Ungraded Lab: Hyperparameter tuning and model training with TFX

In this lab, you will be again doing hyperparameter tuning but this time, it will be within a [Tensorflow Extended (TFX)](https://www.tensorflow.org/tfx) pipeline.

We have already introduced some TFX components in Course 2 of this specialization related to data ingestion, validation, and transformation. In this notebook, you will get to work with two more which are related to model development and training: *Tuner* and *Trainer*.

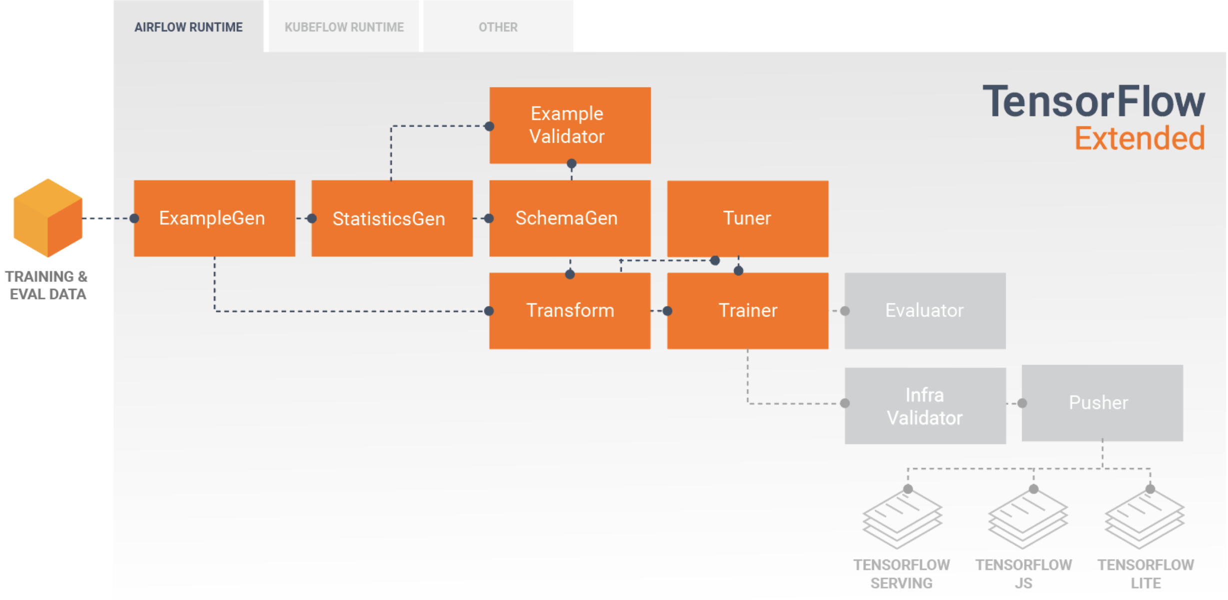

image source: https://www.tensorflow.org/tfx/guide

- The *Tuner* utilizes the Keras Tuner API under the hood to tune your model's hyperparameters.
- You can get the best set of hyperparameters from the Tuner component and feed it into the *Trainer* component to optimize your model for training.

You will again be working with the FashionMNIST dataset and will feed it though the TFX pipeline up to the Trainer component.You will quickly review the earlier components from Course 2, then focus on the two new components introduced.

Let's begin!

Setup

Install TFX

You will first install TFX, a framework for developing end-to-end machine learning pipelines.

```
!pip install -U pip
!pip install tfx==1.3.0
!pip install apache-beam==2.39.0

# These are downgraded to work with the packages used by TFX 1.3
# Please do not delete because it will cause import errors in the next cell
!pip install --upgrade tensorflow-estimator==2.6.0
!pip install --upgrade keras==2.6.0
```

Note: In Google Colab, you need to restart the runtime at this point to finalize updating the packages you just installed. You can do so by clicking the `Restart Runtime` at the end of the output cell above (after installation), or by selecting `Runtime > Restart Runtime` in the Menu bar. ***Please do not proceed to the next section without restarting.*** *You can also ignore the errors about version incompatibility of some of the bundled packages because we won't be using those in this notebook.*

Imports

You will then import the packages you will need for this exercise.

```
import tensorflow as tf
from tensorflow import keras
import tensorflow_datasets as tfds

import os
import pprint

from tfx.components import ImportExampleGen
from tfx.components import ExampleValidator
from tfx.components import SchemaGen
from tfx.components import StatisticsGen
from tfx.components import Transform
from tfx.components import Tuner
from tfx.components import Trainer

from tfx.proto import example_gen_pb2
from tfx.orchestration.experimental.interactive.interactive_context import Inte
```

Download and prepare the dataset

As mentioned earlier, you will be using the Fashion MNIST dataset just like in the previous lab. This will allow you to compare the similarities and differences when using Keras Tuner as a standalone library and within an ML pipeline.

