Chapter 5. Data Preprocessing

The data we use to train our machine learning models is often provided in formats our machine learning models can't consume. For example, in our example project, a feature we want to use to train our model is available only as *Yes* and *No* tags. Any machine learning model requires a numerical representation of these values (e.g., *1* and *0*). In this chapter, we will explain how to convert features into consistent numerical representations so that your machine learning model can be trained with the numerical representations of the features.

One major aspect that we discuss in this chapter is focusing on consistent preprocessing. As shown in Figure 5-1, the preprocessing takes place after data validation, which we discussed in Chapter 4. *TensorFlow Transform* (TFT), the TFX component for data preprocessing, allows us to build our preprocessing steps as TensorFlow graphs. In the following sections, we will discuss why and when this is a good workflow and how to export the preprocessing steps. In Chapter 6, we will use the preprocessed datasets and the preserved transformation graph to train and export our machine learning model, respectively.

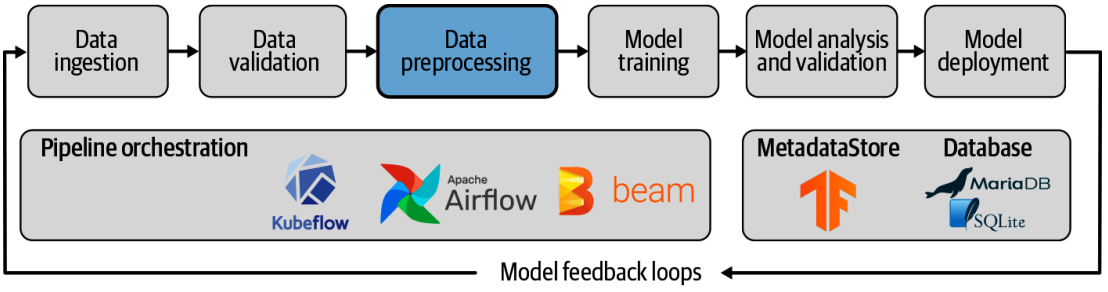

Figure 5-1. Data preprocessing as part of ML pipelines

Data scientists might see the preprocessing steps expressed as TensorFlow operations (operations) as too much overhead. After all, it requires different implementations than you might be used to when you write your preprocessing step with Python's `pandas` or `numpy`. We aren't advocating the use of TFT during the experimentation phase. However, as we demonstrate in the following sections, converting your preprocessing steps to TensorFlow operations when you bring your machine learning

model to a production environment will help avoid training-serving skews as we discussed in Chapter 4.

Why Data Preprocessing?

In our experience, TFT requires the steepest learning curve of any TFX library because expressing preprocessing steps through TensorFlow operations is required. However, there are a number of good reasons why data preprocessing should be standardized in a machine learning pipeline with TFT, including:

- Preprocessing your data efficiently in the context of the entire dataset
- Scaling the preprocessing steps effectively
- Avoiding a potential training-serving skew

Preprocessing the Data in the Context of the Entire Dataset

When we want to convert our data into numerical representations, we often have to do it in the context of the entire dataset. For example, if we want to normalize a numerical feature, we have to first determine the minimum and maximum values of the feature in the training set. With the determined boundaries, we can then normalize our data to values between 0 and 1. This normalization step usually requires two passes over the data: one pass to determine the boundaries and one to convert each feature value. TFT provides functions to manage the passes over the data behind the scenes for us.

Scaling the Preprocessing Steps

TFT uses Apache Beam under the hood to execute preprocessing instructions. This allows us to distribute the preprocessing if needed on the Apache Beam backend of our choice. If you don't have access to Google Cloud's Dataflow product or an Apache Spark or Apache Flink cluster, Apache Beam will default back to its Direct Runner mode.

Avoiding a Training-Serving Skew

TFT creates and saves a TensorFlow graph of the preprocessing steps. First, it will create a graph to process the data (e.g., determine minimum/maximum values). Afterwards, it will preserve the graph with the determined boundaries. This graph can then be used during the inference phase of the model life cycle. This process guarantees that the model in the inference life cycle step sees the same preprocessing steps as the model used during the training.

WHAT IS A TRAINING-SERVING SKEW?

We speak of a training-serving skew when the preprocessing steps used during model training get out of line with the steps used during inference. In many cases, the data used to train models is processed in Python notebooks with `pandas` or in Spark jobs. When the model is deployed to a production setup, the preprocessing steps are implemented in an API before the data hits the model for the prediction. As you can see in Figure 5-2, these two processes require coordination to make sure the steps are always aligned.

