Week 3 Assignment: Data Pipeline Components for Production ML

In this last graded programming exercise of the course, you will put together all the lessons we've covered so far to handle the first three steps of a production machine learning project - Data ingestion, Data Validation, and Data Transformation.

Specifically, you will build the production data pipeline by:

- Performing feature selection
- Ingesting the dataset
- Generating the statistics of the dataset
- Creating a schema as per the domain knowledge
- Creating schema environments
- Visualizing the dataset anomalies
- Preprocessing, transforming and engineering your features
- Tracking the provenance of your data pipeline using ML Metadata

Most of these will look familiar already so try your best to do the exercises by recall or browsing the documentation. If you get stuck however, you can review the lessons in class and the ungraded labs.

Let's begin!

Table of Contents

1 - Imports

```
import tensorflow as tf
import tfx

# TFX components
from tfx.components import CsvExampleGen
from tfx.components import ExampleValidator
from tfx.components import SchemaGen
from tfx.components import StatisticsGen
from tfx.components import Transform
from tfx.components import ImporterNode

# TFX libraries
import tensorflow_data_validation as tfdv
import tensorflow_transform as tft
from tfx.orchestration.experimental.interactive.interactive_context import Interacti

# For performing feature selection
from sklearn.feature_selection import SelectKBest, f_classif

# For feature visualization
import matplotlib.pyplot as plt
import seaborn as sns

# Utilities
from tensorflow.python.lib.io import file_io
from tensorflow_metadata.proto.v0 import schema_pb2
from google.protobuf.json_format import MessageToDict
from  tfx.proto import example_gen_pb2
from tfx.types import standard_artifacts
import os
import pprint
import tempfile
import pandas as pd

# To ignore warnings from TF
tf.get_logger().setLevel('ERROR')

# For formatting print statements
pp = pprint.PrettyPrinter()

# Display versions of TF and TFX related packages
print('TensorFlow version: {}'.format(tf.__version__))
print('TFX version: {}'.format(tfx.__version__))
print('TensorFlow Data Validation version: {}'.format(tfdv.__version__))
print('TensorFlow Transform version: {}'.format(tft.__version__))
```

```
TensorFlow version: 2.3.1
```

```
TFX version: 0.24.0
TensorFlow Data Validation version: 0.24.1
TensorFlow Transform version: 0.24.1
```

2 - Load the dataset

You are going to use a variant of the Cover Type dataset. This can be used to train a model that predicts the forest cover type based on cartographic variables. You can read more about the *original* dataset here and we've outlined the data columns below:

Column Name	Variable Type	Units / Range	Description
Elevation	quantitative	meters	Elevation in meters
Aspect	quantitative	azimuth	Aspect in degrees azimuth
Slope	quantitative	degrees	Slope in degrees
Horizontal_Distance_To_Hydrology	quantitative	meters	Horz Dist to nearest surface water features
Vertical_Distance_To_Hydrology	quantitative	meters	Vert Dist to nearest surface water features
Horizontal_Distance_To_Roadways	quantitative	meters	Horz Dist to nearest roadway
Hillshade_9am	quantitative	0 to 255 index	Hillshade index at 9am, summer solstice
Hillshade_Noon	quantitative	0 to 255 index	Hillshade index at noon, summer soltice
Hillshade_3pm	quantitative	0 to 255 index	Hillshade index at 3pm, summer solstice
Horizontal_Distance_To_Fire_Points	quantitative	meters	Horz Dist to nearest wildfire ignition points
Wilderness_Area (4 binary columns)	qualitative	0 (absence) or 1 (presence)	Wilderness area designation
Soil_Type (40 binary columns)	qualitative	0 (absence) or 1 (presence)	Soil Type designation
Cover_Type (7 types)	integer	1 to 7	Forest Cover Type designation

As you may notice, the qualitative data has already been one-hot encoded (e.g. `Soil_Type` has 40 binary columns where a `1` indicates presence of a feature). For learning, we will use a modified version of this dataset that shows a more raw format. This will let you practice your skills in handling different data types. You can see the code for preparing the dataset here if you want but it is **not required for this assignment**. The main changes include:

- Converting `Wilderness_Area` and `Soil_Type` to strings.
- Converting the `Cover_Type` range to [0, 6]

Run the next cells to load the **modified** dataset to your workspace.

```
# # OPTIONAL: Just in case you want to restart the lab workspace *from scratch*, you
# # can uncomment and run this block to delete previously created files and
# # directories.

# !rm -rf pipeline
# !rm -rf data
```

```
# Declare paths to the data
DATA_DIR = './data'
TRAINING_DIR = f'{DATA_DIR}/training'
TRAINING_DATA = f'{TRAINING_DIR}/dataset.csv'

# Create the directory
!mkdir -p {TRAINING_DIR}
```

```
# download the dataset
!wget -nc https://storage.googleapis.com/workshop-datasets/covertype/full/dataset.csv
```

```
File './data/training/dataset.csv' already there; not retrieving.
```

3 - Feature Selection

For your first task, you will reduce the number of features to feed to the model. As mentioned in Week 2, this will help reduce the complexity of your model and save resources while training. Let's assume that you already have a baseline model that is trained on all features and you want to see if reducing the number of features will generate a better model. You will want to select a subset that has great predictive value to the label (in this case the `Cover_Type`). Let's do that in the following cells.

```
# Load the dataset to a dataframe
df = pd.read_csv(TRAINING_DATA)

# Preview the dataset
df.head()
```

	Elevation	Aspect	Slope	Horizontal_Distance_To_Hydrology	Vertical_Distance_To_Hydrology	Horizontal_Dist
0	2596	51	3	258	0	
1	2590	56	2	212	-6	
2	2804	139	9	268	65	
3	2785	155	18	242	118	
4	2595	45	2	153	-1	

```
# Show the data type of each column
df.dtypes
```

```
Elevation                              int64
Aspect                                 int64
Slope                                  int64
Horizontal_Distance_To_Hydrology       int64
```

```
Vertical_Distance_To_Hydrology        int64
Horizontal_Distance_To_Roadways       int64
Hillshade_9am                         int64
Hillshade_Noon                        int64
Hillshade_3pm                         int64
Horizontal_Distance_To_Fire_Points    int64
Wilderness_Area                      object
Soil_Type                            object
Cover_Type                            int64
dtype: object
```

