Ungraded Lab: TensorFlow Model Analysis

In production systems, the decision to deploy a model usually goes beyond the global metrics (e.g. accuracy) set during training. It is also important to evaluate how your model performs in different scenarios. For instance, does your weather forecasting model perform equally well in summer compared to winter? Or does your camera-based defect detector work only in certain lighting conditions? This type of investigation helps to ensure that your model can handle different cases. More than that, it can help uncover any learned biases that can result in a negative experience for your users. For example, if you're supposed to have a gender-neutral application, you don't want your model to only work well for one while poorly for another.

In this lab, you will be working with TensorFlow Model Analysis (TFMA) -- a library built specifically for analyzing a model's performance across different configurations. It allows you to specify slices of your data, then it will compute and visualize how your model performs on each slice. You can also set thresholds that your model must meet before it is marked ready for deployment. These help you make better decisions regarding any improvements you may want to make to boost your model's performance and ensure fairness.

For this exercise, you will use TFMA to analyze models trained on the Census Income dataset. Specifically, you will:

- study and setup the starter files to use with TFMA
- make a configuration file to tell TFMA what data slices it will analyze and the metrics it will compute
- visualize TFMA's outputs in a notebook environment
- generate a time series of a model's performance
- compare the performance of two models so you can decide which one to push to production

Credits: Some of the code and discussions are based on the TensorFlow team's official tutorial.

Setup

In this section, you will first setup your workspace to have all the modules and files to work with TFMA. You will

- install required libraries,
- download starter files that will contain the dataset, schema, and pretrained models you will analyze
- prepare the dataset so it can be consumed by TFMA
- observe how the models transform the raw features

Install Jupyter Extensions

If running in a local Jupyter notebook, then these Jupyter extensions must be installed in the environment before running Jupyter. These are already available in Colab so we'll just leave the commands here for reference.

```
jupyter nbextension enable --py widgetsnbextension --sys-prefix
jupyter nbextension install --py --symlink tensorflow_model_analysis
--sys-prefix
jupyter nbextension enable --py tensorflow_model_analysis --sys-prefix
```

Install libraries

This will pull in all the dependencies, and will take a minute.

In [ ]:
```
# Upgrade pip to the latest, and install required libraries.
!pip install -U pip
!pip install tensorflow_data_validation==1.1.0
!pip install tensorflow-transform==1.0.0
!pip install tensorflow-model-analysis==0.32.0
```

Note: In Google Colab, you need to restart the runtime at this point to finalize updating the packages you just installed. ***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.*

Check Installation

Running the code below should show the versions of the packages. Please re-run the install if you are seeing errors and don't forget to restart the runtime after re-installation.

In [1]:
```
# Import packages and print versions
import tensorflow as tf
import tensorflow_model_analysis as tfma
import tensorflow_data_validation as tfdv

print('TF version: {}'.format(tf.__version__))
print('TFMA version: {}'.format(tfma.__version__))
print('TFDV version: {}'.format(tfdv.__version__))
```

```
TF version: 2.5.1
TFMA version: 0.32.0
TFDV version: 1.1.0
```

Load The Files

Next, you will download the files you will need for this exercise:

- Test datasets
- Data schema
- Pretrained models

We've also defined some global variables below so you can access these files throughout the notebook more easily.

In [2]:
```
import os

# String variables for file and directory names
URL = 'https://github.com/https-deeplearning-ai/MLEP-public/raw/main/course
TAR_NAME = 'starter_files.tar.gz'
BASE_DIR = 'starter_files'
DATA_DIR = os.path.join(BASE_DIR, 'data')
CSV_DIR = os.path.join(DATA_DIR, 'csv')
TFRECORD_DIR = os.path.join(DATA_DIR, 'tfrecord')
MODELS_DIR = os.path.join(BASE_DIR, 'models')
SCHEMA_FILE = os.path.join(BASE_DIR, 'schema.pbtxt')
```

In [3]:
```
# uncomment this line if you've downloaded the files before and want to res
# !rm -rf starter_files

# Download the tar file from GCP
!wget {URL}

# Extract the tar file to the base directory
!tar xzf {TAR_NAME}

# Delete tar file
!rm {TAR_NAME}
```

```
--2021-08-31 12:27:12--  https://github.com/https-deeplearning-ai/MLEP-publ
ic/raw/main/course3/week4-ungraded-lab/starter_files.tar.gz
Resolving github.com (github.com)... 140.82.112.3
Connecting to github.com (github.com)|140.82.112.3|:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: https://github.com/https-deeplearning-ai/machine-learning-enginee
ring-for-production-public/raw/main/course3/week4-ungraded-lab/starter_file
s.tar.gz [following]
--2021-08-31 12:27:12--  https://github.com/https-deeplearning-ai/machine-l
earning-engineering-for-production-public/raw/main/course3/week4-ungraded-l
ab/starter_files.tar.gz
Reusing existing connection to github.com:443.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/https-deeplearning-ai/machine-l
earning-engineering-for-production-public/main/course3/week4-ungraded-lab/s
tarter_files.tar.gz [following]
--2021-08-31 12:27:12--  https://raw.githubusercontent.com/https-deeplearni
ng-ai/machine-learning-engineering-for-production-public/main/course3/week4
-ungraded-lab/starter_files.tar.gz
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.
111.133, 185.199.109.133, 185.199.108.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.19
9.111.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1449046 (1.4M) [application/octet-stream]
Saving to: ‘starter_files.tar.gz’

starter_files.tar.g 100%[===================>]   1.38M  --.-KB/s    in 0.07
s

2021-08-31 12:27:13 (20.6 MB/s) - ‘starter_files.tar.gz’ saved [1449046/144
9046]
```

You can see the top level file and directories by running the cell below (or just using the file explorer on the left side of this Colab). We'll discuss what each contain in the next sections.

