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.

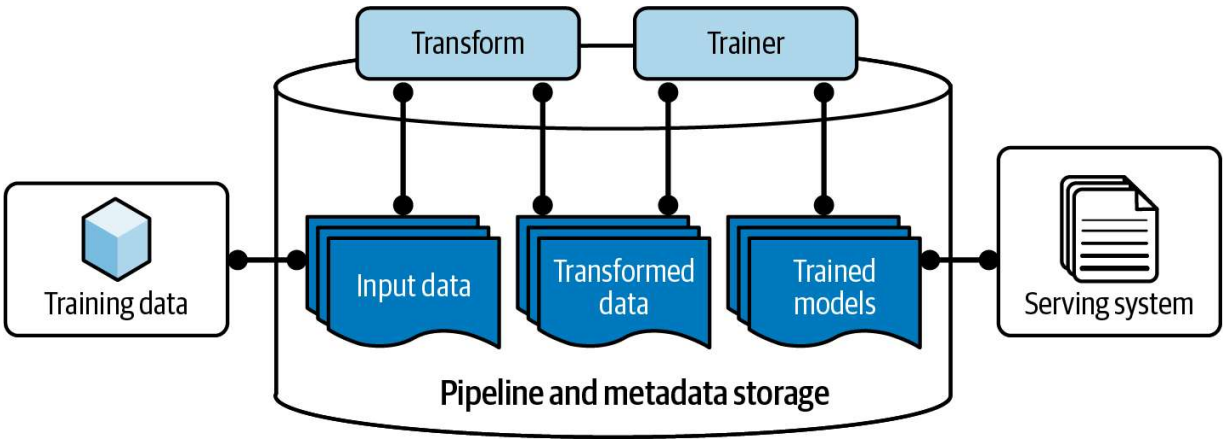

Figure 3-1. A simplified TFX pipeline that includes a Transform component

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.