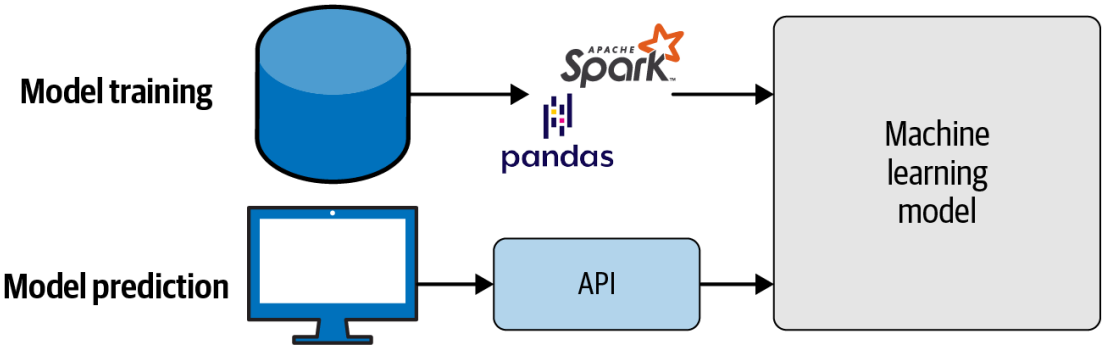

Figure 5-2. A commonly used machine learning setup

With TFT, we can avoid a misalignment of the preprocessing steps. As shown in Figure 5-3, the prediction-requesting client can now submit the raw data, and the preprocessing happens on the deployed model graph.

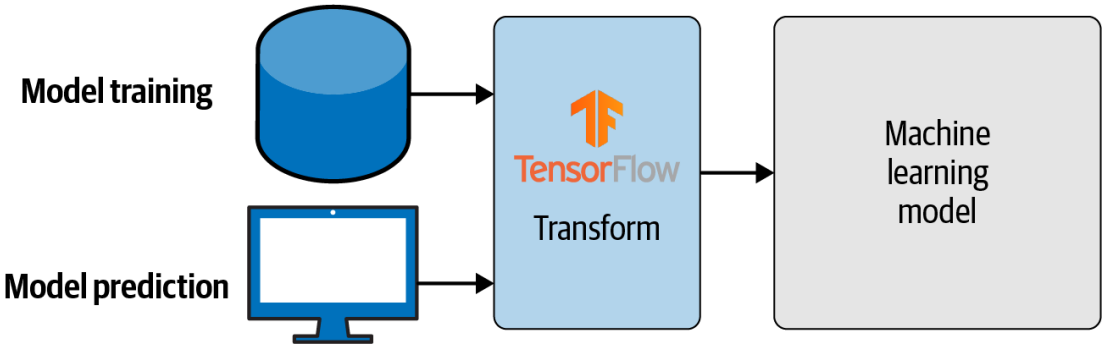

Figure 5-3. Avoiding the training-serving skew with TFT

Such a setup reduces the amount of coordination needed and simplifies deployments.

Deploying Preprocessing Steps and the ML Model as One Artifact

To avoid a misalignment between the preprocessing steps and the trained model, the exported model of our pipeline should include the preprocessing graph and the trained model. We can then deploy the model like any other TensorFlow model, but during our inference, the data will be preprocessed on the model server as part of the model inference. This avoids the requirement that preprocessing happen on the client side and simplifies the development of clients (e.g., web or mobile apps) that request the model predictions. In Chapters 11 and 12, we will discuss how the entire end-to-end pipeline produces such "combined" saved models.

Checking Your Preprocessing Results in Your Pipeline

Implementing the data preprocessing with TFT and the integrating the preprocessing in our pipeline gives us an additional benefit. We can generate statistics from the preprocessed data and check whether they still conform with our requirements to train a machine learning model. An example for this use case is the conversion of text to tokens. If a text contains a lot of new vocabulary, the unknown tokens will be converted to so-called *UNK* or *unknown* tokens. If a certain amount of our tokens are simply unknown, it is often difficult for the machine learning model to effectively generalize from the data, and therefore, the model accuracy will be affected. In our pipelines, we can now check the preprocessing step results by generating statistics (shown in Chapter 4) after the preprocessing step.

THE DIFFERENCE BETWEEN TF.DATA AND TF.TRANSFORM

There is often confusion between *tf.data* and *tf.transform*. *tf.data* is a TensorFlow API for building efficient input pipelines for model training with TensorFlow. The goal of the library is to utilize hardware resources optimally, such as host CPU and RAM, for the data ingestion and preprocessing that happens during training. *tf.transform*, on the other hand, is used to express preprocessing that should happen both in training and inference time. The library makes it possible to perform a full-pass analysis of the input data (e.g., to compute vocabulary or statistics used for data normalization), and this analysis is executed ahead of the training.

Data Preprocessing with TFT

The library for preprocessing data within the TensorFlow ecosystem is TFT. Like TFDV, it is part of the TFX project.

TFT processes the data that we ingested into our pipeline with the earlier generated dataset schema, and it outputs two artifacts:

- Preprocessed training and evaluation datasets in the TFRecord format. The produced datasets can be consumed downstream in the `Trainer` component of our pipeline.
- Exported preprocessing graph (with assets), which will be used when we'll export our machine learning model.

