine, which outputs slightly different CT scan images. Your model learned to rely on the information on the scan machine used to make predictions on whether a scan image shows signs of lung cancer. 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.

Data leakage can happen not only with newcomers to the field, but has also happened to several experienced researchers whose work I admire, and in one of my own projects. Despite its prevalence, data leakage is rarely covered in ML curricula.

### Cautionary Tale: Data Leakage with Kaggle Competition

In 2020, the University of Liverpool launched an **Ion Switching competition on Kaggle**. The task was to identify the number of ion channels open at each time point. They synthesized test data from training data, and some people were able to reverse engineer and obtain test labels from the leak.<sup>13</sup> The two winning teams in this competition are the two teams that were able to exploit the leak, though they might have still been able to win without exploiting the leak.<sup>14</sup>

12 Will Douglas Heaven, “Hundreds of AI Tools Have Been Built to Catch Covid. None of Them Helped,” *MIT Technology Review*, July 30, 2021, <https://oreil.ly/Ig1b1>.

13 Zidmie, “The leak explained!” Kaggle, <https://oreil.ly/1JgLj>.

14 Addison Howard, “Competition Recap—Congratulations to our Winners!” Kaggle, <https://oreil.ly/wVUU4>.

## Common Causes for Data Leakage

In this section, we'll go over some common causes for data leakage and how to avoid them.

### Splitting time-correlated data randomly instead of by time

When I learned ML in college, I was taught to randomly split my data into train, validation, and test splits. This is also how data is often reportedly split in ML research papers. However, this is also one common cause for data leakage.

In many cases, data is time-correlated, which means that the time the data is generated affects its label distribution. Sometimes, the correlation is obvious, as in the case of stock prices. To oversimplify it, the prices of similar stocks tend to move together. If 90% of the tech stocks go down today, it's very likely the other 10% of the tech stocks go down too. When building models to predict the future stock prices, you want to split your training data by time, such as training your 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. We say that the information from the future is leaked into the training process.

However, in many cases, the correlation is nonobvious. Consider the task of predicting whether someone will click on a song recommendation. Whether someone will listen to a song depends not only on their music taste but also on the general music trend that day. If an artist passes away one day, people will be much more likely to listen to that artist. By including samples from a certain day in the train split, information about the music trend that day will be passed into your model, making it easier for it to make predictions on other samples on that same day.

To prevent future information from leaking into the training process and allowing models to cheat during evaluation, split your data by time, instead of splitting randomly, whenever possible. For example, if you have data from five weeks, use the first four weeks for the train split, then randomly split week 5 into validation and test splits as shown in [Figure 5-7](#).

Train split					
Week 1	Week 2	Week 3	Week 4	Week 5	Valid split
X11	X21	X31	X41	X51	
X12	X22	X32	X42	X52	Test split
X13	X23	X33	X43	X53	
X14	X24	X34	X44	X54	
...	...	...	...	...	

Figure 5-7. Split data by time to prevent future information from leaking into the training process

### Scaling before splitting

As discussed in the section [“Scaling” on page 126](#), it’s important to scale your features. Scaling requires global statistics—e.g., mean, variance—of your data. One 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 type of leakage, always split your data first before scaling, then use the statistics from the train split to scale all the splits. Some even suggest that we split our data before any exploratory data analysis 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

One common way to handle the missing values of a feature is to fill (input) them with the mean or median of all values present. Leakage might occur if the mean or median is calculated using entire data instead of just the train split. This type of leakage is similar to the type of leakage caused by scaling, and it can be prevented by using only statistics from the train split to fill in missing values in all the splits.

## Poor handling of data duplication before splitting

If you have duplicates or near-duplicates in your data, failing to remove them before splitting your data might cause the same samples to appear in both train and validation/test splits. Data duplication is quite common in the industry, and has also been found in popular research datasets. For example, CIFAR-10 and CIFAR-100 are two popular datasets used for computer vision research. They were released in 2009, yet it was not until 2019 that Barz and Denzler discovered that 3.3% and 10% of the images from the test sets of the CIFAR-10 and CIFAR-100 datasets have duplicates in the training set.<sup>15</sup>

Data duplication can result from data collection or merging of different data sources. A 2021 *Nature* article listed data duplication as a common pitfall when using ML to detect COVID-19, which happened because “one dataset combined several other datasets without realizing that one of the component datasets already contains another component.”<sup>16</sup> Data duplication can also happen because of data processing—for example, oversampling might result in duplicating certain examples.

