Chapter 20. ML Pipelines for Computer Vision Problems

In this chapter and the next, we will walk through two ML pipelines that demonstrate a holistic set of common ML problems. We will set up the problems and show you how we implemented the solutions. We assume you have read the previous chapters and will refer to details from them.

In this chapter, we will walk through a typical computer vision problem. We are designing an ML pipeline for an image classification problem. The ML model itself isn't earth-shattering, but it isn't the goal to produce a complex model. We wanted to keep the model simple. That way, we can focus on the ML pipeline (the interesting aspect of ML production systems).

In this example, we want to train an ML model to classify images of pets into categories of cats and dogs (shown in Figure 20-1).

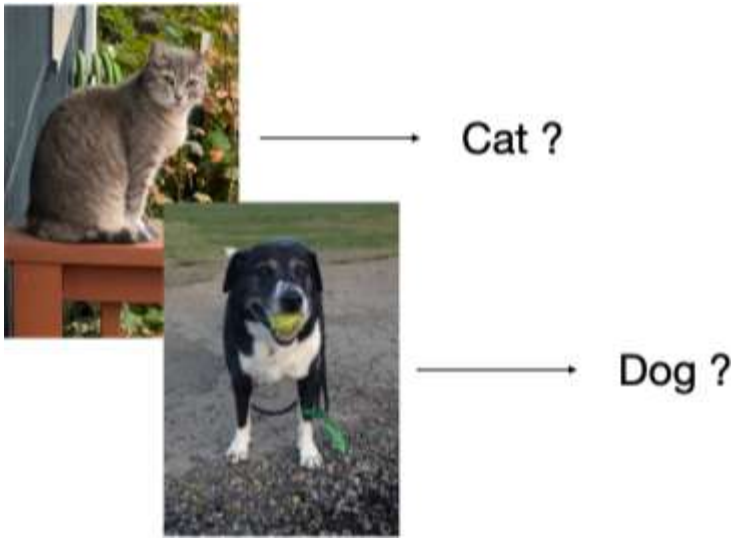

Figure 20-1. The classification problem

In this example, we will briefly discuss the ML models, and then we'll focus on the pipelines, building on the previous chapters. In particular, we'll highlight how to ingest or how to preprocess the image data.

WARNING

At the time of this writing, TFX doesn't support laptops based on Apple's Silicon architecture. If you are using a laptop based on the architecture (e.g., M1s), we highly recommend Google's Colab to work with TFX.

Our Data

For this example, we are using a public dataset compiled by Microsoft Research. The data consists of 25,000 pictures of dogs and cats, separated into two folders. Our example code contains two shell scripts that help you

set up the data for your respective environments (local deployment, Kubeflow, or Google Cloud Vertex). One script downloads the dataset to your local computer. Use this script if you want to follow the example from your computer. We also provide a shell script to download and set up the dataset on a remote Google Cloud bucket (*computer_vision/scripts/set_up_vertex_run.sh*).

Our Model

The example model was implemented using TensorFlow and Keras. We reused a pretrained model from Kaggle, called MobileNet. For a number of years, it was the go-to option for production computer vision problems. The model accepts images in the size of 160 × 160 × 3 pixels. The pretrained model outputs a vector that we then constrain further through a neural network dense layer, and finally through a softmax layer with output nodes (one representing the category “dog” and one representing the category “cat”).

The whole code setup is shown in the following code block:

```
    image_input = tf.keras.layers.Input(
        shape=(constants.PIXELS, constants.PIXELS,
        name=utils.transformed_name(constants.FEATU
        dtype=tf.float32
    )
```

```
    mobilenet_layer = hub.KerasLayer(
        constants.MOBILENET_TFHUB_URL,
        trainable=True,
        arguments=dict(batch_norm_momentum=0.997)
    )
    x = mobilenet_layer(image_input)
    x = tf.keras.layers.Dropout(DROPOUT_RATE)(x)
    x = tf.keras.layers.Dense(256, activation="relu
    output = tf.keras.layers.Dense(num_labels, acti
    model = tf.keras.Model(inputs=image_input, outp
```

NOTE

If you are new to TensorFlow, Keras, or ML in general, we highly recommend *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* by Aurélien Géron (O'Reilly).

Custom Ingestion Component

TFX provides a number of helpful data ingestion components, but unfortunately it provides no component to ingest image data. Therefore, we are using the custom component we discussed in "Reusing Existing Components" in Chapter 19. The custom component reads the images either from a local filesystem or from a remote location. It then compresses the image to reduce the image byte size and creates a base64 representation of the compressed binary image.

