Ungraded Lab: Developing Custom TFX Components

Tensorflow Extended (TFX) provides ready made components for typical steps in a machine learning workflow. Other courses in this specialization focus on specific components and in this lab, you will learn how to make your own. This will be useful in case your project has specific needs that fall outside the standard TFX components. It will make your pipelines more flexible while still leveraging the experiment tracking and orchestration that TFX provides. In particular, you will:

- build a custom component using Python functions
- build a custom component by reusing an existing TFX component
- run a TFX pipeline locally using a built-in pipeline orchestrator

To demonstrate, you will run the pipeline used in this official tutorial then modify it to have a custom component. Some of the discussions here are also taken from that tutorial to explain the motivation and point to additional resources.

Let's begin!

Note: If you haven't taken other courses in this specialization and it's the first time you're using TFX, please see Understanding TFX Pipelines to get an overview of important concepts.

Set Up

We first need to install the TFX Python package and download the dataset which we will use for our model.

Upgrade Pip

To avoid upgrading Pip in a system when running locally, check to make sure that we are running in Colab. Local systems can of course be upgraded separately.

```
try:
  import colab
  !pip install --upgrade pip
except:
  pass
```

Install TFX

```
!pip install -U tfx
```

Note: *Please do not proceed to the next steps without restarting the Runtime after installing* `tfx`*. You can do that by either pressing the* `Restart Runtime` *button at the end of the cell output above, or going to the* `Runtime` *button at the Colab toolbar above and selecting* `Restart Runtime`.

Check the TensorFlow and TFX versions.

```
import tensorflow as tf
print('TensorFlow version: {}'.format(tf.__version__))
from tfx import v1 as tfx
print('TFX version: {}'.format(tfx.__version__))
```

```
TensorFlow version: 2.5.1
TFX version: 1.2.0
```

Set up variables

There are some variables used to define a pipeline. You can customize these variables as you want. By default all output from the pipeline will be generated under the current directory.

```
import os
from absl import logging

# Pipeline label
PIPELINE_NAME = "penguin-simple"

# Output directory to store artifacts generated from the pipeline.
PIPELINE_ROOT = os.path.join('pipelines', PIPELINE_NAME)

# Path to a SQLite DB file to use as an MLMD storage.
METADATA_PATH = os.path.join('metadata', PIPELINE_NAME, 'metadata.db')

# Output directory where created models from the pipeline will be exported.
SERVING_MODEL_DIR = os.path.join('serving_model', PIPELINE_NAME)

# Set default logging level.
logging.set_verbosity(logging.INFO)
```

Prepare example data

The dataset you will use is the Palmer Penguins dataset.

There are four numeric features in this dataset:

- culmen_length_mm
- culmen_depth_mm
- flipper_length_mm

- flipper_length_mm
- body_mass_g

All features were already normalized to have range [0,1]. You will build a classification model which predicts the `species` of penguins. The code below creates a directory and copies the dataset to it. After running it, you should see the dataset in the Colab file explorer under the `data` folder.

```
import urllib.request

# Create directory
DATA_ROOT = 'data'
!mkdir {DATA_ROOT}

# Copy dataset to directory
_data_url = 'https://raw.githubusercontent.com/tensorflow/tfx/master/tfx/examples/penguin/data/labelled/penguins_processed.csv'
_data_filepath = os.path.join(DATA_ROOT, "data.csv")
urllib.request.urlretrieve(_data_url, _data_filepath)
```

```
('data/data.csv', <http.client.HTTPMessage at 0x7f637304e510>)
```

Running the pipeline using standard components

The pipeline will consist of three essential TFX components and the graph will look like this:

```
ExampleGen -> Trainer -> Pusher
```

The pipeline includes the most minimal ML workflow which is importing data (ExampleGen), training a model (Trainer) and exporting the trained model (Pusher).

Model training code

You will first define the trainer module so the `Trainer` component can build the model and train it.

```
_trainer_module_file = 'penguin_trainer.py'
```

```
%%writefile {_trainer_module_file}

from typing import List
from absl import logging
import tensorflow as tf
from tensorflow import keras
from tensorflow_transform.tf_metadata import schema_utils

from tfx import v1 as tfx
from tfx_bsl.public import tfxio
from tensorflow_metadata.proto.v0 import schema_pb2

_FEATURE_KEYS = [
    'culmen_length_mm', 'culmen_depth_mm', 'flipper_length_mm', 'body_mass_g'
]
_LABEL_KEY = 'species'

_TRAIN_BATCH_SIZE = 20
_EVAL_BATCH_SIZE = 10
```

```python
def _build_keras_model() -> tf.keras.Model:
  """Creates a DNN Keras model for classifying penguin data.

  Returns:
    A Keras Model.
  """
  # The model below is built with Functional API, please refer to
  # https://www.tensorflow.org/guide/keras/overview for all API options.
  inputs = [keras.layers.Input(shape=(1,), name=f) for f in _FEATURE_KEYS]
  d = keras.layers.concatenate(inputs)
  for _ in range(2):
    d = keras.layers.Dense(8, activation='relu')(d)
  outputs = keras.layers.Dense(3)(d)

  model = keras.Model(inputs=inputs, outputs=outputs)
  model.compile(
      optimizer=keras.optimizers.Adam(1e-2),
      loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
      metrics=[keras.metrics.SparseCategoricalAccuracy()])

  model.summary(print_fn=logging.info)
  return model

```