Chapter 3. Data Ingestion

With the basic TFX setup and the ML MetadataStore in place, in this chapter, we focus on how to ingest your datasets into a pipeline for consumption in various components, as shown in Figure 3-1.

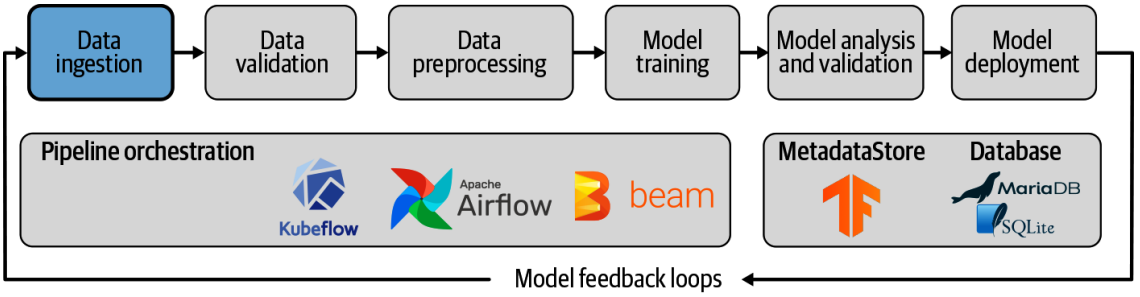

Figure 3-1. Data ingestion as part of ML pipelines

TFX provides us components to ingest data from files or services. In this chapter, we outline the underlying concepts, explain ways to split the datasets into training and evaluation subsets, and demonstrate how to combine multiple data exports into one all-encompassing dataset. We then discuss some strategies to ingest different forms of data (structured, text, and images), which have proven helpful in previous use cases.

Concepts for Data Ingestion

In this step of our pipeline, we read data files or request the data for our pipeline run from an external service (e.g., Google Cloud BigQuery). Before passing the ingested dataset to the next component, we divide the available data into separate datasets (e.g., training and validation datasets) and then convert the datasets into TFRecord files containing the data represented as `tf.Example` data structures.

TFRECORD

TFRecord is a lightweight format optimized for *streaming* large datasets. While in practice, most TensorFlow users store serialized example Protocol Buffers in TFRecord files, the TFRecord file format actually supports any binary data, as shown in the following:

```
import tensorflow as tf

with tf.io.TFRecordWriter("test.tfrecord") as w:
    w.write(b"First record")
    w.write(b"Second record")

for record in tf.data.TFRecordDataset("test.tfrecord"):
    print(record)

tf.Tensor(b'First record', shape=(), dtype=string)
tf.Tensor(b'Second record', shape=(), dtype=string)
```

If TFRecord files contain `tf.Example` records, each record contains one or more features that would represent the columns in our data. The data is then stored in binary files, which can be digested efficiently. If you are interested in the internals of TFRecord files, we recommend the TensorFlow documentation.

Storing your data as *TFRecord* and `tf.Examples` provides a few benefits:

1. The data structure is system independent since it relies on Protocol Buffers, a cross-platform, cross-language library, to serialize data.
2. TFRecord is optimized for downloading or writing large amounts of data quickly.
3. `tf.Example` , the data structure representing every data row within TFRecord, is also the default data structure in the TensorFlow ecosystem and, therefore, is used in all TFX components.

The process of ingesting, splitting, and converting the datasets is performed by the `ExampleGen` component. As we see in the following exam-

ples, datasets can be read from local and remote folders as well as requested from data services like Google Cloud BigQuery.

Ingesting Local Data Files

The `ExampleGen` component can ingest a few data structures, including *comma-separated value* files (CSVs), precomputed TFRecord files, and serialization outputs from Apache Avro and Apache Parquet.

Converting comma-separated data to tf.Example

Datasets for structured data or text data are often stored in CSV files. TFX provides functionality to read and convert these files to `tf.Example` data structures. The following code example demonstrates the ingestion of a folder containing the CSV data of our example project:

```
import os
from tfx.components import CsvExampleGen
from tfx.utils.dsl_utils import external_input

base_dir = os.getcwd()
data_dir = os.path.join(os.pardir, "data")
examples = external_input(os.path.join(base_dir, data_dir)) ❶
example_gen = CsvExampleGen(input=examples) ❷

context.run(example_gen) ❸
```

❶ Define the data path.