As shown in Figure 20-2, we then store the base64-converted image together with the training label in TFRecord files that TFX can consume in the downstream pipeline. We generate the label (cat or dog) by parsing the filepath. It contains information about the type of pet.

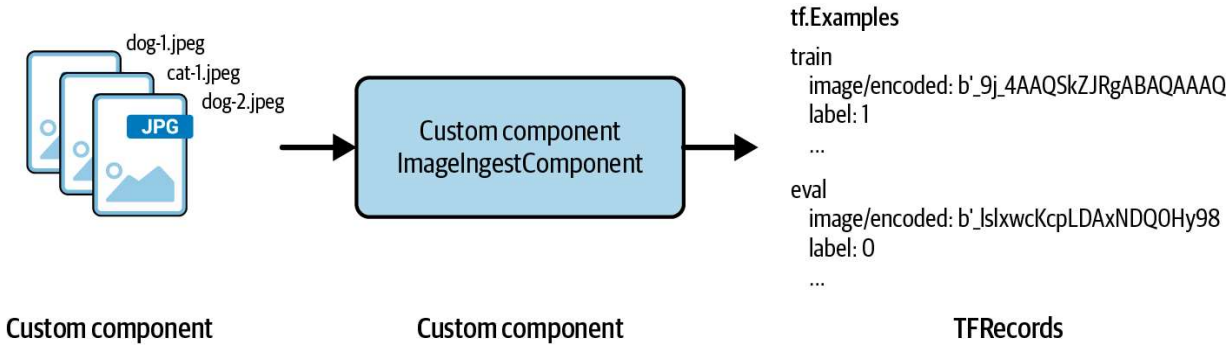

Figure 20-2. Workflow of the custom component

It is important to note that we don't resize images when ingesting the data into the pipeline. You might wonder why we don't convert all images to the 160 × 160 × 3 size our model consumes. If we implement the transformation from an image of an arbitrary size to a size our model can use during our data preprocessing step, we can then reuse that same transformation step when serving inferences using our deployed model. We discuss the preprocessing step in the next section.

Data Preprocessing

In "Consider Instance-Level Versus Full-Pass Transformations", we discussed feature engineering and TF Transform. Here, we want to bring the

knowledge to good use. In the example, the preprocessing step serves three purposes:

- Load the base64-encoded images and resize them to the size our pretrained model can handle.
- Normalize the images to `float32` values between 0 and 1.
- Convert the label information into an `integer` value we can later use for our training purposes.

First, we need to decode a base64-encoded image before we can resize the image to the size our pretrained model can consume. TensorFlow provides a number of utility functions for image preprocessing:

```
def preprocess_image(compressed_image: tf.Tensor
  """
    Preprocess a compressed image by resizing it
    Args:
      compressed_image: A compressed image in the
    Returns:
      A normalized image.
  """
  compressed_image_base64_decoded = tf.io.decode_
  raw_image = tf.io.decode_compressed(
      compressed_image_base64_decoded, compressio
  )
  image = tf.image.decode_jpeg(raw_image, channel
  tf.Assert(  # check that image has 3 channels
```

```
        tf.reduce_all(tf.equal(tf.shape(image)[-1],
        ["TF Preprocess: Check order of image chan
    image = tf.image.resize(  # resize to 160x160
        image, (constants.PIXELS, constants.PIXELS)
        antialias=True)
    image = image / 255  # normalize to [0,1] range
    return image
```

You might wonder why we're using TensorFlow Ops for the image conversion rather than more common image manipulation packages in Python. The reason is that TensorFlow Ops can easily be parallelized with TF Transform. Imagine you want to convert millions of images as part of your pipeline. In that case, parallelization is key.

Second, we can reuse the preprocessing steps when we deploy our TensorFlow model if they are expressed as TensorFlow Ops. In that case, our model server can accept images of any size and the images are conveniently converted ahead of the classification. That simplifies the integration of the model in an application, and it reduces the possibility of training–serving skew.

