Chapter 21. ML Pipelines for Natural Language Processing

In the preceding chapter, we discussed how to create a pipeline for a computer vision production problem, in our case classifying images into categories. In this chapter, we want to demonstrate to you a different type of production problem. But instead of going through all the generic details, we will be focusing on the project-specific aspects.

In this chapter, we are demonstrating the development of an ML model that classifies unstructured text data. In particular, we will be training a transformer model, here a BERT model, to classify the text into categories. As part of the pipeline, we will be spending significant effort on the preprocessing steps of the pipeline. The workflow we present works with any natural language problem, including the latest state-of-the-art large language models (LLMs).

The pipeline will ingest the raw data from an exported CSV file, and we will preprocess the data with TF Transform. After the model is trained, we will combine the preprocessing and the model graph to avoid any training–serving skew.

NOTE

In this chapter, we'll be focusing on novel aspects of the pipeline (e.g., the data ingestion or preprocessing). For more information on how to run Vertex Pipelines, and how to structure your pipeline in general, we highly recommend reviewing the previous chapters.

Our Data

For this example, we are using a public dataset containing 311 call service requests from the City of San Francisco. This is a classic dataset for unstructured text classification and it is available through a number of dataset platforms including Kaggle and Google Cloud public datasets on BigQuery.

We exported the data to the CSV format because not everyone is familiar with Google Cloud BigQuery or has access to it through their cloud provider.

The exported dataset contains 10,000 samples, and the samples contain a status notes column and a category column (showing the JSON structure for better readability):

```
{
  "status_notes": "emailed caller to contact SFM
```

```
    "category": "General Request - 311CUSTOMERSERVI
  }
```

If you want to use the full dataset, we will show you in “Ingestion Component” how to ingest the data directly from Google Cloud BigQuery.

Our Model

Our model will take advantage of the open source version of the pretrained BERT model. The model is provided by Google and Kaggle. BERT, short for Bidirectional Encoder Representations from Transformer, takes three different inputs:

- Input word IDs
- Input masks
- Input type IDs

The BERT model outputs two data structures:

- A pooled vector that represents the whole input data structure
- A sequence vector that represents an embedding for every input token

For our use case, we will be using the pooled vector. If you have limited compute capabilities (e.g., no access to GPUs), we made the BERT layer untrainable. That means no weight updates of the BERT model are happening during the training process. This will save compute resources.

We are training the subsequent dense layers that we added to the top of the pooled layer. To make the training more robust, we added a dropout layer as well. The model is completed with a final softmax layer where we predict the likelihood of the respective categories for the input text.

The whole code setup can be seen in the following code block:

```
bert_layer = hub.KerasLayer(handle=model_url, tra
encoder_inputs = dict(
    input_word_ids=tf.reshape(input_word_ids, (-
    input_mask=tf.reshape(input_mask, (-1, consta
    input_type_ids=tf.reshape(input_type_ids, (-
)
outputs = bert_layer(encoder_inputs)
# Add additional layers depending on your probler
x = tf.keras.layers.Dense(64, activation="relu")
x = tf.keras.layers.Dropout(rate=DROPOUT_RATE)(x
x = tf.keras.layers.Dense(32, activation="relu")
output = tf.keras.layers.Dense(num_labels + 1, ac
model = tf.keras.Model(
    inputs=[
        inputs["input_word_ids"],
        inputs["input_mask"],
        inputs["input_type_ids"]
    ], outputs=output
)
```

Ingestion Component

We mentioned earlier that we are ingesting the data from a CSV file, which we generated for this example. Ingesting CSV files is straightforward, as TFX provides a standard component for it, called CsvExampleGen.

