Chapter 1. Introduction to Machine Learning Production Systems

The field of machine learning engineering is so vast that it can be easy to get lost in the different steps that are necessary to get a model from an experiment into a production deployment. Over the last few years, machine learning, novel machine learning concepts such as attention, and more recently, large language models (LLMs) have been in the news almost every day. However, very little discussion has focused on production machine learning, which brings machine learning into products and applications.

Production machine learning covers all areas of machine learning beyond simply training a machine learning model. Production machine learning can be viewed as a combination of machine learning development practices and modern software development practices. Machine learning pipelines build the foundation for production machine learning. Implementing and executing machine learning pipelines are key aspects of production machine learning.

In this chapter, we will introduce the concept of production machine learning. We'll also introduce what machine learning pipelines are, look at their benefits, and walk through the steps of a machine learning pipeline.

What Is Production Machine Learning?

In an academic or research setting, modeling is relatively straightforward. Typically, you have a dataset (often a standard dataset that is supplied to you, already cleaned and labeled), and you're going to use that dataset to train your model and evaluate the results.

The result you're trying to achieve is simply a model that makes good predictions. You'll probably go through a few iterations to fully optimize the model, but once you're satisfied with the results, you're typically done.

Production machine learning (ML) requires a lot more than just a model. We've found that a model usually contains only about 5% of the code that is required to put an ML application into production. Over their lifetimes, production ML applications will be deployed, maintained, and improved so that you can consistently deliver a high-quality experience to your users.

Let's look at some of the differences between ML in a nonproduction environment (generally research or academia) and ML in a production environment:

- In an academic or research environment, you're typically using a static dataset. Production ML uses real-world data, which is dynamic and usually shifting.

- For academic or research ML, there is one design priority, and usually it is to achieve the highest accuracy over the entire training set. But for production ML, there are several design priorities, including fast inference, fairness, good interpretability, acceptable accuracy, and cost minimization.
- Model training for research ML is based on a single optimal result, and the tuning and training necessary to achieve it. Production ML requires continuous monitoring, assessment, and retraining.
- Interpretability and fairness are very important for any type of ML modeling, but they are absolutely crucial for production ML.
- And finally, while the main challenge with academic and research ML is to find and tune a high-accuracy model, the main challenge with production ML is a high-accuracy model plus the rest of the system that is required to operate the model in production.

In a production ML environment, you're not just producing a single result; you're developing a product or service that is often a mission-critical part of your offering. For example, if you're doing supervised learning, you need to make sure your labels are accurate. You also need to make sure your training dataset has examples that cover the same feature space as the requests your model will receive. In addition, you want to reduce the dimensionality of your feature vector to optimize system performance while retaining or enhancing the predictive information in your data.

Throughout all of this, you need to consider and measure the fairness of your data and model, especially for rare conditions. In fields such as health care, for example, rare but important conditions may be absolutely critical to success.

On top of all that, you're putting a piece of software into production. This requires a system design that includes all the things necessary for any production software deployment, including the following:

- Data preprocessing methods
- Parallelized model training setups
- Repeatable model analysis
- Scalable model deployment

Your production ML system needs to run automatically so that you're continuously monitoring model performance, ingesting new data, retraining as needed, and redeploying to maintain or improve performance.

And of course, you need to try to build your production ML system so that it achieves maximal performance at a minimal cost. That might seem like a daunting task, but the good news is that there are well-established tools and methodologies for doing this.

Benefits of Machine Learning Pipelines

When new training data becomes available, a workflow that includes data validation, preprocessing, model training, analysis, and deployment should be triggered. The key benefit of ML pipelines lies in automation of the steps in the model lifecycle. We have observed too many data science teams manually going through these steps, which is both costly and a source of errors. Throughout this book, we will introduce tools and solutions to automate your ML pipelines.

Let’s take a more detailed look at the benefits of building ML pipelines.

Focus on Developing New Models, Not on Maintaining Existing Models

Automated ML pipelines free up data scientists from maintaining existing models for large parts of their lifecycle. It’s not uncommon for data scientists to spend their days keeping previously developed models up-to-date. They run scripts manually to preprocess their training data, they write one-off deployment scripts, or they manually tune their models. Automated pipelines allow data scientists to develop new models—the fun part of their job. Ultimately, this will lead to higher job satisfaction and retention in a competitive job market.

Prevention of Bugs

Automated pipelines can prevent bugs. As we will explain in later chapters, newly created models will be tied to a set of versioned data, and preprocessing steps will be tied to the developed model. This means that if new data is collected, a new version of the model will be generated. If the preprocessing steps are updated, the training data will become invalid and a new model will be generated.

In manual ML workflows, a common source of bugs is a change in the preprocessing step after a model was trained. In such a case, we would deploy a model with different processing instructions than what we trained the model with. These bugs might be really difficult to debug, since an inference of the model is still possible but is simply incorrect. With automated workflows, these errors can be prevented.

Creation of Records for Debugging and Reproducing Results

In a well-structured pipeline, experiment tracking generates a record of the changes made to a model. This form of model release management enables data scientists to keep track of which model was ultimately selected and deployed. This record is especially valuable if the data science team needs to re-create the model, create a new variant of the model, or track the model’s performance.

Standardization

Standardized ML pipelines improve the work experience of a data science team. Not only can data scientists be onboarded quickly, but they also can move across teams and find the same development environments. This improves efficiency and reduces the time spent getting set up on a new project.

The Business Case for ML Pipelines

In short, the implementation of automated ML pipelines leads to four key benefits for a data science team:

- More development time to spend on novel models
- Simpler processes to update existing models
- Less time spent on reproducing models
- Good information about previously developed models

All of these aspects will reduce the costs of data science projects. Automated ML pipelines will also do the following:

- Help detect potential biases in the datasets or trained models, which can prevent harm to people who interact with the model (e.g., Amazon's ML-powered resume screener was found to be biased against females).

- Create a record (via experiment tracking and model release management) that will assist if questions arise around data protection laws, such as AI regulations in Europe or an AI Bill of Rights in the United States.
- Free up development time for data scientists and increase their job satisfaction.

When to Use Machine Learning Pipelines

Production ML and ML pipelines provide a variety of advantages, but not every data science project needs a pipeline. Sometimes data scientists simply want to experiment with a new model, investigate a new model architecture, or reproduce a recent publication. Pipelines wouldn't be useful in these cases. However, as soon as a model has users (e.g., it is being used in an app), it will require continuous updates and fine-tuning. In these situations, you need an ML pipeline. If you're developing a model that is intended to go into production and you feel fairly confident about the design, starting in a pipeline will save time later when you're ready to graduate your model to production.

Pipelines also become more important as an ML project grows. If the dataset or resource requirements are large, the ML pipeline approach allows for easy infrastructure scaling. If repeatability is important, even when

you’re only experimenting, it is provided through the automation and the audit trail of ML pipelines.

Steps in a Machine Learning Pipeline

An ML pipeline starts with the ingestion of new training data and ends with the receipt of some kind of feedback on how your newly trained model is performing. This feedback can be a production performance metric, or it can be feedback from users of your product. The pipeline comprises a number of steps, including data preprocessing, model training, model analysis, and model deployment.

As you can see in Figure 1-1, the pipeline is actually a recurring cycle. Data can be continuously collected, and therefore, ML models can be updated. More data generally means improved models. And because of this constant influx of data, automation is key.

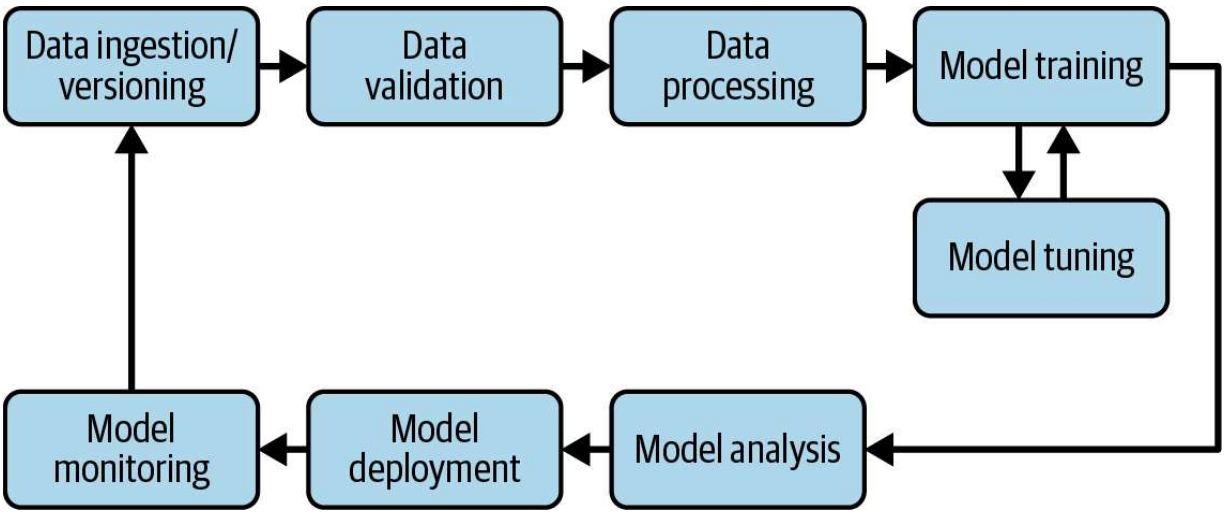


Figure 1-1. The steps in an ML pipeline

In real-world applications, you want to retrain your models frequently. If you don't, in many cases accuracy will decrease because the training data is different from the new data on which the model is making predictions. If retraining is a manual process, where it is necessary to manually validate the new training data or analyze the updated models, a data scientist or ML engineer would have no time to develop new models for entirely different business problems.

Let's discuss the steps that are most commonly included in an ML pipeline.

Data Ingestion and Data Versioning

Data ingestion occurs at the beginning of every ML pipeline. During this step, we process the data into a format that the components that follow can digest. The data ingestion step does not perform any feature engineering; this happens after the data validation step. This is also a good time to

version the incoming data to connect a data snapshot with the trained model at the end of the pipeline.

Data Validation

Before training a new model version, we need to validate the new data. Data validation (discussed in detail in Chapter 2) focuses on checking that the statistics of the new data—for example, the range, number of categories, and distribution of categories—are as expected. It also alerts the data scientist if any anomalies are detected.

For example, say you are training a binary classification model, and 50% of your training data consists of Class X samples and 50% consists of Class Y samples. Data validation tools would alert you if the 50/50 split between these classes changes to, say, 70/30. If a model is being trained with such an imbalanced training set and you haven't adjusted the model's loss function or over-/under-sampled one of the sample categories, the model predictions could be biased toward the dominant category.

Data validation tools will also allow a data scientist to compare datasets and highlight anomalies. If the validation highlights anything out of the ordinary, the pipeline can be stopped and the data scientist can be alerted. If a shift in the data is detected, the data scientist or the ML engineer can either change the sampling of the individual classes (e.g., only pick the

same number of examples from each class), or change the model's loss function, kick off a new model build pipeline, and restart the lifecycle.

Feature Engineering

It is highly likely that you cannot use your freshly collected data and train your ML model directly. In almost all cases, you will need to preprocess the data to use it for your training runs. That preprocessing is referred to as *feature engineering*. Labels often need to be converted to one-hot or multi-hot vectors. The same applies to the model inputs. If you train a model from text data, you want to convert the characters of the text to indices, or convert the text tokens to word vectors. Since preprocessing is only required prior to model training and not with every training epoch, it makes the most sense to run the preprocessing in its own lifecycle step before training the model.

Data preprocessing tools can range from a simple Python script to elaborate graph models. It's important that, when changes to preprocessing steps happen, the previous training data should become invalid and force an update of the entire pipeline.

Model Training and Model Tuning

Model training is the primary goal of most ML pipelines. In this step, we train a model to take inputs and predict an output with the lowest error

possible. With larger models, and especially with large training sets, this step can quickly become difficult to manage. Since memory is generally a finite resource for our computations, efficient distribution of model training is crucial.

Model tuning has seen a great deal of attention lately because it can yield significant performance improvements and provide a competitive edge. Depending on your ML project, you may choose to tune your model before you start to think about ML pipelines, or you may want to tune it as part of your pipeline. Because our pipelines are scalable thanks to their underlying architecture, we can spin up a large number of models in parallel or in sequence. This lets us pick out the optimal model hyperparameters for our final production model.

Model Analysis

Generally, we would use accuracy or loss to determine the optimal set of model parameters. But once we have settled on the final version of the model, it’s extremely useful to carry out a more in-depth analysis of the model’s performance. This may include calculating other metrics such as precision, recall, and area under the curve (AUC), or calculating performance on a larger dataset than the validation set used in training.

An in-depth model analysis should also check that the model’s predictions are fair. It’s impossible to tell how the model will perform for different

groups of users unless the dataset is sliced and the performance is calculated for each slice. We can also investigate the model's dependence on features used in training and explore how the model's predictions would change if we altered the features of a single training example.

Similar to the model-tuning step and the final selection of the best-performing model, this workflow step requires a review by a data scientist. The automation will keep the analysis of the models consistent and comparable against other analyses.

Model Deployment

Once you have trained, tuned, and analyzed your model, it is ready for prime time. Unfortunately, too many models are deployed with one-off implementations, which makes updating models a brittle process.

Model servers allow you to update model versions without redeploying your application. This will reduce your application's downtime and reduce the amount of communication necessary between the application development team and the ML team.

Looking Ahead

In Chapters 20 and 21, we will introduce two examples of a production ML process in which we implement an ML pipeline from end to end. In those

examples, we'll use TensorFlow Extended (TFX), an open source, end-to-end ML platform that lets you implement ML pipelines exactly as you would for production systems.

But first, we will discuss the ML pipeline steps in more detail. We'll start with data collection, labeling, and validation, covered next.

Chapter 2. Collecting, Labeling, and Validating Data

In production environments, you discover some interesting things about the importance of data. We asked ML practitioners at Uber and Gojek, two businesses where data and ML are mission critical, about it. Here's what they had to say:

> *Data is the hardest part of ML and the most important piece to get right...Broken data is the most common cause of problems in production ML systems.*
>
> —ML practitioner at Uber

> *No other activity in the machine learning lifecycle has a higher return on investment than improving the data a model has access to.*
>
> —ML practitioner at Gojek

The truth is that if you ask any production ML team member about the importance of data, you'll get a similar answer. This is why we're talking about data: it's incredibly important to success, and the issues for data in production environments are very different from those in the academic or research environment that you might be familiar with.

OK, now that we've gotten that out of the way, let's dive in!