To avoid this, always check for duplicates before splitting and also after splitting just to make sure. If you oversample your data, do it after splitting.

## Group leakage

A group of examples have strongly correlated labels but are divided into different splits. For example, a patient might have two lung CT scans that are a week apart, which likely have the same labels on whether they contain signs of lung cancer, but one of them is in the train split and the second is in the test split. This type of leakage is common for objective detection tasks that contain photos of the same object taken milliseconds apart—some of them landed in the train split while others landed in the test split. It's hard avoiding this type of data leakage without understanding how your data was generated.

---

<sup>15</sup> Björn Barz and Joachim Denzler, “Do We Train on Test Data? Purging CIFAR of Near-Duplicates,” *Journal of Imaging* 6, no. 6 (2020): 41.

<sup>16</sup> Michael Roberts, Derek Driggs, Matthew Thorpe, Julian Gilbey, Michael Yeung, Stephan Ursprung, Angelica I. Aviles-Rivero, et al. “Common Pitfalls and Recommendations for Using Machine Learning to Detect and Prognosticate for COVID-19 Using Chest Radiographs and CT Scans,” *Nature Machine Intelligence* 3 (2021): 199–217, <https://oreil.ly/TzbKJ>.

## Leakage from data generation process

The example earlier about how information on whether a CT scan shows signs of lung cancer is leaked via the scan machine is an example of this type of leakage. Detecting this type of data leakage requires a deep understanding of the way data is collected. For example, it would be very hard to figure out that the model's poor performance in hospital B is due to its different scan machine procedure if you don't know about different scan machines or that the procedures at the two hospitals are different.

There's no foolproof way to avoid this type of leakage, but you can mitigate the risk by keeping track of the sources of your data and understanding how it is collected and processed. Normalize your data so that data from different sources can have the same means and variances. If different CT scan machines output images with different resolutions, normalizing all the images to have the same resolution would make it harder for models to know which image is from which scan machine. And don't forget to incorporate subject matter experts, who might have more contexts on how data is collected and used, into the ML design process!

## Detecting Data Leakage

Data leakage can happen during many steps, from generating, collecting, sampling, splitting, and processing data to feature engineering. It's 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 (label). If a feature has unusually high correlation, investigate how this feature is generated and whether the correlation makes sense. It's possible that two features independently don't contain leakage, but two features together can contain leakage. For example, when building a model to predict how long an employee will stay at a company, the starting date and the end date separately doesn't tell us much about their tenure, but both together can give us that information.

Do ablation studies to measure how important a feature or a set of features is to your model. If removing a feature causes the model's performance to deteriorate significantly, investigate why that feature is so important. If you have a massive amount of features, say a thousand features, it might be infeasible to do ablation studies on every possible combination of them, but it can still be useful to occasionally do ablation studies with a subset of features that you suspect the most. This is another example of how subject matter expertise can come in handy in feature engineering. Ablation studies can be run offline at your own schedule, so you can leverage your machines during downtime for this purpose.

Keep an eye out for new features added to your model. If adding a new feature significantly improves your model's performance, either that feature is really good or that feature just contains leaked information about labels.

Be very careful every time you look at the test split. If you use the test split in any way other than to report a model's final performance, whether to come up with 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. In my experience, the list of features used for a model in production only grows over time. However, more features doesn't always mean better model performance. Having too many features can be bad both during training and serving your model for the following reasons:

- The more features you have, the more opportunities there are for data leakage.
- Too many features can cause overfitting.
- Too many features can increase memory required to serve a model, which, in turn, might require you to use a more expensive machine/instance to serve your model.
- Too many features can increase inference latency when doing online prediction, especially if you need to extract these features from raw data for predictions online. We'll go deeper into online prediction in [Chapter 7](#).
- Useless features become technical debts. Whenever your data pipeline changes, all the affected features need to be adjusted accordingly. For example, if one day your application decides to no longer take in information about users' age, all features that use users' age need to be updated.