Looking at the data types of each column and the dataset description at the start of this notebook, you can see that most of the features are numeric and only two are not. This needs to be taken into account when selecting the subset of features because numeric and categorical features are scored differently. Let's create a temporary dataframe that only contains the numeric features so we can use it in the next sections.

```
# Copy original dataset
df_num = df.copy()

# Categorical columns
cat_columns = ['Wilderness_Area', 'Soil_Type']

# Label column
label_column = ['Cover_Type']

# Drop the categorical and label columns
df_num.drop(cat_columns, axis=1, inplace=True)
df_num.drop(label_column, axis=1, inplace=True)

# Preview the resuls
df_num.head()
```

	Elevation	Aspect	Slope	Horizontal_Distance_To_Hydrology	Vertical_Distance_To_Hydrology	Horizontal_Dist
0	2596	51	3	258	0	
1	2590	56	2	212	-6	
2	2804	139	9	268	65	
3	2785	155	18	242	118	
4	2595	45	2	153	-1	

You will use scikit-learn's built-in modules to perform univariate feature selection on our dataset's numeric attributes. First, you need to prepare the input and target features:

```
# Set the target values
y = df[label_column].values

# Set the input values
X = df_num.values
```

Afterwards, you will use SelectKBest to score each input feature against the target variable. Be mindful of the scoring function to pass in and make sure it is appropriate for the input (numeric) and target (categorical) values.

Exercise 1: Feature Selection

Complete the code below to select the top 8 features of the numeric columns.

```
### START CODE HERE ###

# Create SelectKBest object using f_classif (ANOVA statistics) for 8 classes
select_k_best = SelectKBest(f_classif, k=8)

# Fit and transform the input data using select_k_best
X_new = select_k_best.fit_transform(X,y)

# Extract the features which are selected using get_support API
features_mask = select_k_best.get_support()

### END CODE HERE ###

# Print the results
reqd_cols = pd.DataFrame({'Columns': df_num.columns, 'Retain': features_mask})
print(reqd_cols)
```

```
                              Columns  Retain
0                           Elevation    True
1                              Aspect   False
2                               Slope    True
3    Horizontal_Distance_To_Hydrology    True
4      Vertical_Distance_To_Hydrology    True
5     Horizontal_Distance_To_Roadways    True
6                       Hillshade_9am    True
7                      Hillshade_Noon    True
8                       Hillshade_3pm   False
9  Horizontal_Distance_To_Fire_Points    True
```

Expected Output:

```
                              Columns  Retain
0                           Elevation    True
1                              Aspect   False
2                               Slope    True
3    Horizontal_Distance_To_Hydrology    True
4      Vertical_Distance_To_Hydrology    True
5     Horizontal_Distance_To_Roadways    True
6                       Hillshade_9am    True
7                      Hillshade_Noon    True
8                       Hillshade_3pm   False
9  Horizontal_Distance_To_Fire_Points    True
```

If you got the expected results, you can now select this subset of features from the original dataframe and save it to a new directory in your workspace.

```
# Set the paths to the reduced dataset
TRAINING_DIR_FSELECT = f'{TRAINING_DIR}/fselect'
TRAINING_DATA_FSELECT = f'{TRAINING_DIR_FSELECT}/dataset.csv'

# Create the directory
!mkdir -p {TRAINING_DIR_FSELECT}
```

```
# Get the feature names from SelectKBest
feature_names = list(df_num.columns[features_mask])

# Append the categorical and label columns
feature_names = feature_names + cat_columns + label_column

# Select the selected subset of columns
df_select = df[feature_names]

# Write CSV to the created directory
df_select.to_csv(TRAINING_DATA_FSELECT, index=False)

# Preview the results
df_select.head()
```

	Elevation	Slope	Horizontal_Distance_To_Hydrology	Vertical_Distance_To_Hydrology	Horizontal_Distance_To_I
0	2596	3	258	0	
1	2590	2	212	-6	
2	2804	9	268	65	
3	2785	18	242	118	
4	2595	2	153	-1	

4 - Data Pipeline

With the selected subset of features prepared, you can now start building the data pipeline. This involves ingesting, validating, and transforming your data. You will be using the TFX components you've already encountered in the ungraded labs and you can look them up here in the official documentation.

4.1 - Setup the Interactive Context

As usual, you will first setup the Interactive Context so you can manually execute the pipeline components from the notebook. You will save the sqlite database in a pre-defined directory in your workspace. Please do not modify this path because you will need this in a later exercise involving ML Metadata.

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

# Declare the InteractiveContext and use a local sqlite file as the metadata store.
context = InteractiveContext(pipeline_root=PIPELINE_DIR)
```

```
WARNING:absl:InteractiveContext metadata_connection_config not provided: using SQLite
ML Metadata database at ./pipeline/metadata.sqlite.
```

4.2 - Generating Examples

The first step in the pipeline is to ingest the data. Using ExampleGen, you can convert raw data to TFRecords for faster computation in the later stages of the pipeline.

Exercise 2: ExampleGen

Use `ExampleGen` to ingest the dataset we loaded earlier. Some things to note:

- The input is in CSV format so you will need to use the appropriate type of `ExampleGen` to handle it.
- This function accepts a *directory* path to the training data and not the CSV file path itself.

This will take a couple of minutes to run.