In Chapter 17, we highlighted how to create the ingestion split of the data. The same applies in this example:

```
output = example_gen_pb2.Output(
    split_config=example_gen_pb2.SplitConfig(
        splits=[
            example_gen_pb2.SplitConfig.Split(nar
            example_gen_pb2.SplitConfig.Split(nar
        ]
    )
)
```

With the output split configured, we can set up the CSV ingestion with a single line of code:

```
from tfx.components import CsvExampleGen
...
example_gen = CsvExampleGen(input_base=data_root,
```

If you want to ingest the data directly from BigQuery, you can simply define a query and then swap out the CsvExampleGen with the BigQueryExampleGen:

```
query = """
SELECT DISTINCT status_notes, category
FROM `bigquery-public-data.san_francisco.311_serv
WHERE status_notes IS NOT NULL
AND status_notes <> ""
LIMIT 10000
"""
example_gen =  BigQueryExampleGen(query=query, ou
```

Assuming that you set up your Google Cloud credentials and added the BigQuery User role to your service account used by your Vertex Pipelines, you can ingest the data directly from BigQuery.

Regardless of how you ingest the data, the generated TFRecords will contain a feature with the `status_notes` and a respective `category`.

We will dive into the conversion from our raw text to the model input features in the following section on preprocessing.

Data Preprocessing

Data preprocessing is the most complex aspect of the ML pipeline because BERT, like other transformer models, requires a specific feature input data structure. But this is a perfect task for tools like TF Transform.

For ML models to understand the raw text, the text needs to be converted to numbers. With Transformer-based models, we started to tokenize text as part of the natural language processing, meaning that the text is broken down into its most frequent character components. There are various different methods of tokenization, which produce different token values. For example, the sentence "Futurama characters like to eat anchovies." would be broken down into the following tokens: "Fu, tura, ma, characters, like, to, eat, an, cho, vies, ."

Looking at the generated tokens, you'll notice that frequent words in the English language, such as *like*, *to*, *an*, and *characters*, are not broken down into subtokens, but less-frequent words, such as *anchovies* and *Futurama*, are broken down into subtokens. That way, the language models can operate on a relatively small vocabulary.

And finally, we can convert the subword tokens to IDs that the BERT model can understand:

```
[14763, 21280, 1918, 2650, 1176, 1106, 3940, 112
```

Transformer models require a fixed-input sequence length. But every input note has a different text length, so we will be padding the remaining sequence length to make up the difference. We tell the model which of the tokens are of interest, by generating an `input_mask`. Because BERT was trained with different objectives, it can handle two sequences with the same feature input. For the model to know the difference between the two sequences, the first sequence is noted with 0 values, and the second sequence is noted with the value 1 in the `input_type_ids` mask. But since we are simply passing only one sequence to the input, the input vector will always contain zero values.

NOTE

The preprocessing steps for other transformer models are very similar. The main difference is often only the type of tokenizer and the model-specific vocabulary. Therefore, the shown example can be used with T5, GPT-X, and other models.

To do the text conversion efficiently, we are using TF Transform in combination with TensorFlow Text (TF Text). TF Text is a library that provides TensorFlow Ops for natural language processing operations such as tokenization of text:

```
import tensorflow_text as tf_text
from utils import load_bert_layer
...
do_lower_case = load_bert_layer().resolved_object
vocab_file_path = load_bert_layer().resolved_obje
...
bert_tokenizer = tf_text.BertTokenizer(
    vocab_lookup_table=vocab_file_path,
    token_out_type=tf.int64,
    lower_case=do_lower_case
)
```

The tokenizer `BertTokenizer` is instantiated with the reference to the vocabulary file of the language model, what type of output format we want (integer IDs or token strings), and whether the tokenizer should lowercase the input text before the tokenization.

We can now apply the tokenizer over the model input by calling:

```
tokens = bert_tokenizer.tokenize(text)
```

Since every input text will probably have a different number of tokens, we need to truncate token lists that are longer than our allowed sequence length from our transformer model, pad all lists that are shorter than our expected token length, and prepend and append control tokens around the input text for our model to know where the text starts and ends. We are accomplishing all of these tasks with the following lines of code:

```
cls_id = tf.constant(101, dtype=tf.int64)
sep_id = tf.constant(102, dtype=tf.int64)
pad_id = tf.constant(0, dtype=tf.int64)
tokens = tokens.merge_dims(1, 2)[:, :sequence_len
start_tokens = tf.fill([tf.shape(text)[0], 1], cl
end_tokens = tf.fill([tf.shape(text)[0], 1], sep_
tokens = tokens[:, :sequence_length - 2]
tokens = tf.concat([start_tokens, tokens, end_tok
tokens = tokens[:, :sequence_length]
tokens = tokens.to_tensor(default_value=pad_id)
pad = sequence_length - tf.shape(tokens)[1]
tokens = tf.pad(tokens, [[0, 0], [0, pad]], const
input_token_ids =  tf.reshape(tokens, [-1, sequen
```