You will first need to setup the directories that you will use to store the dataset, as well as the pipeline artifacts and metadata store.

```
# Location of the pipeline metadata store
_pipeline_root = './pipeline/'

# Directory of the raw data files
_data_root = './data/fmnist'

# Temporary directory
tempdir = './tempdir'
```

```
# Create the dataset directory
!mkdir -p {_data_root}

# Create the TFX pipeline files directory
!mkdir {_pipeline_root}
```

You will now download FashionMNIST from [Tensorflow Datasets](). The `with_info` flag will be set to `True` so you can display information about the dataset in the next cell (i.e. using `ds_info`).

```
# Download the dataset
ds, ds_info = tfds.load('fashion_mnist', data_dir=tempdir, with_info=True)
```

```
# Display info about the dataset
print(ds_info)
```

You can review the downloaded files with the code below. For this lab, you will be using the *train* TFRecord so you will need to take note of its filename. You will not use the *test* TFRecord in this lab.

```
# Define the location of the train tfrecord downloaded via TFDS
tfds_data_path = f'{tempdir}/{ds_info.name}/{ds_info.version}'

# Display contents of the TFDS data directory
os.listdir(tfds_data_path)
```

You will then copy the train split from the downloaded data so it can be consumed by the ExampleGen component in the next step. This component requires that your files are in a directory without extra files (e.g. JSONs and TXT files).

```
# Define the train tfrecord filename
train_filename = 'fashion_mnist-train.tfrecord-00000-of-00001'
```

```

# Copy the train tfrecord into the data root folder
!cp {tfds_data_path}/{train_filename} {_data_root}
```

TFX Pipeline

With the setup complete, you can now proceed to creating the pipeline.

Initialize the Interactive Context

You will start by initializing the [InteractiveContext](https://github.com/tensorflow/tfx/blob/master/tfx/orchestration/experimental/interactive/interactive_context.py) so you can run the components within this Colab environment. You can safely ignore the warning because you will just be using a local SQLite file for the metadata store.

```
# Initialize the InteractiveContext
context = InteractiveContext(pipeline_root=_pipeline_root)
```

ExampleGen

You will start the pipeline by ingesting the TFRecord you set aside. The ImportExampleGen consumes TFRecords and you can specify splits as shown below. For this exercise, you will split the train tfrecord to use 80% for the train set, and the remaining 20% as eval/validation set.

```
# Specify 80/20 split for the train and eval set
output = example_gen_pb2.Output(
    split_config=example_gen_pb2.SplitConfig(splits=[
        example_gen_pb2.SplitConfig.Split(name='train', hash_buckets=8),
        example_gen_pb2.SplitConfig.Split(name='eval', hash_buckets=2),
    ]))

# Ingest the data through ExampleGen
example_gen = ImportExampleGen(input_base=_data_root, output_config=output)

# Run the component
context.run(example_gen)
```

```
# Print split names and URI
artifact = example_gen.outputs['examples'].get()[0]
print(artifact.split_names, artifact.uri)
```

StatisticsGen

Next, you will compute the statistics of the dataset with the StatisticsGen component.

```
# Run StatisticsGen
```

```
statistics_gen = StatisticsGen(
    examples=example_gen.outputs['examples'])

context.run(statistics_gen)
```

SchemaGen

You can then infer the dataset schema with [SchemaGen](). This will be used to validate incoming data to ensure that it is formatted correctly.

```
# Run SchemaGen
schema_gen = SchemaGen(
     statistics=statistics_gen.outputs['statistics'], infer_feature_shape=True
context.run(schema_gen)
```

```
# Visualize the results
context.show(schema_gen.outputs['schema'])
```