```
# # NOTE: Uncomment and run this if you get an error saying there are different
# # headers in the dataset. This is usually because of the notebook checkpoints save
# # that folder.
# !rm -rf {TRAINING_DIR}/.ipynb_checkpoints
# !rm -rf {TRAINING_DIR_FSELECT}/.ipynb_checkpoints
# !rm -rf {SERVING_DIR}/.ipynb_checkpoints
```

```
### START CODE HERE

# Instantiate ExampleGen with the input CSV dataset
example_gen = CsvExampleGen(input_base=TRAINING_DIR_FSELECT)

# Run the component using the InteractiveContext instance
context.run(example_gen)

### END CODE HERE
```

```
▼ExecutionResult at 0x7fa59cc13fd0

        .execution_id   5
           .component   ▶ CsvExampleGen at 0x7fa59cbc8820
    .component.inputs   {}
   .component.outputs   ['examples']   ▶ Channel of type 'Examples' (1 artifact) at
                                       0x7fa59cbc8580
```

4.3 - Computing Statistics

Next, you will compute the statistics of your data. This will allow you to observe and analyze characteristics of your data through visualizations provided by the integrated FACETS library.

Exercise 3: StatisticsGen

Use StatisticsGen to compute the statistics of the output examples of `ExampleGen`.

```python
### START CODE HERE

# Instantiate StatisticsGen with the ExampleGen ingested dataset
statistics_gen = StatisticsGen(examples=example_gen.outputs['examples'])


# Run the component
context.run(statistics_gen)
### END CODE HERE
```

```
▼ExecutionResult at 0x7fa59cbc8250

        .execution_id   6

           .component   ▶ StatisticsGen at 0x7fa59cbc81f0

    .component.inputs
                        ['examples']    ▶ Channel of type 'Examples' (1 artifact) at
                                        0x7fa59cbc8580

   .component.outputs   ['statistics']  ▶ Channel of type 'ExampleStatistics' (1 artifact) at
                                        0x7fa59cbc8670
```

```python
# Display the results
context.show(statistics_gen.outputs['statistics'])
```

Artifact at ./pipeline/StatisticsGen/statistics/6

'train' split:

'eval' split:

Once you've loaded the display, you may notice that the `zeros` column for `Cover_type` is highlighted in red. The visualization is letting us know that this might be a potential issue. In our case though, we know that the `Cover_Type` has a range of [0, 6] so having zeros in this column is something we expect.

4.4 - Inferring the Schema

You will need to create a schema to validate incoming datasets during training and serving. Fortunately, TFX allows you to infer a first draft of this schema with the SchemaGen component.

Exercise 4: SchemaGen

Use `SchemaGen` to infer a schema based on the computed statistics of `StatisticsGen`.

```
### START CODE HERE
# Instantiate SchemaGen with the output statistics from the StatisticsGen
schema_gen = SchemaGen(statistics=statistics_gen.outputs['statistics'])



# Run the component
context.run(schema_gen)
### END CODE HERE
```

```
▼ExecutionResult at 0x7fa59c9bebe0

        .execution_id   7

          .component    ▶SchemaGen at 0x7fa58baa6280

   .component.inputs
                        ['statistics']  ▶Channel of type 'ExampleStatistics' (1 artifact) at
                                        0x7fa59cbc8670

  .component.outputs    ['schema']     ▶Channel of type 'Schema' (1 artifact) at
                                        0x7fa58baa6460
```

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

Artifact at ./pipeline/SchemaGen/schema/7

Feature name	Type	Presence	Valency	Domain
'Soil_Type'	STRING	required	single	'Soil_Type'
'Wilderness_Area'	STRING	required	single	'Wilderness_Area'
'Cover_Type'	INT	required	single	-
'Elevation'	INT	required	single	-
'Hillshade_9am'	INT	required	single	-
'Hillshade_Noon'	INT	required	single	-
'Horizontal_Distance_To_Fire_Points'	INT	required	single	-
'Horizontal_Distance_To_Hydrology'	INT	required	single	-
'Horizontal_Distance_To_Roadways'	INT	required	single	-
'Slope'	INT	required	single	-
'Vertical_Distance_To_Hydrology'	INT	required	single	-

Values

	Domain
'Soil_Type'	'C2702', 'C2703', 'C2704', 'C2705', 'C2706', 'C2717', 'C3501', 'C3502', 'C4201', 'C4703', 'C4704', 'C4744', 'C4758', 'C5101', 'C5151', 'C6101', 'C6102', 'C6731', 'C7101', 'C7102', 'C7103', 'C7201', 'C7202', 'C7700', 'C7701', 'C7702', 'C7709', 'C7710', 'C7745', 'C7746', 'C7755', 'C7756', 'C7757', 'C7790', 'C8703', 'C8707', 'C8708', 'C8771', 'C8772', 'C8776'
'Wilderness_Area'	'Cache', 'Commanche', 'Neota', 'Rawah'

4.5 - Curating the schema

You can see that the inferred schema is able to capture the data types correctly and also able to show the expected values for the qualitative (i.e. string) data. You can still fine-tune this however. For instance, we have features where we expect a certain range:

- `Hillshade_9am` : 0 to 255
- `Hillshade_Noon` : 0 to 255
- `Slope` : 0 to 90
- `Cover_Type` : 0 to 6

You want to update your schema to take note of these so the pipeline can detect if invalid values are being fed to the model.

Exercise 5: Curating the Schema

Use TFDV to update the inferred schema to restrict a range of values to the features mentioned above.

Things to note:

- You can use tfdv.set_domain() to define acceptable values for a particular feature.
- These should still be INT types after making your changes.
- Declare `Cover_Type` as a *categorical* variable. Unlike the other four features, the integers 0 to 6 here correspond to a designated label and not a quantitative measure. You can look at the available flags for `set_domain()` in the official doc to know how to set this.

