Chapter 4. Data Journey and Data Storage

This chapter discusses data evolution throughout the lifecycle of a production pipeline. We'll also look at tools that are available to help manage that process.

As we discussed in the preceding chapters, data is a critical part of the ML lifecycle. As ML data and models change throughout the ML lifecycle, it is important to be able to identify, trace, and reproduce data issues and model changes. As this chapter explains, ML Metadata (MLMD), TensorFlow Metadata (TFMD), and TensorFlow Data Validation (TFDV) are important tools to help you do this. MLMD is a library for recording and retrieving metadata associated with ML workflows, which can help you analyze and debug various parts of an ML system that interact. TFMD provides standard representations of key pieces of metadata used when training ML models, including a schema that describes your expectations for the features in the pipeline's input data. For example, you can specify the expected type, valency, and range of permissible values in TFMD's schema format. You can then use a TFMD-defined schema in TFDV to validate your data, using the data validation process discussed in Chapter 2.

Finally, we'll also introduce some forms of data storage that are particularly relevant to ML, especially for today's increasingly large datasets such as

Common Crawl (380 TiB). In production environments, how you handle your data also determines a large component of your cost structure, the amount of effort required to produce results, and your ability to practice Responsible AI and meet legal requirements.

Data Journey

Understanding data provenance begins with a data journey. A data journey starts with raw features and labels. For supervised learning, the data describes a function that maps the inputs in the training and test sets to the labels. During training, the model learns the functional mapping from input to label in order to be as accurate as possible. The data transforms as part of this training process. Examples of such transformations include changing data formats and applying feature engineering. Interpreting model results requires understanding these transformations. Therefore, it is important to track data changes closely. The *data journey* is the flow of the data from one process to another, from the initial collection of raw data to the final model results, and its transformations along the way. *Data provenance* refers to the linking of different forms of the data as it is transformed and consumed by processes, which enables the tracing back of each instance of the data to the process that created it, and to the previous instance of it.

Artifacts are all the data and other objects produced by the pipeline components. This includes the raw data ingested into the pipeline,

transformed data from different stages, the schema, the model itself, metrics, and so on. Data provenance, or *lineage*, is the sequence of artifacts that are created as we move through the pipeline.

Tracking data provenance is key for debugging, understanding the training process, and comparing different training runs over time. This can help with understanding how particular artifacts were created, tracing through a given training run, and comparing training runs to understand why they produced different results. Data provenance tracking can also help organizations adhere to data protection regulations that require them to closely track personal data, including its origin, changes, and location. Furthermore, since the model itself is an expression of the training set data, we can look at the model as a transformation of the data itself. Data provenance tracking can also help us understand how a model has evolved and perhaps been optimized.

When done properly, ML should produce results that can be reproduced fairly consistently. Like code version control (e.g., using GitHub) and environment versioning (e.g., using Docker or Terraform), data versioning is important. *Data versioning* is version control for datafiles that allows you to trace changes over time and readily restore previous versions. Data versioning tools are just starting to become available, and they include DVC, an open source version control system for ML projects, and Git Large File Storage (Git LFS), an open source Git extension for large file storage versioning.

ML Metadata

Every ML pipeline run generates metadata containing information about pipeline components, their executions, and the artifacts created. You can use this metadata to analyze and debug issues with your pipeline, understanding the interconnections between parts of your pipeline instead of viewing them in isolation. MLMD is a library for recording and accessing ML pipeline metadata, which you can use to track artifacts and pipeline changes during the pipeline lifecycle.

MLMD registers metadata in a Metadata Store, which provides APIs to record metadata in and retrieve metadata from a pluggable storage backend (e.g., SQLite or MySQL). MLMD can register:

- Metadata about artifacts—the inputs and outputs of the ML pipeline components
- Metadata about component executions
- Metadata about contexts, or shared information for a group of artifacts and executions in a workflow (e.g., project name or commit ID)

MLMD also allows you to define types for artifacts, executions, and contexts that describe the properties of those types. In addition, MLMD records information about relationships between artifacts and executions (known as *events*), artifacts and contexts (known as *attributions*), and executions and contexts (known as *associations*).