Important Considerations in Data Collection

In programming language design, a *first-class citizen* in a given programming language is an entity that supports all the operations generally available to other entities. In ML, data is a first-class citizen. Finding data with predictive content might sound easy, but in reality it can be incredibly difficult.

When collecting data, it's important to ensure that the data represents the application you are trying to build and the problem you are trying to solve. By that we mean you need to ensure that the data has feature space coverage that is close to that of the prediction requests you will receive.

Another key part of data collection is sourcing, storing, and monitoring your data responsibly. This means that when you're collecting data, it is important to identify potential issues with your dataset. For example, the data may have come from different measurements of different types (e.g., the dataset may mix some measurements that come from two different types of thermometers that produce different measurements). In addition, simple things like the difference between an integer and a float, or how a missing value is encoded, can cause problems. As another example, if you have a

dataset that measures elevation, does an entry of 0 feet mean no elevation (sea level), or that no elevation data was received for that record? If the output of other ML models is the input dataset for your model, you also need to be aware of the potential for errors to propagate over time. And you want to make sure you're looking for potential problems early in the process by monitoring data sources for system issues and outages.

When collecting data, you will also need to understand data effectiveness by dissecting which features have predictive value. Feature engineering helps maximize the predictive signal of your data, and feature selection helps measure the predictive signal.

Responsible Data Collection

In this section, we will discuss how to responsibly source data. This involves ensuring data security and user privacy, checking for and ensuring fairness, and designing labeling systems that mitigate bias.

ML system data may come from different sources, including synthetic datasets you build, open source datasets, web scraping, and live data collection. When collecting data, data security and data privacy are important. *Data security* refers to the policies, methods, and means to secure personal data. *Data privacy* is about proper usage, collection, retention, deletion, and storage of data.

Data management is not only about the ML product. Users should also have control over what data is being collected. In addition, it is important to establish mechanisms to prevent systems from revealing user data inadvertently. When thinking about user privacy, the key is to protect personal identifiable information (PII). Aggregating, anonymizing, redacting, and giving users control over what data they share can help prevent issues with PII. How you handle data privacy and data security depends on the nature of the data, the operating conditions, and regulations currently in place (an example is the General Data Protection Regulation or GDPR, a European Union regulation on information privacy).

In addition to security and privacy, you must consider fairness. ML systems need to strike a delicate balance in being fair, accurate, transparent, and explainable. However, such systems can fail users in the following ways:

Representational harm

When a system amplifies or reflects a negative stereotype about particular groups

Opportunity denial

When a system makes predictions that have negative real-life consequences, which could result in lasting impacts

Disproportionate product failure

When you have skewed outputs that happen more frequently for a particular group of users

Harm by disadvantage

When a system infers disadvantageous associations between different demographic characteristics and the user behaviors around them

When considering fairness, you need to check that your model does not consistently predict different experiences for some groups in a problematic way, by ensuring group fairness (demographic parity and equalized odds) and equal accuracy.

One aspect of this is looking at potential bias in human-labeled data. For supervised learning, you need accurate labels to train your model on and to serve predictions. These labels usually come from two sources: automated systems and human raters. *Human raters* are people who look at the data and assign a label to it. There are various types of human raters, including generalists, trained subject matter experts, and users. Humans are able to label data in different ways than automated systems can. In addition, the more complicated the data is, the more you may require a human expert to look at that data.

When considering fairness with respect to human-labeled data, there are many things to think about. For instance, you will want to ensure rater pool diversity, and you will want to account for rater context and incentives. In

addition, you'll want to evaluate rater tools and consider cost, as you need a sufficiently large dataset. You will also want to consider data freshness requirements.

Labeling Data: Data Changes and Drift in Production ML

When thinking about data, you must also consider the fact that data changes often. There are numerous potential causes of data changes or problems, which can be categorized as those that cause gradual changes or those that cause sudden changes.

Gradual changes might reflect changes in the data and/or changes in the world that affect the data. Gradual data changes include those due to trends or seasonality, changes in the distribution of features, or changes in the relative importance of features. Changes in the world that affect the data include changes in styles, scope and process changes, changes in competitors, and expansion of a business into different markets or areas.

Sudden changes can involve both data collection problems and system problems. Examples of data collection problems that cause sudden changes in data include moved, disabled, or malfunctioning sensors or cameras, or problems in logging. Examples of system problems that can cause sudden

changes in data include bad software updates, loss of network connectivity, or a system delay or failure.

Thinking about data changes raises the issues of data drift and concept drift. With *data drift*, the distribution of the data input to your model changes. Thus, the data distribution on which the model was trained is different from the current input data to the model, which can cause model performance to decay in time. As an example of data drift, if you have a model that predicts customer clothing preferences that was trained with data collected mainly from teenagers, the accuracy of that model would be expected to degrade if data from older adults is later fed to the model.

With *concept drift*, the relationship between model inputs and outputs changes over time, which can also lead to poorer model performance. For example, a model that predicts consumer clothing preferences might degrade over time as new trends, seasonality, and other previously unseen factors change the customer preferences themselves.

To handle potential data change, you must monitor your data and model performance continuously, and respond to model performance decays over time. When ground truth changes slowly (i.e., over months or years), handling data change tends to be relatively easy. Model retraining can be driven by model improvements, better data, or changes in software or systems. And in this case, you can use curated datasets built using crowd-based labeling.

When ground truth changes more quickly (i.e., over weeks), handling data change tends to become more difficult. In these cases, model retraining can be driven by the factors noted previously, but also by declining model performance. Here, datasets tend to be labeled using direct feedback or crowd-based labeling.

When ground truth changes even more quickly (i.e., over days, hours, or minutes), things become even more difficult. Here, model retraining can be driven by declining model performance, the desire to improve models, better training data availability, or software system changes. Labeling in this scenario could be through direct feedback (discussed next), or through weak supervision for applying labels quickly.

Labeling Data: Direct Labeling and Human Labeling

Training datasets need to be created using the data available to the organization, and models often need to be retrained with new data at some frequency. To create a current training dataset, examples must be labeled. As a result, labeling becomes an ongoing and mission-critical process for organizations doing production ML.

We will start our discussion of labeling data by taking a look at direct labeling and human labeling. *Direct labeling* involves gleaning information

from your system—for example, by tracking click-through rates. *Human labeling* involves having a person label examples with ground truth values—for example, by having a cardiologist label MRI scans as a subject matter expert rater. There are also other methods, including semi-supervised labeling, active learning, and weak supervision, which we will discuss in later chapters that address advanced labeling methods.

Direct labeling has several advantages: it allows for a training dataset to be continuously created, as labels can be added from logs or other system-collected information as data arrives; it allows labels to evolve and adapt quickly as the world changes; and it can provide strong label signals. However, there are situations in which direct labeling is not available or has disadvantages. For example, for some types of ML problems, labels cannot be gleaned from your system. In addition, direct labeling can require custom designs to fit your labeling processes with your systems.

In cases where direct labeling is useful, there are open source tools that you can use for log analysis. Two such tools are Logstash and Fluentd. Logstash is a data processing pipeline for collecting, transforming, and storing logs from different sources. Collected logs can then be sent to one of several types of outputs. Fluentd is a data collector that can collect, parse, transform, and analyze data. Processed data can then be stored or connected with various platforms. In addition, Google Cloud provides log analytics services for storing, searching, analyzing, monitoring, and alerting on logging data and events from Google Cloud and Amazon Web Services

(AWS). Other systems, such as AWS Elasticsearch and Azure Monitor, are also available for log processing and can be used in direct labeling.

With human labeling, raters examine data and manually assign labels. Typically, raters are recruited and given instructions to guide their assignment of ground truth values. Unlabeled data is collected and divided among the raters, often with the same data being assigned to more than one rater to improve quality. The labels are collected, and conflicting labels are resolved.

Human labeling allows more labels to be annotated than might be possible through other means. However, there are disadvantages to this approach. Depending on the dataset, it might be difficult for raters to assign the correct label, resulting in a low-quality dataset. Quality might also suffer due to rater inexperience and other factors. Human labeling can also be an expensive and slow process, and can result in a smaller training dataset than could be created through other methods. This is particularly the case for domains that require significant specialization or expertise to be able to label the data, such as medical imaging. In addition, human labeling is subject to the fairness considerations discussed earlier in this chapter.

Validating Data: Detecting Data Issues

As discussed, there are many ways in which your data can change or in which the systems that impact your data can cause unanticipated issues. Especially in light of the importance of data to ML systems, detecting such issues is essential. In this section, we will discuss common issues to look for in your data, and the concepts involved in detecting those issues. In the next section, we'll explore a specific tool for detecting such data issues.

As we noted earlier, issues can arise due to differences in datasets. One such issue or group of issues is drift, which, as we mentioned previously, involves changes in data over time. With *data drift*, the statistical properties of the input features change due to seasonality, events, or other changes in the world. With *concept drift*, the statistical properties of the labels change over time, which can invalidate the mapping found during training.

Skew involves changes between datasets, often between training datasets and serving datasets. *Schema skew* occurs when the training and serving datasets do not conform to the same schema. *Distribution skew* occurs when the distribution of values in the training and serving datasets differs.

Validating Data: TensorFlow Data

Validation

Now that you understand the basics of data issues and detection workflows, let’s take a look at TensorFlow Data Validation (TFDV), a library that allows you to analyze and validate data using Python and Apache Beam. Google uses TFDV to analyze and validate petabytes of data every day across hundreds or thousands of different applications that are in production. The library helps users maintain the health of their ML pipelines by helping them understand their data and detect data issues like those discussed in this chapter.

TFDV allows users to do the following:

- Generate summary statistics over their data
- Visualize those statistics, including visually comparing two datasets
- Infer a schema to express the expectations for their data
- Check the data for anomalies using the schema
- Detect drift and training–serving skew

Data validation in TFDV starts with generating summary statistics for a dataset. These statistics can include feature presence, values, and valency, among other things. TFDV leverages Apache Beam’s data processing capabilities to compute these statistics over large datasets.

Once TFDV has computed these summary statistics, it can automatically create a schema that describes the data by defining various constraints including feature presence, value count, type, and domain. Although it is useful to have an automatically inferred schema as a starting point, the expectation is that users will tweak or curate the generated schema to better reflect their expectations about their data.

With a refined schema, a user can then run anomaly detection using TFDV. TFDV can do several types of anomaly detection, including comparison of a single set of summary statistics to a schema to ensure that the data from which the statistics were generated conforms to the user's expectations. TFDV can also compare the data distributions between two datasets—again using TFDV-generated summary statistics—to help identify potential drift or training–serving skew (discussed further in the next section).

The results of TFDV's anomaly detection process can help users further refine the schema or identify potentially problematic inconsistencies in their data. The schema can then be maintained over time and used to validate new data as it arrives.

Skew Detection with TFDV

Let's take a closer look at TFDV's ability to detect anomalies such as data drift and training–serving skew between datasets. For our discussion, *drift*

refers to differences across iterations of training data and *skew* refers to differences between training and serving data.

You can use TFDV to detect three types of skew: schema skew, feature skew, and distribution skew, as shown in Figure 2-1.

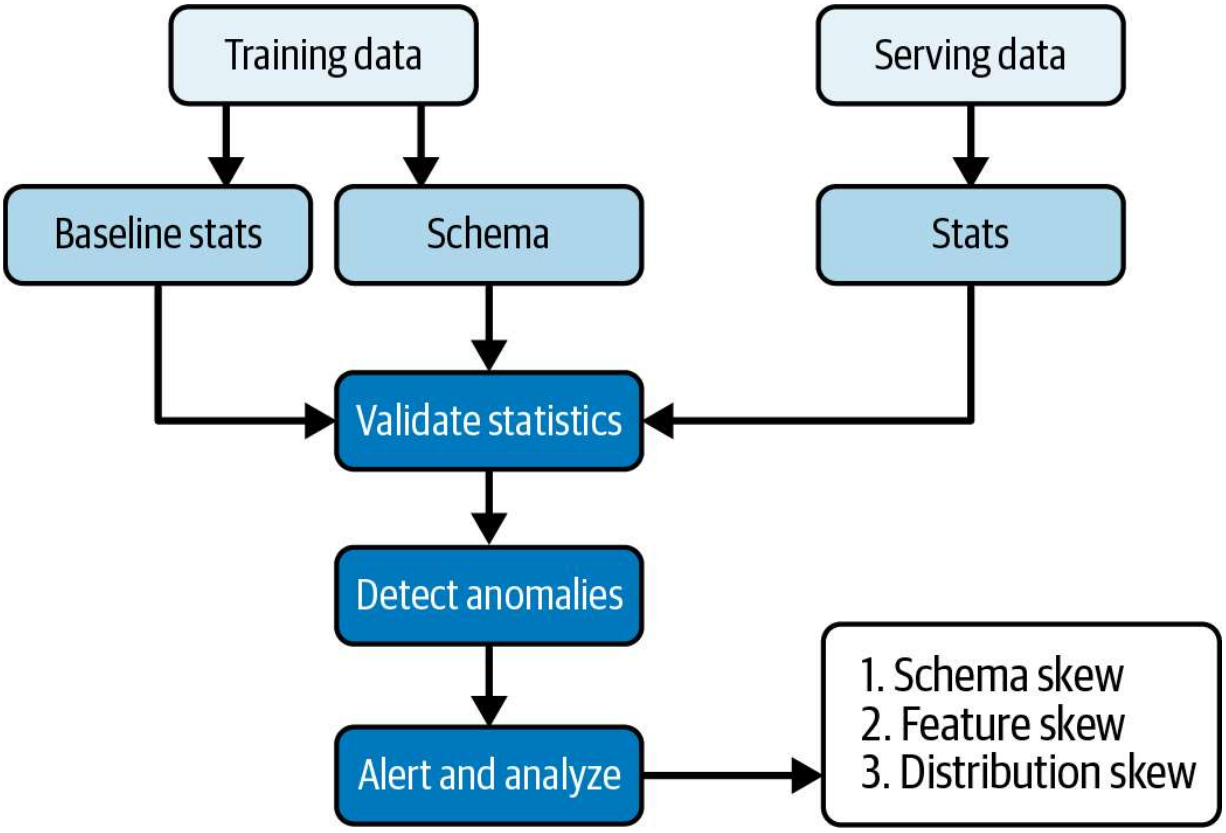


Figure 2-1. Skew detection with TFDV

Types of Skew

Schema skew occurs when the training data and serving data do not conform to the same schema; for example, if Feature A is a float in the training data but an integer in the serving data. Schema skew is detected

similarly to single-dataset anomaly detection, which compares the dataset to a specified schema.

Feature skew occurs where feature values that are supposed to be the same in both training data and serving data differ. To identify feature skew, TFDV joins the training and serving examples on one or more specified identifier features, and then compares the feature values to identify the resulting pairs. If they differ, TFDV reports the difference as feature skew. Because feature skew is computed using examples and not summary statistics, it is computed separately from the other validation steps.