```python
try:
    # Get the schema uri
    schema_uri = schema_gen.outputs['schema']._artifacts[0].uri

# for grading since context.run() does not work outside the notebook
except IndexError:
    print("context.run() was no-op")
    schema_path = './pipeline/SchemaGen/schema'
    dir_id = os.listdir(schema_path)[0]
    schema_uri = f'{schema_path}/{dir_id}'
```

```python
# Get the schema pbtxt file from the SchemaGen output
schema = tfdv.load_schema_text(os.path.join(schema_uri, 'schema.pbtxt'))
```

```
### START CODE HERE ###

# Set the two `Hillshade` features to have a range of 0 to 255
tfdv.set_domain(schema, 'Hillshade_9am', schema_pb2.IntDomain(name='Hillshade_9am', m
tfdv.set_domain(schema, 'Hillshade_Noon', schema_pb2.IntDomain(name='Hillshade_Noon',

# Set the `Slope` feature to have a range of 0 to 90
tfdv.set_domain(schema, 'Slope', schema_pb2.IntDomain(name='Slope', min=0, max=90))

# Set `Cover_Type` to categorical having minimum value of 0 and maximum value of 6
tfdv.set_domain(schema, 'Cover_Type', schema_pb2.IntDomain(name='Cover_Type', min=0,

### END CODE HERE ###

tfdv.display_schema(schema=schema)
```

Feature name	Type	Presence	Valency	Domain
'Soil_Type'	STRING	required	single	'Soil_Type'
'Wilderness_Area'	STRING	required	single	'Wilderness_Area'
'Cover_Type'	INT	required	single	[0,6]
'Elevation'	INT	required	single	-
'Hillshade_9am'	INT	required	single	[0,255]
'Hillshade_Noon'	INT	required	single	[0,255]
'Horizontal_Distance_To_Fire_Points'	INT	required	single	-
'Horizontal_Distance_To_Hydrology'	INT	required	single	-
'Horizontal_Distance_To_Roadways'	INT	required	single	-
'Slope'	INT	required	single	[0,90]
'Vertical_Distance_To_Hydrology'	INT	required	single	-

Domain	Values
'Soil_Type'	'C2702', 'C2703', 'C2704', 'C2705', 'C2706', 'C2717', 'C3501', 'C3502', 'C4201', 'C4703', 'C4704', 'C4744', 'C4758', 'C5101', 'C5151', 'C6101', 'C6102', 'C6731', 'C7101', 'C7102', 'C7103', 'C7201', 'C7202', 'C7700', 'C7701', 'C7702', 'C7709', 'C7710', 'C7745', 'C7746', 'C7755', 'C7756', 'C7757', 'C7790', 'C8703', 'C8707', 'C8708', 'C8771', 'C8772', 'C8776'
'Wilderness_Area'	'Cache', 'Commanche', 'Neota', 'Rawah'

You should now see the ranges you declared in the `Domain` column of the schema.

4.6 - Schema Environments

In supervised learning, we train the model to make predictions by feeding a set of features with its corresponding label. Thus, our training dataset will have both the input features and label, and the schema is configured to detect these.

However, after training and you serve the model for inference, the incoming data will no longer have the label. This will present problems when validating the data using the current version of the schema. Let's demonstrate that in the following cells. You will simulate a serving dataset by getting subset of the training set and dropping the label column (i.e. `Cover_Type`). Afterwards, you will validate this serving

```
# Declare paths to the serving data
SERVING_DIR = f'{DATA_DIR}/serving'
SERVING_DATA = f'{SERVING_DIR}/serving_dataset.csv'

# Create the directory
!mkdir -p {SERVING_DIR}
```

```
# Read a subset of the training dataset
serving_data = pd.read_csv(TRAINING_DATA, nrows=100)

# Drop the `Cover_Type` column
serving_data.drop(columns='Cover_Type', inplace=True)

# Save the modified dataset
serving_data.to_csv(SERVING_DATA, index=False)

# Delete unneeded variable from memory
del serving_data
```

```
# Declare StatsOptions to use the curated schema
stats_options = tfdv.StatsOptions(schema=schema, infer_type_from_schema=True)

# Compute the statistics of the serving dataset
serving_stats = tfdv.generate_statistics_from_csv(SERVING_DATA, stats_options=stats_

# Detect anomalies in the serving dataset
anomalies = tfdv.validate_statistics(serving_stats, schema=schema)

# Display the anomalies detected
tfdv.display_anomalies(anomalies)
```

	Anomaly short description	Anomaly long description
Feature name		
'Cover_Type'	Column dropped	Column is completely missing

As expected, the missing column is flagged. To fix this, you need to configure the schema to detect when it's being used for training or for inference / serving. You can do this by setting schema environments.

Exercise 6: Define the serving environment

Complete the code below to ignore the `Cover_Type` feature when validating in the *SERVING* environment.

```
schema.default_environment.append('TRAINING')

### START CODE HERE ###
# Hint: Create another default schema environment with name SERVING (pass in a strin
schema.default_environment.append('SERVING')

# Remove Cover_Type feature from SERVING using TFDV
# Hint: Pass in the strings with the name of the feature and environment
tfdv.get_feature(schema, 'Cover_Type').not_in_environment.append('SERVING')
### END CODE HERE ###
```

If done correctly, running the cell below should show *No Anomalies*.

```
# Validate the serving dataset statistics in the `SERVING` environment
anomalies = tfdv.validate_statistics(serving_stats, schema=schema, environment='SERVI

# Display the anomalies detected
tfdv.display_anomalies(anomalies)
```

No anomalies found.

We can now save this curated schema in a local directory so we can import it to our TFX pipeline.

```
# Declare the path to the updated schema directory
UPDATED_SCHEMA_DIR = f'{PIPELINE_DIR}/updated_schema'

# Create the said directory
!mkdir -p {UPDATED_SCHEMA_DIR}

# Declare the path to the schema file
schema_file = os.path.join(UPDATED_SCHEMA_DIR, 'schema.pbtxt')

# Save the curated schema to the said file
tfdv.write_schema_text(schema, schema_file)
```

As a sanity check, let's display the schema we just saved and verify that it contains the changes we introduced. It should still show the ranges in the `Domain` column and there should be two environments available.