We'll convert the string labels ("cat" or "dog") to integer values (0 or 1) by computing a vocabulary with TF Transform and then applying the vocabulary across the entire dataset. TF Transform requires only a few lines of code to generate production-grade transformations:

```
def convert_labels(label_tensor: tf.Tensor) -> t
  """Converts a string label tensor into an int
  indexed_vocab_label = tft.compute_and_apply_vo
      label_tensor, top_k=constants.VOCAB_SIZE,
      num_oov_buckets=constants.OOV_SIZE,
      default_value=constants.VOCAB_DEFAULT_INDE
      vocab_filename=constants.LABEL_VOCAB_FILE_
  )
  return indexed_vocab_label
```

Thanks to TF Transform, we can run the transformation locally. Or, in cases where we want to transform terabytes of data, we can parallelize the transformation through services such as Google Dataflow. The transformation code remains the same. No code change is needed; we only need to change the runner for the TFX components.

Exporting the Model

At the end of our pipeline, we'll export the trained and validated TensorFlow model. We could easily call `tf.save_model.save(model)` and consider it done. But we would be missing out on amazing features of the TensorFlow ecosystem.

We can save the model with a model signature that can handle the preprocessing. That way, our deployed ML model can accept images of random sizes and the preprocessing is consistent.

Writing signatures for TensorFlow models looks complicated, but it is actually straightforward. First, we need to define a function that takes our trained model and the preprocessing graph from TF Transform.

The function moves the preprocessing graph to the model graph and then returns a TensorFlow function that accepts an arbitrary number of string inputs (representing our base64-encoded images), applying the preprocessing and inferring the model:

```
def _get_serve_features_signature(model, tf_trans
  """Returns a function that parses a raw input a
  model.tft_layer_input_only = tf_transform_outpu
  @tf.function(
      input_signature=[
          tf.TensorSpec(shape=(None, 1), dtype=tf
      ]
  )
  def serve_tf_raw_fn(image):
    model_input = {constants.FEATURE_KEY: image}
    transformed_features = model.tft_layer_input_
    transformed_features.pop(utils.transformed_na
```

```
      return model(transformed_features)
    return serve_tf_raw_fn
```

After the model is trained, we can save the model with the signature. In fact, TensorFlow models can handle multiple signatures. You could have signatures for different input formats or different output representations:

```
    signatures = {
        "serving_default":
            _get_serve_features_signature(model, t
    }
    tf.save_model.save(
        model, fn_args.serving_model_dir,
        save_format="tf", signatures=signatures)
```

The save method accepts a dictionary with the different signatures. The key represents the name with which it can be called during the inference. If no signature is specified during the inference, TensorFlow expects a signature with the name `serving_default`. Any data now passed to the `serving_default` signature will be transformed according to the steps we defined earlier before it is inferred and the results are returned.

Our Pipeline

Now, let's put all the steps together into a single pipeline. In this section, we dive into the individual aspects of the ML pipeline. If you want to follow along in our example project, we compiled the pipeline definition in the file *pipeline.py*.

Data Ingestion

As the first step in every pipeline, we need to ingest the data to train our model. This is where we'll use our custom ingestion component. Before we use the component, we need to configure the component.

The following lines define that we accept any JPEG image:

```
    input_config = example_gen_pb2.Input(
        splits=[
            example_gen_pb2.Input.Split(name="image
        ]
    )
```

As an output from the ingestion, we expect a dataset with 90% of all data samples being part of the training split and 10% being part of the evaluation split:

```
    output = tfx.v1.proto.Output(
        split_config=tfx.v1.proto.SplitConfig(
            splits=[
                tfx.v1.proto.SplitConfig.Split(name
                tfx.v1.proto.SplitConfig.Split(name
            ]
        )
    )
```

With the two configurations defined, we can set up our custom component. To avoid reinventing the wheel we are reusing the FileBasedExampleGen component provided by TFX. Here, we don't need to reimplement the entire component, but we can focus on swapping out the Executor portion of the component (where the actual magic happens).

We define our component as follows:

```
example_gen = FileBasedExampleGen(
    input_base=data_root,
    input_config=input_config,
    output_config=output,
    custom_executor_spec=executor_spec.BeamExecut
)
```

The `data_root` is the root directory where we stored the images. It can be a local folder or a remote file bucket.

Once the data is ingested, we can generate statistics and a schema describing the data with two lines of code:

```
    # Computes statistics over data for visualizati
    statistics_gen = StatisticsGen(examples=example

    # Generates schema based on statistics files.
    schema_gen = SchemaGen(
        statistics=statistics_gen.outputs["statisti
    )
```

Data Preprocessing

We save the defined preprocessing steps we discussed earlier in a file called *preprocessing.py*. We can now easily call the preprocessing steps through the Transform component from TFX:

```
    transform = Transform(
        examples=example_gen.outputs["examples"],
        schema=schema_gen.outputs["schema"],
```

```
        module_file="preprocessing.py"
    )
```

When the component is being executed, it will load the steps defined in *preprocessing.py* and perform the defined transformations. TFX is looking for a function called `preprocessing_fn` as an entry point to the preprocessing operations.