ExampleValidator

You can assume that the dataset is clean since we downloaded it from TFDS. But just to review, let's run it through [ExampleValidator]() to detect if there are anomalies within the dataset.

```
# Run ExampleValidator
example_validator = ExampleValidator(
    statistics=statistics_gen.outputs['statistics'],
    schema=schema_gen.outputs['schema'])
context.run(example_validator)
```

```
# Visualize the results. There should be no anomalies.
context.show(example_validator.outputs['anomalies'])
```

Transform

Let's now use the [Transform]() component to scale the image pixels and convert the data types to float. You will first define the transform module containing these operations before you run the component.

```
_transform_module_file = 'fmnist_transform.py'
```

```
%%writefile {_transform_module_file}

import tensorflow as tf
import tensorflow_transform as tft

```

```
# Keys
_LABEL_KEY = 'label'
_IMAGE_KEY = 'image'


def _transformed_name(key):
    return key + '_xf'

def _image_parser(image_str):
    '''converts the images to a float tensor'''
    image = tf.image.decode_image(image_str, channels=1)
    image = tf.reshape(image, (28, 28, 1))
    image = tf.cast(image, tf.float32)
    return image


def _label_parser(label_id):
    '''converts the labels to a float tensor'''
    label = tf.cast(label_id, tf.float32)
    return label


def preprocessing_fn(inputs):
    """tf.transform's callback function for preprocessing inputs.
    Args:
        inputs: map from feature keys to raw not-yet-transformed features.
    Returns:
        Map from string feature key to transformed feature operations.
    """

    # Convert the raw image and labels to a float array
    with tf.device("/cpu:0"):
        outputs = {
            _transformed_name(_IMAGE_KEY):
                tf.map_fn(
                    _image_parser,
                    tf.squeeze(inputs[_IMAGE_KEY], axis=1),
                    dtype=tf.float32),
            _transformed_name(_LABEL_KEY):
                tf.map_fn(
                    _label_parser,
                    inputs[_LABEL_KEY],
                    dtype=tf.float32)
        }

    # scale the pixels from 0 to 1
    outputs[_transformed_name(_IMAGE_KEY)] = tft.scale_to_0_1(outputs[_transfor

    return outputs
```

You will run the component by passing in the examples, schema, and transform module file.

Note: You can safely ignore the warnings and `udf_utils` related errors.

```
# Ignore TF warning messages
tf.get_logger().setLevel('ERROR')

# Setup the Transform component
transform = Transform(
    examples=example_gen.outputs['examples'],
    schema=schema_gen.outputs['schema'],
    module_file=os.path.abspath(_transform_module_file))

# Run the component
context.run(transform)
```

Tuner

As the name suggests, the Tuner component tunes the hyperparameters of your model. To use this, you will need to provide a *tuner module file* which contains a `tuner_fn()` function. In this function, you will mostly do the same steps as you did in the previous ungraded lab but with some key differences in handling the dataset.

The Transform component earlier saved the transformed examples as TFRecords compressed in `.gz` format and you will need to load that into memory. Once loaded, you will need to create batches of features and labels so you can finally use it for hypertuning. This process is modularized in the `_input_fn()` below.