```
# Load the schema from the directory we just created
new_schema = tfdv.load_schema_text(schema_file)

# Display the schema. Check that the Domain column still contains the ranges.
tfdv.display_schema(schema=new_schema)
```

	Type	Presence	Valency	Domain
Feature name				
'Soil_Type'	STRING	required	single	'Soil_Type'
'Wilderness_Area'	STRING	required	single	'Wilderness_Area'
'Cover_Type'	INT	required	single	[0,6]
'Elevation'	INT	required	single	-
'Hillshade_9am'	INT	required	single	[0,255]

Feature name	Type	Presence	Valency	Domain
'Hillshade_Noon'	INT	required	single	[0,255]
'Horizontal_Distance_To_Fire_Points'	INT	required	single	-
'Horizontal_Distance_To_Hydrology'	INT	required	single	-
'Horizontal_Distance_To_Roadways'	INT	required	single	-

Domain	Values
'Soil_Type'	'C2702', 'C2703', 'C2704', 'C2705', 'C2706', 'C2717', 'C3501', 'C3502', 'C4201', 'C4703', 'C4704', 'C4744', 'C4758', 'C5101', 'C5151', 'C6101', 'C6102', 'C6731', 'C7101', 'C7102', 'C7103', 'C7201', 'C7202', 'C7700', 'C7701', 'C7702', 'C7709', 'C7710', 'C7745', 'C7746', 'C7755', 'C7756', 'C7757', 'C7790', 'C8703', 'C8707', 'C8708', 'C8771', 'C8772', 'C8776'
'Wilderness_Area'	'Cache', 'Commanche', 'Neota', 'Rawah'

```
# The environment list should show `TRAINING` and `SERVING`.
new_schema.default_environment
```

```
['TRAINING', 'SERVING']
```

4.7 - Generate new statistics using the updated schema

You will now compute the statistics using the schema you just curated. Remember though that TFX components interact with each other by getting artifact information from the metadata store. So you first have to import the curated schema file into ML Metadata. You will do that by using an ImporterNode to create an artifact representing the curated schema.

Exercise 7: ImporterNode

Complete the code below to create a `Schema` artifact that points to the curated schema directory. Pass in an `instance_name` as well and name it `import_user_schema`.

```
### START CODE HERE ###

# Use an ImporterNode to put the curated schema to ML Metadata
user_schema_importer = ImporterNode(instance_name='import_user_schema',
                                   source_uri=UPDATED_SCHEMA_DIR,
                                   artifact_type=standard_artifacts.Schema)



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

### END CODE HERE ###

context.show(user_schema_importer.outputs['result'])
```

Artifact at ./pipeline/updated_schema

Feature name	Type	Presence	Valency	Domain
'Soil_Type'	STRING	required	single	'Soil_Type'
'Wilderness_Area'	STRING	required	single	'Wilderness_Area'
'Cover_Type'	INT	required	single	[0,6]
'Elevation'	INT	required	single	-
'Hillshade_9am'	INT	required	single	[0,255]
'Hillshade_Noon'	INT	required	single	[0,255]
'Horizontal_Distance_To_Fire_Points'	INT	required	single	-
'Horizontal_Distance_To_Hydrology'	INT	required	single	-
'Horizontal_Distance_To_Roadways'	INT	required	single	-
'Slope'	INT	required	single	[0,90]
'Vertical_Distance_To_Hydrology'	INT	required	single	-

Domain	Values
'Soil_Type'	'C2702', 'C2703', 'C2704', 'C2705', 'C2706', 'C2717', 'C3501', 'C3502', 'C4201', 'C4703', 'C4704', 'C4744', 'C4758', 'C5101', 'C5151', 'C6101', 'C6102', 'C6731', 'C7101', 'C7102', 'C7103', 'C7201', 'C7202', 'C7700', 'C7701', 'C7702', 'C7709', 'C7710', 'C7745', 'C7746', 'C7755', 'C7756', 'C7757', 'C7790', 'C8703', 'C8707', 'C8708', 'C8771', 'C8772', 'C8776'
'Wilderness_Area'	'Cache', 'Commanche', 'Neota', 'Rawah'

With the artifact successfully created, you can now use `StatisticsGen` and pass in a `schema` parameter to use the curated schema.

Exercise 8: Statistics with the new schema

Use `StatisticsGen` to compute the statistics with the schema you updated in the previous section.

```
### START CODE HERE ###
# Use StatisticsGen to compute the statistics using the curated schema
statistics_gen_updated = StatisticsGen(examples=example_gen.outputs['examples'],
                                       schema=user_schema_importer.outputs['result'])



# Run the component
context.run(statistics_gen_updated)
### END CODE HERE ###
```

```
▼ExecutionResult at 0x7fa5869d6a60

    .execution_id   9

     .component   ▶ StatisticsGen at 0x7fa5869719a0
```

.component.inputs	**['examples']**	**Channel** of type **'Examples'** (1 artifact) at 0x7fa59cbc8580
	['schema']	**Channel** of type **'Schema'** (1 artifact) at 0x7fa59cc13a30
.component.outputs	**['statistics']**	**Channel** of type **'ExampleStatistics'** (1 artifact) at 0x7fa586971820

```
context.show(statistics_gen_updated.outputs['statistics'])
```

Artifact at ./pipeline/StatisticsGen/statistics/9

'train' split:

'eval' split:

The chart will look mostly the same from the previous runs but you can see that the `Cover Type` is now under the categorical features. That shows that `StatisticsGen` is indeed using the updated schema.

4.8 - Check anomalies

You will now check if the dataset has any anomalies with respect to the schema. You can do that easily with the ExampleValidator component.

Exercise 9: ExampleValidator

Check if there are any anomalies using `ExampleValidator`. You will need to pass in the updated statistics and schema from the previous sections.