The key to TFT is the `preprocessing_fn` function, as shown in Figure 5-4. The function defines all transformations we want to apply to the *raw* data. When we execute the `Transform` component, the `preprocessing_fn` function will receive the raw data, apply the transformation, and return the processed data. The data is provided as TensorFlow Tensors or SparseTensors (depending on the feature). All transformations applied to the tensors have to be TensorFlow operations. This allows TFT to effectively distribute the preprocessing steps.

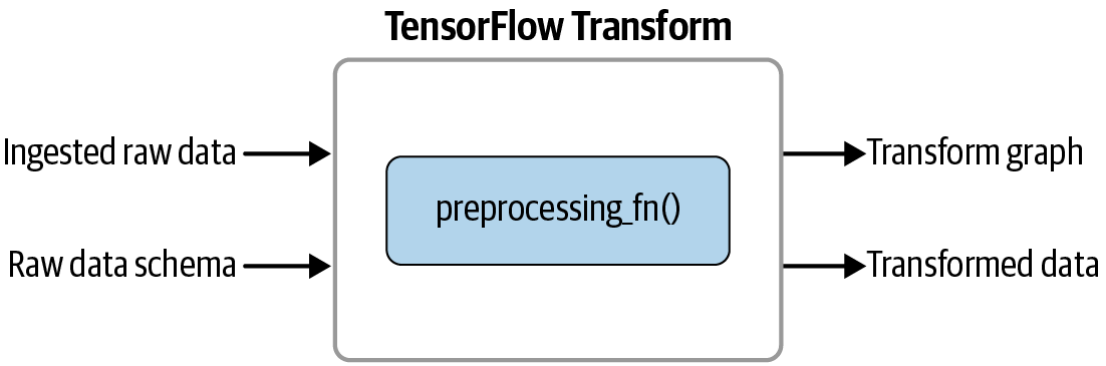

Figure 5-4. Overview of TFT

TFT FUNCTIONS

TFT functions that perform elaborate processing steps behind the scenes like `tft.compute_and_apply_vocabulary` can be spotted by the prefix *tft*. It is common practice to map TFT to the abbreviation *tft* in the Python namespace. Normal TensorFlow operations will be loaded with the common prefix *tf*, as in `tf.reshape`.

TensorFlow Transform also provides useful functions (e.g., `tft.bucketize`, `tft.compute_and_apply_vocabulary`, or `tft.scale_to_z_score`). When these functions are applied to a dataset feature, they will perform the required pass over the data and then apply the obtained boundaries to the data. For example, `tft.compute_and_apply_vocabulary` will generate the vocabulary set of a corpus, apply the created token-to-index mapping to the feature, and return the index value. The function can limit the number of vocab tokens to the top *n* of the most relevant tokens. In the following sections, we will be highlighting some of the most useful TFT operations.

Installation

When we installed the `tfx` package as introduced in Chapter 2, TFT was installed as a dependency. If we would like to use TFT as a standalone package, we can install the PyPI package with:

```
$ pip install tensorflow-transform
```

After installing `tfx` or `tensorflow-transform`, we can integrate our preprocessing steps into our machine learning pipelines. Let's walk through

a couple use cases.

Preprocessing Strategies

As we discussed previously, the applied transformations are defined in a function called `preprocessing_fn()`. The function will then be consumed by our `Transform` pipeline component or by our standalone setup of TFT. Here is an example of a preprocessing function that we will discuss in detail in the following sections:

```
def preprocessing_fn(inputs):
    x = inputs['x']
    x_normalized = tft.scale_to_0_1(x)
    return {
        'x_xf': x_normalized
    }
```

The function receives a batch of inputs as a Python dictionary. The key is the name of the feature and the values representing the raw data before the preprocessing are applied. First, TFT will perform an analysis step, as shown in Figure 5-5. In the case of our little demo example, it will determine the minimum and maximum values of our feature through a full pass over the data. This step can happen in a distributed fashion thanks to the execution of the preprocessing steps on Apache Beam.

In the second pass over the data, the determined values (in our case, the min and max of the feature column) are being used to scale our feature *x* between 0 and 1, as shown in Figure 5-6.

TFT also generates a graph for the prediction with the preserved minimum and maximum values. This will guarantee a consistent execution.