Model Training

The model training works similar to the preprocessing steps. We defined our model training in a file called *model.py*. The Python module contains the model definition, the training setup, and the discussed signatures as well as the model export setup.

TFX expects a function with the name `run_fn` as the entry point to all training operations.

The setup of the component is as simple as the Transform component. We provide the references to the module file, the preprocessed (not the raw) data, and the preprocessing graph (for the export) as well as the data schema information:

```
trainer = Trainer(
    module_file="model.py",
```

```
        examples=transform.outputs["transformed_examp
        transform_graph=transform.outputs["transform_
        schema=schema_gen.outputs["schema"],
    )
```

Model Evaluation

In Chapter 8, we discussed the evaluation of ML models. It is one of the most critical steps during the pipeline run.

If we want to compare the newly trained model against previously produced models, we need to first determine the last exported model for this pipeline. We can do this with the Resolver component, as discussed in Chapter 19:

```
    model_resolver = resolver.Resolver(
        model=Channel(type=Model),
        model_blessing=Channel(type=ModelBlessing),
        strategy_class=latest_blessed_model_resolver
    )
```

With the Resolver component, we can retrieve artifacts from our pipeline artifact store. In our case, we want to load the Model artifact and the artifacts containing the blessing information. Then, we define our strategy

of determining the relevant artifact. In our case, we want to retrieve the last blessed model.

Next, we need to define our evaluation configuration. The configuration consists of three major sections: the `model_specs`, the `slicing_specs`, and the `metrics_specs`.

The `model_specs` define how we interface with the model:

```
model_specs=[
    tfma.ModelSpec(
        signature_name="serving_examples",
        preprocessing_function_names=["transforr
        label_key="label_xf"
    )
]
```

We configure which model signature to use for the evaluation. In our example, we added an example consuming `TF Examples`, instead of raw features. That way, we can easily consume validation sets generated by the pipeline. In our example project, we also defined a model signature that assists with the transformation between raw and preprocessed features. The processing step is very helpful during the model evaluation since we can transform raw datasets and use the preprocessed datasets for the model

evaluation. Lastly, we define our label column. Here we are using the name of the preprocessed label column, in our case `label_xf`.

Next, we can define whether we want to slice the data during the evaluation. Since the example data only contains two populations, cats and dogs, we won't slice the data further. We will evaluate the model on the entire dataset:

```
slicing_specs=[tfma.SlicingSpec()]
```

And lastly, we need to define our model metrics and success criteria. In our example, we wanted to bless any model that fulfills two conditions—the overall sparse categorical accuracy needs to be above 0.6; and the overall accuracy needs to be higher than the previously blessed model:

```
metrics_specs=[
    tfma.MetricsSpec(
        metrics=[
            tfma.MetricConfig(
                class_name="SparseCategoricalAccu
                threshold=tfma.MetricThreshold(
                    value_threshold=tfma.GenericV
                        lower_bound={"value": 0.6
                    ), change_threshold=tfma.Gene
                        direction=tfma.MetricDire
                        absolute={"value": -1e-10
```

```
                        )
                    )
                )
            ]
        )
    ]
```

Once those three specifications are defined, we can create one single configuration:

```
eval_config = tfma.EvalConfig(
    model_specs=[...],
    slicing_specs=[...],
    metrics_specs=[...]
)
```

With the `eval_config`, we can now define the Evaluator component by providing the references to the required artifacts:

```
evaluator = Evaluator(
    model=trainer.outputs["model"],
    examples=example_gen.outputs["examples"],
    baseline_model=model_resolver.outputs["model"
    eval_config=eval_config
)
```

Here, we are evaluating the newly trained model by using the ingested validation dataset, and comparing the model against the resolved, previously blessed model based on the evaluation configuration.