❷ Instantiate the pipeline component.

❸ Execute the component interactively.

If you execute the component as part of an interactive pipeline, the metadata of the run will be shown in the Jupyter Notebook. The outputs of the component are shown in Figure 3-2, highlighting the storage locations of the training and the evaluation datasets.

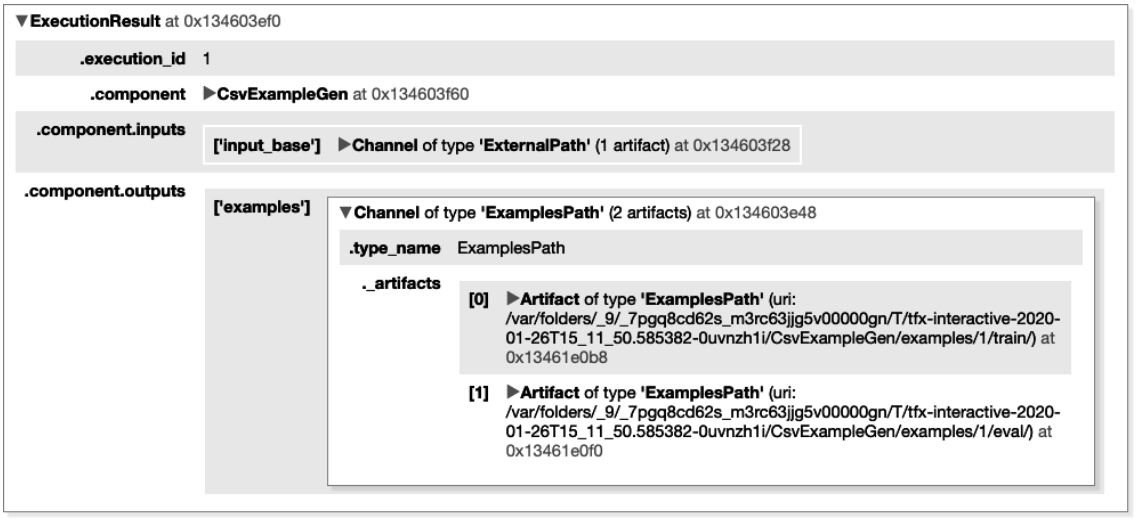

Figure 3-2. `ExampleGen` component output

FOLDER STRUCTURE

It is expected that the input path of *ExampleGen* only contains the data files. The component tries to consume all existing files within the path level. Any additional files (e.g., metadata files) can't be consumed by the component and make the component step fail. The component is also not stepping through existing subdirectories unless it is configured as an input pattern.

Importing existing TFRecord Files

Sometimes our data can't be expressed efficiently as CSVs (e.g., when we want to load images for computer vision problems or large corpora for natural language processing problems). In these cases, it is recommended to convert the datasets to TFRecord data structures and then load the saved TFRecord files with the `ImportExampleGen` component. If you would like to perform the conversion of your data to TFRecord files as part of the pipeline, take a look at Chapter 10, in which we discuss the development of custom TFX components including a data ingestion component. TFRecord files can be ingested as shown in the following example:

```
import os
from tfx.components import ImportExampleGen
from tfx.utils.dsl_utils import external_input

base_dir = os.getcwd()
data_dir = os.path.join(os.pardir, "tfrecord_data")
```

```
examples = external_input(os.path.join(base_dir, data_dir))
example_gen = ImportExampleGen(input=examples)

context.run(example_gen)
```

Since the datasets are already stored as `tf.Example` records within the TFRecord files, they can be imported and don't need any conversion. The `ImportExampleGen` component handles this import step.

Converting Parquet-serialized data to tf.Example

In Chapter 2, we discussed the internal architecture of TFX components and the behavior of a component, which is driven by its `executor`. If we would like to load new file types into our pipeline, we can override the `executor_class` instead of writing a completely new component.