Once we have converted our input texts to token IDs, we can easily generate the input masks and input type IDs that are required for the BERT embedding generation:

```
input_mask = tf.cast(input_word_ids > 0, tf.int64
input_mask = tf.reshape(input_mask, [-1, constant
zeros_dims = tf.stack(tf.shape(input_mask))
input_type_ids = tf.fill(zeros_dims, 0)
input_type_ids = tf.cast(input_type_ids, tf.int64
```

The conversion of our labels works the same way we did it in Chapter 17. We will be using TF Transform's `compute_and_apply_vocabulary` function and applying it across the label column of our data:

```
indexed_vocab_label = tft.compute_and_apply_vocab
    label_tensor,
    top_k=constants.VOCAB_SIZE,
    num_oov_buckets=constants.OOV_SIZE,
    default_value=constants.VOCAB_DEFAULT_INDEX,
    vocab_filename=constants.LABEL_VOCAB_FILE_NAM
)
```

Now that we have converted our training labels into integers, we are done with the preprocessing setup. We'll wrap everything up in a `preprocessing_fn` function (the expected function name) and store it in our preprocessing module called *preprocessing.py* (you can choose your module name).

NOTE

We have written an in-depth article on combining TF Transform and TF Text for BERT preprocessing. If you are interested in a more in-depth review that goes beyond the example introduction, we highly recommend the two-part series (part 1, part 2).

Putting the Pipeline Together

The remainder of the pipeline setup is identical to our previous example. First, we create each of our components, assemble a list of the instantiated component objects, and then create our pipeline object. See Chapter 18 for specific details.

Executing the Pipeline

Running our pipeline is exactly the same as we discussed in Chapter 18. If you're using Apache Beam, you can simply run the following line of code and the pipeline will be executed wherever you run the line of code:

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

If you are running your pipeline on Google Cloud Vertex Pipelines, you will need to build your container image for your pipeline following the naming pattern: `gcr.io/$PROJECT_ID/$IMAGE_NAME:$IMAGE_TAG`.

Here is an example Dockerfile that can be used for the example project:

```
FROM tensorflow/tfx:1.14.0
RUN pip install tensorflow-text==2.13.0
WORKDIR /pipeline
COPY ./components ./components
ENV PYTHONPATH="/pipeline:${PYTHONPATH}"
```

Unfortunately, the default TFX image doesn't contain the TF Text library. Therefore, we'll need to build a custom image. Check Chapter 20 for more details on how to build custom pipeline Docker images.

Once you have created your custom pipeline image, you can convert your pipeline definition to the Vertex pipeline description by executing the pipeline runner for Vertex Pipelines:

```
cpu_container_image_uri = "gcr.io/$PROJECT_ID/$I
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=
...
```

Once the pipeline definition is written out, you submit the pipeline definition to Vertex Pipelines with `job.submit`, as we discussed it in Chapters 8 and 20:

```
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_ACCOUN
)
```

By submitting the pipeline to Vertex Pipelines, it will be executed immediately and you can follow the pipeline progress in the Vertex Pipelines user interface.

Model Deployment with Google Cloud Vertex

Once you have executed the ML model, trained it, and done in-depth validation, it is time to deploy the model. In the preceding chapter, we focused on local deployment with TF Serving. In this chapter, we want to focus on a more scalable deployment solution: using Google Cloud Vertex Model Endpoints.

When the pipeline completes its run successfully, you can deploy the model through a three-step workflow. First, register the model with the Vertex Model Registry. Then, create the model endpoint if it doesn't already exist. Finally, deploy the registered model on the available endpoint. With the last step, the endpoint will be available to accept model requests and provide predictions for your applications.