Distribution skew occurs when there is a shift in the distribution of feature values across two datasets. TFDV uses L-infinity distance (for categorical features only) and Jensen–Shannon divergence (for numeric and categorical features) to identify and measure such shifts. If the measure exceeds a user-specified threshold, TFDV will raise a distribution skew anomaly noting the difference.

Various factors can cause the distribution of serving and training datasets to differ significantly, including faulty sampling during training, use of different data sources for training and serving, and trend, seasonality, or other changes over time. Once TFDV helps identify potential skew, you can investigate the shift to determine whether it’s a problem that needs to be remedied.

Example: Spotting Imbalanced Datasets with TensorFlow Data Validation

Let’s say you want to visually and programmatically detect whether your dataset is imbalanced. We consider datasets to be *imbalanced* if the sample quantities per label are vastly different (e.g., you have 100 samples for one category and 1,000 samples for another category). Real-world datasets will almost always be imbalanced for various reasons—for example, because the costs of acquiring samples for a certain category might be too high—but datasets that are too imbalanced hinder the model training process to generalize the overall problem.

TFDV offers simple ways to generate statistics of your datasets and check for imbalance. In this section, we’ll take you through the steps of using TFDV to spot imbalanced datasets.

Let’s start by installing the TFDV library:

```
$ pip install tensorflow-data-validation
```

If you have TFX installed, TFDV will automatically be installed as one of the dependencies.

With a few lines of code, we can analyze the data. First, let's generate the data statistics:

```
import tensorflow_data_validation as tfdv
stats = tfdv.generate_statistics_from_csv(
    data_location='your_data.csv',
    delimiter=',')
```

TFDV provides functions to load the data from a variety of formats, such as Pandas data frames (`generate_statistics_from_dataframe`) or TensorFlow's TFRecords (`generate_statistics_from_tfrecord`):

```
stats = tfdv.generate_statistics_from_tfrecord(
    data_location='your_data.tfrecord')
```

It even allows you to define your own data connectors. For more information, refer to the TFDV documentation.

If you want to programmatically check the label distribution, you can read the generated statistics. In our example, we loaded a spam detection dataset with data samples marked as `spam` or `ham` . As in every real-world example, the dataset contains more nonspam examples than spam examples. But how many? Let's check:

```
print(stats.datasets[0].features[0].string_stats
buckets {
  label: "ham"
  sample_count: 4827.0
}
buckets {
  low_rank: 1
  high_rank: 1
  label: "spam"
  sample_count: 747.0
}
```

The output shows that our dataset contains 747 spam examples and 4,827 ham (benign) examples.

Furthermore, you can use TFDV to quickly generate a visualization of statistics, as shown in Figure 2-2 for another dataset, with the following function:

```
tfdv.visualize_statistics(stats)
```

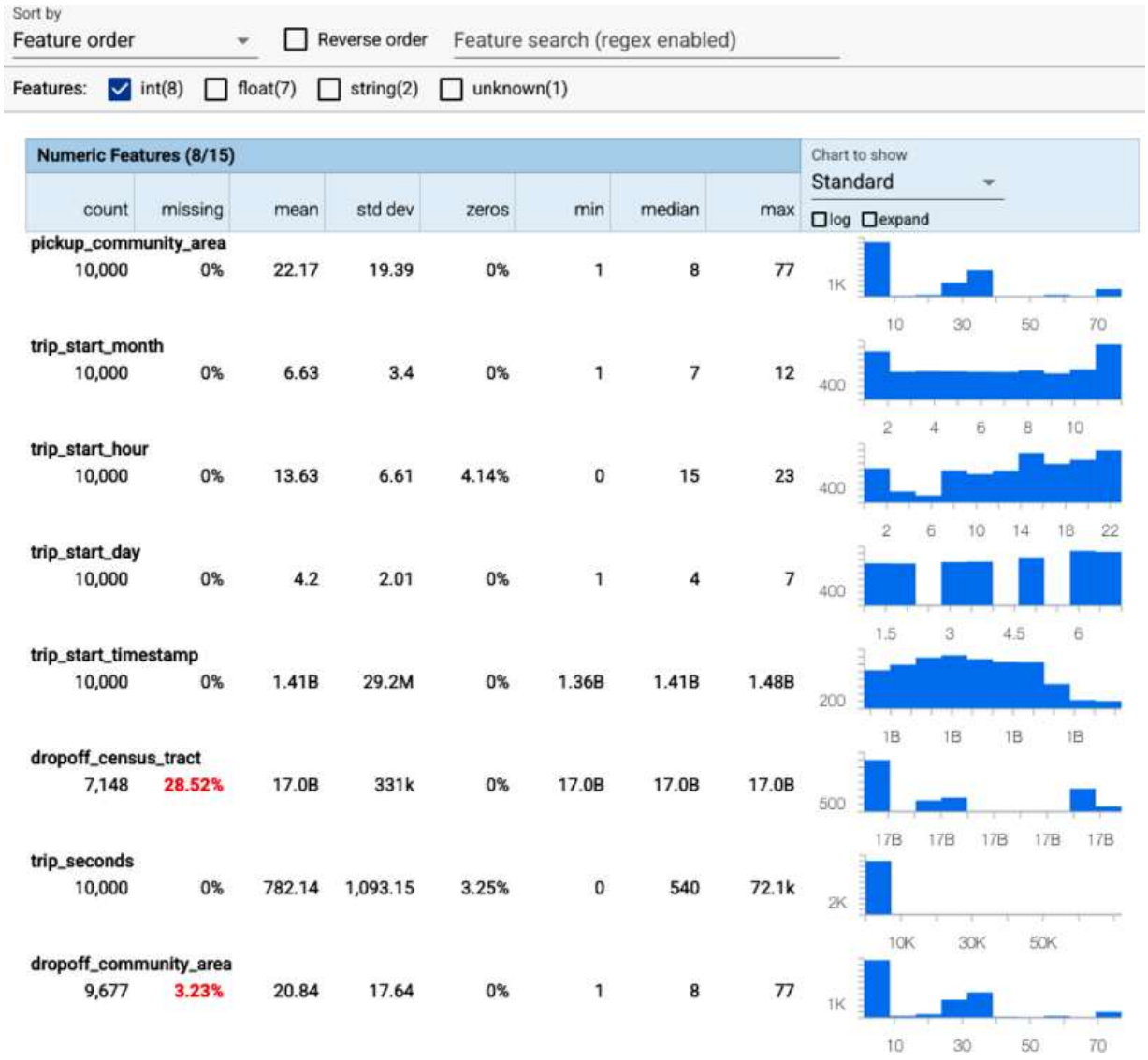


Figure 2-2. Visualizing a dataset

ALTERNATIVES TO TENSORFLOW DATA VALIDATION

While the simplicity of TFDV is amazing, data scientists might prefer a different analysis tool, especially if they don't use TensorFlow as their ML framework of choice. A number of open source data analysis tools have been released alongside TFDV. Following are some alternatives:

Great Expectations

Started as an open source project, but is now a commercial cloud solution. It allows you to connect with a number of data sources out of the box, including in-memory databases.

Evidently

Allows users to analyze and visualize datasets with a focus on dataset monitoring. It supports drift detection for unstructured text data.

Conclusion

In this chapter, we discussed the many things to consider when collecting and labeling the data used to train ML models. Given the importance of data to the health of your ML system, the potential issues with collecting and labeling data, and the potential for data changes for various and sometimes

difficult-to-foresee reasons, it is imperative to develop effective systems for managing and validating your data.

Chapter 3. Feature Engineering and Feature Selection

Feature engineering and feature selection are at the heart of data preprocessing for ML, especially for model training. Feature engineering is also required when performing inference, and it’s critical that the preprocessing that is done during inference matches the preprocessing that was done during training.

Some of the material in this chapter may seem like a review, especially if you’ve worked in ML in a nonproduction context such as in an academic or research setting. But we’ll be focusing on production issues in this chapter. One major issue we’ll discuss is how to perform feature engineering at scale in a reproducible and consistent way.

We’ll also discuss feature selection and why it’s important in a production context. Often, you will have more features than you actually need for your model, and your goal should be to only include those features that offer the most predictive information for the problem you’re trying to solve. Including more than that adds cost and complexity and can contribute to quality issues such as overfitting.

Introduction to Feature Engineering

> *Coming up with features is difficult, time-consuming, and requires expert knowledge. Applied machine learning often requires careful engineering of the features and dataset.*
>
> —Andrew Ng

Feature engineering is a type of preprocessing that is intended to help your model learn. Feature engineering is critical for making maximum use of your data, and it's a bit of an art form. The goal is to extract as much information as possible from your data, in a form that helps your model learn. The way that data is represented can have a big influence on how well a model is able to learn from it. For example, models tend to converge much more quickly and reliably when numerical data has been normalized. Therefore, the techniques for selecting and transforming the input data are key to increasing the predictive quality of the models, and dimensionality reduction is recommended whenever possible.

In feature engineering, we need to make sure the most relevant information is preserved, while both the representation and the predictive signal are enhanced and the required compute resources are reduced. Remember, in production ML, compute resources are a key contributor to the cost of running a model, both in training and in inference.

The art of feature engineering is to improve your model's ability to learn while reducing, if possible, the compute resources your model requires. It does this by transforming, projecting, eliminating, and/or combining the features in your raw data to form a new version of your dataset. Like many things in ML, this tends to be an iterative process that evolves over time as your data and model evolve.

Feature engineering is usually applied in two fairly different ways. During training, you typically have the entire dataset available to you. This allows you to use global properties of individual features in your feature engineering transformations. For example, you can compute the standard deviation of a feature across all your examples and then use it to perform standardization.

When you serve your trained model, you must do exactly the same feature engineering on the incoming prediction requests so that you give your model the same types of data it was trained on. For example, if you created a one-hot vector for a categorical feature when you trained, you need to also create an equivalent one-hot vector when you serve your model.

But when serving, you don't have the entire dataset to work with, and you typically process each request individually, so it's important that your serving process has access to the global properties of your features, such as the standard deviation. This means that if you used standard deviation during training, you need to include it with the feature engineering you do

when serving. Failing to do this is a very common source of problems in production systems, known as *training–serving skew*, and often these errors are difficult to find. We'll discuss this in more detail later in this chapter.

So, to review some key points, feature engineering can be very difficult and time-consuming, but it is also very important to success. You want to squeeze the most out of your data, and you do that using feature engineering. By doing this, you enable your models to learn better. You also want to make sure you concentrate predictive information and condense your data into as few features as possible to make the best and most cost-efficient use of your compute resources. And you need to make sure you apply the same feature engineering while serving as you applied during training.

Preprocessing Operations

Once, when we were first starting out, we got the idea that we could just skip normalizing our data. So we did. We trained a model, and of course, it wasn't converging. We started worrying about the model and code, and we forgot about the decision not to normalize, so we tried adjusting hyperparameters, changing the layers of the model, and looking for issues with the data. It took us a while to remember: *Oh yeah, we didn't normalize!* So we added the normalization, and of course the model started converging. D'oh! Well, we haven't made that particular mistake again.

In this section, we'll discuss the following preprocessing operations, which represent the main operations to perform on your data:

- Data wrangling and data cleansing
- Normalizing
- Bucketizing
- One-hot encoding
- Dimensionality reduction
- Image transformations

The first step in preprocessing is almost always some amount of data cleanup, which is commonly referred to as *data wrangling*. This includes basic things like making sure each feature in all the examples is of the correct data type and that the values are valid. Some of this can also spill over into feature engineering. During this step, we start, of course, with mapping raw data into features. Then, we look at different types of features, such as numerical features and categorical features. Our knowledge of the data should help guide the way toward our goal of engineering better features.

Also during this step, we perform *data cleansing*, which in broad terms consists of eliminating or correcting erroneous data. Part of this is domain dependent. For example, if your data is collected while a store is open and you know the store is not open at midnight, any data you have with a timestamp of midnight should probably be discarded.

You’ll often improve your results by performing per-feature transformations on your data, such as scaling, normalizing, or bucketizing your numeric values. For example, integer data can be mapped to floats, numerical data can be normalized, and one-hot vectors can be created from categorical values. Normalizing in particular helps with gradient descent.

Other types of transformation are more global in nature, affecting multiple features. For example, dimensionality reduction involves reducing the number of features, sometimes by projecting features to a different space. New features can be created by using several different techniques, including combining or deriving features from other features.

Text is an example of a class of data that has a whole world of transformations that are used for preprocessing. Models can only work with numerical data, so for text features, there are a number of techniques for creating numerical data from text. For example, if the text represents a category, techniques such as one-hot encoding are used. If there is a large number of categories, or if each text value may be unique, a vocabulary is generally used, with the feature converted to an index in the vocabulary. If the text is used in natural language processing (NLP) and the meaning of the text is important, an embedding space is used and the words in the feature value are represented as coordinates in the space. Text preprocessing also includes operations such as stemming and lemmatization, and normalization techniques such as term frequency–inverse document frequency (TF-IDF) and n-grams.

Images are similar to text in that a whole world of transformations can be applied to them during preprocessing. Techniques have been developed that can improve the predictive quality of images. These include rotating, flipping, scaling, clipping, resizing, cropping, or blurring images; using specialized filters such as Canny filters or Sobel filters; or implementing other photometric distortions. Transformations of image data are also widely used for data augmentation.

Feature Engineering Techniques

Feature engineering covers a wide range of operations on data that were originally applied in statistics and data science, as well as new techniques that were developed specifically for ML. A discussion of them all could easily be a book by itself, and in fact, several books have been written on this very topic. So in this section, we will highlight some of the most common techniques and provide you with a basic understanding of what feature engineering is and why it's important.

Normalizing and Standardizing

In general, all your numerical feature values should be normalized or standardized. As shown in the following equation, *normalization*, aka *min-max scaling*, shifts and scales your feature values to a range of [0,1]. *Standardization*, aka *z-score*, shifts and scales your feature values to a mean

of 0 with a standard deviation of 1, which is also shown in the following equation. Both normalizing and standardizing help your model learn by improving the ability of gradient descent to find minimas:

$$
\begin{array}{lll}
X_{\text{norm}} & = \dfrac{X - X_{\min}}{X_{\max} - X_{\min}} & \qquad X_{\text{std}} = \dfrac{X - \mu}{\sigma} \qquad (\text{z-score}) \\
X_{\text{norm}} & \in [0, 1] & \qquad X_{\text{std}} \sim \mathcal{N}(0, \sigma) \\
\text{Normalization (min-max)} & & \qquad \text{Standardization (z-score)}
\end{array}
$$

In both normalization and standardization, you need global attributes of your feature values. Normalization requires knowing both the min and max values, and standardization requires knowing both the mean and standard deviation. That means you must do a full pass over your data, examining every example in your dataset, to calculate those values. For large datasets, this can require a significant amount of processing.