TFX includes `executor` classes for loading different file types, including Parquet serialized data. The following example shows how you can override the `executor_class` to change the loading behavior. Instead of using the `CsvExampleGen` or `ImportExampleGen` components, we will use a generic file loader component `FileBasedExampleGen`, which allows an override of the `executor_class`:

```
from tfx.components import FileBasedExampleGen ❶
from tfx.components.example_gen.custom_executors import parquet_executor ❷
from tfx.utils.dsl_utils import external_input

examples = external_input(parquet_dir_path)
example_gen = FileBasedExampleGen(
    input=examples,
    executor_class=parquet_executor.Executor) ❸
```

❶ Import generic file loader component.

❷ Import Parquet-specific executor.

❸ Override the executor.

Converting Avro-serialized data to tf.Example

The concept of overriding the `executor_class` can, of course, be expanded to almost any other file type. TFX provides additional classes, as shown in the following example for loading Avro-serialized data:

```
from tfx.components import FileBasedExampleGen ❶
from tfx.components.example_gen.custom_executors import avro_executor ❷
from tfx.utils.dsl_utils import external_input

examples = external_input(avro_dir_path)
example_gen = FileBasedExampleGen(
    input=examples,
    executor_class=avro_executor.Executor) ❸
```

❶ Import generic file loader component.

❷ Import the Avro-specific executor.

❸ Override the executor.

In case we want to load a different file type, we could write our custom executor specific to our file type and apply the same concepts of overriding the executor earlier. In Chapter 10, we will guide you through two examples of how to write your own custom data ingestion component and executor.

Converting your custom data to TFRecord data structures

Sometimes it is simpler to convert existing datasets to TFRecord data structures and then ingest them with the `ImportExampleGen` component as we discussed in "Importing existing TFRecord Files". This approach is useful if our data is not available through a data platform that allows efficient data streaming. For example, if we are training a computer vision model and we load a large number of images into our pipeline, we have to convert the images to TFRecord data structures in the first place (more

on this in the later section on "Image Data for Computer Vision Problems").

In the following example, we convert our structured data into TFRecord data structures. Imagine our data isn't available in a CSV format, only in JSON or XML. The following example can be used (with small modifications) to convert these data formats before ingesting them to our pipeline with the `ImportExampleGen` component.

To convert data of any type to TFRecord files, we need to create a `tf.Example` structure for every data record in the dataset. `tf.Example` is a simple but highly flexible data structure, which is a key-value mapping:

```
{"string": value}
```

In the case of TFRecord data structure, a `tf.Example` expects a `tf.Features` object, which accepts a dictionary of features containing key-value pairs. The key is always a string identifier representing the feature column, and the value is a `tf.train.Feature` object.

Example 3-1. TFRecord data structure

```
Record 1:
tf.Example
    tf.Features
        'column A': tf.train.Feature
        'column B': tf.train.Feature
        'column C': tf.train.Feature
```

`tf.train.Feature` allows three data types:

- `tf.train.BytesList`
- `tf.train.FloatList`
- `tf.train.Int64List`

To reduce code redundancy, we'll define helper functions to assist with converting the data records into the correct data structure used by

`tf.Example`:

```
import tensorflow as tf

def _bytes_feature(value):
    return tf.train.Feature(bytes_list=tf.train.BytesList(value=[value]))

def _float_feature(value):
    return tf.train.Feature(float_list=tf.train.FloatList(value=[value]))

def _int64_feature(value):
    return tf.train.Feature(int64_list=tf.train.Int64List(value=[value]))
```

