C2_W2_Lab_1_Simple_Feature_Engineering

January 1, 2025

1 Ungraded Lab: Simple Feature Engineering

In this lab, you will get some hands-on practice with the Tensorflow Transform library (or `tf.Transform`). This serves to show what is going on under the hood when you get to use the TFX Transform component within a TFX pipeline in the next labs. The code snippets and main discussions are taken from this TensorFlow official notebook but we have expounded on a few key points.

Preprocessing is often required in ML projects because the raw data is not yet in a suitable format for training a model. Not doing so usually results in the model not converging or having poor performance. Some standard transformations include normalizing pixel values, bucketizing, one-hot encoding, and the like. Consequently, these same transformations should also be done during inference to ensure that the model is computing the correct predictions.

With Tensorflow Transform, you can preprocess data using the same code for both training a model and serving inferences in production. It provides several utility functions for common preprocessing tasks including creating features that require a full pass over the training dataset. The outputs are the transformed features and a TensorFlow graph which you can use for both training and serving. Using the same graph for both training and serving can prevent feature skew, since the same transformations are applied in both stages.

For this introductory exercise, you will walk through the "Hello World" of using TensorFlow Transform to preprocess input data. As you've seen in class, the main steps are to:

1. Collect raw data
2. Define metadata
3. Create a preprocessing function
4. Generate a constant graph with the required transformations

Let's begin!

1.1 Imports

[1]:
```
import tensorflow as tf
import tensorflow_transform as tft
import tensorflow_transform.beam as tft_beam

from tensorflow_transform.tf_metadata import dataset_metadata
from tensorflow_transform.tf_metadata import schema_utils
```

```
import pprint
import tempfile

print(f'TensorFlow version: {tf.__version__}')
print(f'TFX Transform version: {tft.__version__}')
```

```
TensorFlow version: 2.6.0
TFX Transform version: 1.3.0
```

1.2 Collect raw data

First, you will need to load your data. For simplicity, we will not use a real dataset in this exercise. You will do that in the next lab. For now, you will just use dummy data so you can inspect the transformations more easily.

[2]:
```
# define sample data
raw_data = [
      {'x': 1, 'y': 1, 's': 'hello'},
      {'x': 2, 'y': 2, 's': 'world'},
      {'x': 3, 'y': 3, 's': 'hello'}
  ]
```

1.3 Define the metadata

Next, you will define the metadata. This contains the schema that tells the types of each feature column (or key) in `raw_data`. You need to take note of a few things:

- The transform function later expects the metadata to be packed in a DatasetMetadata object.
- The constructor for the `DatasetMetadata` class expects a Schema protocol buffer data type. You can use the schema_from_feature_spec() method to generate that from a dictionary.
- To build the said dictionary, you will use the keys/column names of `raw_data` and assign a FeatureSpecType as values. This allows you to specify if the input is fixed or variable length (using tf.io classes), as well as to define the shape and data type.

See how this is implemented in the cell below.

[3]:
```
# define the schema as a DatasetMetadata object
raw_data_metadata = dataset_metadata.DatasetMetadata(

    # use convenience function to build a Schema protobuf
    schema_utils.schema_from_feature_spec({

        # define a dictionary mapping the keys to its feature spec type
        'y': tf.io.FixedLenFeature([], tf.float32),
        'x': tf.io.FixedLenFeature([], tf.float32),
        's': tf.io.FixedLenFeature([], tf.string),
    }))

raw_data_metadata
```

[3]:
```
{'_schema': feature {
  name: "s"
  type: BYTES
  presence {
    min_fraction: 1.0
  }
  shape {
  }
}
feature {
  name: "x"
  type: FLOAT
  presence {
    min_fraction: 1.0
  }
  shape {
  }
}
feature {
  name: "y"
  type: FLOAT
  presence {
    min_fraction: 1.0
  }
  shape {
  }
}
}
```

[4]:
```
# preview the schema
print(raw_data_metadata._schema)
```

```
feature {
  name: "s"
  type: BYTES
  presence {
    min_fraction: 1.0
  }
  shape {
  }
}
feature {
  name: "x"
  type: FLOAT
  presence {
    min_fraction: 1.0
  }
```

```
  shape {
  }
}
feature {
  name: "y"
  type: FLOAT
  presence {
    min_fraction: 1.0
  }
  shape {
  }
}
```

1.4 Create a preprocessing function

The *preprocessing function* is the most important concept of `tf.Transform`. A preprocessing function is where the transformation of the dataset really happens. It accepts and returns a dictionary of tensors, where a tensor means a Tensor or SparseTensor. There are two main groups of API calls that typically form the heart of a preprocessing function:

1. **TensorFlow Ops:** Any function that accepts and returns tensors. These add TensorFlow operations to the graph that transforms raw data into transformed data one feature vector at a time. These will run for every example, during both training and serving.
2. **TensorFlow Transform Analyzers:** Any of the analyzers provided by `tf.Transform`. Analyzers also accept and return tensors, but unlike TensorFlow ops they only run once during training, and typically make a full pass over the entire training dataset. They create tensor constants, which are added to your graph. For example, `tft.min` computes the minimum of a tensor over the training dataset.