Registering Your ML Model

Your first step to deploy your ML model is to register the model and its new version with the Vertex Model Registry. You can register your model through the Vertex user interface, through a number of Vertex SDKs (e.g., Python, Java), or through Google Cloud's command-line interface. In the following example, we use Google Cloud's CLI:

```
export PROJECT=<YOUR_PROJECT_NAME>
export REGION=us-central1
export MODEL_NAME=311-call-classification
export IMAGE_URI=us-docker.pkg.dev/vertex-ai/pre
export PATH_TO_MODEL= \
gs://<BUCKET_NAME>/<PIPELINE_NAME>/<RUN_ID>/\
<PIPELINE_NAME>-<TIMESTAMP>/Pusher_-<COMPONENT_I
```

The `PATH_TO_MODEL` is the Google Cloud Storage path where the pipeline Pusher component will ship the trained and validated model.

Next, we need to register the model with the model registry. We can perform this step via the following CLI command and the Vertex SDK. The model registration step connects the model with an underlying container that contains all the dependency for inference tasks. In our example, we are using a Docker container with all the TensorFlow dependencies.

If you are using the CLI, you can use the following command:

```
gcloud ai models upload \
  --region=$REGION \
  --display-name=$MODEL_NAME \
  --container-image-uri=$IMAGE_URI \
  --artifact-uri=$PATH_TO_MODEL
```

When you execute the command, Google will kick off a task to register the model. The command will return the operation ID and the final status:

```
Using endpoint [https://us-central1-aiplatform.go
Waiting for operation [1013299264639467520]...done
```

You can list all available models in the registry with the following `list` command:

```
$ gcloud ai models list  --region=$REGION
Using endpoint [https://us-central1-aiplatform.go
MODEL_ID             DISPLAY_NAME
4976724978360647680  311-call-classification
```

The model ID will become handy in a future step.

If you prefer to use the Vertex Python SDK, the following code will perform the same model registration:

```
from google.cloud import aiplatform

def upload_model(project_id, region, model_name,
    """Uploads a model to Vertex AI."""
    client_options = {"api_endpoint": f"{region}
    # Initialize Vertex AI client
    aiplatform.init(project=project_id,
        location=region,
        client_options=client_options)
    model = aiplatform.Model.upload(
        display_name=model_name,
        artifact_uri=artifact_uri,
        serving_container_image_uri=image_uri,
    )
    model.wait()
    print(f"Model uploaded: {model.resource_name

# Set your values for the following variables
project_id = "your-project-id"
region = "your-region"
model_name = "your-model-name"
image_uri = "your-image-uri"
artifact_uri = "your-path-to-model"
upload_model(project_id, region, model_name, imag
```

Creating a New Model Endpoint

For now, we need to create a model endpoint where we can deploy the model to. If you already have an endpoint created, you can skip this step:

```
gcloud ai endpoints create \
  --project=$PROJECT \
    --region=$REGION \
    --display-name=311-call-classifications
```

The command will return output similar to the following:

```
Using endpoint [https://us-central1-aiplatform.go
Waiting for operation [4713015944891334656]...don
Created Vertex AI endpoint: projects/49811700686
```

The equivalent Python code is the following:

```
from google.cloud import aiplatform

def create_endpoint(project_id, region, display_n
    """Creates a Vertex AI endpoint."""
    client_options = {"api_endpoint": f"{region}-
    # Initialize Vertex AI client
    aiplatform.init(project=project_id,
       location=region,
       client_options=client_options)
    endpoint = aiplatform.Endpoint.create(display
    print(f"Endpoint created: {endpoint.resource_

# Set your values for the following variables
project_id = "your-project-id"
region = "your-region"
```

```
display_name = "311-call-classifications"
create_endpoint(project_id, region, display_name
```