The choice between normalization and standardization can often be based on experimenting to see which one produces better results, but it can also be informed by what you know about your data. If your feature values seem to be a Gaussian distribution, then standardization is probably a better choice. Otherwise, normalization is often a better choice. Note that normalization is also often applied as a layer in a neural network architecture, which helps with backpropagation by improving gradient descent.

Bucketizing

Numerical features can be transformed into categorical features through bucketizing. *Bucketizing* creates ranges of values, and each feature is assigned to a corresponding bucket if it falls into the range for that bucket.

Buckets can be uniformly spaced, or they can be spaced based on the number of values that fall into them to make them contain the same number of examples, which is referred to as *quantile bucketing*. Equally spaced buckets only require choosing the bucket size, but may result in some buckets having many more examples than others, and even some empty buckets. Quantile buckets require a full pass over the data to calculate the number of examples that would fall into each bucket of different sizes. Thus, in choosing how to bucketize, it is important to consider the distribution of your data. With more even distributions, use of equally spaced buckets—which will not require a full pass over the data—may be appropriate. If your data distribution is skewed, however, it may be worthwhile to do the full pass over your data to implement quantile bucketing.

Bucketizing is useful for features that are numerical but are really more categorical in nature for the model. For example, for geographical data, predicting the exact latitude and longitude may mask global characteristics of the data, while grouping into regions may reveal patterns.

Feature Crosses

Feature crosses combine multiple features together into a new feature. They encode nonlinearity in the feature space, or encode the same information with fewer features. We can create many different kinds of feature crosses, and it really depends on our data. It requires a little bit of imagination to look for ways to try to combine the features we have. For example, if we have numerical features, we could multiply two features and produce one feature that expresses the information in those two features. We can also take categorical features or even numerical features and combine them in ways that make sense semantically, capturing the meaning in fewer features.

For example, if we have two different features, the day of the week and the hour of the day, and we put them together, we can express this as the hour of the week. This results in a single feature that preserves the information that was previously in two features.

Dimensionality and Embeddings

Dimensionality reduction techniques are useful for reducing the number of input features in your models while retaining the greatest variance. Principal component analysis (PCA), the most widely known dimensionality reduction algorithm, projects your data into a lower-dimensional space along the principal components to reduce the data's dimensionality. Both t-distributed stochastic neighbor embedding (t-SNE)

and Uniform Manifold Approximation and Projection (UMAP) are also dimensionality reduction techniques, but they are often used for visualizing high-dimensional data in two or three dimensions.

Projecting your data into a lower-dimensional space for visualization is one kind of embedding. But often when we discuss embeddings, we're really referring to *semantic embedding spaces*, or *word embeddings*. These capture semantic relationships between different items in your data, most commonly for natural language. For example, the word *apple* will be much closer in meaning to the word *orange* since both are fruits, and more distant from the word *sailboat* since the two concepts have little in common. This kind of semantic embedding is widely used in natural language models, but it can also be used with images or any other item with a conceptual meaning. Data is projected into a semantic embedding space by training a model to understand the relationships between items, often through self-supervised training on very large datasets or *corpora*.

Visualization

Being able to visualize your data in a lower dimension is often very helpful for understanding the characteristics of your data, such as any clustering that might not be noticeable otherwise. In other words, it helps you develop an intuitive sense of your data. This is really where some of the art of feature engineering comes into play, where you as a developer form an understanding of your data. It's especially important for high-dimensional

data, because we as humans can visualize maybe three dimensions before things get really weird. Even four dimensions is hard, and 20 is impossible. Tools such as the TensorFlow embedding projector can be really valuable for this. This tool is free and a lot of fun to play with, but it's also a great tool to help you understand your data.

Feature Transformation at Scale

As you move from studying ML in a classroom setting or working as a researcher to doing production ML, you'll discover that it's one thing to do feature engineering in a notebook with maybe a few megabytes of data and quite another thing to do it in a production environment with maybe a couple of terabytes of data, implementing a repeatable, automated process.

In the past, when ML pipelines were in their infancy, data scientists would often use notebooks to create models in one language, such as Python, and then deploy them on a different platform, potentially rewriting their feature engineering code in a different language, such as Java. This translation from development to deployment would often create issues that were difficult to identify and resolve. A better approach has since developed in which ML practitioners use *pipelines*, unified frameworks to both train and deploy with consistent and reproducible results. Let's take a look at how to leverage such a system and do feature engineering at scale.

Choose a Framework That Scales Well

At scale, your training datasets could be terabytes of data, and you want each transformation to be as efficient as possible and make optimal use of your computing resources. So, when you're first writing your feature engineering code, it's often a good idea to start with a subset of your data and work out as many issues as possible before proceeding to the full dataset. You can use data processing frameworks on your development machine or in a notebook that are no different from what you're going to use at scale, as long as you choose a framework that scales well. But for production, it will be configured somewhat differently.

Apache Beam, for example, includes a Direct Runner, which can run directly on your laptop, and you can then swap that out for a Google Dataflow Runner or an Apache Flink Runner to scale up to your full dataset. In this way, Apache Beam scales well. Pandas, unfortunately, does not scale well, since it assumes that the entire dataset fits in memory and has no provision for distributed processing.

Avoid Training–Serving Skew

Consistent transformations between training and serving are incredibly important. Remember that any transformations you do on your training data will also need to be applied in exactly the same way to data from prediction requests when you serve your model. If you do different transformations

when you’re serving your model than you did when you were training it, or even if you use different code that *should* do the same thing, you are going to have problems, and those problems will often be very hard to find or even be aware of. Your model results may look reasonable and there may be no errors thrown, when in fact your model results are far below what you expect them to be because you’re giving your model bad data, or data that doesn’t match what the model was trained with. This is referred to as *training–serving skew*.

Inconsistencies in feature engineering, or training–serving skew, often result from using different code for transforming data for training and serving. When you are training your model, you have code that you’re using for training. If the codebase is different, such as using Python for training and Java for serving, that’s a potential source of problems. Initially, the solution to this problem might seem simple: just use the same code in both training and serving. But that might not be possible depending on your deployment scenario. For example, you might be deploying your model to a server cluster and using it on an Internet of Things (IoT) device, and you might not be able to use the same code in both environments due to differences in the configuration and resources available.

Consider Instance-Level Versus Full-Pass

Transformations

Depending on the transformations you're doing on your data, you may be able to take each example and transform it separately without referencing any other examples in the dataset, or you may need to analyze the entire dataset before doing any transformations. These are referred to, respectively, as *instance-level transformations* and *full-pass transformations*. Obviously, the compute requirements for full-pass transformations are much higher than for instance-level transformations, so full-pass transformations need to be carefully designed.

Even for something as basic as normalization, you need to determine the min, max, and standard deviation of your feature, and that requires examining every example, which means you need to do a full-pass transformation. If you have terabytes of data, that's a lot of processing. Contrast this with doing a simple multiplication for a feature cross, which can be done at the instance level. Bucketizing can similarly be done at the instance level, assuming you know ahead of time what the buckets are going to be; sometimes you need to do a full pass to determine which buckets make sense.

Once you've made a full pass to collect statistics like the min, max, and standard deviation of a numerical feature, it's best to save those values and include them in the configuration for your serving process so that you can use them at the instance level when doing transformations for prediction

requests. For normalization again, if you already have the min, max, and standard deviation, you can process each request separately. In fact, for online serving, since each request arrives at your server separately, it's usually very difficult to do anything analogous to a full pass. For batch serving, you can do a full pass, assuming your batch size is large and representative enough to be valid, but it's better if you can avoid this.

Using TensorFlow Transform

To do feature engineering at scale, we need good tools that scale well. TensorFlow Transform is a widely used and efficient tool for just this purpose. In this section, we'll go a bit deeper into how TensorFlow Transform (from this point on, simply referred to as "TF Transform") works, what it does, and why it does it. We'll look at the benefits of using TF Transform and how it applies feature transformations, and we'll look at some of TF Transform's analyzers and the role they play in doing feature engineering. Although TF Transform is a separate open source library that you can use by itself, we're going to primarily focus on using TF Transform in the context of a TensorFlow Extended (TFX) pipeline. We'll go into detail on TFX pipelines in Chapters 18 and 19, but for now, think of them as a complete training process designed to be used for production deployments.

TF Transform can be used for processing both the training data and the serving requests, especially if you're developing your model in TensorFlow. If you're not working with TensorFlow, you can still use TF Transform, but for serving requests you will need to use it outside of the model. When you use it with TensorFlow, the transformations done by TF Transform can be included in your model, which means you will have exactly the same transformations regardless of where you deploy your trained model for serving.

Looking at this in the context of a typical TFX pipeline, we're starting with our raw training data. (Although we'll be discussing a typical pipeline, TFX allows you to create nearly any pipeline architecture you can imagine.) We split it with ExampleGen, the first component in the pipeline. ExampleGen ingests and splits our data into training and eval splits by default, but that split is configurable.

The split dataset is then fed to the StatisticsGen component. StatisticsGen calculates statistics for our data, making a full pass over the dataset. For numeric features, for example, it calculates the mean, standard deviation, min, max, and so forth. For categorical features, it collects the valid categorical values that are included in the training data.

Those statistics get fed to the SchemaGen component, which infers the types of each feature. SchemaGen creates a schema that is then used by downstream components including ExampleValidator, which takes those

previously generated statistics and schema and looks for problems in the data. For instance, if we have examples that are the wrong type in a particular feature—perhaps we have an integer where we expected a float—ExampleValidator will flag that.

Transform is the next component in our typical pipeline. Transform will take the schema that was generated from the original training dataset and do our feature engineering based on the code we give it. The resulting transformed data is given to the Trainer and other downstream components.

Figure 3-1 shows a simplified TFX pipeline, with training data flowing through it and a trained model flowing to a serving system. Along the way, the data and various artifacts flow into and out of a metadata storage system. The details of the process are omitted from Figure 3-1 in order to present a high-level view. We'll cover those details in later chapters.

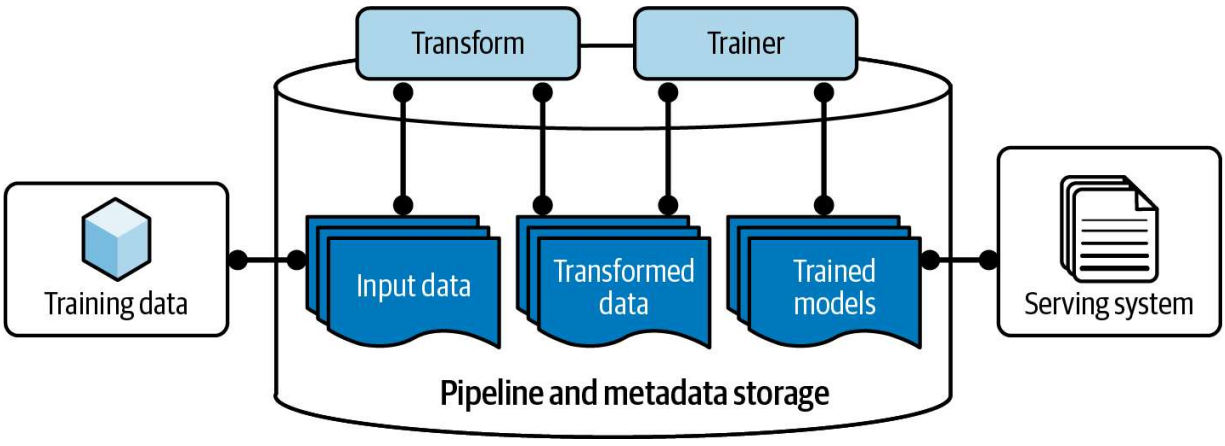


Figure 3-1. A simplified TFX pipeline that includes a Transform component

The Transform component gets inputs from ExampleGen, StatisticsGen, and SchemaGen, which include a dataset and a schema for the dataset. That schema, by the way, may very well have been reviewed and improved by a developer who knew more about what to expect from the data than can really be inferred by SchemaGen. That process is referred to as *curating the schema*. TF Transform also needs your user code because you need to express the feature engineering you want to do. For example, if you're going to normalize a feature, you need to give TF Transform user code to do that.

TF Transform creates the following:

- A TensorFlow graph, which is referred to as the *transform graph*
- The new schema and statistics for the transformed data
- The transformed data itself

The transform graph expresses all the transformations we are doing on our data, as a TensorFlow graph. The transformed data is simply the result of doing our transformations. Both the graph and the data are given to the Trainer component, which will use the transformed data for training and will include the transform graph prepended to the trained model.

Training a TensorFlow model creates a TensorFlow graph as a SavedModel. This is the computation graph of the model parameters and operations. Prepending the transform graph to the SavedModel is important because it

means we always do exactly the same transformations when we serve the model, regardless of where and how it is served, so there is no potential for training–serving skew. The transform graph is also optimized to capture the results of invariant transformations as constants, such as the standard deviation of numerical features.

Because TF Transform is designed to scale to very large datasets, it performs processing by using Apache Beam. This enables TF Transform to scale from running on a single CPU all the way to running on a large compute cluster, typically with changes in only one line of code.

Analyzers

Many data transformations require *calculations*, or the collection of statistics on the entire dataset. For example, whether you're doing something as simple as calculating the minimum value of a numerical feature or something as relatively advanced as PCA on a space described by a set of features, you require a full pass over the dataset, and since datasets can potentially comprise many terabytes of data, this can require extensive compute resources.

To perform these kinds of computations, TF Transform defines the concept of *Analyzers*. Analyzers perform individual operations on data, which include the following:

Functionality	Analyzer
Scaling	`scale_to_z_score` `scale_to_0_1`

Bucketizing	`quantiles`
	`apply_buckets`
	`bucketize`

Vocabulary	`bag_of_words`
	`tfidf`
	`ngrams`

Dimensionality reduction	`pca`

Analyzers use Apache Beam for processing, which enables scalability. Analyzers only run once for each model training workflow, and they do not run during serving. Instead, the results produced by each Analyzer are captured as constants in the transform graph and included with the SavedModel. Those constants are then used as part of transforming individual examples during both training and serving.