In theory, if a feature doesn't help a model make good predictions, regularization techniques like L1 regularization should reduce that feature's weight to 0. However, in practice, it might help models learn faster if the features that are no longer useful (and even possibly harmful) are removed, prioritizing good features.

You can store removed features to add them back later. You can also just store general feature definitions to reuse and share across teams in an organization. When talking about feature definition management, some people might think of feature stores as the solution. However, not all feature stores manage feature definitions. We'll discuss feature stores further in [Chapter 10](#).

There are two factors you might want to consider when evaluating whether a feature is good for a model: importance to the model and generalization to unseen data.

## Feature Importance

There are many different methods for measuring a feature's importance. If you use a classical ML algorithm like boosted gradient trees, the easiest way to measure the importance of your features is to use built-in feature importance functions implemented by XGBoost.<sup>17</sup> For more model-agnostic methods, you might want to look into SHAP (SHapley Additive exPlanations).<sup>18</sup> [InterpretML](#) is a great open source package that leverages feature importance to help you understand how your model makes predictions.

The exact algorithm for feature importance measurement is complex, but intuitively, a feature's importance to a model is measured by how much that model's performance deteriorates if that feature or a set of features containing 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. [Figures 5-8 and 5-9](#) show how SHAP can help you understand the contribution of each feature to a model's predictions.

---

<sup>17</sup> With XGBoost function `get_score`.

<sup>18</sup> A great open source Python package for calculating SHAP can be found on [GitHub](#).



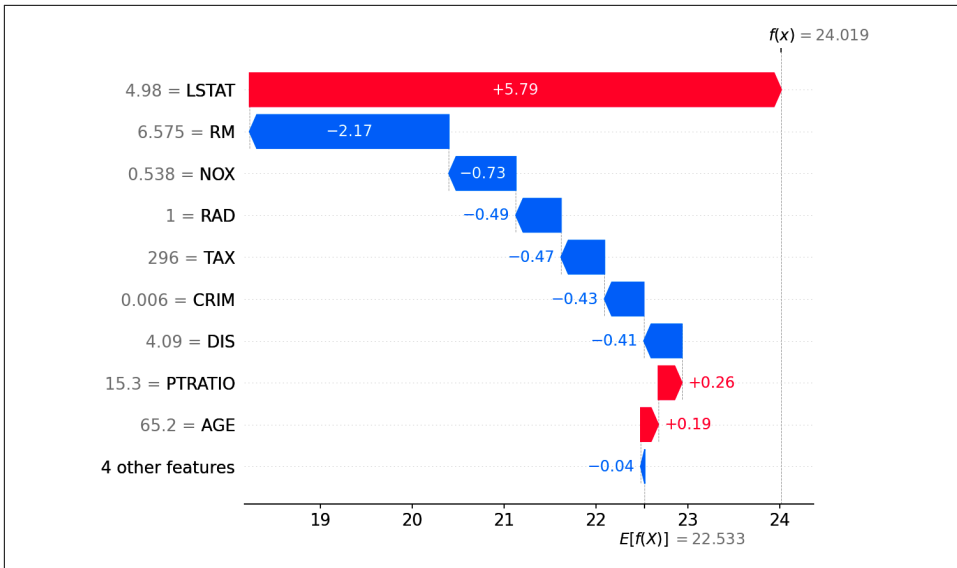


Figure 5-8. How much each feature contributes to a model's single prediction, measured by SHAP. The value  $LSTAT = 4.98$  contributes the most to this specific prediction. Source: Scott Lundberg<sup>19</sup>