With the helper functions in place, let's take a look at how we could convert our demo dataset to files containing the TFRecord data structure. First, we need to read our original data file and convert every data record into a `tf.Example` data structure and then save all records in a TFRecord file. The following code example is an abbreviated version. The complete example can be found in the book's GitHub repository under *chapters/data_ingestion*.

```
import csv
import tensorflow as tf

original_data_file = os.path.join(
    os.pardir, os.pardir, "data",
    "consumer-complaints.csv")
tfrecord_filename = "consumer-complaints.tfrecord"
tf_record_writer = tf.io.TFRecordWriter(tfrecord_filename) ❶

with open(original_data_file) as csv_file:
    reader = csv.DictReader(csv_file, delimiter=",", quotechar='"')
    for row in reader:
        example = tf.train.Example(features=tf.train.Features(feature={ ❷
            "product": _bytes_feature(row["product"]),
            "sub_product": _bytes_feature(row["sub_product"]),
            "issue": _bytes_feature(row["issue"]),
            "sub_issue": _bytes_feature(row["sub_issue"]),
            "state": _bytes_feature(row["state"]),
            "zip_code": _int64_feature(int(float(row["zip_code"]))),
```

```
                "company": _bytes_feature(row["company"]),
                "company_response": _bytes_feature(row["company_response"]),
                "consumer_complaint_narrative": \
                    _bytes_feature(row["consumer_complaint_narrative"]),
                "timely_response": _bytes_feature(row["timely_response"]),
                "consumer_disputed": _bytes_feature(row["consumer_disputed"]),
            }))
            tf_record_writer.write(example.SerializeToString()) ❸
        tf_record_writer.close()
```

❶ Creates a `TFRecordWriter` object that saves to the path specified in *`tfrecord_filename`*

❷ `tf.train.Example` for every data record

❸ Serializes the data structure

The generated TFRecord file *consumer-complaints.tfrecord* can now be imported with the `ImportExampleGen` component.

Ingesting Remote Data Files

The `ExampleGen` component can read files from remote cloud storage buckets like Google Cloud Storage or AWS Simple Storage Service (S3).[1] TFX users can provide the bucket path to the `external_input` function, as shown in the following example:

```
examples = external_input("gs://example_compliance_data/")
example_gen = CsvExampleGen(input=examples)
```

Access to private cloud storage buckets requires setting up the cloud provider credentials. The setup is provider specific. AWS is authenticating users through a user-specific *access key* and *access secret*. To access private AWS S3 buckets, you need to create a user access key and secret.[2] In contrast, the Google Cloud Platform (GCP) authenticates users through *service accounts*. To access private GCP Storage buckets, you need to cre-

ate a service account file with the permission to access the storage bucket.[3]

Ingesting Data Directly from Databases

TFX provides two components to ingest datasets directly from databases. In the following sections, we introduce the `BigQueryExampleGen` component to query data from BigQuery tables and the `PrestoExampleGen` component to query data from Presto databases.

Google Cloud BigQuery

TFX provides a component to ingest data from Google Cloud's BigQuery tables. This is a very efficient way of ingesting structured data if we execute our machine learning pipelines in the GCP ecosystem.

GOOGLE CLOUD CREDENTIALS

Executing the `BigQueryExampleGen` component requires that we have set the necessary Google Cloud credentials in our local environment. We need to create a service account with the required roles (at least *BigQuery Data Viewer* and *BigQuery Job User*). If you execute the component in the interactive context with Apache Beam or Apache Airflow, you have to specify the path to the service account credential file through the environment variable `GOOGLE_APPLICATION_CREDENTIALS` , as shown in the following code snippet. If you execute the component through Kubeflow Pipelines, you can provide the service account information through OpFunc functions introduced at "OpFunc Functions".

You can do this in Python with the following:

```
import os
os.environ["GOOGLE_APPLICATION_CREDENTIALS"] =
    "/path/to/credential_file.json"
```

For more details, see the Google Cloud documentation.

The following example shows the simplest way of querying our BigQuery tables:

```
from tfx.components import BigQueryExampleGen

query = """
    SELECT * FROM `<project_id>.<database>.<table_name>`
"""
example_gen = BigQueryExampleGen(query=query)
```

Of course, we can create more complex queries to select our data, for example, joining multiple tables.