In [4]:
```
print("Here's what we downloaded:")
!ls {BASE_DIR}
```

```
Here's what we downloaded:
data  models  schema.pbtxt
```

Preview the dataset

The `data/csv` directory contains the test split of the Census Income dataset. We've divided it into several files for this demo notebook:

- `data_test.csv` - 15000 rows of test data
- `data_test_1.csv` - first 5000 rows of `data_test.csv`
- `data_test_2.csv` - next 5000 rows of `data_test.csv`
- `data_test_3.csv` - last 5000 rows of `data_test.csv`

You can see the description of each column here (please open link in a new window if Colab prevents the download). Also for simplicity, we've already preprocessed the `label` column as binary (i.e. `0` or `1`) to match the model's output. In your own projects, your labels might be in a different data type (e.g. string) and you want to transform that first so you can evaluate your model properly. You can preview the first few rows below:

In [5]:
```
# Path to the full test set
TEST_DATA_PATH = os.path.join(CSV_DIR, 'data_test.csv')

# Preview the first few rows
!head {TEST_DATA_PATH}
```

```
age,workclass,fnlwgt,education,education-num,marital-status,occupation,rela
tionship,race,sex,capital-gain,capital-loss,hours-per-week,native-country,l
abel
25,Private,226802,11th,7,Never-married,Machine-op-inspct,Own-child,Black,Ma
le,0,0,40,United-States,0
38,Private,89814,HS-grad,9,Married-civ-spouse,Farming-fishing,Husband,Whit
e,Male,0,0,50,United-States,0
28,Local-gov,336951,Assoc-acdm,12,Married-civ-spouse,Protective-serv,Husban
d,White,Male,0,0,40,United-States,1
44,Private,160323,Some-college,10,Married-civ-spouse,Machine-op-inspct,Husb
and,Black,Male,7688,0,40,United-States,1
18,?,103497,Some-college,10,Never-married,?,Own-child,White,Female,0,0,30,U
nited-States,0
34,Private,198693,10th,6,Never-married,Other-service,Not-in-family,White,Ma
le,0,0,30,United-States,0
29,?,227026,HS-grad,9,Never-married,?,Unmarried,Black,Male,0,0,40,United-St
ates,0
63,Self-emp-not-inc,104626,Prof-school,15,Married-civ-spouse,Prof-specialt
y,Husband,White,Male,3103,0,32,United-States,1
24,Private,369667,Some-college,10,Never-married,Other-service,Unmarried,Whi
te,Female,0,0,40,United-States,0
```

Parse the Schema

You also downloaded a schema generated by TensorFlow Data Validation. You should be familiar with this file type already from previous courses. You will load it now so you can use it in the later parts of the notebook.

In [6]:

```
# Load the schema as a protocol buffer
SCHEMA = tfdv.load_schema_text(SCHEMA_FILE)

# Display the schema
tfdv.display_schema(SCHEMA)
```

Feature name	Type	Presence	Valency	Domain
'age'	INT	required		-
'capital-gain'	INT	required		-
'capital-loss'	INT	required		-
'education'	STRING	required		'education'
'education-num'	INT	required		-
'fnlwgt'	INT	required		-
'hours-per-week'	INT	required		-
'label'	INT	required		-
'marital-status'	STRING	required		'marital-status'
'native-country'	STRING	required		'native-country'
'occupation'	STRING	required		'occupation'
'race'	STRING	required		'race'
'relationship'	STRING	required		'relationship'
'sex'	STRING	required		'sex'
'workclass'	STRING	required		'workclass'

```
/usr/local/lib/python3.7/dist-packages/tensorflow_data_validation/utils/display_util.py:180: FutureWarning: Passing a negative integer is deprecated in version 1.0 and will not be supported in future version. Instead, use None to not limit the column width.
  pd.set_option('max_colwidth', -1)
```

Domain	Values
'education'	'10th', '11th', '12th', '1st-4th', '5th-6th', '7th-8th', '9th', 'Assoc-acdm', 'Assoc-voc', 'Bachelors', 'Doctorate', 'HS-grad', 'Masters', 'Preschool', 'Prof-school', 'Some-college'
'marital-status'	'Divorced', 'Married-AF-spouse', 'Married-civ-spouse', 'Married-spouse-absent', 'Never-married', 'Separated', 'Widowed'
'native-country'	'?', 'Cambodia', 'Canada', 'China', 'Columbia', 'Cuba', 'Dominican-Republic', 'Ecuador', 'El-Salvador', 'England', 'France', 'Germany', 'Greece', 'Guatemala', 'Haiti', 'Holand-Netherlands', 'Honduras', 'Hong', 'Hungary', 'India', 'Iran', 'Ireland', 'Italy', 'Jamaica', 'Japan', 'Laos', 'Mexico', 'Nicaragua', 'Outlying-US(Guam-USVI-etc)', 'Peru', 'Philippines', 'Poland', 'Portugal', 'Puerto-Rico', 'Scotland', 'South', 'Taiwan', 'Thailand', 'Trinadad&Tobago', 'United-States', 'Vietnam', 'Yugoslavia'
'occupation'	'?', 'Adm-clerical', 'Armed-Forces', 'Craft-repair', 'Exec-managerial', 'Farming-fishing', 'Handlers-cleaners', 'Machine-op-inspct', 'Other-service', 'Priv-house-serv', 'Prof-specialty', 'Protective-serv', 'Sales', 'Tech-support', 'Transport-moving'
'race'	'Amer-Indian-Eskimo', 'Asian-Pac-Islander', 'Black', 'Other', 'White'