Code Example

Now let’s look at some code. We’re going to start by creating a preprocessing function, which is used to define the user code that expresses

the feature engineering you're going to do:

```
import tensorflow_transform as tft
def preprocessing_fn(inputs):
  ...
    <feature engineering code>
```

For example, we might want to normalize numeric features using a z-score:

```
for key in DENSE_FLOAT_FEATURE_KEYS:
    outputs[key] = tft.scale_to_z_score(inputs[ke
```

This is just an example. `DENSE_FLOAT_FEATURE_KEYS` is a list of feature names that you defined in advance. You're going to do whatever feature engineering you have to do, but it's this style of Python code that you're working with. Developing a vocabulary for a text-based categorical feature is very similar:

```
for key in VOCAB_FEATURE_KEYS:
    outputs[key] = tft.vocabulary(inputs[key], vo
```

We might also want to create some *bucket features*, which are numerical features that are assigned to a "bucket" based on ranges of values, to then become categorical features:

```
  for key in BUCKET_FEATURE_KEYS:
      outputs[key] = tft.bucketize(inputs[key], FEA
```

These are just examples, and not everything needs to be done in this “for loop” style.

In a production deployment, TF Transform typically uses Apache Beam to distribute processing across a compute cluster. During development, you can also use Beam on a single system—for example, you can just run it on your laptop, using the Direct Runner. In development, that’s pretty useful.

Feature Selection

In production, you will have various sources of data that you can give to your model. It’s almost always the case that some of the data available to you does not help your model learn and generate predictions. For example, if you’re trying to predict which ads a user in France will be interested in on a web page, giving your model data about the current temperature in Japan is unlikely to help your model learn.

Feature selection is a set of algorithms and techniques designed to improve the quality of your data by determining which features in your data actually help your model learn. In this section, we’ll discuss feature selection techniques, but we’ll start with a related concept, the idea of feature spaces.

Feature Spaces

A *feature space* is the *n*-dimensional space defined by your features. If you have two features, your feature space is two dimensional. If you have three features, it's three dimensional, and so forth. Feature spaces do not include the target label.

Feature spaces are easiest to understand for numeric features. The min and max values of each feature determine the range of each dimension of the space. Your model will only actually learn to predict from values in those ranges, although it will try to predict if you give it examples with values outside those ranges. How well it does this depends on the robustness of your model, which we will discuss later.

So, feature space coverage is important. Let's refer to the feature space defined by your training data as your *training feature space*, and the feature space defined by the data in prediction requests that your model will receive when you serve it in production as your *serving feature space*. Ideally, your training feature space should cover your entire serving feature space. It's even better if your training feature space is slightly larger than your serving feature space.

Keep in mind that the ranges of values for your serving features will change as your data drifts, so it's important to have monitoring in place to signal

when your prediction requests have drifted too much and your model needs to be retrained with new data.

The density of your training data in different regions of your feature space is also important. Your model is likely to be more accurate in regions with many examples than in regions with few examples. Often, the sheer number of examples in your training data is less important than the variety of examples and their coverage of your feature space. Beginning developers often make the mistake of assuming that more data is just automatically better, but if there are many duplicates or near duplicates in your data, your model is unlikely to be improved by more data.

Feature Selection Overview

Let’s get back to the main topic of this section, feature selection. You can think of feature selection as one part of optimizing your data. The goal is to only include the minimum number of features that provide the maximum amount of predictive information that will help your model learn.

We try to select features we actually need and eliminate the ones we don’t. That reduces the size of the feature space. Reducing the dimensionality in turn reduces the amount of training data required, and often increases the density of feature space coverage.

Each feature we include also adds resource requirements for gathering and maintaining the systems, bandwidth, and storage we need in order to create training datasets and supply that feature during serving. It also adds to model complexity and can even degrade model accuracy. And it increases the cost and complexity of serving the model, since there is more data to feed and more compute required for a larger, more complex model.

There are many feature selection algorithms, and (just like modeling) they can be both supervised and unsupervised. We'll now discuss some of the factors that will help you decide whether to choose supervised or unsupervised feature selection.

As the name implies, *unsupervised feature selection* does not consider the relationship between the features and the label. Instead, it's really looking for features that are correlated. When you have two or more features that are highly correlated, you really only need one of them, and you're going to try to select the one that gives you the best result.

Supervised feature selection is focused on the relationship between each feature and the label. It tries to assess the amount of predictive information (often referred to as *feature importance*) in each feature. Supervised feature selection algorithms include filter methods, wrapper methods, and embedded methods. The following sections introduce each class of algorithm.

Filter Methods

For filter methods, we're primarily using correlation to look for the features that contain the information we're going to use to predict our target. This may be univariate or multivariate, with univariate requiring less computation.

There are different ways to measure correlation, including the following:

- *Pearson correlation* is a way to measure correlation for linear relationships and is probably the most commonly used.
- *Kendall's Tau* is a rank correlation coefficient that looks at monotonic relationships and is usually used with a fairly small sample size for efficiency.
- *Spearman correlation* measures the strength and direction of monotonic association between two variables.

Besides correlation, there are other metrics that are used by some algorithms, including mutual information, F-test, and chi-squared.

Here's how to use Pandas to calculate the Pearson correlation for feature selection:

```
# Pearson correlation by default
cor = df.corr()
cor_target = abs(cor["feature_name"])
```

```
# Selecting highly correlated features to elimina
redundant_features = cor_target[cor_target>0.8]
```

Now let's look at univariate feature selection, using the scikit-learn package. This package offers several univariate algorithms, including SelectKBest, SelectPercentile, and GenericUnivariateSelect, which we assume is fairly generic. These support the use of statistical tests, including mutual information and F-tests for regression problems. For classification, scikit-learn offers chi-squared, a version of F-test for classification, and a version of mutual information for classification. Let's look at how univariate feature selection gets implemented in code:

```
def univariate_selection():
    X_train, X_test, Y_train, Y_test = train_test
                                           X, Y,
                                           test_s
                                           strati
                                           random


    X_train_scaled = StandardScaler().fit_transfo
    X_test_scaled = StandardScaler().fit_transfor
    min_max_scaler = MinMaxScaler()
    Scaled_X = min_max_scaler.fit_transform(X_tra
    selector = SelectKBest(chi2, k=20) # Use chi-
    X_new = selector.fit_transform(Scaled_X, Y_tr
    feature_idx = selector.get_support()
```

```
        feature_names = df.drop("diagnosis_int", axis
                                .columns[feature_idx]
        return feature_names
```

The preceding code represents a typical pattern for doing feature selection using scikit-learn.

Wrapper Methods

Wrapper methods are *supervised*, meaning they require the dataset to be labeled. Wrapper methods use models to measure the impact of either iteratively adding or removing features from the dataset. The heart of all wrapper methods is a process that:

- Chooses a set of features to include in the iteration
- Trains and evaluates a model using this set of features
- Compares the evaluation metric with metrics for other sets of features to determine the starting set of features for the next iteration

Wrapper methods tend to be more computationally demanding than other feature selection techniques, especially for large sets of potential features. The three main types of wrapper methods are forward selection, backward elimination, and recursive feature elimination.

Forward selection

Forward selection is an iterative, greedy search algorithm. We start with one feature, train a model, and evaluate the model performance. We repeat that process, keeping the previously added features and adding additional features, one at a time. In each round of tests, we're trying all the remaining features one by one, measuring the performance, and keeping the feature that gives the best performance for the next round. We keep repeating this until there's no improvement, at which point we know we've generated the best subset of our features.

You can see that forward selection requires training a new model for every iteration, and that the number of iterations grows exponentially with the number of potential features. Forward selection is a good choice to consider if you think your final feature set will be fairly small compared to the set of potential features.

Backward elimination

As the name implies, backward elimination is basically the opposite of forward selection. Backward elimination starts with all the features and evaluates the model performance when removing each feature. We remove the next feature, trying to get to better performance with fewer features, and we keep doing that until there's no improvement.

You can see that, like forward selection, backward elimination requires training a new model for every iteration, and that the number of iterations grows exponentially with the number of potential features. Backward elimination is a good choice to consider if you think your final feature set will be a majority of the set of potential features.

Recursive feature elimination

Recursive feature elimination uses feature importance to select which features to keep, rather than model performance. We begin by selecting the desired number of features that we want in the resulting set. Then, starting with the whole set of potential features, we train the model and eliminate one feature at a time. We rank the features by feature importance, which means we need to have a method of assigning importance to features. We then discard the least important features. We keep doing that until we get down to the number of features we intend to keep.

An important aspect of this is that we need to have a measurement of feature importance in our model, and not all models are able to do that. The most common class of models that offers the ability to measure feature importance is tree-based models. Another aspect is that we need to somehow decide in advance how many features we want to keep, which isn’t always obvious. Forward selection and backward elimination both find that number automatically, stopping when performance no longer improves.

Code example

For recursive feature elimination, this is what the code might look like when using scikit-learn:

```
def run_rfe(label_name, X, Y, num_to_keep):
  X_train, X_test, y_train, y_test = train_test_s
                                        X, Y,
                                        test_siz
                                        random_s

  X_train_scaled = StandardScaler().fit_transforr
  X_test_scaled = StandardScaler().fit_transform(
  model = RandomForestClassifier(criterion = 'ent
                                 random_state = 4

  rfe = RFE(model, n_features_to_select = num_to_
  rfe = rfe.fit(X_train_scaled, y_train)

  feature_names = df.drop(label_name, axis = 1)
                          .columns[rfe.get_support
  return feature_names
```

This code example uses a random forest classifier, which is one of the model types that measures feature importance.

Embedded Methods

Embedded methods for feature selection are largely a function of the model design itself. For example, L1 or L2 regularization is essentially an embedded method for doing a crude and inefficient form of feature selection, since they can have the effect of disabling features that do not significantly contribute to the result.

A much better example is the use of feature importance, which is a property of most tree-based model architectures, to select important features. This is well supported in many common frameworks, including scikit-learn, where the `SelectFromModel` method can be used for feature selection.

Notice that to use embedded methods, the model must be trained, at least to a reasonable level, to measure the impact of each feature on the result as expressed by feature importance. This leads to an iterative process, similar to forward selection, backward elimination, and recursive elimination, to measure the effectiveness of different sets of features.

Feature and Example Selection for LLMs and GenAI

The discussion so far has been on feature selection techniques that are more focused on classic and deep learning applications, with a goal of improving the quality of the training dataset. Recognition of the importance of data

quality has been extended to large language models (LLMs) and other generative AI (GenAI) applications, where it has been shown that improving the quality of a dataset has a significant impact on the results. This has led to the development of new techniques that are specifically focused on GenAI datasets, but in these cases the focus is usually on *example selection* instead of feature selection.

GenAI datasets, such as those that are used to pretrain LLMs, are typically huge collections of data that have been scraped from the internet. For example, the Common Crawl dataset can range in size from hundreds of terabytes to petabytes of data. However, the number of features in these datasets is very small, usually only a single feature for text-only data that is used for training LLMs.

Techniques to select which examples from the original dataset to include in the final dataset have shown increasingly impressive results. For example, as this book was going to press, Google DeepMind published a paper on multimodal contrastive learning with joint example selection (JEST), in which the authors introduce a batch-based algorithm for identifying high-quality training data. By using their technique, the authors were able to demonstrate substantial efficiency gains in multimodal learning. Among other advantages, these improvements significantly reduce the amount of power required to train a state-of-the-art GenAI model, simply as a result of improving data quality.

Example: Using TF Transform to Tokenize Text

Since text is such a common type of data and language models can be so powerful, let's look at an example of a form of feature engineering applied for all language models. Earlier we discussed how you can use TF Transform to preprocess your datasets ahead of model training. In this example, we are diving a bit deeper into a common preprocessing step: the tokenization of unstructured text.

Token-based language models such as BERT, T5, and LLaMa require conversion of the raw text to tokens, and more specifically to token IDs. Language models are trained with a vocabulary, usually limited to the top most frequently used word fragments and control tokens.

If you would like to train a BERT model to classify the sentiment of a text, you need to use a tokenizer to preprocess the input text to token IDs:

```
Text: "I like pistachio ice cream."
Tokens: ['i', 'like', 'pi', '##sta', '##chio', '
Token IDs: [1045, 2066, 14255, 9153, 23584, 3256,
```

Furthermore, the language models expect "control tokens" such as start, stop, or pad tokens. In this example, we demonstrate how you can

preprocess your text data to be ready for fine-tuning a BERT model. However, the steps extend (with slight modifications) to other language models such as T5 and LLaMa.

ML frameworks such as TensorFlow and PyTorch provide framework-specific libraries to support such conversions. In this example, we are using TensorFlow Text together with TF Transform. If you prefer PyTorch, check out TorchText.

Before converting text into tokens, it is recommended to normalize the text to the supported character encoding (e.g., UTF-8). At the same time, you can “clean” the text, for example, by removing common text patterns that occur in every sample.

Once the text data is normalized and cleaned, we’ll tokenize the text. Depending on what natural language library you use, you can either tokenize directly to token IDs or tokenize first to token strings and then convert the tokens to token IDs. In our case, TensorFlow Text allows the conversion directly to token IDs. The prominent BERT model uses WordPiece tokenization, while more recent models such as T5 and LLaMa rely on SentencePiece tokenization.

WHICH TOKENIZER SHOULD YOU USE?

The type of tokenization to use is driven by the foundational model, which in this example is BERT. Your tokenization needs to match the tokenizer that was used for the initial training of the language model. You also need to use the same underlying vocabulary from the initial training; otherwise, the token IDs from the fine-tuning won't match the token IDs generated during the initial training. This will cause catastrophic forgetting and impact your model's performance.

The types of tokenizers differ in tokenization speed, handling of whitespaces, and multilanguage support.

Language models also expect a set of control tokens to notate the start or end of the model input, as well as any number of pad tokens or unknown tokens. *Unknown tokens* are tokens that the tokenizer couldn't convert into token IDs. It therefore notates such tokens with a fixed ID.

Language models expect a `fixed` model input. That means texts with fewer tokens need to be padded. In this case, we simply fill up the text with the maximum number of tokens the language model expects as input. For BERT models, that is generally 512 tokens (unless otherwise defined).

Transformer-based language models also often expect an `input_mask` and sometimes even `input_type_ids`. The `input_mask` ultimately

speeds up the computations within the language model by focusing on the relevant parts of the data input. In the case of BERT, the model was trained with different objectives (e.g., whether the second sentence is a follow-up sentence to the first sentence). To support such objectives, the model needs to distinguish between the different sentences, and that is done through the `input_type_ids`.