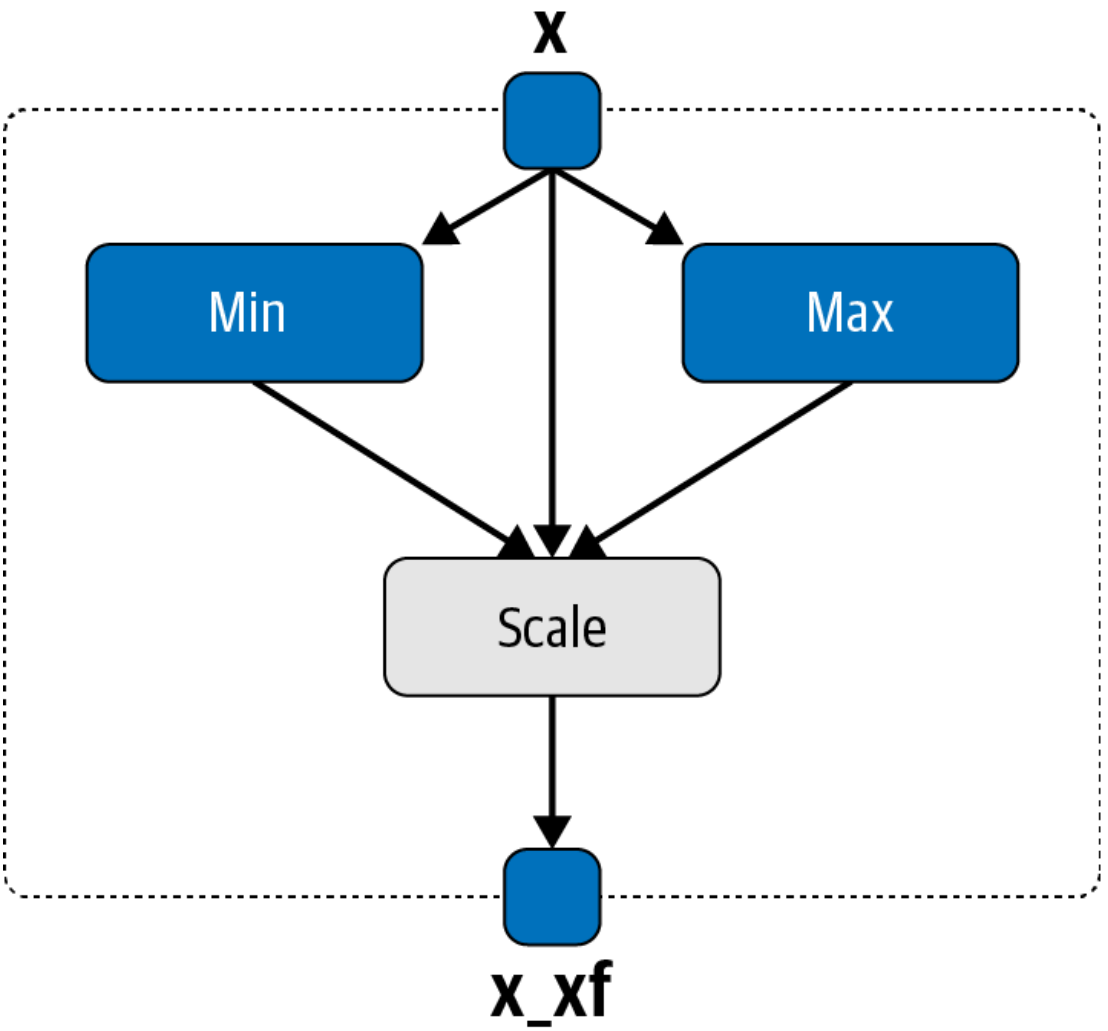

Figure 5-5. Analysis step during the TFT execution

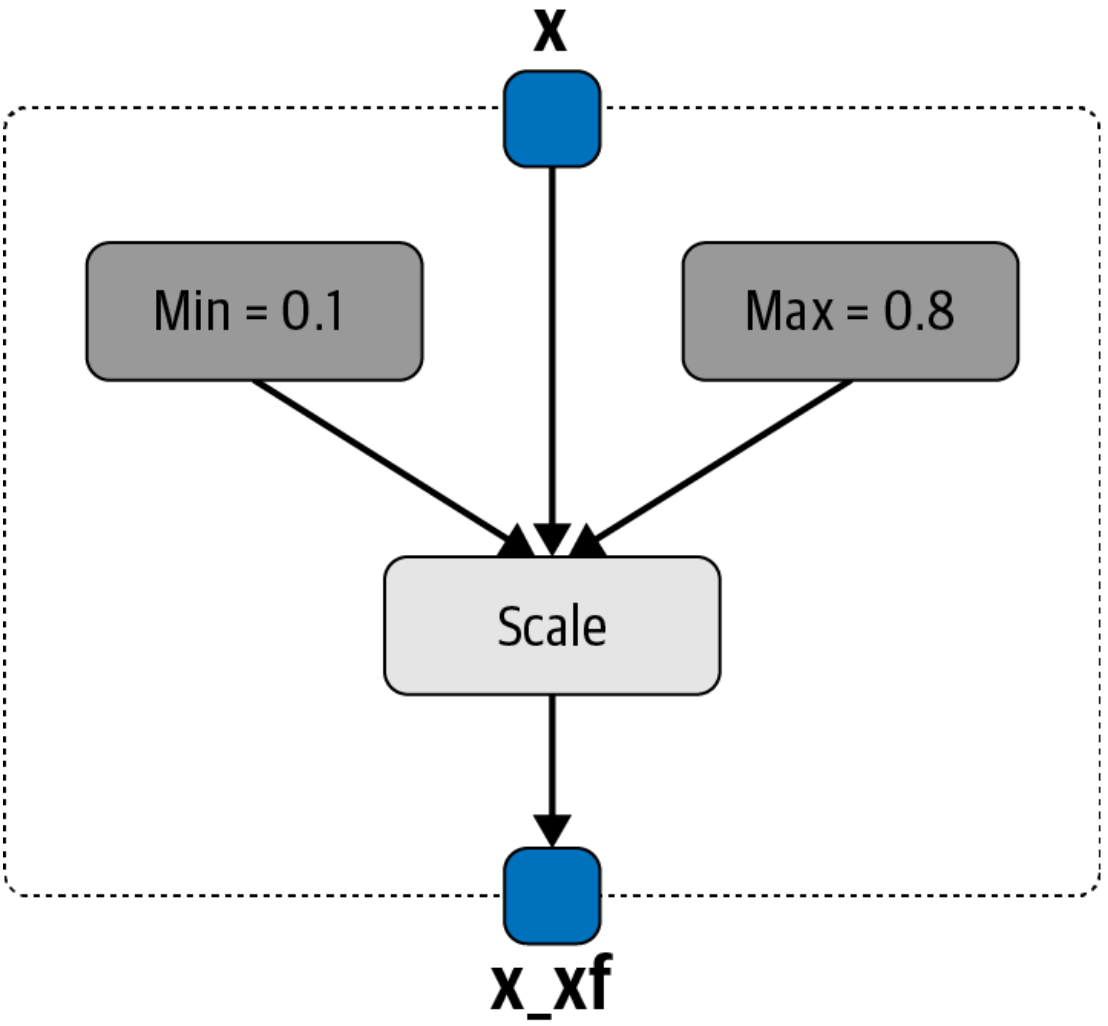

Figure 5-6. Applying the results of the analysis step

PREPROCESSING_FN()

Please note that TFT will build a graph out of the `preprocessing_fn()` function and it will run in its own session. It is expected that the function returns a dictionary with the transformed features as values of the Python dictionary.

Best Practices

During our work with TFT, we have learned a good number of lessons. Here are some of them:

Feature names matter

The naming of the output features of the preprocessing is important. As you will see in the following TFT implementations, we reuse the name of the input feature and append `_xf` . Also, the names of the input nodes of the TensorFlow models need to match the

names of the output features from the `preprocessing_fn` function.

Consider the data types

TFT limits the data types of the output features. It exports all preprocessed features as either `tf.string`, `tf.float32`, or `tf.int64` values. This is important in case your model can't consume these data types. Some models from TensorFlow Hub require inputs to be presented as `tf.int32` values (e.g., BERT models). We can avoid that situation if we cast the inputs to the correct data types inside our models or if we convert the data types in the estimator input functions.

Preprocessing happens in batches

When you write preprocessing functions, you might think of it as processing one data row at a time. In fact, TFT performs the operations in batches. This is why we will need to reshape the output of the `preprocessing_fn()` function to a Tensor or SparseTensor when we use it in the context of our `Transform` component.