Domain	Values
'relationship'	'Husband', 'Not-in-family', 'Other-relative', 'Own-child', 'Unmarried', 'Wife'
'sex'	'Female', 'Male'

Use the Schema to Create TFRecords

TFMA needs a TFRecord file input so you need to convert the CSVs in the data directory. If you've done the earlier labs, you will know that this can be easily done with `ExampleGen`. For this notebook however, you will use the helper function below instead to demonstrate how it can be done outside a TFX pipeline. You will pass in the schema you loaded in the previous step to determine the correct type of each feature.

In [7]:

```
# imports for helper function
import csv
from tensorflow.core.example import example_pb2
from tensorflow_metadata.proto.v0 import schema_pb2

def csv_to_tfrecord(schema, csv_file, tfrecord_file):
  ''' Converts a csv file into a tfrecord
  Args:
    schema (schema_pb2) - Schema protobuf from TFDV
    csv_file (string) - file to convert to tfrecord
    tfrecord_file (string) - filename of tfrecord to create

  Returns:
    filename of tfrecord
  '''

  # Open CSV file for reading. Each row is mapped as a dictionary.
  reader = csv.DictReader(open(csv_file, 'r'))

  # Initialize TF examples list
  examples = []

  # For each row in CSV, create a TF Example based on
  # the Schema and append to the list
  for line in reader:

    # Intialize example
    example = example_pb2.Example()

    # Loop through features in the schema
    for feature in schema.feature:

      # Get current feature name
      key = feature.name

      # Populate values based on data type of current feature
      if feature.type == schema_pb2.FLOAT:
        example.features.feature[key].float_list.value[:] = (
            [float(line[key])] if len(line[key]) > 0 else [])
      elif feature.type == schema_pb2.INT:
        example.features.feature[key].int64_list.value[:] = (
            [int(line[key])] if len(line[key]) > 0 else [])
      elif feature.type == schema_pb2.BYTES:
        example.features.feature[key].bytes_list.value[:] = (
            [line[key].encode('utf8')] if len(line[key]) > 0 else [])

    # Append to the list
    examples.append(example)

  # Write examples to tfrecord file
  with tf.io.TFRecordWriter(tfrecord_file) as writer:
    for example in examples:
      writer.write(example.SerializeToString())

  return tfrecord_file
```

The code below will do the conversion and we've defined some more global variables that you will use in later exercises.

In [8]:
```
# Create tfrecord directory
!mkdir {TFRECORD_DIR}

# Create list of tfrecord files
tfrecord_files = [csv_to_tfrecord(SCHEMA, f'{CSV_DIR}/{name}', f"{TFRECORD_
  for name in os.listdir(CSV_DIR)]

# Print created files
print(f'files created: {tfrecord_files}')

# Create variables for each tfrecord
TFRECORD_FULL = os.path.join(TFRECORD_DIR, 'data_test.tfrecord')
TFRECORD_DAY1 = os.path.join(TFRECORD_DIR, 'data_test_1.tfrecord')
TFRECORD_DAY2 = os.path.join(TFRECORD_DIR, 'data_test_2.tfrecord')
TFRECORD_DAY3 = os.path.join(TFRECORD_DIR, 'data_test_3.tfrecord')

# Delete unneeded variable
del tfrecord_files
```

```
files created: ['starter_files/data/tfrecord/data_test_3.tfrecord', 'starte
r_files/data/tfrecord/data_test_1.tfrecord', 'starter_files/data/tfrecord/d
ata_test.tfrecord', 'starter_files/data/tfrecord/data_test_2.tfrecord']
```

Pretrained models

Lastly, you also downloaded pretrained Keras models and they are stored in the `models/` directory. TFMA supports a number of different model types including TF Keras models, models based on generic TF2 signature APIs, as well TF estimator based models. The get_started guide has the full list of model types supported and any restrictions. You can also consult the FAQ for examples on how to configure these models.

We have included three models and you can choose to analyze any one of them in the later sections. These were saved in SavedModel format which is the default when saving with the Keras Models API.

In [9]:
```
# list model directories
!ls {MODELS_DIR}

# Create string variables for each model directory
MODEL1_FILE = os.path.join(MODELS_DIR, 'model1')
MODEL2_FILE = os.path.join(MODELS_DIR, 'model2')
MODEL3_FILE = os.path.join(MODELS_DIR, 'model3')
```

```
model1  model2  model3
```

As mentioned earlier, these models were trained on the Census Income dataset. The label is `1` if a person earns more than 50k USD and `0` if less than or equal. You can load one of the models and look at the summary to get a sense of its architecture. All three models use the same architecture but were trained with different epochs to simulate varying performance.