CHANGES TO THE BIGQUERYEXAMPLEGEN COMPONENT

In TFX versions greater than 0.22.0, the *BigQueryExampleGen* component needs to be imported from `tfx.extensions.google_cloud_big_query`:

```
from tfx.extensions.google_cloud_big_query.example_gen \
    import component as big_query_example_gen_component
big_query_example_gen_component.BigQueryExampleGen(query=query)
```

Presto databases

If we want to ingest data from a Presto database, we can use `PrestoExampleGen`. The usage is very similar to `BigQueryExampleGen`, in which we defined a database query and then executed the query. The `PrestoExampleGen` component requires additional configuration to specify the database's connection details:

```
from proto import presto_config_pb2
from presto_component.component import PrestoExampleGen

query = """
    SELECT * FROM `<project_id>.<database>.<table_name>`
"""
presto_config = presto_config_pb2.PrestoConnConfig(
    host='localhost',
```

```
    port=8080)
example_gen = PrestoExampleGen(presto_config, query=query)
```

PRESTOEXAMPLEGEN REQUIRES SEPARATE INSTALLATION

Since TFX version 0.22, the *PrestoExampleGen* requires a separate installation process. After installing the `protoc` compiler,[4] you can install the component from source with the steps below:

```
$ git clone git@github.com:tensorflow/tfx.git && cd tfx/
$ git checkout v0.22.0
$ cd tfx/examples/custom_components/presto_example_gen
$ pip install -e .
```

After the installation, you will be able to import the *PrestoExampleGen* component and its protocol buffer definitions.

Data Preparation

Each of the introduced `ExampleGen` components allows us to configure input settings (`input_config`) and output settings (`output_config`) for our dataset. If we would like to ingest datasets incrementally, we can define a span as the input configuration. At the same time, we can configure how the data should be split. Often we would like to generate a training set together with an evaluation and test set. We can define the details with the output configuration.

Splitting Datasets

Later in our pipeline, we will want to evaluate our machine learning model during the training and test it during the model analysis step. Therefore, it is beneficial to split the dataset into the required subsets.

Splitting one dataset into subsets

The following example shows how we can extend our data ingestion by requiring a three-way split: training, evaluation, and test sets with a ratio of 6:2:2. The ratio settings are defined through the `hash_buckets`:

```
from tfx.components import CsvExampleGen
from tfx.proto import example_gen_pb2
from tfx.utils.dsl_utils import external_input

base_dir = os.getcwd()
data_dir = os.path.join(os.pardir, "data")
output = example_gen_pb2.Output(
    split_config=example_gen_pb2.SplitConfig(splits=[ ❶
        example_gen_pb2.SplitConfig.Split(name='train', hash_buckets=6), ❷
        example_gen_pb2.SplitConfig.Split(name='eval', hash_buckets=2),
        example_gen_pb2.SplitConfig.Split(name='test', hash_buckets=2)
    ]))

examples = external_input(os.path.join(base_dir, data_dir))
example_gen = CsvExampleGen(input=examples, output_config=output) ❸

context.run(example_gen)
```

❶ Define preferred splits.

❷ Specify the ratio.

❸ Add `output_config` argument.

After the execution of the `example_gen` object, we can inspect the generated artifacts by printing the list of the artifacts:

```
for artifact in example_gen.outputs['examples'].get():
    print(artifact)

Artifact(type_name: ExamplesPath,
    uri: /path/to/CsvExampleGen/examples/1/train/, split: train, id: 2)
Artifact(type_name: ExamplesPath,
    uri: /path/to/CsvExampleGen/examples/1/eval/, split: eval, id: 3)
```

```
Artifact(type_name: ExamplesPath,
    uri: /path/to/CsvExampleGen/examples/1/test/, split: test, id: 4)
```

In the following chapter, we will discuss how we investigate the produced datasets for our data pipeline.

DEFAULT SPLITS

If we don't specify any output configuration, the `ExampleGen` component splits the dataset into a training and evaluation split with a ratio of 2:1 by default.

Preserving existing splits

In some situations, we have already generated the subsets of the datasets externally, and we would like to preserve these splits when we ingest the datasets. We can achieve this by providing an input configuration.