Remember, no eager execution

The functions inside of the `preprocessing_fn()` function need to be represented by TensorFlow ops. If you want to lower an input string, you couldn't use `lower()`. You have to use the TensorFlow operation `tf.strings.lower()` to perform the same procedure in a graph mode. Eager execution isn't supported; all operations rely on pure TensorFlow graph operations.

`tf.function` can be used in `preprocessing_fn()` functions, but with restrictions: You can only use a `tf.function` that accepts Tensors (i.e., `lower()` wouldn't work since it doesn't work on a tensor). You can't call a TFT analyzer (or a mapper that relies on an analyzer, such as `tft.scale_to_z_score`).

TFT Functions

TFT provides a variety of functions to facilitate efficient feature engineering. The list of provided functions is extensive and constantly growing. This is why we don't claim to present a complete list of supported functions, but we want to highlight useful operations in relation to vocabulary generation, normalization, and bucketization:

`tft.scale_to_z_score()`

If you want to normalize a feature with a mean of 0 and standard deviation of 1, you can use this useful TFT function.

`tft.bucketize()`

This useful function lets you bucketize a feature into bins. It returns a bin or bucket index. You can specify the argument `num_buckets` to set the number of buckets. TFT will then divide the equal-sized buckets.

`tft.pca()`

This function lets you compute the *principal component analysis* (PCA) for a given feature. PCA is a common technique to reduce dimensionality by linearly projecting the data down to the subspace that best preserves the variance of the data. It requires the argument `output_dim` to set the dimensionality of your PCA representation.