In [10]:
```
# Load model 1
model = tf.keras.models.load_model(MODEL1_FILE)

# Print summary. You can ignore the warnings at the start.
model.summary()
```

```
WARNING:tensorflow:SavedModel saved prior to TF 2.5 detected when loading K
eras model. Please ensure that you are saving the model with model.save() o
r tf.keras.models.save_model(), *NOT* tf.saved_model.save(). To confirm, th
ere should be a file named "keras_metadata.pb" in the SavedModel directory.
WARNING:tensorflow:SavedModel saved prior to TF 2.5 detected when loading K
eras model. Please ensure that you are saving the model with model.save() o
r tf.keras.models.save_model(), *NOT* tf.saved_model.save(). To confirm, th
ere should be a file named "keras_metadata.pb" in the SavedModel directory.
WARNING:tensorflow:Inconsistent references when loading the checkpoint into
this object graph. Either the Trackable object references in the Python pro
gram have changed in an incompatible way, or the checkpoint was generated i
n an incompatible program.

Two checkpoint references resolved to different objects (<tensorflow.pytho
n.keras.saving.saved_model.load.TensorFlowTransform>TransformFeaturesLayer
object at 0x7f94a20dfe50> and <tensorflow.python.keras.engine.input_layer.I
nputLayer object at 0x7f94a21d95d0>).
WARNING:tensorflow:Inconsistent references when loading the checkpoint into
this object graph. Either the Trackable object references in the Python pro
gram have changed in an incompatible way, or the checkpoint was generated i
n an incompatible program.

Two checkpoint references resolved to different objects (<tensorflow.pytho
n.keras.saving.saved_model.load.TensorFlowTransform>TransformFeaturesLayer
object at 0x7f94a20dfe50> and <tensorflow.python.keras.engine.input_layer.I
nputLayer object at 0x7f94a21d95d0>).
Model: "model"
__________________________________________________________________________
_____
Layer (type)                 Output Shape              Param #     Connected
to
==========================================================================
=====
fnlwgt_xf (InputLayer)       [(None, 1)]               0
__________________________________________________________________________
_____
education-num_xf (InputLayer) [(None, 1)]              0
__________________________________________________________________________
_____
capital-gain_xf (InputLayer) [(None, 1)]               0
__________________________________________________________________________
_____
capital-loss_xf (InputLayer) [(None, 1)]               0
__________________________________________________________________________
_____
hours-per-week_xf (InputLayer) [(None, 1)]             0
__________________________________________________________________________
_____
concatenate (Concatenate)    (None, 5)                 0           fnlwgt_xf
[0][0]
                                                                   education-
num_xf[0][0]
                                                                   capital-ga
in_xf[0][0]
                                                                   capital-lo
ss_xf[0][0]
                                                                   hours-per-
week_xf[0][0]
__________________________________________________________________________
_____
dense (Dense)                (None, 100)               600         concatenat
e[0][0]
__________________________________________________________________________
_____
dense_1 (Dense)              (None, 70)                7070        dense
[0][0]
__________________________________________________________________________
_____
education_xf (InputLayer)    [(None, 21)]              0
```

```
_____________________________
marital-status_xf (InputLayer)  [(None, 12)]         0
_____________________________
occupation_xf (InputLayer)      [(None, 20)]         0
_____________________________
race_xf (InputLayer)            [(None, 10)]         0
_____________________________
relationship_xf (InputLayer)    [(None, 11)]         0
_____________________________
workclass_xf (InputLayer)       [(None, 14)]         0
_____________________________
sex_xf (InputLayer)             [(None, 7)]          0
_____________________________
native-country_xf (InputLayer)  [(None, 47)]         0
_____________________________
age_xf (InputLayer)             [(None, 4)]          0
_____________________________
dense_2 (Dense)                 (None, 48)           3408        dense_1
[0][0]
_____________________________
concatenate_1 (Concatenate)     (None, 146)          0           education_
xf[0][0]                                                         marital-st
atus_xf[0][0]                                                    occupation
_xf[0][0]                                                        race_xf
[0][0]                                                           relationsh
ip_xf[0][0]                                                      workclass_
xf[0][0]                                                         sex_xf
[0][0]                                                           native-cou
ntry_xf[0][0]                                                    age_xf
[0][0]
_____________________________
dense_3 (Dense)                 (None, 34)           1666        dense_2
[0][0]
_____________________________
dense_4 (Dense)                 (None, 128)          18816       concatenat
e_1[0][0]
_____________________________
concatenate_2 (Concatenate)     (None, 162)          0           dense_3
[0][0]                                                           dense_4
[0][0]
_____________________________
dense_5 (Dense)                 (None, 1)            163         concatenat
e_2[0][0]
_____________________________
transform_features_layer (Tenso multiple             0
_________________________________________________________________
```

```
======================
Total params: 31,723
Trainable params: 31,723
Non-trainable params: 0
_____________________________________________________________________________
```

You can see the code to build these in the next lab. For now, you'll only need to take note of a few things. First, the output is a single dense unit with a sigmoid activation (i.e. `dense_5` above). This is standard for binary classification problems.