For the following configuration, let's assume that our dataset has been split externally and saved in subdirectories:

```
└── data
    ├── train
    │   └── 20k-consumer-complaints-training.csv
    ├── eval
    │   └── 4k-consumer-complaints-eval.csv
    └── test
        └── 2k-consumer-complaints-test.csv
```

We can preserve the existing input split by defining this input configuration:

```
import os

from tfx.components import CsvExampleGen
from tfx.proto import example_gen_pb2
from tfx.utils.dsl_utils import external_input

base_dir = os.getcwd()
```

```
data_dir = os.path.join(os.pardir, "data")

input = example_gen_pb2.Input(splits=[
    example_gen_pb2.Input.Split(name='train', pattern='train/*'), ❶
    example_gen_pb2.Input.Split(name='eval', pattern='eval/*'),
    example_gen_pb2.Input.Split(name='test', pattern='test/*')
])

examples = external_input(os.path.join(base_dir, data_dir))
example_gen = CsvExampleGen(input=examples, input_config=input)
❷
```

❶ Set existing subdirectories.

❷ Add the `input_config` argument.

After defining the input configuration, we can pass the settings to the `ExampleGen` component by defining the `input_config` argument.

Spanning Datasets

One of the significant use cases for machine learning pipelines is that we can update our machine learning models when new data becomes available. For this scenario, the `ExampleGen` component allows us to use *spans*. Think of a span as a snapshot of data. Every hour, day, or week, a batch *extract, transform, load* (ETL) process could make such a data snapshot and create a new span.

A span can replicate the existing data records. As shown in the following, *export-1* contains the data from the previous *export-0* as well as newly created records that were added since the *export-0* export:

```
└── data
    ├── export-0
    │   └── 20k-consumer-complaints.csv
    ├── export-1
    │   └── 24k-consumer-complaints.csv
```

```
    └── export-2
        └── 26k-consumer-complaints.csv
```

We can now specify the patterns of the spans. The input configuration accepts a `{SPAN}` placeholder, which represents the number (0, 1, 2, ...) shown in our folder structure. With the input configuration, the `ExampleGen` component now picks up the "latest" span. In our example, this would be the data available under folder *export-2*:

```
from tfx.components import CsvExampleGen
from tfx.proto import example_gen_pb2
from tfx.utils.dsl_utils import external_input

base_dir = os.getcwd()
data_dir = os.path.join(os.pardir, "data")

input = example_gen_pb2.Input(splits=[
    example_gen_pb2.Input.Split(pattern='export-{SPAN}/*')
])

examples = external_input(os.path.join(base_dir, data_dir))
example_gen = CsvExampleGen(input=examples, input_config=input)
context.run(example_gen)
```

Of course, the input definitions can also define subdirectories if the data is already split:

```
input = example_gen_pb2.Input(splits=[
    example_gen_pb2.Input.Split(name='train',
                                pattern='export-{SPAN}/train/*'),
    example_gen_pb2.Input.Split(name='eval',
                                pattern='export-{SPAN}/eval/*')
])
```

Versioning Datasets

In machine learning pipelines, we want to track the produced models together with the used datasets, which were used to train the machine

learning model. To do this, it is useful to version our datasets.

Data versioning allows us to track the ingested data in more detail. This means that we not only store the file name and path of the ingested data in the ML MetadataStore (because it's currently supported by the TFX components) but also that we track more metainformation about the raw dataset, such as a hash of the ingested data. Such version tracking would allow us to verify that the dataset used during the training is still the dataset at a later point in time. Such a feature is critical for end-to-end ML reproducibility.

However, such a feature is currently not supported by the TFX `ExampleGen` component. If you would like to version your datasets, you can use third-party data versioning tools and version the data before the datasets are ingested into the pipeline. Unfortunately, none of the available tools will write the metadata information to the TFX ML MetadataStore directly.

If you would like to version your datasets, you can use one of the following tools:

Data Version Control (DVC)

DVC is an open source version control system for machine learning projects. It lets you commit hashes of your datasets instead of the entire dataset itself. Therefore, the state of the dataset is tracked (e.g., via `git`), but the repository isn't cluttered with the entire dataset.

