

Week 2 Assignment: Feature Engineering

For this week's assignment, you will build a data pipeline using [Tensorflow Extended \(TFX\)](#) to prepare features from the [Metro Interstate Traffic Volume dataset](#). Try to only use the documentation and code hints to accomplish the tasks but feel free to review the 2nd ungraded lab this week in case you get stuck.

Upon completion, you will have:

- created an InteractiveContext to run TFX components interactively
- used TFX ExampleGen component to split your dataset into training and evaluation datasets
- generated the statistics and the schema of your dataset using TFX StatisticsGen and SchemaGen components
- validated the evaluation dataset statistics using TFX ExampleValidator
- performed feature engineering using the TFX Transform component

Let's begin!

Table of Contents

- [1 - Setup](#)
 - [1.1 - Imports](#)
 - [1.2 - Define Paths](#)
 - [1.3 - Preview the Dataset](#)
 - [1.4 - Create the InteractiveContext](#)
- [2 - Run TFX components interactively](#)
 - [2.1 - ExampleGen](#)
 - [Exercise 1 - ExampleGen](#)
 - [Exercise 2 - get_records\(\)](#)
 - [2.2 - StatisticsGen](#)
 - [Exercise 3 - StatisticsGen](#)
 - [2.3 - SchemaGen](#)
 - [Exercise 4 - SchemaGen](#)
 - [2.4 - ExampleValidator](#)
 - [Exercise 5 - ExampleValidator](#)
 - [2.5 - Transform](#)
 - [Exercise 6 - preprocessing_fn\(\)](#)
 - [Exercise 7 - Transform](#)

1 - Setup

As usual, you will first need to import the necessary packages. For reference, the lab environment uses *TensorFlow version: 2.3.1* and *TFX version: 0.24.0*.

1.1 Imports

```
import tensorflow as tf

from tfx.components import import CsvExampleGen
from tfx.components import import ExampleValidator
from tfx.components import import SchemaGen
from tfx.components import import StatisticsGen
from tfx.components import import Transform

from tfx.orchestration.experimental.interactive.interactive_context import import InteractiveContext
from google.protobuf.json_format import import MessageToDict

import os
import pprint

pp = pprint.PrettyPrinter()
```

1.2 - Define paths

You will define a few global variables to indicate paths in the local workspace.

```
# location of the pipeline metadata store
_pipeline_root = 'metro_traffic_pipeline/'

# directory of the raw data files
_data_root = 'metro_traffic_pipeline/data'

# path to the raw training data
_data_filepath = os.path.join(_data_root, 'metro_traffic_volume.csv')
```

1.3 - Preview the dataset

The [Metro Interstate Traffic Volume dataset](#) contains hourly traffic volume of a road in Minnesota from 2012-2018. With this data, you can develop a model for predicting the traffic volume given the date, time, and weather conditions. The attributes are:

- **holiday** - US National holidays plus regional holiday, Minnesota State Fair
- **temp** - Average temp in Kelvin
- **rain_1h** - Amount in mm of rain that occurred in the hour
- **snow_1h** - Amount in mm of snow that occurred in the hour
- **clouds_all** - Percentage of cloud cover
- **weather_main** - Short textual description of the current weather
- **weather_description** - Longer textual description of the current weather
- **date_time** - DateTime Hour of the data collected in local CST time
- **traffic_volume** - Numeric Hourly I-94 ATR 301 reported westbound traffic volume
- **month** - taken from date_time
- **day** - taken from date_time
- **day_of_week** - taken from date_time
- **hour** - taken from date_time

Disclaimer: We added the last four attributes shown above (i.e. month, day, day_of_week, hour) to the original dataset to increase the features you can transform later.