Another is that the model is exported with a transformation layer. You can see this in the summary above at the bottom row named `transform_features_layer` and it is not connected to the other layers. From previous labs, you will know that this is taken from the graph generated by the `Transform` component. It helps to avoid training-serving skews by making sure that raw inputs are transformed in the same way that the model expects. It is also available as a `tft_layer` property of the model object.

In [11]:
```
# Transformation layer can be accessed in two ways. These are equivalent.
model.get_layer('transform_features_layer') is model.tft_layer
```

Out[11]:
```
True
```

TFMA invokes this layer automatically for your raw inputs but we've included a short snippet below to demonstrate how the transformation works. You can see that the raw features are indeed reformatted to an acceptable input for the model. The raw numeric features are scaled and the raw categorical (string) features are encoded to one-hot vectors.

In [12]:
```
from tensorflow_transform.tf_metadata import schema_utils

# Load one tfrecord
tfrecord_file = tf.data.TFRecordDataset(TFRECORD_DAY1)

# Parse schema object as a feature spec
feature_spec = schema_utils.schema_as_feature_spec(SCHEMA).feature_spec

# Create a batch from the dataset
for records in tfrecord_file.batch(1).take(1):

  # Parse the batch to get a dictionary of raw features
  parsed_examples = tf.io.parse_example(records, feature_spec)

  # Print the results
  print("\nRAW FEATURES:")
  for key, value in parsed_examples.items():
    print(f'{key}: {value.numpy()}')

  # Pop the label since the model does not expect a label input
  parsed_examples.pop('label')

  # Transform the rest of the raw features using the transform layer
  transformed_examples = model.tft_layer(parsed_examples)

  # Print the input to the model
  print("\nTRANSFORMED FEATURES:")
  for key, value in transformed_examples.items():
    print(f'{key}: {value.numpy()}')
```

```
RAW FEATURES:
age: [[25]]
capital-gain: [[0]]
capital-loss: [[0]]
education: [[b'11th']]
education-num: [[7]]
fnlwgt: [[226802]]
hours-per-week: [[40]]
label: [[0]]
marital-status: [[b'Never-married']]
native-country: [[b'United-States']]
occupation: [[b'Machine-op-inspct']]
race: [[b'Black']]
relationship: [[b'Own-child']]
sex: [[b'Male']]
workclass: [[b'Private']]

TRANSFORMED FEATURES:
capital-gain_xf: [0.]
age_xf: [[1. 0. 0. 0.]]
education-num_xf: [0.4]
occupation_xf: [[0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0.]]
marital-status_xf: [[0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
native-country_xf: [[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0. 0. 0. 0. 0. 0.
  0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
fnlwgt_xf: [0.14569008]
race_xf: [[0. 1. 0. 0. 0. 0. 0. 0. 0.]]
workclass_xf: [[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]]
relationship_xf: [[0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]]
sex_xf: [[1. 0. 0. 0. 0. 0. 0.]]
education_xf: [[0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
0.]]
hours-per-week_xf: [0.39795917]
capital-loss_xf: [0.]
```

The transformed features can be passed into the model to get the predictions. The snippet below demonstrates this and we used a low-threshold BinaryAccuracy metric to compare the true labels and model predictions.

In [13]:

```
from tensorflow_transform.tf_metadata import schema_utils

# Load one tfrecord
tfrecord_file = tf.data.TFRecordDataset(TFRECORD_DAY1)

# Parse schema object as a feature spec
feature_spec = schema_utils.schema_as_feature_spec(SCHEMA).feature_spec

# Create a batch from the dataset
for records in tfrecord_file.batch(5).take(1):

  # Get the label values from the raw input
  parsed_examples = tf.io.parse_example(records, feature_spec)
  y_true = parsed_examples.pop('label')
  print(f'labels:\n {y_true.numpy()}\n')

  # Transform the raw features and pass to the model to get predictions
  transformed_examples = model.tft_layer(parsed_examples)
  y_pred = model(transformed_examples)
  print(f'predictions:\n {y_pred.numpy()}\n')

  # Measure the binary accuracy
  metric = tf.keras.metrics.BinaryAccuracy(threshold=0.3)
  metric.update_state(y_true, y_pred)
  print(f'binary accuracy: {metric.result().numpy()}\n')
```

```
labels:
 [[0]
 [0]
 [1]
 [1]
 [0]]

predictions:
 [[1.6402992e-34]
 [3.4708142e-02]
 [5.1936507e-03]
 [3.3919078e-01]
 [2.3632433e-15]]

binary accuracy: 0.800000011920929
```

Last thing to note is the model is also exported with a serving signature. You will know more about this in the next lab and in later parts of the specialization but for now, you can think of it as a configuration for when the model is deployed for inference. For this particular model, the default signature is configured to transform batches of serialized raw features before feeding to the model input. That way, you wouldn't have to explicitly code the transformations as previously shown in the snippet above. You can just pass in batches of data directly as shown below.

In [14]:
```
# Load one tfrecord
tfrecord_file = tf.data.TFRecordDataset(TFRECORD_DAY1)

# Print available signatures
print(f'model signatures: {model.signatures}\n')

# Create a batch
for records in tfrecord_file.batch(5).take(1):

  # Pass the batch to the model serving signature to get predictions
  output = model.signatures['serving_default'](examples=records)

  # Print results
  print(f"predictions:\n {output['output_0']}\n")
```

```
model signatures: _SignatureMap({'serving_default': <ConcreteFunction signa
ture_wrapper(*, examples) at 0x7F949E6C8B50>})

predictions:
 [[1.6402867e-34]
 [3.4708112e-02]
 [5.1937401e-03]
 [3.3919084e-01]
 [2.3632433e-15]]
```

TFMA accesses this model signature so it can work with the raw data and evaluate the model to get the metrics. Not only that, it can also extract specific features and domain values from your dataset before it computes these metrics. Let's see how this is done in the next section.