Model Export

If the evaluation model is successful and the model is blessed, we are exporting the model to our export location defined as `serving_model_dir`. TFX provides the Pusher component for this task:

```
pusher = Pusher(
    model=trainer.outputs["model"],
    model_blessing=evaluator.outputs["blessing"],
    push_destination=pusher_pb2.PushDestination(
        filesystem=pusher_pb2.PushDestination.Fi
            base_directory=serving_model_dir
        )
    )
)
```

The model blessing is an optional flag. If you always want to export the model, regardless of the evaluation result, feel free to leave out the optional argument.

Putting It All Together

Regardless of what orchestrator we use, we need to define a pipeline object. In our example projects, we provide you with a little helper function to create your pipeline components. The function is called `create_components`:

```
components = create_components(
    data_root=constants.LOCAL_DATA_ROOT,
    serving_model_dir=constants.LOCAL_SERVING_MOD
)
```

We then define our optional pipeline configurations for Apache Beam and our metadata store:

```
beam_pipeline_args = [
    "--direct_num_workers=0",
]
metadata_path = os.path.join(
    constants.LOCAL_PIPELINE_ROOT, "metadata",
    constants.PIPELINE_NAME, "metadata.db"
)
```

TFX now allows us to convert the list of components into a directed pipeline graph and turn it into a generic pipeline object:

```
p = pipeline.Pipeline(
    components=components,
    pipeline_name=constants.PIPELINE_NAME,
    pipeline_root=constants.LOCAL_PIPELINE_ROOT,
    enable_cache=True,
    metadata_connection_config=metadata.
    sqlite_metadata_connection_config(metadata_pa
    beam_pipeline_args=beam_pipeline_args
)
```

With the generic pipeline now defined, let's focus on the execution of the pipeline.

Executing on Apache Beam

As we discussed in Chapter 18, running a TFX pipeline is as simple as executing the generated pipeline object:

```
from tfx.orchestration.beam.beam_dag_runner impor
…
BeamDagRunner().run(p)
```

This will execute the pipeline on the machine where you run your Python environment.

You will see the execution of the different components in sequential order:

```
INFO:absl:Successfully built user code wheel dist
68f9e690d01fe806b442cb18f7cee955ff5ab60941346c553
-py3-none-any.whl'; target user module is 'model
INFO:absl:Full user module path is  ...
68f9e690d01fe806b442cb18f7cee955ff5ab60941346c553
-py3-none-any.whl'
INFO:absl:Using deployment config:
 executor_specs {
  key: "Evaluator"
  value {
    ...
```

If you want to follow the Apache Beam example, you can execute the Python script *runner_beam.py* in the *computer vision* project.

Executing on Vertex Pipelines

We introduced Vertex Pipelines in "Executing Vertex Pipelines" in Chapter 18. The execution consists of two steps:

1. Convert the TFX pipeline into a graph definition.
2. Submit the graph definition to Vertex Pipelines.

In this section, we will focus on the project-specific details regarding the execution of Vertex Pipelines.

In Chapter 15, we mentioned that the `KubeflowV2DagRunnerConfig` gets configured with a `default_image`. We used the generic and publicly available Docker image `gcr.io/tfx-oss-public/tfx`; however, the image won't contain our custom component, preprocessing, and model modules. Generating a custom Docker image for your project isn't complicated. Here is how you do it for your project.

First, create a `Dockerfile` as follows in your project root directory. Update the TFX version if needed and adjust the components folder if you use a different project structure. If you have specific project dependencies, you can install them during the container build process:

```
FROM tensorflow/tfx:1.14.0
WORKDIR /pipeline
COPY ./components ./components
ENV PYTHONPATH="/pipeline:${PYTHONPATH}"
```

Once you define your `Dockerfile`, you need to build the image. You can do this by running `docker build` as follows:

```
$ PROJECT_ID="<your gcp project id>"
$ IMAGE_NAME="computer-vision-example"
$ IMAGE_TAG="1.0"

# Build the Docker image
$ docker build -t gcr.io/$PROJECT_ID/$IMAGE_NAME
```

If you are using Google Cloud for your repository of Docker images, you need to authenticate your local Docker client with the Google Cloud repository. You can do this by running:

```
$ gcloud auth configure-docker
```

Afterward, you can push the image to the Google Cloud repository with the following:

```
$ docker push gcr.io/$PROJECT_ID/$IMAGE_NAME:$IMA
```