`tft.compute_and_apply_vocabulary()`

This is one of the most amazing TFT functions. It computes all unique values of a feature column and then maps the most frequent values to an index. This index mapping is then used to convert the feature to a numerical representation. The function generates all assets for your graph behind the scenes. We can configure *most frequent* in two ways: either by defining the *n* highest-ranked unique items with `top_k` or by using the `frequency_threshold` above each element for consideration in the vocabulary.

`tft.apply_saved_model()`

This function lets you apply entire TensorFlow models on a feature. We can load a saved model with a given `tag` and `signature_name`

and then the `inputs` will be passed to the model. The predictions from the model execution will then be returned.

Text data for natural language problems

If you are working on natural language processing problems and you would like to utilize TFT for your corpus preprocessing to turn your documents into numerical representations, TFT has a good number of functions at your disposal. Besides the introduced function `tft.compute_and_apply_vocabulary()`, you could also use the following TFT functions.

`tft.ngrams()`

This will generate *n-grams*. It takes a SparseTensor of string values as inputs. For example, if you want to generate one-grams and bi-grams for the list `['Tom', 'and', 'Jerry', 'are', 'friends']`), the function returns `[b'Tom', b'Tom and', b'and', b'and Jerry', b'Jerry', b'Jerry are', b'are', b'are friends', b'friends']`. Besides the sparse input tensor, the function takes two additional arguments: `ngram_range` and `separator`. `ngram_range` sets the range of n-grams. If your n-grams should contain one-grams and bi-grams, set the `ngram_range` to `(1, 2)`. The `separator` allows us to set the joining string or character. In our example, we set the `separator` to `" "`.

`tft.bag_of_words()`

This function uses `tft.ngrams` and generates a bag-of-words vector with a row for each unique n-gram. The original order of the n-grams may not be preserved if, for example, the tokens repeat within an input.

`tft.tfidf()`

A frequently used concept in natural language processing is TFIDF, or term frequency inverse document frequency. It generates two outputs: a vector with the token indices and a vector representing their TFIDF weights. The function expects a sparse input vector representing the token indices (the result of the `tft.compute_and_apply_vocabulary()` function). The dimension-

ality of these vectors is set by the `vocab_size` input argument. The weight for every token index is calculated by the document frequency of the token in a document times the inverse document frequency. This computation is often resource intensive. Therefore, distributing the computation with TFT is of great advantage.

TensorFlow Text also lets you use all available functions from the TensorFlow Text library. The library provides extensive TensorFlow support for text normalization, text tokenization, n-gram computation, and modern language models like BERT.

Image data for computer vision problems

If you are working on computer vision models, TFT can preprocess the image datasets for you. TensorFlow provides various image preprocessing operations with the `tf.images` and the `tf.io` APIs.

`tf.io` provides useful functions to open images as part of a model graph (e.g., `tf.io.decode_jpeg` and `tf.io.decode_png`). `tf.images` provides functions to crop or resize images, convert color schemes, adjust the images (e.g., contrast, hue, or brightness), or perform image transformations like image flipping, transposing, etc.

In Chapter 3, we discussed strategies to ingest images into our pipelines. In TFT, we can now read the encoded images from TFRecord files and, for example, resize them to a fixed size or reduce color images to grayscale images. Here is an implementation example of such a `preprocessing_fn` function:

```
def process_image(raw_image):
    raw_image = tf.reshape(raw_image, [-1])
    img_rgb = tf.io.decode_jpeg(raw_image, channels=3) ❶
    img_gray = tf.image.rgb_to_grayscale(img_rgb) ❷
    img = tf.image.convert_image_dtype(img, tf.float32)
    resized_img = tf.image.resize_with_pad( ❸
        img,
        target_height=300,
        target_width=300
    )
```

```
    img_grayscale = tf.image.rgb_to_grayscale(resized_img) ❹
    return tf.reshape(img_grayscale, [-1, 300, 300, 1])
```

❶ Decode the JPEG image format.

❷ Convert the loaded RGB image to grayscale.

❸ Resize the image to 300 × 300 pixels.

❹ Convert the image to grayscale.

One note regarding the `tf.reshape()` operation as part of the `return` statement: TFT might process inputs in batches. Since the batch size is handled by TFT (and Apache Beam), we need to reshape the output of our function to handle any batch size. Therefore, we are setting the first dimension of our return tensor to *–1*. The remaining dimensions represent our images. We are resizing them to 300 × 300 pixels and reducing the RGB channels to a grayscale channel.

Standalone Execution of TFT

After we have defined our `preprocessing_fn` function, we need to focus on how to execute the `Transform` function. For the execution, we have two options. We can either execute the preprocessing transformations in a standalone setup or as part of our machine learning pipeline in the form of a TFX component. Both types of executions can be performed on a local Apache Beam setup or on Google Cloud's Dataflow service. In this section, we will discuss the standalone executions of TFT. This would be the recommended situation if you would like to preprocess data effectively outside of pipelines. If you are interested in how to integrate TFT into your pipelines, feel free to jump to "Integrate TFT into Your Machine Learning Pipeline".