By recording this information, MLMD enables functionality to help understand, synthesize, and debug complex ML pipelines over time, such as:

- Finding all models trained from a given dataset
- Comparing artifacts of a given type (e.g., comparing models)
- Examining how a given artifact was created
- Determining whether a component has already processed a given input
- Constructing a directed acyclic graph (DAG) of the component executions in a pipeline

Using a Schema

Another key tool for managing data in an ML pipeline is a *schema*, which describes expectations for the features in the pipeline's input data and can be used to ensure that all input data meets those expectations.

A schema-based data validation process can help you understand how your ML pipeline data is evolving, assisting you in identifying and correcting data errors or updating the schema when the changes are valid. By examining schema evolution over time, you can gain an understanding of how the underlying input data has changed. In addition, you can use schemas to facilitate other processes that involve pipeline data, including things like feature engineering.

The TFMD library includes a schema protocol buffer, which can be used to store schema information, including:

- Names of all features in the dataset
- Feature type (int, float, string)
- Whether a feature is required in each example in the dataset
- Feature valency
- Value ranges or expected values
- How much the distribution of feature values is expected to shift across iterations of the dataset

TFMD and TFDV are closely related. You can use the schemas that you define with the TFMD-supplied protocol buffer in TFDV to efficiently ensure that every dataset you run through an ML pipeline conforms to the constraints articulated in that schema. For example, with a TFMD schema that specifies required feature values and types, you can use TFDV to identify as early as possible whether your dataset has anomalies—such as missing required values, values of the wrong type, and so on—that could negatively impact model training or serving. To do so, use TFDV's `generate_statistics_from_tfrecord()` function (or another input format–specific statistics generation function) to generate summary statistics for your dataset, and then pass those statistics and a schema to TFDV's `validate_statistics()` function. TFDV will return an Anomalies protocol buffer describing how (if at all) the input data deviates

from the schema. This process of checking your data against your schema is described in greater detail in Chapter 2.

Schema Development

TFMD and TFDV are closely related with respect to schema development as well as schema validation. Given the size of many input datasets, it may be cumbersome to generate a new schema manually. To help with schema generation, TFDV provides the `infer_schema()` function, which infers an initial TFMD schema based on summary statistics for an individual dataset. Although it is useful to have an auto-generated schema as a starting point, it is important to curate the schema to ensure that it fully and accurately describes expectations for the pipeline data. For example, schema inference will generate an initial list (or range) of valid values, but because it is generated from statistics for only a single dataset, it might not be comprehensive. Expert curation will ensure that a complete list is used.

TFDV includes various utility functions (e.g., `get_feature()` and `set_domain()`) to help you update the TFMD schema. You can also use TFDV's `display_schema()` function to visualize a schema in a Jupyter Notebook to further assist in the schema development process.

Schema Environments

Although schemas help ensure that your ML datasets conform to a shared set of constraints, it might be necessary to introduce variations in those constraints across different data (e.g., training versus serving data). Schema environments can be used to support these variations. You can associate a given feature with one or more environments using the `default_environment`, `in_environment`, and `not_in_environment` fields in the schema. You can then specify an environment to use for a given set of input statistics in `validate_statistics()`, and TFDV will filter the schema constraints applied based on the specified environment.

As an example, you can use schema environments where your data has a label feature that is required for training but will be missing in serving. To do this, have two default environments in your schema: Training and Serving. In the schema, associate the label feature only with the Training environment using the `not_in_environment` field, as follows:

```
default_environment: "Training"
default_environment: "Serving"
feature {
  name: "some_feature"
  type: BYTES
  presence {
```

```
      min_fraction: 1.0
    }
  }
  feature {
    name: "label_feature"
    type: BYTES
    presence {
      min_fraction: 1.0
    }
    not_in_environment: "Serving"
  }
```

Then, when you call `validate_statistics()` with training data, specify the Training environment, and when you call it with serving data, specify the Serving environment. Using the schema, TFDV will check that the label feature is present in every example in the training data and that the label feature is not present in the serving data.