```
### START CODE HERE ###

example_validator = ExampleValidator(statistics=statistics_gen_updated.outputs['stat
                                     schema=user_schema_importer.outputs['result'])



# Run the component.
context.run(example_validator)

### END CODE HERE ###
```

```
▼ExecutionResult at 0x7fa57a02c760

        .execution_id   10

           .component   ▶ExampleValidator at 0x7fa57a318a30

    .component.inputs
                          ['statistics']   ▶Channel of type 'ExampleStatistics' (1 artifact) at
                                           0x7fa586971820

                              ['schema']   ▶Channel of type 'Schema' (1 artifact) at
                                           0x7fa59cc13a30

   .component.outputs     ['anomalies']   ▶Channel of type 'ExampleAnomalies' (1
                                          artifact) at 0x7fa57a318e80
```

```
# Visualize the results
context.show(example_validator.outputs['anomalies'])
```

Artifact at ./pipeline/ExampleValidator/anomalies/10

'train' split:

No anomalies found.

'eval' split:

No anomalies found

4.10 - Feature engineering

You will now proceed to transforming your features to a form suitable for training a model. This can include several methods such as scaling and converting strings to vocabulary indices. It is important for these transformations to be consistent across your training data, and also for the serving data when the model is deployed for inference. TFX ensures this by generating a graph that will process incoming data both during training and inference.

Let's first declare the constants and utility function you will use for the exercise.

```
# Set the constants module filename
_cover_constants_module_file = 'cover_constants.py'
```

```
%%writefile {_cover_constants_module_file}

SCALE_MINMAX_FEATURE_KEYS = [
        "Horizontal_Distance_To_Hydrology",
        "Vertical_Distance_To_Hydrology",
    ]

SCALE_01_FEATURE_KEYS = [
        "Hillshade_9am",
        "Hillshade_Noon",
        "Horizontal_Distance_To_Fire_Points",
    ]

SCALE_Z_FEATURE_KEYS = [
        "Elevation",
        "Slope",
        "Horizontal_Distance_To_Roadways",
    ]

VOCAB_FEATURE_KEYS = ["Wilderness_Area"]

HASH_STRING_FEATURE_KEYS = ["Soil_Type"]

LABEL_KEY = "Cover_Type"

# Utility function for renaming the feature
def transformed_name(key):
    return key + '_xf'
```

```
Overwriting cover_constants.py
```

Next you will define the `preprocessing_fn` to apply transformations to the features.

Exercise 10: Preprocessing function

Complete the module to transform your features. Refer to the code comments to get hints on what operations to perform.

Here are some links to the docs of the functions you will need to complete this function:

- `tft.scale_by_min_max`
- `tft.scale_to_0_1`
- `tft.scale_to_z_score`
- `tft.compute_and_apply_vocabulary`
- `tft.hash_strings`

```
# Set the transform module filename
_cover_transform_module_file = 'cover_transform.py'
```

```
%%writefile {_cover_transform_module_file}

import tensorflow as tf
import tensorflow_transform as tft

import cover_constants

_SCALE_MINMAX_FEATURE_KEYS = cover_constants.SCALE_MINMAX_FEATURE_KEYS
_SCALE_01_FEATURE_KEYS = cover_constants.SCALE_01_FEATURE_KEYS
_SCALE_Z_FEATURE_KEYS = cover_constants.SCALE_Z_FEATURE_KEYS
_VOCAB_FEATURE_KEYS = cover_constants.VOCAB_FEATURE_KEYS
_HASH_STRING_FEATURE_KEYS = cover_constants.HASH_STRING_FEATURE_KEYS
_LABEL_KEY = cover_constants.LABEL_KEY
_transformed_name = cover_constants.transformed_name

def preprocessing_fn(inputs):

    features_dict = {}

    ### START CODE HERE ###
    for feature in _SCALE_MINMAX_FEATURE_KEYS:
        data_col = inputs[feature]
        # Transform using scaling of min_max function
        # Hint: Use tft.scale_by_min_max by passing in the respective column
        features_dict[_transformed_name(feature)] = tft.scale_by_min_max(data_col)

    for feature in _SCALE_01_FEATURE_KEYS:
        data_col = inputs[feature]
        # Transform using scaling of 0 to 1 function
        # Hint: tft.scale_to_0_1
        features_dict[_transformed_name(feature)] = tft.scale_to_0_1(data_col)

    for feature in _SCALE_Z_FEATURE_KEYS:
        data_col = inputs[feature]
        # Transform using scaling to z score
        # Hint: tft.scale_to_z_score
        features_dict[_transformed_name(feature)] = tft.scale_to_z_score(data_col)

    for feature in _VOCAB_FEATURE_KEYS:
        data_col = inputs[feature]
        # Transform using vocabulary available in column
        # Hint: Use tft.compute_and_apply_vocabulary
        features_dict[_transformed_name(feature)] = tft.compute_and_apply_vocabulary

    for feature in _HASH_STRING_FEATURE_KEYS:
        data_col = inputs[feature]
        # Transform by hashing strings into buckets
        # Hint: Use tft.hash_strings with the param hash_buckets set to 10
        features_dict[_transformed_name(feature)] = tft.hash_strings(data_col, hash_

    ### END CODE HERE ###

    # No change in the label
    features_dict[_LABEL_KEY] = inputs[_LABEL_KEY]

    return features_dict
```

```
Overwriting cover_transform.py
```

Exercise 11: Transform

Use the TFX Transform component to perform the transformations and generate the transformation graph. You will need to pass in the dataset examples, *curated* schema, and the module that contains the preprocessing function.