Take a quick look at the first few rows of the CSV file.

```
# Preview the dataset
!head {_data_filepath}
```

```
holiday,temp,rain_1h,snow_1h,clouds_all,weather_main,weather_description,date_time,traffic_volume,month,day,day_of
_week,hour
None,288.28,0.0,0.0,40,Clouds,scattered clouds,2012-10-02 09:00:00,5545,10,2,1,9
None,289.36,0.0,0.0,75,Clouds,broken clouds,2012-10-02 10:00:00,4516,10,2,1,10
None,289.58,0.0,0.0,90,Clouds,overcast clouds,2012-10-02 11:00:00,4767,10,2,1,11
None,290.13,0.0,0.0,90,Clouds,overcast clouds,2012-10-02 12:00:00,5026,10,2,1,12
None,291.14,0.0,0.0,75,Clouds,broken clouds,2012-10-02 13:00:00,4918,10,2,1,13
None,291.72,0.0,0.0,1,Clear,sky is clear,2012-10-02 14:00:00,5181,10,2,1,14
None,293.17,0.0,0.0,1,Clear,sky is clear,2012-10-02 15:00:00,5584,10,2,1,15
None,293.86,0.0,0.0,1,Clear,sky is clear,2012-10-02 16:00:00,6015,10,2,1,16
None,294.14,0.0,0.0,20,Clouds,few clouds,2012-10-02 17:00:00,5791,10,2,1,17
```

1.4 - Create the InteractiveContext

You will need to initialize the `InteractiveContext` to enable running the TFX components interactively. As before, you will let it create the metadata store in the `_pipeline_root` directory. You can safely ignore the warning about the missing metadata config file.

```
# Declare the InteractiveContext and use a local sqlite file as the metadata store.
# You can ignore the warning about the missing metadata config file
context = InteractiveContext(pipeline_root=_pipeline_root)
```

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

2 - Run TFX components interactively

In the following exercises, you will create the data pipeline components one-by-one, run each of them, and visualize their output artifacts. Recall that we refer to the outputs of pipeline components as *artifacts* and these can be inputs to the next stage of the pipeline.

2.1 - ExampleGen

The pipeline starts with the [ExampleGen](#) component. It will:

- split the data into training and evaluation sets (by default: 2/3 train, 1/3 eval).
- convert each data row into `tf.train.Example` format. This [protocol buffer](#) is designed for Tensorflow operations and is used by the TFX components.
- compress and save the data collection under the `_pipeline_root` directory for other components to access. These examples are stored in `TFRecord` format. This optimizes read and write operations within Tensorflow especially if you have a large collection of data.

Exercise 1: ExampleGen

Fill out the code below to ingest the data from the CSV file stored in the `_data_root` directory.

```
### START CODE HERE

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

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


### END CODE HERE
```

▼ **ExecutionResult** at 0x7ff545c58070

.execution_id 8

.component  **CsvExampleGen** at 0x7ff4ac8d42b0

.component.inputs {}

.component.outputs **['examples']**  **Channel** of type '**Examples**' (1 artifact) at 0x7ff4ac8d42e0

You should see the output cell of the `InteractiveContext` above showing the metadata associated with the component execution. You can expand the items under `.component.outputs` and see that an `Examples` artifact for the train and eval split is created in `metro_traffic_pipeline/CsvExampleGen/examples/{execution_id}`.

You can also check that programmatically with the following snippet. You can focus on the `try` block. The `except` and `else` block is needed mainly for grading. `context.run()` yields no operation when executed in a non-interactive environment (such as the grader script that runs outside of this notebook). In such scenarios, the URI must be manually set to avoid errors.

```
try:
    # get the artifact object
    artifact = example_gen.outputs['examples'].get()[0]

    # print split names and uri
    print(f'split names: {artifact.split_names}')
    print(f'artifact uri: {artifact.uri}')

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

else:
    artifact_uri = artifact.uri
```

```
split names: ["train", "eval"]
artifact uri: metro_traffic_pipeline/CsvExampleGen/examples/8
```

The ingested data has been saved to the directory specified by the artifact Uniform Resource Identifier (URI). As a sanity check, you can take a look at some of the training examples. This requires working with Tensorflow data types, particularly `tf.train.Example` and `TFRecord` (you can read more about them [here](#)). Let's first load the `TFRecord` into a variable:

```
# Get the URI of the output artifact representing the training examples, which is a directory
train_uri = os.path.join(artifact_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
dataset = tf.data.TFRecordDataset(tfrecord_filenames, compression_type="GZIP")
```

Exercise 2: get_records()

Complete the helper function below to return a specified number of examples.