Setup and Run TFMA

With the dataset and model now available, you can now move on to use TFMA. There are some additional steps needed:

- Create a `tfma.EvalConfig` protocol message containing details about the models, metrics, and data slices you'd like to analyze
- Create a `tfma.EvalSharedModel` that points to your saved models.
- Specify an output path where the analysis results will be stored

Create EvalConfig

The tfma.EvalConfig() is a protocol message that sets up the analysis. Here, you will specify:

- `model_specs` - message containing at least the label key so it can be extracted from the evaluation/test data
- `metrics_specs` - containing the metrics you would like to evaluate. A comprehensive guide can be found here and you will use the binary classification metrics for this exercise.
- `slicing_specs` - containing the feature slices you would like to compute metrics for. A short description of different types of slices is shown here

The eval config should be passed as a protocol message and you can use the

google.protobuf.text_format module for that as shown below

In [15]:

```
import tensorflow_model_analysis as tfma
from google.protobuf import text_format

# Setup tfma.EvalConfig settings
eval_config = text_format.Parse("""
  ## Model information
  model_specs {
    # For keras (and serving models), you need to add a `label_key`.
    label_key: "label"
  }

  ## Post training metric information. These will be merged with any built-
  ## metrics from training.
  metrics_specs {
    metrics { class_name: "ExampleCount" }
    metrics { class_name: "BinaryAccuracy" }
    metrics { class_name: "BinaryCrossentropy" }
    metrics { class_name: "AUC" }
    metrics { class_name: "AUCPrecisionRecall" }
    metrics { class_name: "Precision" }
    metrics { class_name: "Recall" }
    metrics { class_name: "MeanLabel" }
    metrics { class_name: "MeanPrediction" }
    metrics { class_name: "Calibration" }
    metrics { class_name: "CalibrationPlot" }
    metrics { class_name: "ConfusionMatrixPlot" }
    # ... add additional metrics and plots ...
  }

  ## Slicing information

  # overall slice
  slicing_specs {}

  # slice specific features
  slicing_specs {
    feature_keys: ["sex"]
  }
  slicing_specs {
    feature_keys: ["race"]
  }

  # slice specific values from features
  slicing_specs {
    feature_values: {
      key: "native-country"
      value: "Cambodia"
    }
  }
  slicing_specs {
    feature_values: {
      key: "native-country"
      value: "Canada"
    }
  }

  # slice feature crosses
  slicing_specs {
    feature_keys: ["sex", "race"]
  }
""", tfma.EvalConfig())
```

Create EvalSharedModel

TFMA also requires an EvalSharedModel instance that points to your model so it can be shared between multiple threads in the same process. This instance includes information about the type of model (keras, etc) and how to load and configure the model from its saved location on disk (e.g. tags, etc). The tfma.default_eval_shared_model() API can be used to create this given the model location and eval config.

In [16]:
```
# Create a tfma.EvalSharedModel that points to your model.
# You can ignore the warnings generated.
eval_shared_model = tfma.default_eval_shared_model(
    eval_saved_model_path=MODEL1_FILE,
    eval_config=eval_config)
```

```
WARNING:tensorflow:SavedModel saved prior to TF 2.5 detected when loading K
eras model. Please ensure that you are saving the model with model.save() o
r tf.keras.models.save_model(), *NOT* tf.saved_model.save(). To confirm, th
ere should be a file named "keras_metadata.pb" in the SavedModel directory.
WARNING:tensorflow:SavedModel saved prior to TF 2.5 detected when loading K
eras model. Please ensure that you are saving the model with model.save() o
r tf.keras.models.save_model(), *NOT* tf.saved_model.save(). To confirm, th
ere should be a file named "keras_metadata.pb" in the SavedModel directory.
WARNING:tensorflow:Inconsistent references when loading the checkpoint into
this object graph. Either the Trackable object references in the Python pro
gram have changed in an incompatible way, or the checkpoint was generated i
n an incompatible program.

Two checkpoint references resolved to different objects (<tensorflow.pytho
n.keras.saving.saved_model.load.TensorFlowTransform>TransformFeaturesLayer
object at 0x7f949d137c50> and <tensorflow.python.keras.engine.input_layer.I
nputLayer object at 0x7f949d17ad10>).
WARNING:tensorflow:Inconsistent references when loading the checkpoint into
this object graph. Either the Trackable object references in the Python pro
gram have changed in an incompatible way, or the checkpoint was generated i
n an incompatible program.

Two checkpoint references resolved to different objects (<tensorflow.pytho
n.keras.saving.saved_model.load.TensorFlowTransform>TransformFeaturesLayer
object at 0x7f949d137c50> and <tensorflow.python.keras.engine.input_layer.I
nputLayer object at 0x7f949d17ad10>).
```

In [17]:
```
# Show properties of EvalSharedModel
print(f'EvalSharedModel contents: {eval_shared_model}')
```

```
EvalSharedModel contents: EvalSharedModel(model_path='starter_files/models/
model1', add_metrics_callbacks=[], include_default_metrics=True, example_we
ight_key=None, additional_fetches=None, model_loader=<tensorflow_model_anal
ysis.types.ModelLoader object at 0x7f949d8cdbe0>, model_name='', model_type
='tf_keras', rubber_stamp=False, is_baseline=False)
```

Run TFMA

With the setup complete, you just need to declare an output directory then run TFMA. You will pass in the eval config, shared model, dataset, and output directory to `tfma.run_model_analysis()` as shown below. This will create a `tfma.EvalResult` which you can use later for rendering metrics and plots.