Deploying Your ML Model

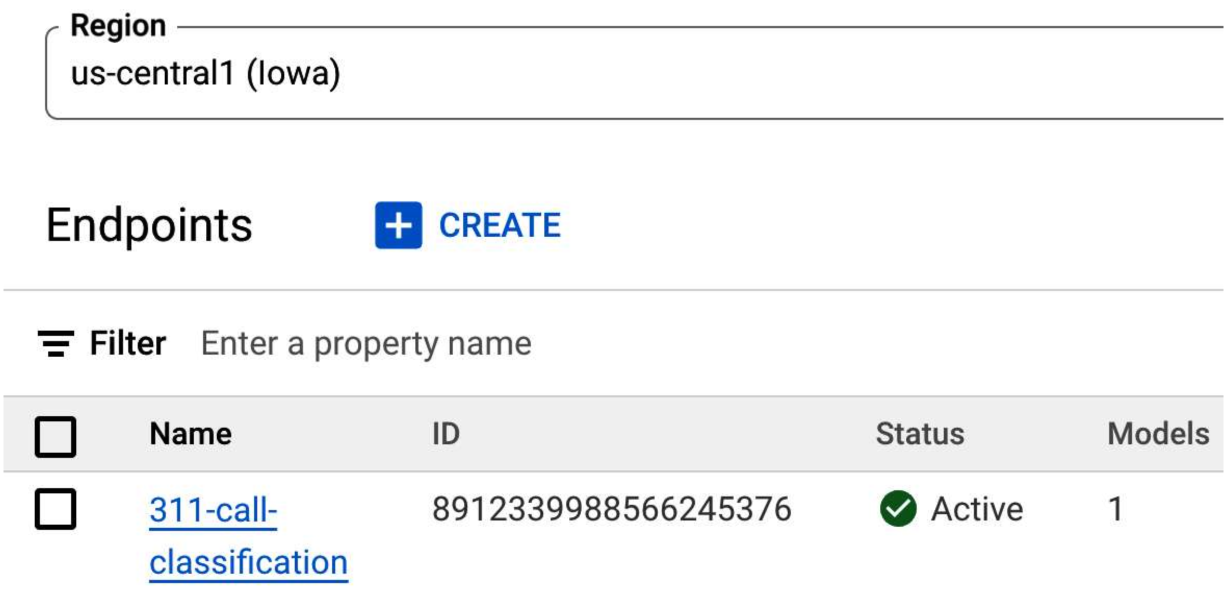
Once we have an endpoint instantiated, we can now deploy the registered model to the new endpoint. In the following command, we deploy the model with the ID `4976724978360647680` to the endpoint with the ID `7662248044343066624`:

```
gcloud ai endpoints deploy-model 766224804434306
  --project=$PROJECT \
  --region=$REGION \
  --model=4976724978360647680 \
  --display-name=311-call-classification-model
```

NOTE

The model deployment offers a number of configuration options that are constantly extended. We highly recommend the Google documentation for details around accelerator configuration, scaling options, and available machine instance types.

Once the deployment is completed, you will see the Active checkmark in the Vertex Online Prediction User Interface under the given endpoint, as shown in Figure 21-1.


Region
us-central1 (Iowa)

Endpoints

	Name	ID	Status	Models
☐	311-call-classification	8912339988566245376	✓ Active	1

Figure 21-1. List of Vertex endpoints in Google Cloud

If you prefer the Python SDK option, you can achieve the same result with the following code:

```
from google.cloud import aiplatform

def deploy_model_with_id(
    project_id, region, endpoint_id, model_id, de
    machine_type="n1-standard-4", min_replica_cou
):
    """Deploys a model with specific ID to a spec
    client_options = {"api_endpoint": f"{region}-
    # Initialize Vertex AI client
    aiplatform.init(project=project_id,
        location=region,
        client_options=client_options)
    endpoint = aiplatform.Endpoint(endpoint_name=
    model = aiplatform.Model(model_name=model_id)

    # Define deployment configuration
    traffic_percentage = 100  # Initial traffic p
    machine_type = machine_type
    min_replica_count = min_replica_count
    max_replica_count = max_replica_count
    # Deploy the model
    endpoint.deploy(
        model=model,
        deployed_model_display_name=deployed_mode
        traffic_percentage=traffic_percentage,
        machine_type=machine_type,
        min_replica_count=min_replica_count,
        max_replica_count=max_replica_count,
    )
    print(f"Model deployed to endpoint {endpoint_

# Set your values for the following variables
project_id = "your-project-id"
region = "your-region"
endpoint_id = "7662248044343066624"  # Replace wi
model_id = "4976724978360647680"  # Replace with
deployed_model_display_name = "311-call-classifi
deploy_model_with_id(
      project_id,
      region, endpoint_id,
      model_id,
      deployed_model_display_name)
```