Hints: You may find the [MessageToDict](#) helper function and `tf.train.Example`'s [ParseFromString\(\)](#) method useful here. You can also refer [here](#) for a refresher on `TFRecord` and `tf.train.Example()`

```
def get_records(dataset, num_records):
    """Extracts records from the given dataset.
    Args:
        dataset (TFRecordDataset): dataset saved by ExampleGen
        num_records (int): number of records to preview
    """

    # initialize an empty list
    records = []

    ### START CODE HERE
    # Use the `take()` method to specify how many records to get
    for tfrecord in dataset.take(num_records):

        # Get the numpy property of the tensor
        serialized_example = tfrecord.numpy()

        # Initialize a `tf.train.Example()` to read the serialized data
        example = tf.train.Example()

        # Read the example data (output is a protocol buffer message)
        example.ParseFromString(serialized_example)

        # convert the protocol bufffer message to a Python dictionary
        example_dict = MessageToDict(example)

        # append to the records list
        records.append(example_dict)

    ### END CODE HERE
    return records

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

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

```
[{'features': {'feature': {'clouds_all': {'int64List': {'value': ['40']}},
                           'date_time': {'bytesList': {'value': ['MjAxMi0xMCOwMiAwOTowMDowMA==']}},
                           'day': {'int64List': {'value': ['2']}},
                           'day_of_week': {'int64List': {'value': ['1']}},
                           'holiday': {'bytesList': {'value': ['Tm9uZQ==']}},
                           'hour': {'int64List': {'value': ['9']}},
                           'month': {'int64List': {'value': ['10']}},
                           'rain_1h': {'floatList': {'value': [0.0]}},
                           'snow_1h': {'floatList': {'value': [0.0]}},
                           'temp': {'floatList': {'value': [288.28]}},
                           'traffic_volume': {'int64List': {'value': ['5545']}},
                           'weather_description': {'bytesList': {'value': ['c2NhdHRlcmVkJGNsb3Vkcw==']}},
                           'weather_main': {'bytesList': {'value': ['Q2xvdWRz']}}}}},
 {'features': {'feature': {'clouds_all': {'int64List': {'value': ['75']}},
                           'date_time': {'bytesList': {'value': ['MjAxMi0xMCOwMiAxMDowMDowMA==']}},
                           'day': {'int64List': {'value': ['2']}},
                           'day_of_week': {'int64List': {'value': ['1']}},
                           'holiday': {'bytesList': {'value': ['Tm9uZQ==']}},
                           'hour': {'int64List': {'value': ['10']}},
                           'month': {'int64List': {'value': ['10']}},
                           'rain_1h': {'floatList': {'value': [0.0]}},
                           'snow_1h': {'floatList': {'value': [0.0]}},
                           'temp': {'floatList': {'value': [289.36]}},
                           'traffic_volume': {'int64List': {'value': ['4516']}},
                           'weather_description': {'bytesList': {'value': ['YnJva2VuIGNsb3Vkcw==']}},
                           'weather_main': {'bytesList': {'value': ['Q2xvdWRz']}}}}}]
```

```
{'features': {'feature': {'clouds_all': {'int64List': {'value': ['90']}},
    'date_time': {'bytesList': {'value': ['MjAxMi0xMC0wMiAxMTowMDowMA==']}},
    'day': {'int64List': {'value': ['2']}},
    'day_of_week': {'int64List': {'value': ['1']}},
    'holiday': {'bytesList': {'value': ['Tm9uZQ==']}},
    'hour': {'int64List': {'value': ['11']}},
    'month': {'int64List': {'value': ['10']}},
    'rain_1h': {'floatList': {'value': [0.0]}},
    'snow_1h': {'floatList': {'value': [0.0]}},
    'temp': {'floatList': {'value': [289.58]}},
    'traffic_volume': {'int64List': {'value': ['4767']}},
    'weather_description': {'bytesList': {'value': ['b3ZlcmNhc3QgY2xvdWRz']}},
    'weather_main': {'bytesList': {'value': ['02xvdWRz']}}}}}
```

You should see three of the examples printed above. Now that `ExampleGen` has finished ingesting the data, the next step is data analysis.

2.2 - StatisticsGen

The `StatisticsGen` component computes statistics over your dataset for data analysis, as well as for use in downstream components. It uses the `TensorFlow Data Validation` library.

`StatisticsGen` takes as input the dataset ingested using `CsvExampleGen`.

Exercise 3: StatisticsGen

Fill the code below to generate statistics from the output examples of `CsvExampleGen`.