Pachyderm

Pachyderm is an open source machine learning platform running on Kubernetes. It originated with the concept of versioning for data ("Git for data") but has now expanded into an entire data platform, including pipeline orchestration based on data versions.

Ingestion Strategies

So far, we have discussed a variety of ways to ingest data into our machine learning pipelines. If you are starting with an entirely new project, it might be overwhelming to choose the right data ingestion strategy. In the following sections, we will provide a few suggestions for three data types: structured, text, and image data.

Structured Data

Structured data is often stored in a database or on a disk in file format, supporting tabular data. If the data exists in a database, we can either export it to CSVs or consume the data directly with the `PrestoExampleGen` or the `BigQueryExampleGen` components (if the services are available).

Data available on a disk stored in file formats supporting tabular data should be converted to CSVs and ingested into the pipeline with the `CsvExampleGen` component. Should the amount of data grow beyond a few hundred megabytes, you should consider converting the data into TFRecord files or store the data with Apache Parquet.

Text Data for Natural Language Problems

Text corpora can snowball to a considerable size. To ingest such datasets efficiently, we recommend converting the datasets to TFRecord or Apache Parquet representations. Using performant data file types allows an efficient and incremental loading of the corpus documents. The ingestion of the corpora from a database is also possible; however, we recommend considering network traffic costs and bottlenecks.

Image Data for Computer Vision Problems

We recommend converting image datasets from the image files to TFRecord files, but not to decode the images. Any decoding of highly compressed images only increases the amount of disk space needed to store the intermediate `tf.Example` records. The compressed images can be stored in `tf.Example` records as byte strings:

```
import tensorflow as tf

base_path = "/path/to/images"
filenames = os.listdir(base_path)

def generate_label_from_path(image_path):
    ...
    return label

def _bytes_feature(value):
    return tf.train.Feature(bytes_list=tf.train.BytesList(value=[value]))

def _int64_feature(value):
    return tf.train.Feature(int64_list=tf.train.Int64List(value=[value]))

tfrecord_filename = 'data/image_dataset.tfrecord'

with tf.io.TFRecordWriter(tfrecord_filename) as writer:
    for img_path in filenames:
        image_path = os.path.join(base_path, img_path)
        try:
            raw_file = tf.io.read_file(image_path)
        except FileNotFoundError:
            print("File {} could not be found".format(image_path))
            continue
        example = tf.train.Example(features=tf.train.Features(feature={
            'image_raw': _bytes_feature(raw_file.numpy()),
            'label': _int64_feature(generate_label_from_path(image_path))
        }))
        writer.write(example.SerializeToString())
```

The example code reads images from a provided path */path/to/images* and stores the image as byte strings in the `tf.Example`. We aren't preprocessing our images at this point in the pipeline. Even though we might save a considerable amount of disk space, we want to perform these tasks later in our pipeline. Avoiding the preprocessing at this point helps us to prevent bugs and potential training/serving skew later on.

We store the raw image together with labels in the `tf.Examples`. We derive the label for each image from the file name with the function

`generate_label_from_path` in our example. The label generation is dataset specific; therefore, we haven't included it in this example.

After converting the images to TFRecord files, we can consume the datasets efficiently with the `ImportExampleGen` component and apply the same strategies we discussed in "Importing existing TFRecord Files".

Summary

In this chapter, we discussed various ways of ingesting data into our machine learning pipeline. We highlighted the consumption of datasets stored on a disk as well as in databases. In the process, we also discussed that ingested data records, which are converted to `tf.Example` (store in TFRecord files) for the consumption of the downstream components.

In the following chapter, we will take a look at how we can consume the generated `tf.Example` records in our data validation step of the pipeline.

1 Reading files from AWS S3 requires Apache Beam 2.19 or higher, which is supported since TFX version 0.22.

2 See the documentation for more details on managing AWS Access Keys.

3 See the documentation for more details on how to create and manage service accounts.

4 Visit the proto-lens GitHub for details on the `protoc` installation.