Now let's put the following four steps into one example:

1. Text normalization
2. Text tokenization
3. Token truncation/padding
4. Creating input masks and type IDs

```
import tensorflow as tf
import tensorflow_hub as hub
import tensorflow_text as tf_text
…
START_TOKEN_ID = 101
END_TOKEN_ID = 102
TFHUB_URL = ("https://www.kaggle.com/models/tens
             "en-uncased-l-12-h-768-a-12/3")

def load_bert_model(model_url=TFHUB_URL):
    bert_layer = hub.KerasLayer(handle=model_url,
    return bert_layer
```

```
def _preprocessing_fn(inputs):
      vocab_file_path = load_bert_model().resolve

      bert_tokenizer = tf_text.BertTokenizer(
          vocab_lookup_table=vocab_file_path,
          token_out_type=tf.int64,
          lower_case=True)

      text = inputs['message']
      category = inputs['category']

      # Normalize text
      text = tf_text.normalize_utf8(text)

      # Tokenization
      tokens = bert_tokenizer.tokenize(text).merg

      # Add control tokens
      tokens, input_type_ids = tf_text.combine_se
          tokens,
          start_of_sequence_id=START_TOKEN_ID,
          end_of_segment_id=END_TOKEN_ID)

      # Token truncation / padding
      tokens, input_mask_ids =
      tf_text.pad_model_inputs(
        tokens, max_seq_length=128)
```

```
        # Convert categories to labels
        labels = tft.compute_and_apply_vocabulary(
            label, vocab_filename="category")

        return {
          "labels": labels,
          "input_ids": tokens,
          "input_mask_ids": input_mask_ids,
          "input_type_ids": input_type_ids,
        }
```

Using the presented preprocessing function allows you to prepare text data to fine-tune a BERT model. To fine-tune a different language model, update the tokenizer function and the expected output data structure from the preprocessing step.

Benefits of Using TF Transform

Earlier, we noted that the strength of TF Transform lies in its efficient preprocessing. However, unlike our previous examples, in this example each conversion is happening row by row, and the analysis pass performed by TF Transform may not be necessary. Nevertheless, there are still several reasons to use TF Transform in such a case:

- Converting categories to labels often necessitates an analysis pass, so token conversion is effectively an added bonus.

- It prevents training–serving skew, ensuring consistency between the training and serving data.
- It scales with the data due to its preprocessing graph computation capabilities, allowing parallelization of preprocessing through tools such as Apache Beam and Google Cloud Dataflow.
- By separating the feature preprocessing from the actual training, it helps keep complex models more understandable and maintainable.
- It is integrated with TFX via the Transform standard pipeline component.

However, there is an initial implementation investment required. If the TF Transform setup is too complex, we recommend checking out the alternatives listed in the following section.

Alternatives to TF Transform

TF Transform isn’t the only library you can use for working with text and language models. A number of other natural language libraries exist for the various ML frameworks, including the following:

KerasNLP

KerasNLP abstracts the tokenization and creation of the data structures. At the time of this writing, it supports TensorFlow models and is limited to a set of language models. However, it allows for fast bootstrapping of prototype models.

SpaCy

: This framework-agnostic NLP library offers a wide range of preprocessing functions. It is a great option if you need an ML framework–independent solution.

TorchText

: TorchText is the perfect NLP library choice if you are developing PyTorch models. It provides similar functionality as TensorFlow Text for PyTorch-based ML projects.

Conclusion

This chapter continued our discussion of data, focusing on techniques to improve the data we have in order to achieve a better result. As we write this in 2024, there has been a renewed focus in the ML community on the importance of data for ML, leading Andrew Ng to launch the "Data-centric AI movement". In generative AI, there has also been an emerging focus on developing highly curated, high-quality datasets for the fine-tuning of pretrained foundation models such as PaLM and LLaMa.

Why are people focusing on data? The reasons are fairly simple. The increasingly large datasets that have become available have tended to lead many people to focus on data quantity instead of data quality. Leaders in the field are now encouraging developers to focus more on data quality because

ultimately what is important is not the amount of data, but the information contained in the data. In human terms, you could read a thousand books on Antarctica and learn nothing about computer science, but reading one book on computer science could teach you much about computer science. It is the information contained in those books, or in your dataset, that is important for you, or your model, to learn.

The feature engineering we discussed in this chapter is intended to make that information more accessible to your model so that it learns more easily. The feature selection we discussed in this chapter is intended to concentrate the information in your data in the highest-quality form and enable you to make trade-offs for the efficient use of your computing resources.

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.

Chapter 5. Advanced Labeling, Augmentation, and Data Preprocessing

The topics in this chapter are especially important to shaping your data to get the most value from it for your model, especially in a supervised learning setting. Labeling in particular can easily be one of the most expensive and time-consuming activities in the creation, maintenance, and evolution of an ML application. A good understanding of the options available will help you make the most of your resources and budget.

To that end, in this chapter we will discuss data augmentation, a class of methods in which you add more data to your training dataset in order to improve training, usually to improve generalization in particular. Data augmentation is almost always based on manipulating your current data to create new, but still valid, variations of your examples.

We will also discuss data preprocessing, but in this chapter we'll focus on domain-specific preprocessing. Different domains, such as time series, text, and images, have specialized forms of feature engineering. We discussed one of these, tokenizing text, in "Consider Instance-Level Versus Full-Pass Transformations". In this chapter, we'll review common methods for working with time series data.

But first, let’s address an important question: How can we assign labels in ways other than going through each example manually? In other words, can we automate the process even at the expense of introducing inaccuracies in the labeling process? The answer is yes, and the way we do it is through advanced labeling.

Advanced Labeling

Why is advanced labeling important? Well, the use of ML is growing worldwide, and ML requires training data. If you’re doing supervised learning, that training data needs labels, and supervised learning represents the vast majority of ML in production today.

But manually labeling data is often expensive and difficult, and unlabeled data is typically pretty cheap and easy to get and contains a lot of information that can help improve our model. So, advanced labeling techniques help us reduce the cost of labeling data while leveraging the information in large amounts of unlabeled data.

In this section, we’ll start with a discussion of how semi-supervised labeling works and how you can use it to improve your model’s performance by expanding your labeled dataset in directions that provide the most predictive information. We’ll follow this with a discussion of active learning, which uses intelligent sampling to assign to unlabeled data labels based on the existing data. Then, we’ll introduce weak supervision,

which is an advanced technique for programmatically labeling data, typically by using heuristics that are designed by subject matter experts.

Semi-Supervised Labeling

With semi-supervised labeling, you start with a relatively small dataset that's been labeled by humans. You then combine that labeled data with a large amount of unlabeled data, inferring the labels by looking at how the different human-labeled classes are clustered within the feature space. Then, you train your model using the combination of the two datasets. This method is based on the assumption that different label classes will cluster together within the feature space, which is typically—but not always—a good assumption.

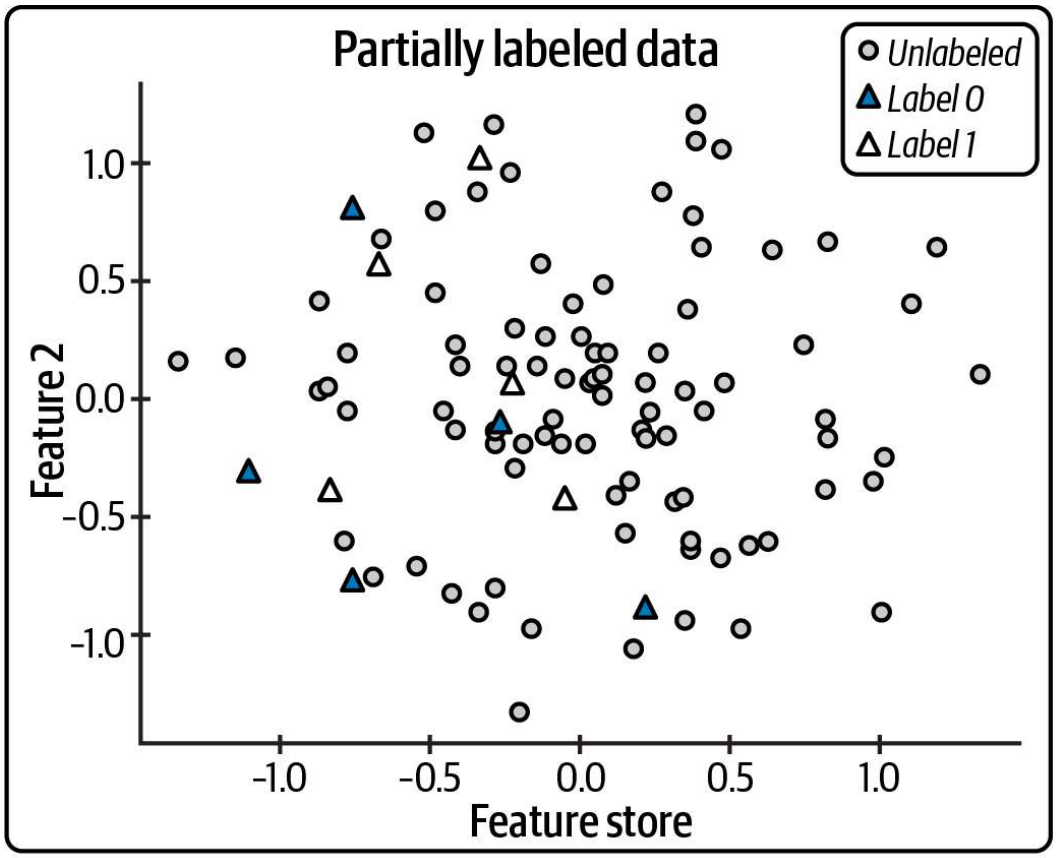
Using semi-supervised labeling is advantageous for two main reasons. First, combining labeled and unlabeled data can increase feature space coverage, which, as described in "Feature Selection", can improve the accuracy of ML models. Second, getting unlabeled data is often very inexpensive because it doesn't require people to assign labels. Often, unlabeled data is easily available in large quantities.

By the way, don't confuse semi-supervised labeling with semi-supervised training, which is very different. We'll discuss semi-supervised training in a later chapter.

Label propagation

Label propagation is an algorithm for assigning labels to previously unlabeled examples. This makes it a semi-supervised algorithm, where a subset of data points have labels. The algorithm propagates the labels to data points without labels based on the similarity or community structure of the labeled data points and the unlabeled data points. This similarity or structure is used to assign labels to the unlabeled data.

In Figure 5-1, you can see some labeled data (the triangles) and a lot of unlabeled data (the circles). With label propagation, you assign labels to the unlabeled data based on how they cluster with their neighbors.


Partially labeled data
Unlabeled
Label 0
Label 1
Feature 2
1.0
0.5
0.0
-0.5
-1.0
-1.0
-0.5
0.0
0.5
1.0
Feature store

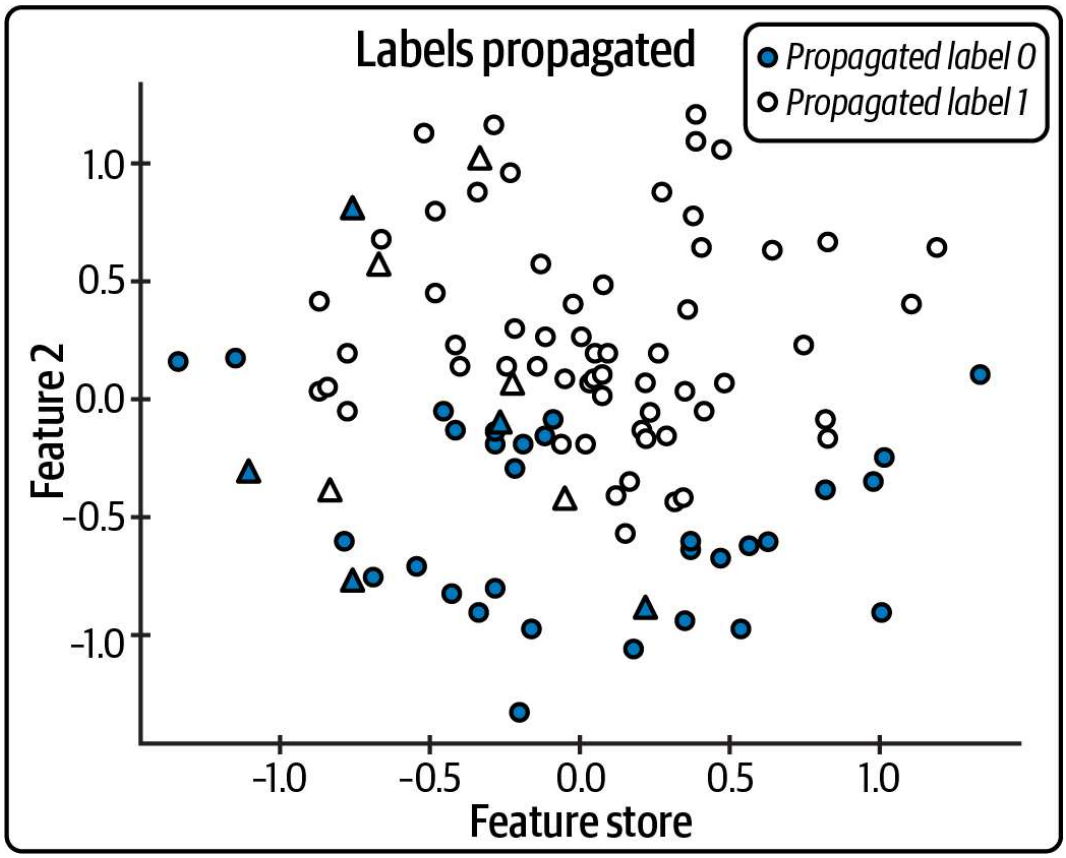

Labels propagated
Propagated label 0
Propagated label 1
Feature 2
1.0
0.5
0.0
-0.5
-1.0
-1.0
-0.5
0.0
0.5
1.0
Feature store

Figure 5-1. Label propagation

The labels are then propagated to the rest of the clusters, as indicated with different shades. We should mention that there are many different ways to do label propagation—graph-based label propagation is only one of several techniques. Label propagation itself is considered *transductive learning*, meaning we are mapping from the examples themselves, without learning a function for the mapping.

Sampling techniques

Typically, your labeled dataset will be much smaller than the available unlabeled dataset. If you're going to add to your labeled dataset by labeling new data, you need some way to decide which unlabeled examples to label. You could just select them randomly, which is referred to as *random sampling*. Or you could try to somehow select the best examples, which are those that improve your model the most. There are a variety of techniques for trying to select the best examples, and we'll introduce a few of these next.

Active Learning

Active learning is a way to intelligently sample your data, selecting the unlabeled points that would bring the most predictive value to your model. This is very helpful in a variety of contexts, including when you have a

limited data budget. It costs money to label data, especially when you're using human experts to look at the data and assign a label to it. Active learning helps you make sure you focus your resources on the data that will give you the most bang for your buck.

If you have an imbalanced dataset, active learning is an efficient way to select rare classes at the training stage. And if standard sampling strategies do not help improve accuracy and other target metrics, active learning can often offer a way to achieve the desired accuracy.