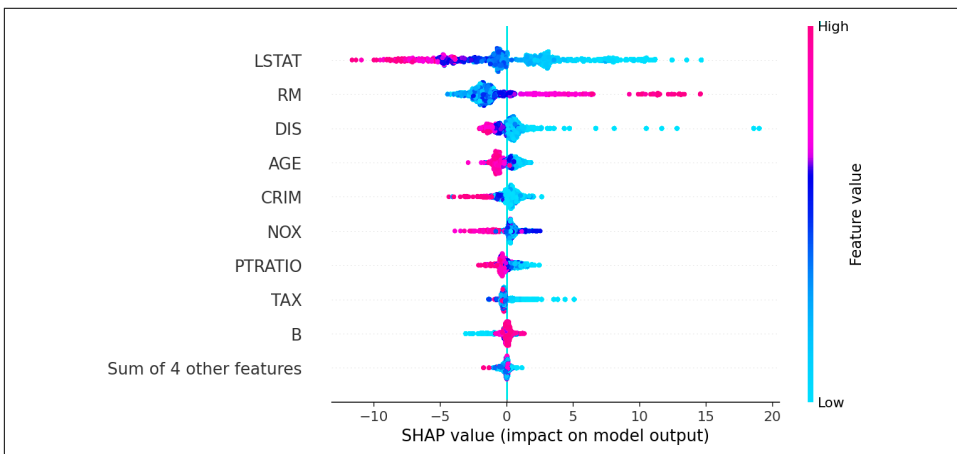


Figure 5-9. How much each feature contributes to a model, measured by SHAP. The feature  $LSTAT$  has the highest importance. Source: Scott Lundberg

<sup>19</sup> Scott Lundberg, SHAP (SHapley Additive exPlanations), GitHub repository, last accessed 2021, <https://oreil.ly/c8qqE>.

Often, a small number of features accounts for a large portion of your model's feature importance. When measuring feature importance for a click-through rate prediction model, the ads team at Facebook found out that the top 10 features are responsible for about half of the model's total feature importance, whereas the last 300 features contribute less than 1% feature importance, as shown in [Figure 5-10](#).<sup>20</sup>

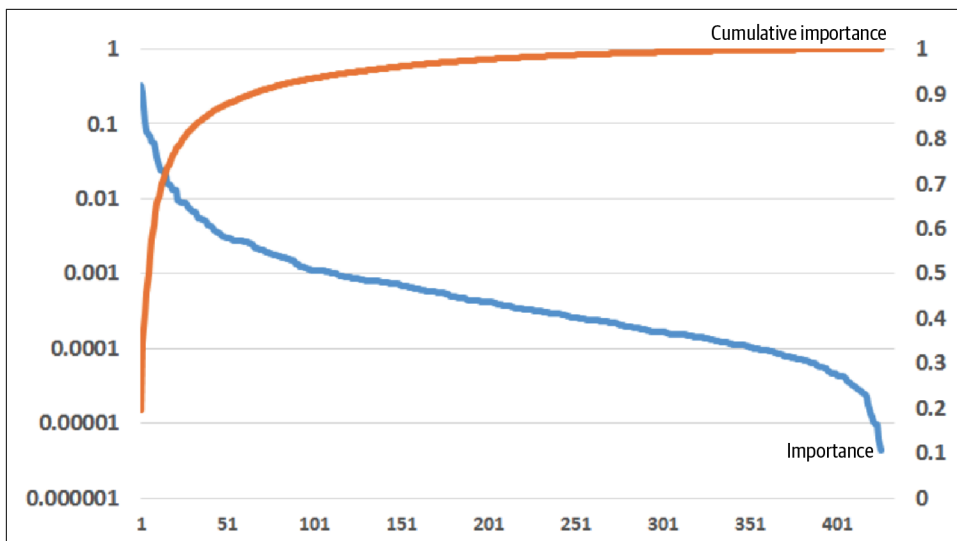


Figure 5-10. Boosting feature importance. X-axis corresponds to the number of features. Feature importance is in log scale. Source: He et al.

Not only good for choosing the right features, feature importance techniques are also great for interpretability as they help you understand how your models work under the hood.

## Feature Generalization

Since the goal of an ML model is to make correct predictions on unseen data, features used for the model should generalize to unseen data. Not all features generalize equally. For example, for the task of predicting whether a comment is spam, the identifier of each comment is not generalizable at all and shouldn't be used as a feature for the model. However, the identifier of the user who posts the comment, such as username, might still be useful for a model to make predictions.