Going back, the `tuner_fn()` function will return a `TunerFnResult` namedtuple containing your `tuner` object and a set of arguments to pass to `tuner.search()` method. You will see these in action in the following cells. When reviewing the module file, we recommend viewing the `tuner_fn()` first before looking at the other auxiliary functions.

```
# Declare name of module file
_tuner_module_file = 'tuner.py'
```

```
%%writefile {_tuner_module_file}

# Define imports
from kerastuner.engine import base_tuner
import kerastuner as kt
from tensorflow import keras
from typing import NamedTuple, Dict, Text, Any, List
from tfx.components.trainer.fn_args_utils import FnArgs, DataAccessor
import tensorflow as tf
import tensorflow_transform as tft

# Declare namedtuple field names
TunerFnResult = NamedTuple('TunerFnResult', [('tuner', base_tuner.BaseTuner),
                                             ('fit_kwargs', Dict[Text, Any])])

# Label key
```

```
LABEL_KEY = 'label_xf'

# Callback for the search strategy
stop_early = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5)


def _gzip_reader_fn(filenames):
  '''Load compressed dataset

  Args:
    filenames - filenames of TFRecords to load

  Returns:
    TFRecordDataset loaded from the filenames
  '''

  # Load the dataset. Specify the compression type since it is saved as `.gz`
  return tf.data.TFRecordDataset(filenames, compression_type='GZIP')


def _input_fn(file_pattern,
              tf_transform_output,
              num_epochs=None,
              batch_size=32) -> tf.data.Dataset:
  '''Create batches of features and labels from TF Records

  Args:
    file_pattern - List of files or patterns of file paths containing Example r
    tf_transform_output - transform output graph
    num_epochs - Integer specifying the number of times to read through the dat
            If None, cycles through the dataset forever.
    batch_size - An int representing the number of records to combine in a sing

  Returns:
    A dataset of dict elements, (or a tuple of dict elements and label).
    Each dict maps feature keys to Tensor or SparseTensor objects.
  '''

  # Get feature specification based on transform output
  transformed_feature_spec = (
      tf_transform_output.transformed_feature_spec().copy())

  # Create batches of features and labels
  dataset = tf.data.experimental.make_batched_features_dataset(
      file_pattern=file_pattern,
      batch_size=batch_size,
      features=transformed_feature_spec,
      reader=_gzip_reader_fn,
      num_epochs=num_epochs,
      label_key=LABEL_KEY)

  return dataset


def model_builder(hp):
```

```
  '''
  Builds the model and sets up the hyperparameters to tune.

  Args:
    hp - Keras tuner object

  Returns:
    model with hyperparameters to tune
  '''

  # Initialize the Sequential API and start stacking the layers
  model = keras.Sequential()
  model.add(keras.layers.Flatten(input_shape=(28, 28, 1)))

  # Tune the number of units in the first Dense layer
  # Choose an optimal value between 32-512
  hp_units = hp.Int('units', min_value=32, max_value=512, step=32)
  model.add(keras.layers.Dense(units=hp_units, activation='relu', name='dense_1

  # Add next layers
  model.add(keras.layers.Dropout(0.2))
  model.add(keras.layers.Dense(10, activation='softmax'))

  # Tune the learning rate for the optimizer
  # Choose an optimal value from 0.01, 0.001, or 0.0001
  hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])

  model.compile(optimizer=keras.optimizers.Adam(learning_rate=hp_learning_rate)
                loss=keras.losses.SparseCategoricalCrossentropy(),
                metrics=['accuracy'])

  return model

def tuner_fn(fn_args: FnArgs) -> TunerFnResult:
  """Build the tuner using the KerasTuner API.
  Args:
    fn_args: Holds args as name/value pairs.

      - working_dir: working dir for tuning.
      - train_files: List of file paths containing training tf.Example data.
      - eval_files: List of file paths containing eval tf.Example data.
      - train_steps: number of train steps.
      - eval_steps: number of eval steps.
      - schema_path: optional schema of the input data.
      - transform_graph_path: optional transform graph produced by TFT.

  Returns:
    A namedtuple contains the following:
      - tuner: A BaseTuner that will be used for tuning.
      - fit_kwargs: Args to pass to tuner's run_trial function for fitting the
                    model , e.g., the training and validation dataset. Required
                    args depend on the above tuner's implementation.
  """

  # Define tuner search strategy
```

```
    tuner = kt.Hyperband(model_builder,
                       objective='val_accuracy',
                       max_epochs=10,
                       factor=3,
                       directory=fn_args.working_dir,
                       project_name='kt_hyperband')

    # Load transform output
    tf_transform_output = tft.TFTransformOutput(fn_args.transform_graph_path)

    # Use _input_fn() to extract input features and labels from the train and val
    train_set = _input_fn(fn_args.train_files[0], tf_transform_output)
    val_set = _input_fn(fn_args.eval_files[0], tf_transform_output)


    return TunerFnResult(
        tuner=tuner,
        fit_kwargs={
            "callbacks":[stop_early],
            'x': train_set,
            'validation_data': val_set,
            'steps_per_epoch': fn_args.train_steps,
            'validation_steps': fn_args.eval_steps
        }
    )
```

With the module defined, you can now setup the Tuner component. You can see the description of each argument [here](#).

Notice that we passed a `num_steps` argument to the train and eval args and this was used in the `steps_per_epoch` and `validation_steps` arguments in the tuner module above. This can be useful if you don't want to go through the entire dataset when tuning. For example, if you have 10GB of training data, it would be incredibly time consuming if you will iterate through it entirely just for one epoch and one set of hyperparameters. You can set the number of steps so your program will only go through a fraction of the dataset.