Caution: When you apply your preprocessing function to serving inferences, the constants that were created by analyzers during training do not change. If your data has trend or seasonality components, plan accordingly.

You can see available functions to transform your data here.

[5]:
```
def preprocessing_fn(inputs):
    """Preprocess input columns into transformed columns."""

    # extract the columns and assign to local variables
    x = inputs['x']
    y = inputs['y']
    s = inputs['s']

    # data transformations using tft functions
    x_centered = x - tft.mean(x)
    y_normalized = tft.scale_to_0_1(y)
    s_integerized = tft.compute_and_apply_vocabulary(s)
    x_centered_times_y_normalized = (x_centered * y_normalized)
```

```
    # return the transformed data
    return {
        'x_centered': x_centered,
        'y_normalized': y_normalized,
        's_integerized': s_integerized,
        'x_centered_times_y_normalized': x_centered_times_y_normalized,
    }
```

1.5 Generate a constant graph with the required transformations

Now you're ready to put everything together and transform your data. Like TFDV last week, Tensorflow Transform also uses Apache Beam for deployment scalability and flexibility. As you'll see below, Beam uses the pipe (|) operator to stack the different stages of the pipeline. In this case, you will just feed the data (and metadata) to the AnalyzeAndTransformDataset class and use the preprocessing function you defined above to transform the data.

For a closer look at Beam syntax for tranform pipelines, you can refer to the documentation here and try the short Colab here.

Note: You can safely ignore the warning about unparseable args shown after running the cell below.

[6]:
```
# Ignore the warnings
tf.get_logger().setLevel('ERROR')

# a temporary directory is needed when analyzing the data
with tft_beam.Context(temp_dir = tempfile.mkdtemp()):

    # define the pipeline using Apache Beam syntax
    transformed_dataset, transform_fn = (

        # analyze and transform the dataset using the preprocessing function
        (raw_data, raw_data_metadata) | tft_beam.AnalyzeAndTransformDataset(
            preprocessing_fn)
    )

# unpack the transformed dataset
transformed_data, transformed_metadata = transformed_dataset

# print the results
print('\nRaw data:\n{}\n'.format(pprint.pformat(raw_data)))
print('Transformed data:\n{}'.format(pprint.pformat(transformed_data)))
```

```
WARNING:apache_beam.options.pipeline_options:Discarding unparseable args:
['/opt/conda/lib/python3.8/site-packages/ipykernel_launcher.py', '-f',
'/home/jovyan/.local/share/jupyter/runtime/kernel-
edb86a81-18b4-43a5-9b10-3ec930be80fc.json']
WARNING:root:Make sure that locally built Python SDK docker image has Python 3.8
interpreter.
```

```
Raw data:
[{'s': 'hello', 'x': 1, 'y': 1},
 {'s': 'world', 'x': 2, 'y': 2},
 {'s': 'hello', 'x': 3, 'y': 3}]

Transformed data:
[{'s_integerized': 0,
  'x_centered': -1.0,
  'x_centered_times_y_normalized': -0.0,
  'y_normalized': 0.0},
 {'s_integerized': 1,
  'x_centered': 0.0,
  'x_centered_times_y_normalized': 0.0,
  'y_normalized': 0.5},
 {'s_integerized': 0,
  'x_centered': 1.0,
  'x_centered_times_y_normalized': 1.0,
  'y_normalized': 1.0}]
```

1.6 Is this the right answer?

Previously, you used `tf.Transform` to do this:

```
x_centered = x - tft.mean(x)
y_normalized = tft.scale_to_0_1(y)
s_integerized = tft.compute_and_apply_vocabulary(s)
x_centered_times_y_normalized = (x_centered * y_normalized)
```

x_centered With input of `[1, 2, 3]` the mean of `x` is 2, and you subtract it from `x` to center your `x` values at 0. So the result of `[-1.0, 0.0, 1.0]` is correct. #### y_normalized Next, you scaled your `y` values between 0 and 1. Your input was `[1, 2, 3]` so the result of `[0.0, 0.5, 1.0]` is correct. #### s_integerized You mapped your strings to indexes in a vocabulary, and there were only 2 words in your vocabulary ("hello" and "world"). So with input of `["hello", "world", "hello"]` the result of `[0, 1, 0]` is correct. #### x_centered_times_y_normalized You created a new feature by crossing `x_centered` and `y_normalized` using multiplication. Note that this multiplies the results, not the original values, and the new result of `[-0.0, 0.0, 1.0]` is correct.

1.6.1 Wrap Up

In this lab, you went through the fundamentals of using Tensorflow Transform to turn raw data into features. This code can be used to transform both the training and serving data. However, the code can be quite complex if you'll be using this as a standalone library to build a pipeline (see this notebook for reference). Now that you know what is going on under the hood, you can use a higher-level set of tools like Tensorflow Extended to simplify the process. This will abstract some of the steps you did here like manually defining schemas and using `tft_beam` functions. It will also leverage other libraries, such as TFDV, to perform other processes in the usual machine learning pipeline like detecting anomalies. You will get to see these in the next lab.