<sup>20</sup> Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, et al., "Practical Lessons from Predicting Clicks on Ads at Facebook," in *ADKDD '14: Proceedings of the Eighth International Workshop on Data Mining for Online Advertising* (August 2014): 1–9, <https://oreil.ly/dHXeC>.

Measuring feature generalization is a lot less scientific than measuring feature importance, and it requires both intuition and subject matter expertise on top of statistical knowledge. Overall, there are two aspects you might want to consider with regards to generalization: feature coverage and 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. A rough rule of thumb is that if this feature appears in a very small percentage of your data, it's not going to be very generalizable. For example, if you want to build a model to predict whether someone will buy a house in the next 12 months and you think that the number of children someone has will be a good feature, but you can only get this information for 1% of your data, this feature might not be very useful.

This rule of thumb is rough because some features can still be useful even if they are missing in most of your data. This is especially true when the missing values are not at random, which means having the feature or not might be a strong indication of its value. For example, if a feature appears only in 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 (such as it appears in 90% of the examples in the train split but only in 20% of the examples in the 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. If the set of values that appears in the seen data (such as the train split) has no overlap with the set of values that appears in the unseen data (such as the test split), this feature might even hurt your model's performance.

As a concrete example, imagine you want to build a model to estimate the time it will take for a given taxi ride. You retrain this model every week, and you want to use the data from the last six days to predict the ETAs (estimated time of arrival) for today. One of the features is `DAY_OF_THE_WEEK`, which you think is useful because the traffic on weekdays is usually worse than on the weekend. This feature coverage is 100%, because it's present in every feature. However, in the train split, the values for this feature are Monday to Saturday, whereas in the test split, the value for this feature is Sunday. If you include this feature in your model without a clever scheme to encode the days, it won't generalize to the test split, and might harm your model's performance.

On the other hand, `HOUR_OF_THE_DAY` is a great feature, because the time in the day affects the traffic too, and the range of values for this feature in the train split overlaps with the test split 100%.

When considering a feature's generalization, there's a trade-off between generalization and specificity. You might realize that the traffic during an hour only changes depending on whether that hour is the rush hour. So you generate the feature `IS_RUSH_HOUR` and set it to 1 if the hour is between 7 a.m. and 9 a.m. or between 4 p.m. and 6 p.m. `IS_RUSH_HOUR` is more generalizable but less specific than `HOUR_OF_THE_DAY`. Using `IS_RUSH_HOUR` without `HOUR_OF_THE_DAY` might cause models to lose important information about the hour.

## Summary

Because the success of today's ML systems still depends on their features, it's important for organizations interested in using ML in production to invest time and effort into feature engineering.

How to engineer good features is a complex question with no foolproof answers. The best way to learn is through experience: trying out different features and observing how they affect your models' performance. It's also possible to learn from experts. I find it extremely useful to read about how the winning teams of Kaggle competitions engineer their features to learn more about their techniques and the considerations they went through.

Feature engineering often involves subject matter expertise, and subject matter experts might not always be engineers, so it's important to design your workflow in a way that allows nonengineers to contribute to the process.

Here is a summary of best practices for feature engineering:

- Split data by time into train/valid/test splits instead of doing it randomly.
- If you oversample your data, do it after splitting.
- Scale and normalize your data after splitting to avoid data leakage.
- Use statistics from only the train split, instead of the entire data, to scale your features and handle missing values.

- Understand how your data is generated, collected, and processed. Involve domain experts if possible.
- Keep track of your data's lineage.
- Understand feature importance to your model.
- Use features that generalize well.
- Remove no longer useful features from your models.

With a set of good features, we'll move to the next part of the workflow: training ML models. Before we move on, I just want to reiterate that moving to modeling doesn't mean we're done with handling data or feature engineering. We are never done with data and features. In most real-world ML projects, the process of collecting data and feature engineering goes on as long as your models are in production. We need to use new, incoming data to continually improve models, which we'll cover in [Chapter 9](#).