Changes Across Datasets

You can use the schema to define your expectations about how data will change across datasets, both with respect to value distributions for individual features and with respect to the number of examples in the dataset as a whole.

As we discussed in Chapter 2, you can use TFDV to detect skew and drift between datasets, where skew looks at differences between two different data sources (e.g., training and serving data) and drift looks at differences across iterations of data from the same source (e.g., successive iterations of training data). You can articulate your expectations for how much feature value distributions should change across datasets using the `skew_comparator` and `drift_comparator` fields in the schema. If the feature value distributions shift more than the threshold specified in those fields, TFDV will raise an anomaly to flag the issue.

In addition to articulating the bounds of permissible feature value distribution shifts, the schema can specify expectations for how datasets as a whole differ. In particular, you can use the schema to express expectations about how the number of examples can change over time using the `num_examples_drift_comparator` field in the schema. TFDV will check that the ratio of the current dataset's number of examples to the previous dataset's number of examples is within the bounds specified by the `num_examples_drift_comparator`'s thresholds.

The schema can be used to articulate constraints beyond those noted in this discussion. Refer to the documentation in the TFMD schema protocol buffer file for the most current information about what the TFMD schema can express.

Enterprise Data Storage

Data is central to any ML effort. The quality of your data will strongly influence the quality of your models. Managing data in production environments affects the cost and resources required for your ML project, as well as your ability to satisfy ethical and legal requirements. Data storage is one aspect of that. The following sections should give you a basic understanding of some of the main types of data storage systems used for ML in production environments.

Feature Stores

A *feature store* is a central repository for storing documented, curated, and access-controlled features. A feature store makes it easy to discover and consume features that can be both online or offline, for both serving and training.

In practice, many modeling problems use identical or similar features, so the same data is often used in multiple modeling scenarios. In many cases, a feature store can be seen as the interface between feature engineering and model development. Feature stores are typically shared, centralized feature repositories that reduce redundant work among teams. They enable teams to both share data and discover data that is already available. It’s common to have different teams in an organization with different business problems

that they’re trying to solve; they’re pursuing different modeling efforts, but they’re using identical data or data that’s very similar. For these reasons, feature stores are becoming the predominant choice for enterprise data storage.

Feature stores often allow transformations of data so that you can avoid duplicating that processing in different individual pipelines. The access to the data in feature stores can be controlled based on role-based permissions. The data in the feature stores can be aggregated to form new features. The data can potentially be anonymized and even purged for things like wipeouts for General Data Protection Regulations (GDPR) compliance, for example. Feature stores typically allow for feature processing offline, which can be done on a regular basis, perhaps in a cron job, for example.

Imagine that you’re going to run a job to ingest data, and then maybe do some feature engineering on it and produce additional features from it (e.g., for feature crosses). These new features will also be published to the feature store, and other developers can discover and leverage them, often using metadata added with the new features. You might also integrate that with monitoring tools as you are processing and adjusting your data. Those processed features are stored for offline use. They can also be part of a prediction request, perhaps by doing a join with the raw data provided in the prediction request in order to pull in additional information.

Metadata

Metadata is a key component of all the features in the data that you store in a feature store. Feature metadata helps you discover the features you need. The metadata that describes the data you are keeping is a tool—and often the main tool for trying to discover the data you're looking for and understand its characteristics. The specific type of feature store you use will dictate how the metadata that describes your data can be added and searched within a feature store.

Precomputed features

For online feature usage where predictions must be returned in real time, the latency requirements are typically fairly strict. You're going to need to make sure you have fast access to that data. If you're going to do a join, for example, maybe with user account information along with individual requests, that join has to happen quickly, but it's often challenging to compute features in a performant manner online. So having precomputed features is often a good idea. If you precompute and store those features, you can use them later, and typically that's at fairly low latency. You can also do the precomputing in a batch environment.