Apache Beam provides a depth of functionality that is outside the scope of this book. It deserves its own publication. However, we want to walk you through the "Hello World" example of preprocessing with Apache Beam.

In our example, we would like to apply the normalization preprocessing function that we introduced earlier on our tiny raw dataset, shown in the following source code:

```
raw_data = [
    {'x':   1.20},
    {'x':   2.99},
    {'x': 100.00}
]
```

First, we need to define a data schema. We can generate a schema from a feature specification, as shown in the following source code. Our tiny dataset only contains one feature named `x`. We define the feature with the `tf.float32` data type:

```
import tensorflow as tf
from tensorflow_transform.tf_metadata import dataset_metadata
from tensorflow_transform.tf_metadata import schema_utils

raw_data_metadata = dataset_metadata.DatasetMetadata(
    schema_utils.schema_from_feature_spec({
        'x': tf.io.FixedLenFeature([], tf.float32),
    }))
```

With the dataset loaded and the data schema generated, we can now execute the preprocessing function `preprocessing_fn`, which we defined earlier. TFT provides bindings for the execution on Apache Beam with the function `AnalyzeAndTransformDataset`. This function is performing the two-step process we discussed earlier: first analyze the dataset and then transform it. The execution is performed through the Python context manager `tft_beam.Context`, which allows us to set, for example, the desired batch size. However, we recommend using the default batch size because it is more performant in common use cases. The following example shows the usage of the AnalyzeAndTransformDataset function:

```
import tempfile
import tensorflow_transform.beam.impl as tft_beam

with beam.Pipeline() as pipeline:
```

```
with tft_beam.Context(temp_dir=tempfile.mkdtemp()):

    tfrecord_file = "/your/tf_records_file.tfrecord"
    raw_data = (
        pipeline | beam.io.ReadFromTFRecord(tfrecord_file))

    transformed_dataset, transform_fn = (
        (raw_data, raw_data_metadata) | tft_beam.AnalyzeAndTransformDataset
            preprocessing_fn))
```

The syntax of the Apache Beam function calls is a bit different from usual Python calls. In the earlier example, we apply the `preprocessing_fn` function with the Apache Beam function `AnalyzeAndTransformDataset()` and provide the two arguments with our data `raw_data` and our defined metadata schema `raw_data_metadata`. `AnalyzeAndTransformDataset()` then returns two artifacts: the preprocessed dataset and a function, here named `transform_fn`, representing the transform operations applied to our dataset.

If we test our "Hello World" example, execute the preprocessing steps, and print the results, we will see the tiny processed dataset:

```
transformed_data, transformed_metadata = transformed_dataset
print(transformed_data)
[
    {'x_xf': 0.0},
    {'x_xf': 0.018117407},
    {'x_xf': 1.0}
]
```

In our "Hello World" example, we completely ignored the fact that the data isn't available as a Python dictionary, which often needs to be read from a disk. Apache Beam provides functions to handle file ingestions effectively (e.g., with `beam.io.ReadFromText()` or `beam.io.ReadFromTFRecord()`) in the context of building TensorFlow models.

As you can see, defining Apache Beam executions can get complex quickly, and we understand that data scientists and machine learning en-

gineers aren't in the business of writing execution instructions from scratch. This is why TFX is so handy. It abstracts all the instructions under the hood and lets the data scientist focus on their problem-specific setups like defining the `preprocessing_fn()` function. In the next section, we will take a closer look into the `Transform` setup for our example project.

Integrate TFT into Your Machine Learning Pipeline

In the final section of this chapter, we discuss how to apply TFT capabilities to our example project. In Chapter 4, we investigated the dataset and determined which features are categorical or numerical, which features should be bucketized, and which feature we want to embed from string representation to a vector representation. This information is crucial for defining our feature engineering.

In the following code, we define our features. For simpler processing later on, we group the input feature names in dictionaries representing each transform output data type: one-hot encoded features, bucketized features, and raw string representations:

```
import tensorflow as tf
import tensorflow_transform as tft

LABEL_KEY = "consumer_disputed"

# Feature name, feature dimensionality.
ONE_HOT_FEATURES = {
    "product": 11,
    "sub_product": 45,
    "company_response": 5,
    "state": 60,
    "issue": 90
}

# Feature name, bucket count.
BUCKET_FEATURES = {
    "zip_code": 10
}
```

```
# Feature name, value is unused.
TEXT_FEATURES = {
    "consumer_complaint_narrative": None
}
```

Before we can loop over these input feature dictionaries, let's define a few helper functions to transform the data efficiently. It is a good practice to rename the features by appending a suffix to the feature name (e.g., `_xf`). The suffix will help distinguish whether errors are originating from input or output features and prevent us from accidentally using a nontransformed feature in our actual model:

```
def transformed_name(key):
    return key + '_xf'
```