An active learning strategy relies on being able to select the examples to label that will best help the model learn. In a fully supervised setting, the training dataset consists of only those examples that have been labeled. In a semi-supervised setting, you leverage your labeled examples to label some additional, previously unlabeled examples in order to increase the size of your labeled dataset. Active learning is a way to select which unlabeled examples to label.

A typical active learning cycle proceeds as follows:

1. You start with a labeled dataset, which you use to train a model, and a pool of unlabeled data.
2. Active learning selects a few unlabeled examples, using intelligent sampling (as described in more detail in the sections that follow).

3. You label the examples that were selected with human annotators, or by leveraging other techniques. This gives you a larger labeled dataset.
4. You use this larger labeled dataset to retrain the model, potentially starting a new iteration of the active learning cycle.

But this begs the question: How do we do intelligent sampling?

Margin sampling

Margin sampling is one widely used technique for doing intelligent sampling. Margin sampling is a valuable technique for active learning that focuses on querying the most uncertain samples, those closest to the decision boundary, to improve the model's learning efficiency and performance.

In Figure 5-2, the data belongs to two classes. Additionally, there are unlabeled data points. In this setting, the simplest strategy is to train a binary linear classification model on the labeled data, which outputs a decision boundary.

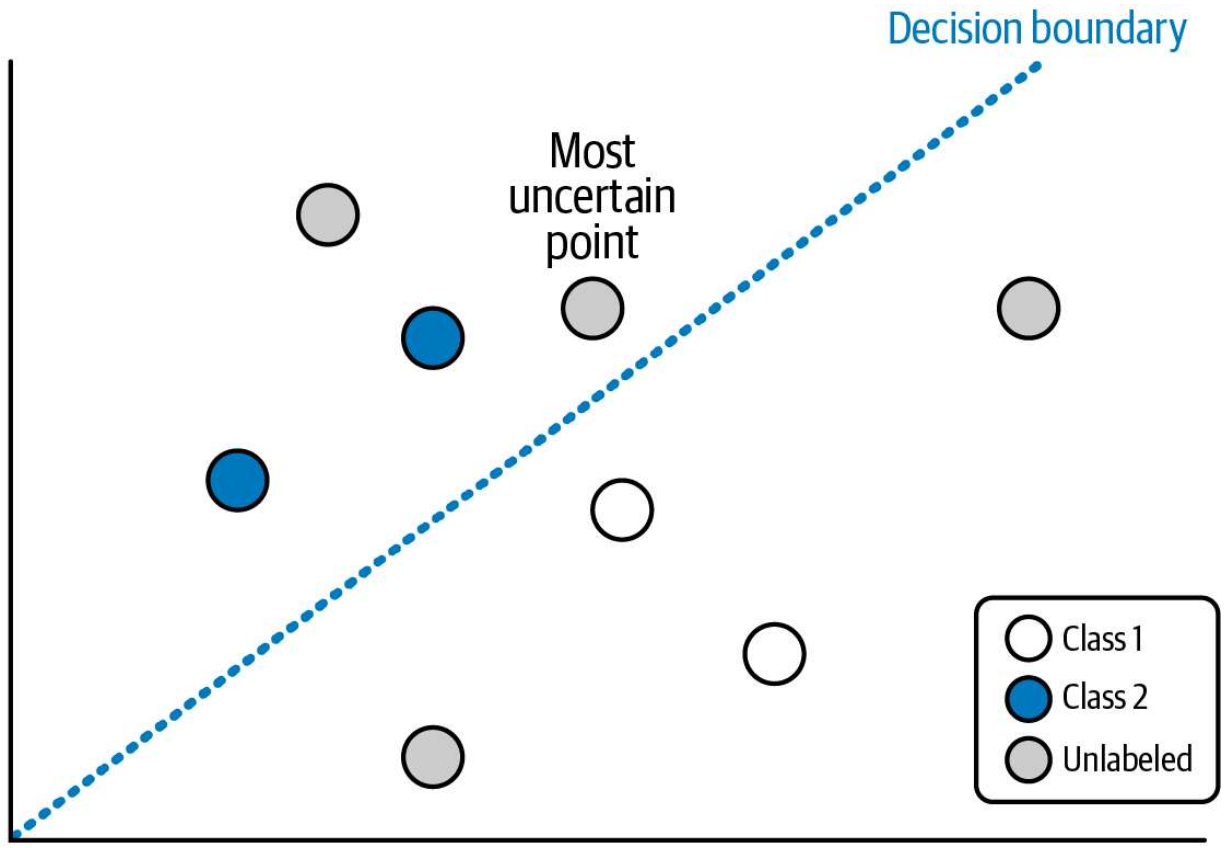


Figure 5-2. Margin sampling, initial state

With active learning, you select the most uncertain point to be labeled next and added to the dataset. Margin sampling defines the most uncertain point as the one that is closest to the decision boundary.

As shown in Figure 5-3, using this new labeled data point, you retrain the model to learn a new classification boundary. By moving the boundary, the model learns a bit better to separate the classes. Next, you find the next most uncertain data point, and you repeat the process until the model doesn’t improve.

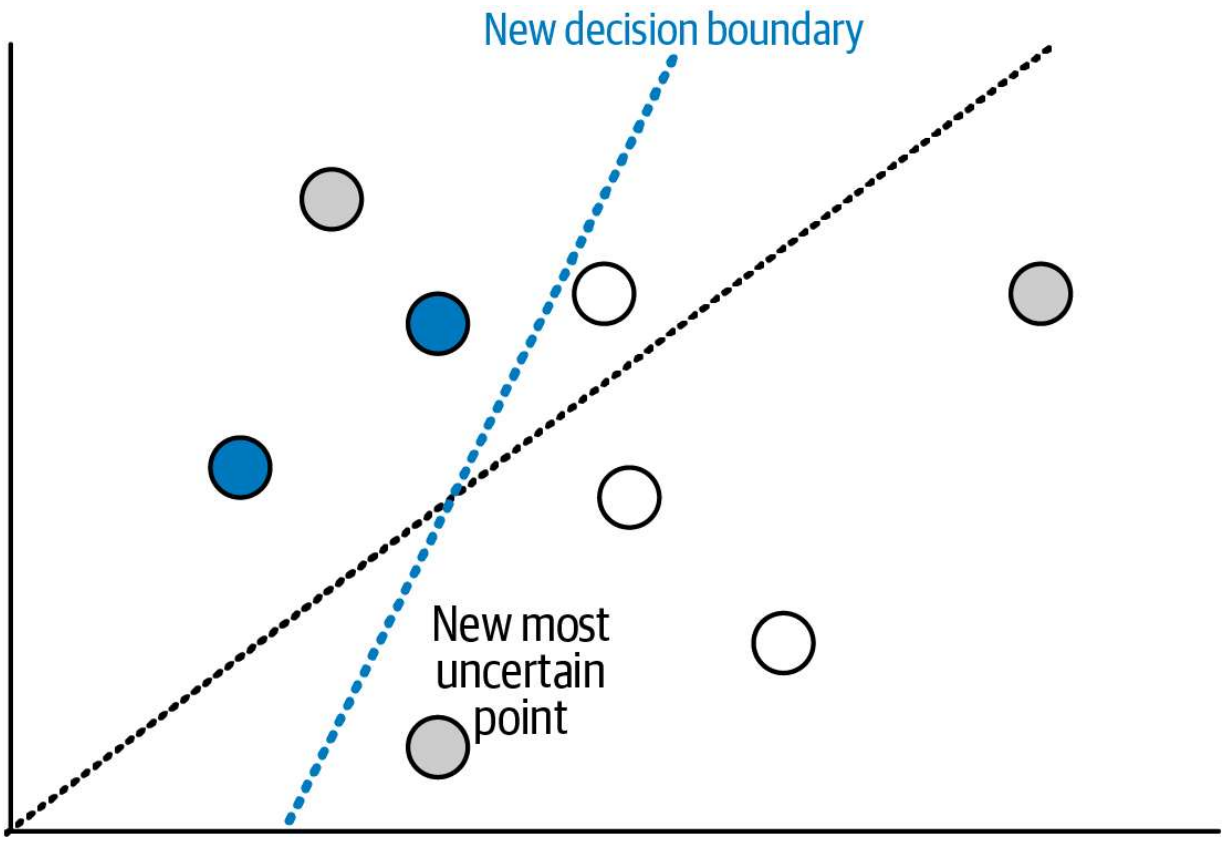


Figure 5-3. Margin sampling, after first iteration

Figure 5-4 shows model accuracy as a function of the number of training examples for different sampling techniques. The bottom line shows the results of random sampling. The top two lines show the performance of two margin sampling algorithms using active learning (the difference between the two is not important right now).

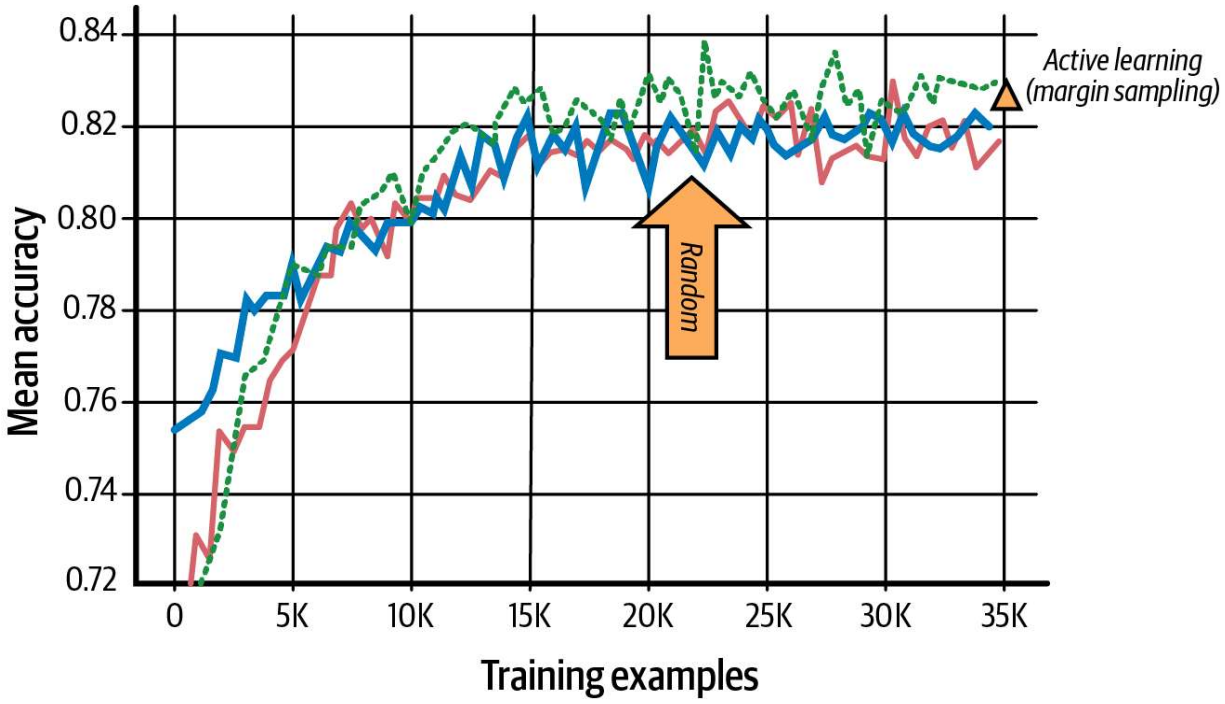


Figure 5-4. Intelligent sampling results

Looking at the x-axis you can see that margin sampling achieves higher accuracy with fewer training examples than random sampling. Eventually, as a higher percentage of the unlabeled data is labeled with random sampling, it catches up to margin sampling. This agrees with what we would expect if margin sampling intelligently selects the best examples to label.

Other sampling techniques

Margin sampling is only one intelligent sampling technique. With margin sampling, as you saw, you assign labels to the most uncertain points based on their distance from the decision boundary. Another technique is *cluster-based sampling*, in which you select a diverse set of points by using

clustering methods over your feature space. Yet another technique is *query by committee*, in which you train several models and select the data points with the highest disagreement among them. And finally, *region-based sampling* is a relatively new algorithm. At a high level, this algorithm works by dividing the input space into separate regions and running an active learning algorithm on each region.

Weak Supervision

> *Hand-labeling training data for machine learning problems is effective, but very labor and time intensive. This work explores how to use algorithmic labeling systems relying on other sources of knowledge that can provide many more labels but which are noisy.*
>
> —Jeff Dean, SVP, Google Research and AI, March 14, 2019

Weak supervision is a way to generate labels by using information from one or more sources, usually subject matter experts and/or heuristics. The resulting labels are noisy and probabilistic, rather than the deterministic labels that we're used to. They provide a signal of what the actual label should be, but they aren't expected to be 100% correct. Instead, there is some probability that they're correct.

More rigorously, weak supervision comprises one or more noisy conditional distributions over unlabeled data, and the main objective is to learn a generative model that determines the relevance of each of these noisy sources.

Starting with unlabeled data for which you don’t know the true labels, you add to the mix one or more weak supervision sources. These sources are a list of heuristic procedures that implement noisy and imperfect automated labeling. Subject matter experts are the most common sources for designing these heuristics, which typically consist of a coverage set and an expected probability of the true label over the coverage set. By “noisy” we mean that the label has a certain probability of being correct, rather than the 100% certainty that we’re used to for the labels in our typical supervised labeled data. The main goal is to learn the trustworthiness of each weak supervision source. This is done by training a generative model.

The Snorkel framework came out of Stanford in 2016 and is the most widely used framework for implementing weak supervision. It does not require manual labeling, so the system programmatically builds and manages training datasets. Snorkel provides tools to clean, model, and integrate the resulting training data that is generated by the weak supervision pipeline. Snorkel uses novel, theoretically grounded techniques to get the job done quickly and efficiently. Snorkel also offers data augmentation and slicing, but our focus here is on weak supervision.

With Snorkel, you start with unlabeled data and apply labeling functions (the heuristics that are designed by subject matter experts) to generate noisy labels. You then use a generative model to denoise the noisy labels and assign importance weights to different labeling functions. Finally, you train a discriminative model—*your* model—with the denoised labels.

Let's take a look at what a couple of simple labeling functions might look like in code. Here is an easy way to create functions to label spam using Snorkel:

```
from snorkel.labeling import labeling_function
 @labeling_function()
def lf_contains_my(x):
    # Many spam comments talk about 'my channel',
    return SPAM if "my" in x.text.lower() else AB
 @labeling_function()
def lf_short_comment(x):
    # Non-spam comments are often short, such as
    return NOT_SPAM if len(x.text.split()) < 5 el
```

The first step is to import the `labeling_function` from Snorkel. With the first function (`lf_contains_my`), we label a message as spam if it contains the word *my*. Otherwise, the function returns `ABSTAIN` , which means it has no opinion on what the label should be. The second

function (`lf_short_comment`) labels a message as not spam if it is shorter than five words.