```
### START CODE HERE ###
# Instantiate the Transform component
transform = Transform(examples=example_gen.outputs['examples'],
                      schema=user_schema_importer.outputs['result'],
                      module_file=os.path.abspath(_cover_transform_module_file)
                     )



### END CODE HERE ###

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

```
WARNING:root:This output type hint will be ignored and not used for type-checking pur
poses. Typically, output type hints for a PTransform are single (or nested) types wra
pped by a PCollection, PDone, or None. Got: Tuple[Dict[str, Union[NoneType, _Datase
t]], Union[Dict[str, Dict[str, PCollection]], NoneType]] instead.
WARNING:root:This output type hint will be ignored and not used for type-checking pur
poses. Typically, output type hints for a PTransform are single (or nested) types wra
pped by a PCollection, PDone, or None. Got: Tuple[Dict[str, Union[NoneType, _Datase
t]], Union[Dict[str, Dict[str, PCollection]], NoneType]] instead.
WARNING:apache_beam.typehints.typehints:Ignoring send_type hint: <class 'NoneType'>
WARNING:apache_beam.typehints.typehints:Ignoring return_type hint: <class 'NoneType'>
WARNING:apache_beam.typehints.typehints:Ignoring send_type hint: <class 'NoneType'>
WARNING:apache_beam.typehints.typehints:Ignoring return_type hint: <class 'NoneType'>
WARNING:apache_beam.typehints.typehints:Ignoring send_type hint: <class 'NoneType'>
WARNING:apache_beam.typehints.typehints:Ignoring return_type hint: <class 'NoneType'>
WARNING:apache_beam.typehints.typehints:Ignoring send_type hint: <class 'NoneType'>
WARNING:apache_beam.typehints.typehints:Ignoring return_type hint: <class 'NoneType'>
WARNING:apache_beam.typehints.typehints:Ignoring send_type hint: <class 'NoneType'>
WARNING:apache_beam.typehints.typehints:Ignoring return_type hint: <class 'NoneType'>
WARNING:apache_beam.typehints.typehints:Ignoring send_type hint: <class 'NoneType'>
WARNING:apache_beam.typehints.typehints:Ignoring return_type hint: <class 'NoneType'>
```

▼**ExecutionResult** at 0x7fa4f05f9040

.execution_id	14
.component	▶ **Transform** at 0x7fa4f04709a0
.component.inputs	**['examples']** ▶ **Channel** of type **'Examples'** (1 artifact) at 0x7fa59cbc8580 **['schema']** ▶ **Channel** of type **'Schema'** (1 artifact) at 0x7fa59cc13a30
.component.outputs	**['transform_graph']** ▶ **Channel** of type **'TransformGraph'** (1 artifact) at 0x7fa4f0470eb0 **['transformed_examples']** ▶ **Channel** of type **'Examples'** (1 artifact) at 0x7fa4f0470520

['updated_analyzer_cache'] ▶ **Channel** of type **'TransformCache'** (1 artifact) at 0x7fa4f04709d0

Let's inspect a few examples of the transformed dataset to see if the transformations are done correctly.

```
try:
    transform_uri = transform.outputs['transformed_examples'].get()[0].uri

# for grading since context.run() does not work outside the notebook
except IndexError:
    print("context.run() was no-op")
    examples_path = './pipeline/Transform/transformed_examples'
    dir_id = os.listdir(examples_path)[0]
    transform_uri = f'{examples_path}/{dir_id}'
```

```
# Get the URI of the output artifact representing the transformed examples
train_uri = os.path.join(transform_uri, 'train')

# Get the list of files in this directory (all compressed TFRecord files)
tfrecord_filenames = [os.path.join(train_uri, name)
                      for name in os.listdir(train_uri)]

# Create a `TFRecordDataset` to read these files
transformed_dataset = tf.data.TFRecordDataset(tfrecord_filenames, compression_type="
```

```
# import helper function to get examples from the dataset
from util import get_records

# Get 3 records from the dataset
sample_records_xf = get_records(transformed_dataset, 3)

# Print the output
pp.pprint(sample_records_xf)
```

```
[{'features': {'feature': {'Cover_Type': {'int64List': {'value': ['4']}},
                           'Elevation_xf': {'floatList': {'value': [-1.2982628]}},
                           'Hillshade_9am_xf': {'floatList': {'value': [0.8700787
4]}},
                           'Hillshade_Noon_xf': {'floatList': {'value': [0.913385
8]}},
                           'Horizontal_Distance_To_Fire_Points_xf': {'floatList': {'v
alue': [0.875366]}},
                           'Horizontal_Distance_To_Hydrology_xf': {'floatList': {'val
ue': [0.18468146]}},
                           'Horizontal_Distance_To_Roadways_xf': {'floatList': {'valu
e': [-1.1803539]}},
                           'Slope_xf': {'floatList': {'value': [-1.483387]}},
                           'Soil_Type_xf': {'int64List': {'value': ['4']}},
                           'Vertical_Distance_To_Hydrology_xf': {'floatList': {'value
': [0.22351421]}},
                           'Wilderness_Area_xf': {'int64List': {'value': ['0']}}}}},
 {'features': {'feature': {'Cover_Type': {'int64List': {'value': ['4']}},
                           'Elevation_xf': {'floatList': {'value': [-1.3197033]}},
                           'Hillshade_9am_xf': {'floatList': {'value': [0.8661417
4]}},
                           'Hillshade_Noon_xf': {'floatList': {'value': [0.925196
8]}},
                           'Horizontal_Distance_To_Fire_Points_xf': {'floatList': {'v
alue': [0.8678377]}},
                           'Horizontal_Distance_To_Hydrology_xf': {'floatList': {'val
ue': [0.15175375]}},
                           'Horizontal_Distance_To_Roadways_xf': {'floatList': {'valu
e': [-1.2572862]}},
                           'Slope_xf': {'floatList': {'value': [-1.6169325]}},
                           'Soil_Type_xf': {'int64List': {'value': ['4']}},
                           'Vertical_Distance_To_Hydrology_xf': {'floatList': {'value
': [0.21576227]}},
                           'Wilderness_Area_xf': {'int64List': {'value': ['0']}}}}},
 {'features': {'feature': {'Cover_Type': {'int64List': {'value': ['1']}},
                           'Elevation_xf': {'floatList': {'value': [-0.5549895]}},
                           'Hillshade_9am_xf': {'floatList': {'value': [0.9212598]}},
                           'Hillshade_Noon_xf': {'floatList': {'value': [0.9370078
4]}},
                           'Horizontal_Distance_To_Fire_Points_xf': {'floatList': {'v
alue': [0.8533389]}},
                           'Horizontal_Distance_To_Hydrology_xf': {'floatList': {'val
ue': [0.19183965]}},
                           'Horizontal_Distance_To_Roadways_xf': {'floatList': {'valu
e': [0.53138816]}},
                           'Slope_xf': {'floatList': {'value': [-0.6821134]}},
                           'Soil_Type_xf': {'int64List': {'value': ['4']}},
                           'Vertical_Distance_To_Hydrology_xf': {'floatList': {'value
': [0.30749354]}},
                           'Wilderness_Area_xf': {'int64List': {'value': ['0']}}}}}]
```

5 - ML Metadata

TFX uses ML Metadata under the hood to keep records of artifacts that each component uses. This makes it easier to track how the pipeline is run so you can troubleshoot if needed or want to reproduce results.