Requesting Predictions from the Deployed Model

Once the model is deployed to your endpoint, you can request predictions from your applications. Google Cloud provides a number of SDKs for Python, Java, or GoLang. In the following example, we want to stay language agnostic and we request a prediction through Google Cloud’s CLI tool.

First, create a JSON file with the request inputs. The following snippet shows an example format (we stored the file as *requests.json*):

```
  {
     "instances":[
        {
           "text":["Garbage pick up required"]
        }
     ]
  }
```

With the requests now in place, we can request the predictions via `gcloud`. To request predictions from an endpoint (in our case, endpoint `7662248044343066624`), we run this command:

```
$ export ENDPOINT_ID=7662248044343066624
$ gcloud ai endpoints predict $ENDPOINT_ID \
  --region=$REGION \
  --json-request=requests.json
```

The command line will then return the prediction results as follows:

```
Using endpoint [https://us-central1-prediction-ai
[[0.154666945, 0.169343904, 0.0821105, 0.08182372
0.10572128, 0.0635185838, 0.0764537, 0.082392193
0.0797150582, 0.0443297, 0.0599244162]]
```

Using the Python SDK, the inference code looks as follows:

```
import json
from google.cloud import aiplatform
def predict_on_endpoint(project_id, region, endpo
    """Sends prediction requests to a given endpo
    client_options = {"api_endpoint": f"{region}-
    # Initialize Vertex AI client
    aiplatform.init(project=project_id,
            location=region,
            client_options=client_options)
        endpoint = aiplatform.Endpoint(endpoint_name=
        response = endpoint.predict(instances=instanc
        print("Prediction results:")
        for prediction in response.predictions:
            print(prediction)
# Set your values for the following variables
project_id = "your-project-id"
region = "your-region"
endpoint_id = "7662248044343066624"  # Replace wi
# Load instances from a JSON file
with open("requests.json", "r") as f:
    instances = json.load(f)
predict_on_endpoint(project_id, region, endpoint_
```

Cleaning Up Your Deployed Model

If you want to control your costs, we highly recommend deleting idle endpoints. The following commands let you clean up your project by first removing the model from the endpoint and then deleting the endpoint itself:

```
$ gcloud ai endpoints undeploy-model 766224804434
  --project=$PROJECT \
  --region=$REGION \
  --deployed-model-id=4976724978360647680
```

```
$ gcloud ai endpoints delete 7662248044343066624
```

Cleaning up your endpoints using the Python SDK is possible with the following Python code:

```
from google.cloud import aiplatform

def undeploy_and_delete(project_id, region, endpo
    """Undeploys a model from an endpoint and the
    client_options = {"api_endpoint": f"{region}-
    # Initialize Vertex AI client
    aiplatform.init(project=project_id,
       location=region,
       client_options=client_options)
    endpoint = aiplatform.Endpoint(endpoint_name=

    # Undeploy the model
    endpoint.undeploy(deployed_model_id=deployed_
    print(f"Model {deployed_model_id} undeployed
    # Delete the endpoint
    endpoint.delete()
    print(f"Endpoint {endpoint_id} deleted.")

# Set your values for the following variables
project_id = "your-project-id"
region = "your-region"
```

```
endpoint_id = "7662248044343066624"  # Replace wi
deployed_model_id = "4976724978360647680"  # Repl
undeploy_and_delete(project_id, region, endpoint_
```

Conclusion

In this chapter, we demonstrated how a pipeline can be built for a natural language problem like text classification with a transformer model like BERT. The steps apply to all natural language problems, only with minor updates to the preprocessing steps.

Over the previous two chapters, we introduced two basic pipelines for very common ML problems. But nothing says those two pipelines couldn't be combined. This will be more and more important as multimodal models will be applied to more applications.