You can compute for the total number of steps in one epoch by: `number of examples / batch size`. For this particular example, we have `48000 examples / 32 (default size)` which equals `1500` steps per epoch for the train set (compute val steps from 12000 examples). Since you passed `500` in the `num_steps` of the train args, this means that some examples will be skipped. This will likely result in lower accuracy readings but will save time in doing the hypertuning. Try modifying this value later and see if you arrive at the same set of hyperparameters.

```
from tfx.proto import trainer_pb2

# Setup the Tuner component
tuner = Tuner(
    module_file=_tuner_module_file,
```

```
    examples=transform.outputs['transformed_examples'],
    transform_graph=transform.outputs['transform_graph'],
    schema=schema_gen.outputs['schema'],
    train_args=trainer_pb2.TrainArgs(splits=['train'], num_steps=500),
    eval_args=trainer_pb2.EvalArgs(splits=['eval'], num_steps=100)
    )
```

```
# Run the component. This will take around 10 minutes to run.
# When done, it will summarize the results and show the 10 best trials.
context.run(tuner, enable_cache=False)
```

Trainer

Like the Tuner component, the [Trainer](https://www.tensorflow.org/tfx/guide/trainer) component also requires a module file to setup the training process. It will look for a `run_fn()` function that defines and trains the model. The steps will look similar to the tuner module file:

- Define the model - You can get the results of the Tuner component through the `fn_args.hyperparameters` argument. You will see it passed into the `model_builder()` function below. If you didn't run `Tuner`, then you can just explicitly define the number of hidden units and learning rate.
- Load the train and validation sets - You have done this in the Tuner component. For this module, you will pass in a `num_epochs` value (10) to indicate how many batches will be prepared. You can opt not to do this and pass a `num_steps` value as before.
- Setup and train the model - This will look very familiar if you're already used to the Keras Models Training API. You can pass in callbacks like the TensorBoard callback so you can visualize the results later.
- Save the model - This is needed so you can analyze and serve your model. You will get to do this in later parts of the course and specialization.

```
# Declare trainer module file
_trainer_module_file = 'trainer.py'
```

```
%%writefile {_trainer_module_file}

from tensorflow import keras
from typing import NamedTuple, Dict, Text, Any, List
from tfx.components.trainer.fn_args_utils import FnArgs, DataAccessor
import tensorflow as tf
import tensorflow_transform as tft

# Define the label key
LABEL_KEY = 'label_xf'

def _gzip_reader_fn(filenames):
```

```
  '''Load compressed dataset

  Args:
    filenames - filenames of TFRecords to load

  Returns:
    TFRecordDataset loaded from the filenames
  '''

  # Load the dataset. Specify the compression type since it is saved as `.gz`
  return tf.data.TFRecordDataset(filenames, compression_type='GZIP')


def _input_fn(file_pattern,
              tf_transform_output,
              num_epochs=None,
              batch_size=32) -> tf.data.Dataset:
  '''Create batches of features and labels from TF Records

  Args:
    file_pattern - List of files or patterns of file paths containing Example r
    tf_transform_output - transform output graph
    num_epochs - Integer specifying the number of times to read through the dat
            If None, cycles through the dataset forever.
    batch_size - An int representing the number of records to combine in a sing

  Returns:
    A dataset of dict elements, (or a tuple of dict elements and label).
    Each dict maps feature keys to Tensor or SparseTensor objects.
  '''
  transformed_feature_spec = (
      tf_transform_output.transformed_feature_spec().copy())

  dataset = tf.data.experimental.make_batched_features_dataset(
      file_pattern=file_pattern,
      batch_size=batch_size,
      features=transformed_feature_spec,
      reader=_gzip_reader_fn,
      num_epochs=num_epochs,
      label_key=LABEL_KEY)

  return dataset


def model_builder(hp):
  '''
  Builds the model and sets up the hyperparameters to tune.

  Args:
    hp - Keras tuner object

  Returns:
    model with hyperparameters to tune
  '''