In [ ]:
```
# Specify output path for the evaluation results
OUTPUT_DIR = os.path.join(BASE_DIR, 'output')

# Run TFMA. You can ignore the warnings generated.
eval_result = tfma.run_model_analysis(
    eval_shared_model=eval_shared_model,
    eval_config=eval_config,
    data_location=TFRECORD_FULL,
    output_path=OUTPUT_DIR)
```

Visualizing Metrics and Plots

You can visualize the results also using TFMA methods. In this section, you will view the returned metrics and plots for the different slices you specified in the eval config.

Rendering Metrics

You can view the metrics with the `tfma.view.render_slicing_metrics()` method. By default, the views will display the `Overall` slice. To view a particular slice you can pass in a feature name to the `slicing_column` argument as shown below. You can visualize the different metrics through the `Show` dropdown menu and you can hover over the bar charts to show the exact value measured.

We encourage you to try the different options you see and also modify the command. Here are some examples:

- Removing the `slicing_column` argument will produce the overall slice.
- You can also pass in `race` (since it was specified in the eval config) to see the results for that particular slice.
- Using the `Examples (Weighted) Threshold` slider above 5421 will remove the `Female` slice because it has less examples than that.
- Toggling the `View` dropdown to `Metrics Histogram` will show the results divided into buckets. For example, if you're slicing column is `sex` and the `Histogram Type` dropdown is at `Slice Counts`, then you will one slice in two of the 10 (default) buckets since we only have two values for that feature ('Male' and 'Female'). The x-axis show the values for the metric in the `Select Metric` dropdown. This is the default view when the number of slices is large.
- At the bottom of the screen, you will notice the measurements also presented in tabular format. You can sort it by clicking on the feature name headers.

In [19]:
```
# Render metrics for a feature
tfma.view.render_slicing_metrics(eval_result, slicing_column='sex')
```

More Slices

If you haven't yet, you can also pass in the `native-country` to the slicing column. The difference in this visualization is we only specified two of its values in the eval config earlier.

This is useful if you just want to study a subgroup of a particular feature and not the entire domain.

In [20]:
```
# Render metrics for feature. Review EvalConfig message to see what values
tfma.view.render_slicing_metrics(eval_result, slicing_column='native-countr
```

TFMA also supports creating feature crosses to analyze combinations of features. Our original settings created a cross between `sex` and `race` and you can pass it in as a SlicingSpec as shown below.

In [21]:
```
# Render metrics for feature crosses
tfma.view.render_slicing_metrics(
    eval_result,
    slicing_spec=tfma.SlicingSpec(
        feature_keys=['sex', 'race']))
```

In some cases, crossing the two columns creates a lot of combinations. You can narrow down the results to only look at specific values by specifying it in the `slicing_spec` argument. Below shows the results for the `sex` feature for the `Other` race.

In [22]:
```
# Narrow down the feature crosses by specifying feature values
tfma.view.render_slicing_metrics(
    eval_result,
    slicing_spec=tfma.SlicingSpec(
        feature_keys=['sex'], feature_values={'race': 'Other'}))
```

Rendering Plots

Any plots that were added to the `tfma.EvalConfig` as post training `metric_specs` can be displayed using `tfma.view.render_plot`.

As with metrics, plots can be viewed by slice. Unlike metrics, only plots for a particular slice value can be displayed so the `tfma.SlicingSpec` must be used and it must specify both a slice feature name and value. If no slice is provided then the plots for the `Overall` slice is used.

The example below displays the plots that were computed for the `sex:Male` slice. You can click on the names at the bottom of the graph to see a different plot type. Alternatively, you can tick the `Show all plots` checkbox to show all the plots in one screen.

In [23]:
```
# Render plots
tfma.view.render_plot(
    eval_result,
    tfma.SlicingSpec(feature_values={'sex': 'Male'}))
```

Tracking Model Performance Over Time

Your training dataset will be used for training your model, and will hopefully be

representative of your test dataset and the data that will be sent to your model in production. However, while the data in inference requests may remain the same as your training data, it can also start to change enough so that the performance of your model will change. That means that you need to monitor and measure your model's performance on an ongoing basis so that you can be aware of and react to changes.

Let's take a look at how TFMA can help. You will load three different datasets and compare the model analysis results using the `render_time_series()` method.