In this final section of the assignment, you will demonstrate going through this metadata store to retrieve related artifacts. This skill is useful for when you want to recall which inputs are fed to a particular stage of the pipeline. For example, you can know where to locate the schema used to perform feature transformation, or you can determine which set of examples were used to train a model.

You will start by importing the relevant modules and setting up the connection to the metadata store. We have also provided some helper functions for displaying artifact information and you can review its code in the external `util.py` module in your lab workspace.

```python
# Import mlmd and utilities
import ml_metadata as mlmd
from ml_metadata.proto import metadata_store_pb2
from util import display_types, display_artifacts, display_properties

# Get the connection config to connect to the metadata store
connection_config = context.metadata_connection_config

# Instantiate a MetadataStore instance with the connection config
store = mlmd.MetadataStore(connection_config)

# Declare the base directory where All TFX artifacts are stored
base_dir = connection_config.sqlite.filename_uri.split('metadata.sqlite')[0]
```

5.1 - Accessing stored artifacts

With the connection setup, you can now interact with the metadata store. For instance, you can retrieve all artifact types stored with the `get_artifact_types()` function. For reference, the API is documented

```
# Get the artifact types
types = store.get_artifact_types()

# Display the results
display_types(types)
```

	id	name
0	5	Examples
1	7	ExampleStatistics
2	9	Schema
3	12	ExampleAnomalies
4	14	TransformGraph
5	15	TransformCache

You can also get a list of artifacts for a particular type to see if there are variations used in the pipeline. For example, you curated a schema in an earlier part of the assignment so this should appear in the records. Running the cell below should show at least two rows: one for the inferred schema, and another for the updated schema. If you ran this notebook before, then you might see more rows because of the different schema artifacts saved under the `./SchemaGen/schema` directory.

```
# Retrieve the transform graph list
schema_list = store.get_artifacts_by_type('Schema')

# Display artifact properties from the results
display_artifacts(store, schema_list, base_dir)
```

	artifact id	type	uri
0	3	Schema	./SchemaGen/schema/3
1	4	Schema	./updated_schema
2	7	Schema	./SchemaGen/schema/7

Moreover, you can also get the properties of a particular artifact. TFX declares some properties automatically for each of its components. You will most likely see `name`, `state` and `producer_component` for each artifact type. Additional properties are added where appropriate. For example, a `split_names` property is added in `ExampleStatistics` artifacts to indicate which splits the statistics are generated for.

```
# Get the latest TransformGraph artifact
statistics_artifact = store.get_artifacts_by_type('ExampleStatistics')[-1]

# Display the properties of the retrieved artifact
display_properties(store, statistics_artifact)
```

	property	value
0	split_names	["train", "eval"]

	property	value
1	name	statistics
2	state	published

5.2 - Tracking artifacts

For this final exercise, you will build a function to return the parent artifacts of a given one. For example, this should be able to list the artifacts that were used to generate a particular `TransformGraph` instance.

Exercise 12: Get parent artifacts

Complete the code below to track the inputs of a particular artifact.

Tips:

- You may find get_events_by_artifact_ids() and get_events_by_execution_ids() useful here.
- Some of the methods of the MetadataStore class (such as the two given above) only accepts iterables so remember to convert to a list (or set) if you only have an int (e.g. pass `[x]` instead of `x`).

```
def get_parent_artifacts(store, artifact):

    ### START CODE HERE ###

    # Get the artifact id of the input artifact
    artifact_id = artifact.id
    #artifact_id = store.get_artifact_by_id(artifact).id

    # Get events associated with the artifact id
    artifact_id_events = store.get_events_by_artifact_ids([artifact_id])

    # From the `artifact_id_events`, get the execution ids of OUTPUT events.
    # Cast to a set to remove duplicates if any.
    execution_id = set(
        event.execution_id
        for event in artifact_id_events
        if event.type == metadata_store_pb2.Event.OUTPUT
    )

    # Get the events associated with the execution_id
    execution_id_events = store.get_events_by_execution_ids(execution_id)

    # From execution_id_events, get the artifact ids of INPUT events.
    # Cast to a set to remove duplicates if any.
    parent_artifact_ids = set(
        event.artifact_id
        for event in execution_id_events
        if event.type == metadata_store_pb2.Event.INPUT
    )


    # Get the list of artifacts associated with the parent_artifact_ids
    parent_artifact_list = store.get_artifacts_by_id(parent_artifact_ids)


    ### END CODE HERE ###

    return parent_artifact_list
```

```
# Get an artifact instance from the metadata store
artifact_instance = store.get_artifacts_by_type('TransformGraph')[0]

# Retrieve the parent artifacts of the instance
parent_artifacts = get_parent_artifacts(store, artifact_instance)

# Display the results
display_artifacts(store, parent_artifacts, base_dir)
```

	artifact id	type	uri
0	4	Schema	./updated_schema
1	5	Examples	./CsvExampleGen/examples/5

Expected Output:

Note: The ID numbers may differ.

artifact id	type	uri

artifact id	type	uri
1	Examples	./CsvExampleGen/examples/1

Congratulations! You have now completed the assignment for this week. You've demonstrated your skills in selecting features, performing a data pipeline, and retrieving information from the metadata store. Having the ability to put these all together will be critical when working with production grade machine learning projects. For next week, you will work on more data types and see how these can be prepared in an ML pipeline. **Keep it up!**