Some of our features are of a sparse nature, but TFT expects the transformation outputs to be dense. We can use the following helper function to convert sparse to dense features and to fill the missing values with a default value:

```
def fill_in_missing(x):
    default_value = '' if x.dtype == tf.string or to_string else 0
    if type(x) == tf.SparseTensor:
        x = tf.sparse.to_dense(
            tf.SparseTensor(x.indices, x.values, [x.dense_shape[0], 1]),
                            default_value)
    return tf.squeeze(x, axis=1)
```

In our model, we represent most input features as one-hot encoded vectors. The following helper function converts a given index to a one-hot encoded representation and returns the vector:

```
def convert_num_to_one_hot(label_tensor, num_labels=2):
    one_hot_tensor = tf.one_hot(label_tensor, num_labels)
    return tf.reshape(one_hot_tensor, [-1, num_labels])
```

Before we can process our features, we need one more helper function to convert zip codes represented as strings to float values. Our dataset lists

zip codes as follows:

```
zip codes
97XXX
98XXX
```

To bucketize records with missing zip codes correctly, we replaced the placeholders with zeros and bucketized the resulting floats into 10 buckets:

```
def convert_zip_code(zip_code):
    if zip_code == '':
        zip_code = "00000"
    zip_code = tf.strings.regex_replace(zip_code, r'X{0,5}', "0")
    zip_code = tf.strings.to_number(zip_code, out_type=tf.float32)
    return zip_code
```

With all the helper functions in place, we can now loop over each feature column and transform it depending on the type. For example, for our features to be converted to one-hot features, we convert the category names to an index with `tft.compute_and_apply_vocabulary()` and then convert the index to a one-hot vector representation with our helper function `convert_num_to_one_hot()`. Since we are using `tft.compute_and_apply_vocabulary()`, TensorFlow Transform will first loop over all categories and then determine a complete category to index mapping. This mapping will then be applied during our evaluation and serving phase of the model:

```
def preprocessing_fn(inputs):
    outputs = {}
    for key in ONE_HOT_FEATURES.keys():
        dim = ONE_HOT_FEATURES[key]
        index = tft.compute_and_apply_vocabulary(
            fill_in_missing(inputs[key]), top_k=dim + 1)
        outputs[transformed_name(key)] = convert_num_to_one_hot(
            index, num_labels=dim + 1)
    ...
    return outputs
```

Our processing of the bucket features is very similar. We decided to bucketize the zipcodes because one-hot encoded zip codes seemed too sparse. Each feature is bucketized into, in our case, 10 buckets, and we encode the index of the bucket as one-hot vectors:

```
for key, bucket_count in BUCKET_FEATURES.items():
    temp_feature = tft.bucketize(
            convert_zip_code(fill_in_missing(inputs[key])),
            bucket_count,
            always_return_num_quantiles=False)
    outputs[transformed_name(key)] = convert_num_to_one_hot(
            temp_feature,
            num_labels=bucket_count + 1)
```

Our text input features as well as our label column don't require any transformations; therefore, we simply convert them to dense features in case a feature might be sparse:

```
for key in TEXT_FEATURES.keys():
    outputs[transformed_name(key)] = \
        fill_in_missing(inputs[key])

outputs[transformed_name(LABEL_KEY)] = fill_in_missing(inputs[LABEL_KEY])
```

WHY WE DIDN'T EMBED THE TEXT FEATURES TO VECTORS

You might wonder why we haven't embedded our text features into a fixed vector as part of our transform step. This is certainly possible. But we decided to load the TensorFlow Hub model as a part of the model instead of the preprocessing. The key reason for that decision was that we could make the embeddings trainable and refine the vector representations during our training phase. Therefore, they cannot be hard-coded into the preprocessing step and represented as a fixed graph during the training phase.

If we use the `Transform` component from TFX in our pipeline, it expects the transformation code to be provided in a separate Python file. The name of the module file can be set by the user (e.g., in our case

`module.py`), but the entry point `preprocessing_fn()` needs to be contained in the module file and the function can't be renamed:

```
transform = Transform(
    examples=example_gen.outputs['examples'],
    schema=schema_gen.outputs['schema'],
    module_file=os.path.abspath("module.py"))
context.run(transform)
```

When we execute the `Transform` component, TFX will apply the transformations defined in our `module.py` module file to the loaded input data, which was converted to TFRecord data structures during the data ingestion step. The component will then output our transformed data, a transform graph, and the required metadata.

The transformed data and the transform graph can be consumed during our next step, the `Trainer` component. Check out "Running the Trainer Component" for how to consume the outputs of our `Transform` component. The following chapter also highlights how the generated transform graph can be combined with the trained model to export a saved model. More details can be found in Example 6-2.

Summary

In this chapter, we discussed how to effectively preprocess data in our machine learning pipelines with TFT. We introduced how to write `preprocessing_fn` functions, provided an overview of some available functions provided by TFT, and discussed how to integrate the preprocessing steps into the TFX pipeline. Now that the data has been preprocessed, it is time to train our model.