Advanced Labeling Review

Supervised learning requires labeled data, but labeling data is often an expensive, difficult, and slow process. Let’s review the key points of advanced labeling techniques that offer benefits over supervised learning:

Semi-supervised learning

Falls between unsupervised learning and supervised learning. It works by combining a small amount of labeled data with a large amount of unlabeled data. This improves learning accuracy.

Active learning

Relies on intelligent sampling techniques that select the most important examples to label and add to the dataset. Active learning improves predictive accuracy while minimizing labeling cost.

Weak supervision

Leverages noisy, limited, or inaccurate label sources inside a supervised learning environment that tests labeling accuracy. Snorkel is a compact and user-friendly system to manage all these operations and to establish training datasets using weak supervision.

Data Augmentation

In the previous section, we explored methods for getting more labeled data by labeling unlabeled data, but another way to do this is to augment your existing data to create more labeled examples. With data augmentation, you can expand a dataset by adding slightly modified copies of existing data, or by creating new synthetic data from your existing data.

With the existing data, it is possible to create more data by making minor alterations/perturbations in the existing examples. Simple variations such as flips or rotations in images are an easy way to double or triple the number of images in a dataset, while retaining the same label for all the variants.

Data augmentation is a way to improve your model's performance, and often its ability to generalize. This adds new, valid examples that fall into regions of the feature space that aren't covered by your real examples.

Keep in mind that if you add invalid examples, you run the risk of learning the wrong answer, or at least introducing unwanted noise, so be careful to only augment your data in valid ways! For example, consider the images in Figure 5-5.

Figure 5-5. An invalid variant

Let’s begin with a concrete example of data augmentation using CIFAR-10, a famous and widely used dataset. We’ll then continue with a discussion of some other augmentation techniques.

Example: CIFAR-10

The CIFAR-10 dataset (from the Canadian Institute for Advanced Research) is a collection of images commonly used to train ML models and computer vision algorithms. It is one of the most widely used datasets for ML research.

CIFAR-10 contains 60,000 color images measuring 32 × 32 pixels. There are 10 different classes with 6,000 images in each class. Let’s take a practical look at data augmentation with the CIFAR-10 dataset:

```
def augment(x, height, width, num_channels):
    x = tf.image.resuize_with_crop_or_pad(x, heig
    x = tf.image.random_crop(x, [height, width, n
    x = tf.image.random_flip_left_right(x)
    return x
```

This code creates new examples that are perfectly valid. It starts by cropping the padded image to a given height and width, adding a padding of 8 pixels. It then creates random translated images by cropping again, and then randomly flips the images horizontally.

Other Augmentation Techniques

Apart from simple image manipulation, there are other advanced techniques for data augmentation that you may want to consider. Although we won't be discussing them here, these are some techniques for you to research on your own:

- Semi-supervised data augmentation
- Unsupervised Data Augmentation (UDA)
- Policy-based data augmentation (e.g., with AutoAugment)

While generating valid variations of images is easy to imagine and fairly easy to implement, for other kinds of data the augmentation techniques and the types of variants generated may not be as straightforward. The

applicability of different augmentation techniques tends to be specific to the type of data, and sometimes to the domain you're working in. This is another one of those areas where the ML engineering team's skill and knowledge of the data and domain are critical.

Data Augmentation Review

Data augmentation is a great way to increase the number of labeled examples in your dataset. Data augmentation increases the size of your dataset, and the sample diversity, which results in better feature space coverage. Data augmentation can reduce overfitting and increase the ability of your model to generalize.

Preprocessing Time Series Data: An Example

Data comes in a lot of different shapes, sizes, and formats, and each is analyzed, processed, and modeled differently. Some common types of data include images, video, text, audio, time series, and sensor data. Preprocessing for each of these tends to be very specialized and can easily fill a book, so instead of discussing all of them, we're going to look at only one: time series data.

A *time series* is a sequence of data points in time, often from events, where the time dimension indicates when the event occurred. The data points may or may not be ordered in the raw data, but you will almost always want to order them by time for modeling. Inherently, time series problems are almost always about predicting the future.

> *It is difficult to make predictions, especially about the future.*
>
> —Danish proverb

Time series forecasting does exactly that: it tries to predict the future. It does this by analyzing data from the past. Time series is often an important type of data and modeling for many business applications, such as financial forecasting, demand forecasting, and other types of forecasting that are important for business planning and optimization.

For example, to predict the future temperature at a given location we could use other meteorological variables, such as atmospheric pressure, wind direction, and wind velocity, that have been recorded previously. In fact, we would probably be using a weather time series dataset similar to the one that was recorded by the Max Planck Institute for Biogeochemistry. That dataset contains 14 different features including air temperature, atmospheric pressure, and humidity. The features were recorded every 10 minutes beginning in 2003.

Let’s take a closer look at how that data is organized and collected. There are 14 variables including measurements related to humidity, wind velocity and direction, temperature, and atmospheric pressure. The target for prediction is the temperature. The sampling rate is 1 observation every 10 minutes, so there are 6 observations per hour and 144 in a given day (6 × 24). The time dimension gives us the order, and order is important for this dataset since there is a lot of information in how each weather feature changes between observations. For time series, order is almost always important.

Figure 5-6 shows a plot of a temperature feature over time. You can see that there’s a pattern to this that repeats over specific intervals of time. This kind of repeating pattern is referred to as *seasonality*, but it can be any kind of repeating pattern and does not need to have anything to do with the seasons of the year. There’s clear seasonality here, which we need to consider when doing feature engineering for this data.

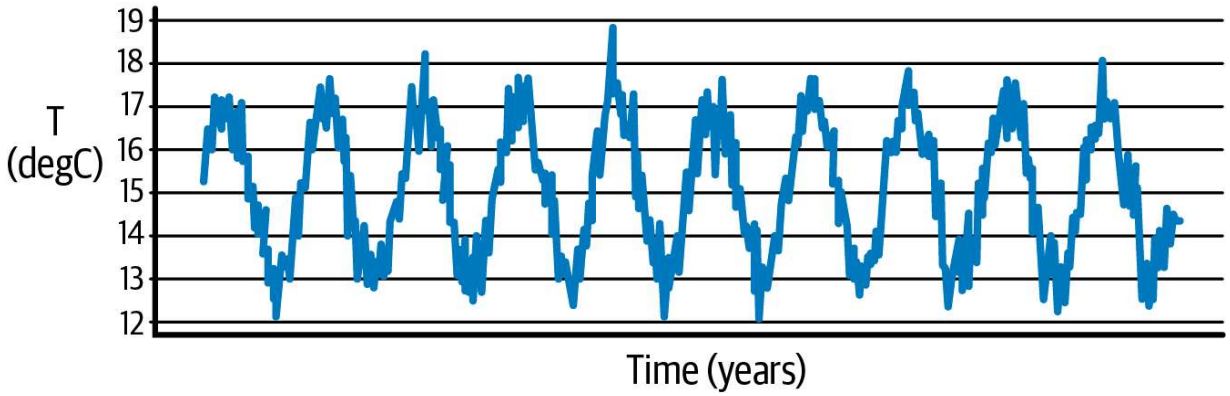


Figure 5-6. Weather periodicity showing seasonality

We should consider doing seasonal decomposition, but to keep things simple in this example we won't be doing that. Instead, we'll be focusing on windowing and sampling, which can be used with or without seasonal decomposition. Seasonal decomposition is used to improve the data and focus on the residual, and is often used in anomaly detection.

Windowing

Using a windowing strategy to look at dependencies with past data seems to be a natural path to take. Windowing strategies in time series data become pretty important, and they're kind of unique to time series and similar types of sequence data. The example in Figure 5-7 shows one windowing strategy that you might use for a model you want to use to make a prediction one hour into the future, given a history of six hours.

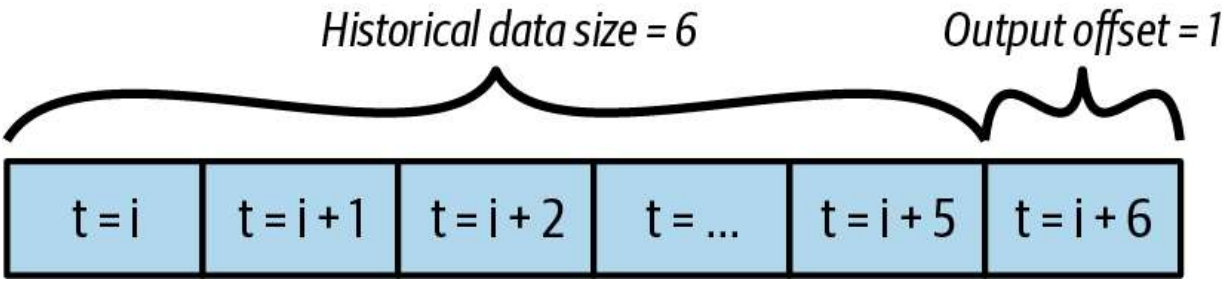


Figure 5-7. An example of a windowing strategy for making a prediction one hour into the future, given a history of six hours

Figure 5-8 shows a windowing strategy that you might use if you want to make a prediction 24 hours into the future, given 24 hours of history, so in that case, your history size is 24.

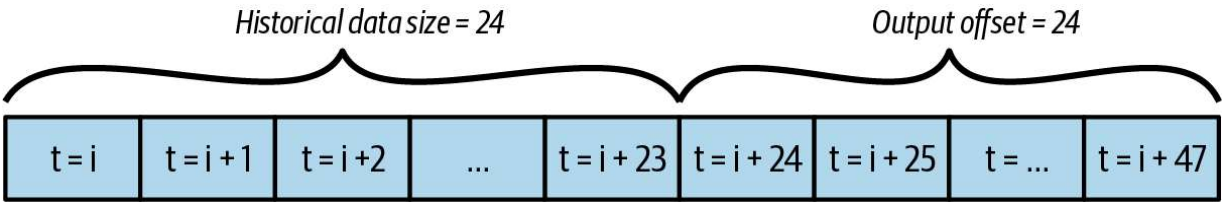


Figure 5-8. An example of a windowing strategy for making a prediction 24 hours into the future, given a history of 24 hours

In Figure 5-8, the offset size is also 24, so you could use a total window size of 48, which would be the history plus the output offset. It's also important to consider when "now" is, and to make sure you omit data pertaining to the future (i.e., time travel). In this example, if "now" is at t = 24, we need to be careful not to include the data from t = 25 to t = 47 in our training data. We could do that in feature engineering, or by reducing the window to include only the history and the label. If during training we were to include data about the future in our features, we would not have that data available when we use the model for inference, since the future hasn't happened yet.

Sampling

It's also important to design a sampling strategy. You already know that there are six observations per hour in our example, one observation every 10 minutes. In one day, there will be 144 observations. If you take five days of past observations and make a prediction six hours into the future, that means our history size will be 5 × 144 or 720 observations, the output offset will be 6 × 6 or 36, and the total time window size will be 792. Figure 5-9 shows visually what we mean by the total window size, history, and offset.

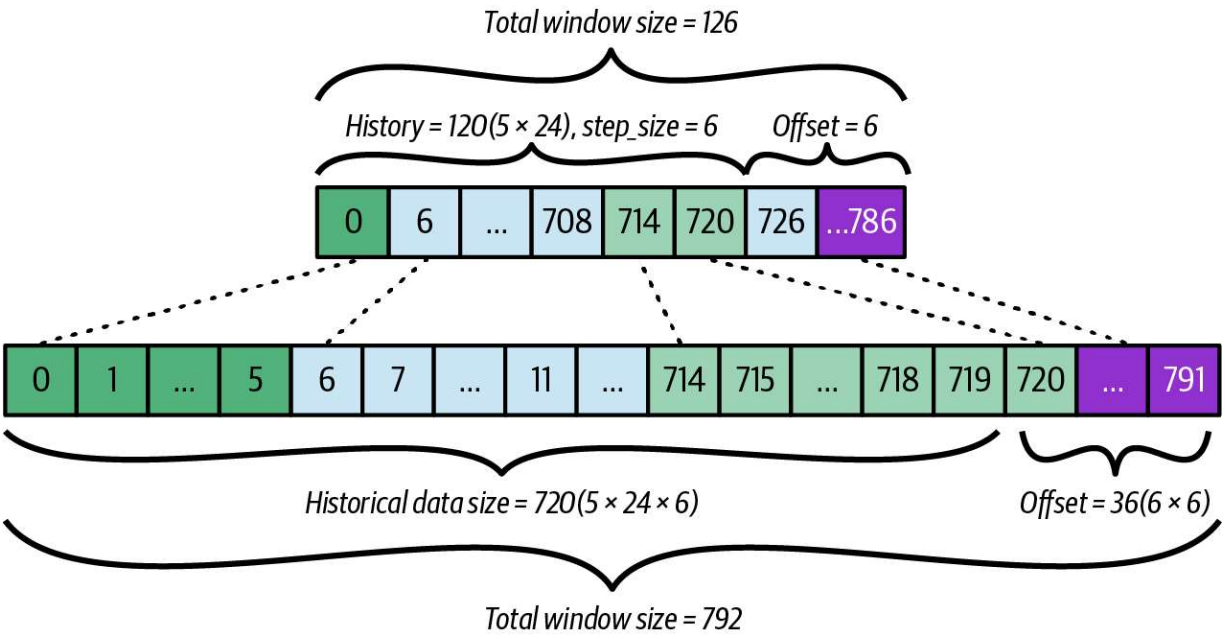


Figure 5-9. Improving a sampling strategy

Since observations in one hour are unlikely to change much, let's sample one observation per hour. We could take the first observation in the hour as a sample, or even better, we could take the median of the observations for each hour.

Then our history size becomes 5 × 24 × 1 or 120, and our output offset will be 6, so our total window size becomes 126. In this way, we've reduced the size of our feature vector from 792 to 126 by either sampling within each hour, or aggregating the data for each hour by taking the median.

This example is intended to be a short introduction to time series data. For a more in-depth look at time series data, you can refer to the TensorFlow documentation.

Conclusion

This chapter provided some background and perspective on the importance of labeling and preprocessing to successful modeling, along with the advantages of data augmentation as a method to expand on the information in the dataset. With new modeling techniques being developed at an amazing pace for generative AI and new forms of deep learning, new techniques for labeling, preprocessing, and data augmentation are also being developed. While this chapter did not cover all the techniques that exist today, it should give you a good understanding of the kinds of approaches to take and ways to think about these important areas. Remember, your model is only as good as the information in your data, and anything you can do to make it easier for your model to learn from that information will result in a better model.