Time travel

However, when you're training your model, you need to make sure you only include data that will be available when a serving request is made. Including data that is only available at some time after a serving request is referred to as *time travel*, and many feature stores include safeguards to avoid that. For example, consider data about events, where each example has a timestamp. Including examples with a timestamp that is after the point in time that the model is predicting would provide information that will not be available to the model when it is served. For example, when trying to predict the weather for tomorrow, you should not include data from tomorrow.

Data Warehouses

Data warehouses were originally developed for big data and business intelligence applications, but they're also valuable tools for production ML. A *data warehouse* is a technology that aggregates data from one or more sources so that it can be processed and analyzed. A data warehouse is usually meant for long-running batch jobs, and their storage is optimized for read operations. Data entering the warehouse may not be in real time.

When you're storing data in a data warehouse, your data needs to follow a consistent schema. A data warehouse is subject oriented, and the information stored in it revolves around a topic. For example, data stored in

a data warehouse may be focused on the organization’s customers or its vendors. The data in a data warehouse is often collected from multiple types of sources, such as relational databases or files. The data collected in a data warehouse is usually timestamped to maintain the context of when it was generated.

Data warehouses are nonvolatile, which means the previous versions of data are not erased when new data is added. That means you can access the data stored in a data warehouse as a function of time, and understand how that data has evolved.

Data warehouses offer an enhanced ability to analyze your data by timestamping your data. A data warehouse can help you maintain contexts. When you store your data in a data warehouse, it follows a consistent schema, and that helps improve the data’s quality and consistency. Studies have shown that the return on investment for data warehouses tends to be fairly high for many use cases. Lastly, the read and query efficiency from data warehouses is typically high, giving you fast access to your data.

DATA WAREHOUSE OR DATABASE?

You're probably familiar with databases. A natural question is, what's the difference between a data warehouse and a database?

Data warehouses are meant for analyzing data, whereas databases are often used for transaction purposes. Inside a data warehouse, there may be a delay between storing the data and the data becoming available for read operations. In a database, data is usually available immediately after it's stored. Data warehouses store data as a function of time, and therefore, historical data is also available. Data warehouses are typically capable of storing a larger amount of data compared to databases. Queries in data warehouses are complex in nature and tend to run for a long time, whereas queries in databases are relatively simple and tend to run in real time. Normalization is not necessary for data warehouses, but it should be used with databases.

Data Lakes

A *data lake* stores data in its raw format, which is usually in the form of binary large objects (blobs) or files. A data lake, like a data warehouse, aggregates data from various sources of enterprise data. A data lake can include structured data such as relational databases, semi-structured data such as CSV files, or unstructured data such as a collection of images or

documents. Since data lakes store data in its raw format, they don’t do any processing, and they usually don’t follow a schema.

It is important to be aware of the potential for a data lake to turn into a data swamp if it is not properly managed. A *data swamp* occurs when it becomes difficult to retrieve useful or relevant data, undermining the purpose of storing your data in the first place. Thus, when setting up a data lake, it is important to understand how the stored data will be identified and retrieved and to ensure that the data is added to the lake with the metadata necessary to support such identification and retrieval.

DATA LAKE OR DATA WAREHOUSE?

The primary difference between a data lake and a data warehouse is that in a data warehouse, data is stored in a consistent format that follows a schema, whereas in data lakes, the data is usually in its raw format. In data lakes, the reason for storing the data is often not determined ahead of time. This is usually not the case for a data warehouse, where it’s usually stored for a specific purpose. Data warehouses are often used by business professionals as well, whereas data lakes are typically used only by data professionals such as data scientists. Since the data in data warehouses is stored in a consistent format, changes to the data can be complex and costly. Data lakes, however, are more flexible, and they make it easier to make changes to the data.

Conclusion

This chapter discussed data journeys in production ML pipelines and outlined how tools such as MLMD, TFMD, and TFDV can help you identify, understand, and debug how data and models change throughout the ML lifecycle in those pipelines. It also described the main types of data storage systems used in production ML, and considerations for determining the right place to store your production ML data.