```

```
  # Initialize the Sequential API and start stacking the layers
  model = keras.Sequential()
  model.add(keras.layers.Flatten(input_shape=(28, 28, 1)))

  # Get the number of units from the Tuner results
  hp_units = hp.get('units')
  model.add(keras.layers.Dense(units=hp_units, activation='relu'))

  # Add next layers
  model.add(keras.layers.Dropout(0.2))
  model.add(keras.layers.Dense(10, activation='softmax'))

  # Get the learning rate from the Tuner results
  hp_learning_rate = hp.get('learning_rate')

  # Setup model for training
  model.compile(optimizer=keras.optimizers.Adam(learning_rate=hp_learning_rate)
                loss=keras.losses.SparseCategoricalCrossentropy(),
                metrics=['accuracy'])

  # Print the model summary
  model.summary()

  return model


def run_fn(fn_args: FnArgs) -> None:
  """Defines and trains the model.
  Args:
    fn_args: Holds args as name/value pairs. Refer here for the complete attrib
    https://www.tensorflow.org/tfx/api_docs/python/tfx/components/trainer/fn_ar
  """

  # Callback for TensorBoard
  tensorboard_callback = tf.keras.callbacks.TensorBoard(
      log_dir=fn_args.model_run_dir, update_freq='batch')

  # Load transform output
  tf_transform_output = tft.TFTransformOutput(fn_args.transform_graph_path)

  # Create batches of data good for 10 epochs
  train_set = _input_fn(fn_args.train_files[0], tf_transform_output, 10)
  val_set = _input_fn(fn_args.eval_files[0], tf_transform_output, 10)

  # Load best hyperparameters
  hp = fn_args.hyperparameters.get('values')

  # Build the model
  model = model_builder(hp)

  # Train the model
  model.fit(
      x=train_set,
      validation_data=val_set,
      callbacks=[tensorboard_callback]
```

```
    )

# Save the model
model.save(fn_args.serving_model_dir, save_format='tf')
```

You can pass the output of the `Tuner` component to the `Trainer` by filling the `hyperparameters` argument with the `Tuner` output. This is indicated by the `tuner.outputs['best_hyperparameters']` below. You can see the definition of the other arguments [here](#).

```
# Setup the Trainer component
trainer = Trainer(
    module_file=_trainer_module_file,
    examples=transform.outputs['transformed_examples'],
    hyperparameters=tuner.outputs['best_hyperparameters'],
    transform_graph=transform.outputs['transform_graph'],
    schema=schema_gen.outputs['schema'],
    train_args=trainer_pb2.TrainArgs(splits=['train']),
    eval_args=trainer_pb2.EvalArgs(splits=['eval']))
```

Take note that when re-training your model, you don't always have to retune your hyperparameters. Once you have a set that you think performs well, you can just import it with the ImporterNode as shown in the [official docs](#):

```
hparams_importer = ImporterNode(
    instance_name='import_hparams',
    # This can be Tuner's output file or manually edited file. The file contains
    # text format of hyperparameters (kerastuner.HyperParameters.get_config())
    source_uri='path/to/best_hyperparameters.txt',
    artifact_type=HyperParameters)

trainer = Trainer(
    ...
    # An alternative is directly use the tuned hyperparameters in Trainer's user
    # module code and set hyperparameters to None here.
    hyperparameters = hparams_importer.outputs['result'])
```

```
# Run the component
context.run(trainer, enable_cache=False)
```

Your model should now be saved in your pipeline directory and you can navigate through it as shown below. The file is saved as `saved_model.pb`.

```
# Get artifact uri of trainer model output
model_artifact_dir = trainer.outputs['model'].get()[0].uri
```

```

# List subdirectories artifact uri
print(f'contents of model artifact directory:{os.listdir(model_artifact_dir)}')

# Define the model directory
model_dir = os.path.join(model_artifact_dir, 'Format-Serving')

# List contents of model directory
print(f'contents of model directory: {os.listdir(model_dir)}')
```

You can also visualize the training results by loading the logs saved by the Tensorboard callback.

```
model_run_artifact_dir = trainer.outputs['model_run'].get()[0].uri

%load_ext tensorboard
%tensorboard --logdir {model_run_artifact_dir}
```

Congratulations! You have now created an ML pipeline that includes hyperparameter tuning and model training. You will know more about the next components in future lessons but in the next section, you will first learn about a framework for automatically building ML pipelines: AutoML. Enjoy the rest of the course!