In [ ]:
```
import os

# Put data paths we prepared earlier in a list
TFRECORDS = [TFRECORD_DAY1, TFRECORD_DAY2, TFRECORD_DAY3]

# Initialize output paths list for each result
output_paths = []

# Run eval on each tfrecord separately
for num, tfrecord in enumerate(TFRECORDS):

  # Use the same model as before
  eval_shared_model = tfma.default_eval_shared_model(
      eval_saved_model_path=MODEL1_FILE,
      eval_config=eval_config)

  # Prepare output path name
  output_path = os.path.join('.', 'time_series', str(num))
  output_paths.append(output_path)

  # Run TFMA on the current tfrecord in the loop
  tfma.run_model_analysis(eval_shared_model=eval_shared_model,
                          eval_config=eval_config,
                          data_location=tfrecord,
                          output_path=output_path)
```

First, imagine that you've trained and deployed your model yesterday. And now, you want to see how it's doing on the new data coming in today. The visualization will start by displaying AUC. From the UI, you can:

- Add other metrics using the "Add metric series" menu.
- Close unwanted graphs by clicking on x
- Hover over data points (the ends of line segments in the graph) to get more details

Note: In the metric series charts, the x-axis is just the model directory name of the model that you're examining.

In [25]:
```
# Load results for day 1 and day 2 datasets
eval_results_from_disk = tfma.load_eval_results(output_paths[:2])

# Visualize results
tfma.view.render_time_series(eval_results_from_disk)
```

Now imagine that another day has passed and you want to see how it's doing on the new data coming in today.

In [26]:

```
# Load results for all three days
eval_results_from_disk = tfma.load_eval_results(output_paths)

# Visualize the results
tfma.view.render_time_series(eval_results_from_disk)
```

This type of investigation lets you see if your model is behaving poorly on new data. You can make the decision to retrain your production model based on these results. Retraining might not always produce the best results and you also need a way to detect that. You will see how TFMA helps in that regard in the next section.

Model Validation

TFMA can be configured to evaluate multiple models at the same time. Typically, this is done to compare a candidate model against a baseline (such as the currently serving model) to determine what the performance differences in metrics are. When thresholds are configured, TFMA will produce a `tfma.ValidationResult` record indicating whether the performance matches expecations.

Below, you will re-configure the EvalConfig settings to compare two models: a candidate and a baseline. You will also validate the candidate's performance against the baseline by setting a `tmfa.MetricThreshold` on the `BinaryAccuracy` metric. This helps in determining if your new model can indeed replace your currently deployed model.

In [ ]:

```
# Setup tfma.EvalConfig setting with metric thresholds
eval_config_with_thresholds = text_format.Parse("""
  ## Model information
  model_specs {
    name: "candidate"
    label_key: "label"
  }
  model_specs {
    name: "baseline"
    label_key: "label"
    is_baseline: true
  }

  ## Post training metric information
  metrics_specs {
    metrics { class_name: "ExampleCount" }
    metrics {
      class_name: "BinaryAccuracy"
      threshold {
        # Ensure that metric is always > 0.9
        value_threshold {
          lower_bound { value: 0.9 }
        }
        # Ensure that metric does not drop by more than a small epsilon
        # e.g. (candidate - baseline) > -1e-10 or candidate > baseline - 1e
        change_threshold {
          direction: HIGHER_IS_BETTER
          absolute { value: -1e-10 }
        }
      }
    }
    metrics { class_name: "BinaryCrossentropy" }
    metrics { class_name: "AUC" }
    metrics { class_name: "AUCPrecisionRecall" }
    metrics { class_name: "Precision" }
    metrics { class_name: "Recall" }
    metrics { class_name: "MeanLabel" }
    metrics { class_name: "MeanPrediction" }
    metrics { class_name: "Calibration" }
    metrics { class_name: "CalibrationPlot" }
    metrics { class_name: "ConfusionMatrixPlot" }
    # ... add additional metrics and plots ...
  }

  ## Slicing information
  slicing_specs {}  # overall slice
  slicing_specs {
    feature_keys: ["race"]
  }
  slicing_specs {
    feature_keys: ["sex"]
  }
""", tfma.EvalConfig())

# Create tfma.EvalSharedModels that points to the candidate and baseline
candidate_model_path = MODEL1_FILE
baseline_model_path = MODEL2_FILE

eval_shared_models = [
  tfma.default_eval_shared_model(
      model_name=tfma.CANDIDATE_KEY,
      eval_saved_model_path=candidate_model_path,
```

```
# Specify validation path
validation_output_path = os.path.join(OUTPUT_DIR, 'validation')

# Run TFMA on the two models
eval_result_with_validation = tfma.run_model_analysis(
    eval_shared_models,
    eval_config=eval_config_with_thresholds,
    data_location=TFRECORD_FULL,
    output_path=validation_output_path)
```

When running evaluations with one or more models against a baseline, TFMA automatically adds different metrics for all the metrics computed during the evaluation. These metrics are named after the corresponding metric but with the string `_diff` appended to the metric name. A positive value for these `_diff` metrics indicates an improved performance against the baseline.

Like in the previous section, you can view the results with `render_time_series()`.

In [28]:
```
# Render results
tfma.view.render_time_series(eval_result_with_validation)
```

You can use `tfma.load_validator_result` to view the validation results you specified with the threshold settings. For this example, the validation fails because `BinaryAccuracy` is below the threshold.

In [29]:
```
# Print validation result
validation_result = tfma.load_validation_result(validation_output_path)
print(validation_result.validation_ok)
```

```
False
```

Congratulations! You have now explored the different methods of model analysis using TFMA. In the next section, you will see how these can fit into a TFX pipeline so you can automate the process and store the results in your pipeline directory and metadata store.