Now, you can use the image `gcr.io/$PROJECT_ID/$IMAGE_NAME:$IMAGE_TAG` in your pipeline configuration:

```
...
cpu_container_image_uri = \
```

```
    "gcr.io/<your project id>/computer-vision-exa
# Create a Kubeflow V2 runner
runner_config = kubeflow_v2_dag_runner.KubeflowV2
    default_image=cpu_container_image_uri)
runner = kubeflow_v2_dag_runner.KubeflowV2DagRunn
    config=runner_config,
    output_filename=pipeline_definition_file
)
runner.run(pipeline=create_pipeline(), write_out=
…
```

The remainder of the pipeline setup is exactly as we discussed it in Chapter 18. After executing the runner, you submit the pipeline definition to Vertex Pipelines with `job.submit`:

```
aiplatform.init(
    project=constants.GCP_PROJECT_ID,
    location=constants.VERTEX_REGION,
)
job = aiplatform.PipelineJob(
    display_name=constants.PIPELINE_NAME + "-pipe
    template_path=pipeline_definition_file,
    pipeline_root=constants.GCS_PIPELINE_ROOT,
    enable_caching=True,
)
job.submit(
```

```
        service_account=constants.GCP_SERVICE_ACCOUNT
    )
```

If you want to follow the Vertex Pipelines example, you can execute the Python script *runner_vertex.py* in the *computer vision* project.

Model Deployment with TensorFlow Serving

If you want to deploy the trained model through your ML pipeline, you can easily do this by using TensorFlow Serving (TF Serving), as we explained in Chapters 12 through 14.

NOTE

While the example in this chapter focuses on local deployment with TF Serving, the next chapter demonstrates model deployment with Google Cloud Vertex.

For our deployment case, let's assume that you pushed your model to a local path defined in `serving_model_dir` when you created your Pusher component. TFX will save the trained model using protocol buffers for serializing the model. Make sure your `serving_model_dir` contains the model name and a version number (e.g.,

`cats_and_dogs_classification/1`). In the example, we are saving the first version.

You can deploy the model by using TF Serving's Docker container image. You can pull the TF Serving Docker image from the Docker Hub by running the following bash command:

```
$ docker pull tensorflow/serving
```

NOTE

Install Docker in your system if you haven't installed it already. You can download Docker from the Docker website.

With the container image now available, you can create a Docker container by running the following command. It will serve your model using TF Serving, open port 8501, and bind-mount the model directory to the container:

```
$ docker run -p 8501:8501 \
       --name=cats_and_dogs_classification \
       --mount type=bind, \
            source=$(pwd)/cats_and_dogs_classif
            target=/models/tf_model \
```

```
        -e MODEL_NAME=cats_and_dogs_classificatio
        -t tensorflow/serving
```

Once the container starts up, you'll see output similar to the following:

```
2024-04-15 01:02:52.825696:
I tensorflow_serving/model_servers/server.cc:77]
Building single TensorFlow model file config:
  model_name: cats_and_dogs_classification model_
   /models/cats_and_dogs_classification
2024-04-15 01:02:52.826118:
I tensorflow_serving/model_servers/server_core.cc
Adding/updating models.
2024-04-15 01:02:52.826137:
I tensorflow_serving/model_servers/server_core.cc
(Re-)adding model: cats_and_dogs_classification
2024-04-15 01:02:53.010338:
I tensorflow_serving/core/basic_manager.cc:740]
Successfully reserved resources to load servable
{name: cats_and_dogs_classification version: 1}
…
2024-04-15 01:02:54.514855:
I tensorflow_serving/model_servers/server.cc:444]
Exporting HTTP/REST API at:localhost:8501 ...
[evhttp_server.cc : 250] NET_LOG: Entering the ev
```

With the model server now running inside the Docker container and port 8501 open for us to communicate with the server, we can request model predictions from the server. Here is an example inference:

```
$ curl -d '{
    "signature_name": "serving_default",
    "instances": [$(base64 -w 0 cat_example.jpg)]
}' -X POST http://localhost:8501/v1/models/cats_a
```

You should see a result similar to ours:

```
{
    "predictions": [[0.15466693, 0.84533307]]
}
```

NOTE

When you want to stop your Docker container again, you can use the following command: `docker stop \`docker ps -q\``.

Conclusion

In this chapter, we reviewed the implementation of a TFX pipeline end to end for a computer vision problem. First, we implemented a custom

component to ingest the image data. We especially focused on the preprocessing steps. After a walkthrough of the setup of every pipeline component, we discussed how to execute the pipeline on two different orchestration platforms: Apache Beam and Google Cloud Vertex Pipelines. As a result, we produced a computer vision model that can decide whether a pet in a photo is a cat or a dog.