```
### 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 0x7ff4ac8d4eb0

.execution_id 9

.component  **StatisticsGen** at 0x7ff5458453a0

.component.inputs **['examples']**  **Channel** of type '**Examples**' (1 artifact) at 0x7ff4ac8d42e0

.component.outputs **['statistics']**  **Channel** of type '**ExampleStatistics**' (1 artifact) at 0x7ff545845be0

```
# Plot the statistics generated
context.show(statistics_gen.outputs['statistics'])
```

Artifact at metro_traffic_pipeline/StatisticsGen/statistics/9

'train' split:

```
WARNING:tensorflow:From /opt/conda/lib/python3.8/site-packages/tensorflow_data_validation/utils/stats_util.py:229:
tf_record_iterator (from tensorflow.python.lib.io.tf_record) is deprecated and will be removed in a future versio
n.
Instructions for updating:
Use eager execution and:
`tf.data.TFRecordDataset(path)`
```



'eval' split:

2.3 - SchemaGen

The [SchemaGen](#) component also uses TFDV to generate a schema based on your data statistics. As you've learned previously, a schema defines the expected bounds, types, and properties of the features in your dataset.

`SchemaGen` will take as input the statistics that we generated with `StatisticsGen`, looking at the training split by default.

Exercise 4: SchemaGen

```
### 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 0x7ff54456bdf0

.execution_id 10

.component ▶ **SchemaGen** at 0x7ff544566d30

.component.inputs
['statistics'] ▶ **Channel** of type 'ExampleStatistics' (1 artifact) at 0x7ff545845be0

.component.outputs
['schema'] ▶ **Channel** of type 'Schema' (1 artifact) at 0x7ff544566a30

If all went well, you can now visualize the generated schema as a table.

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

Artifact at metro_traffic_pipeline/SchemaGen/schema/10

Feature name	Type	Presence	Valency	Domain
--------------	------	----------	---------	--------

Feature name	Type	Presence	Valency	Domain	Values
'clouds_all'	INT	required	single	-	
'date_time'	BYTES	required	single	-	
'day'	INT	required	single	-	
'day_of_week'	INT	required	single	-	
'holiday'	STRING	required	single	'holiday'	
'hour'	INT	required	single	-	
'month'	INT	required	single	-	
'rain_1h'	FLOAT	required	single	-	
'snow_1h'	FLOAT	required	single	-	
'temp'	FLOAT	required	single	-	
'traffic volume'	INT	required	single	-	
					Values
Domain					
'holiday'	'Christmas Day', 'Columbus Day', 'Independence Day', 'Labor Day', 'Martin Luther King Jr Day', 'Memorial Day', 'New Years Day', 'None', 'State Fair', 'Thanksgiving Day', 'Veterans Day', 'Washingtons Birthday'				
'weather_description'	'SQUALLS', 'Sky is Clear', 'broken clouds', 'drizzle', 'few clouds', 'fog', 'freezing rain', 'haze', 'heavy intensity drizzle', 'heavy intensity rain', 'heavy snow', 'light intensity drizzle', 'light intensity shower rain', 'light rain', 'light rain and snow', 'light shower snow', 'light snow', 'mist', 'moderate rain', 'overcast clouds', 'proximity shower rain', 'proximity thunderstorm', 'proximity thunderstorm with drizzle', 'proximity thunderstorm with rain', 'scattered clouds', 'shower drizzle', 'sky is clear', 'sleet', 'smoke', 'snow', 'thunderstorm', 'thunderstorm with heavy rain', 'thunderstorm with light drizzle', 'thunderstorm with light rain', 'thunderstorm with rain', 'very heavy rain', 'shower snow', 'thunderstorm with drizzle'				
'weather_main'	'Clear', 'Clouds', 'Drizzle', 'Fog', 'Haze', 'Mist', 'Rain', 'Smoke', 'Snow', 'Squall', 'Thunderstorm'				

Each attribute in your dataset shows up as a row in the schema table, alongside its properties. The schema also captures all the values that a categorical feature takes on, denoted as its domain.

This schema will be used to detect anomalies in the next step.

2.4 - ExampleValidator

The [ExampleValidator](#) component detects anomalies in your data based on the generated schema from the previous step. Like the previous two components, it also uses TFDV under the hood.

`ExampleValidator` will take as input the statistics from `StatisticsGen` and the schema from `SchemaGen`. By default, it compares the statistics from the evaluation split to the schema from the training split.

Exercise 5: ExampleValidator

Fill the code below to detect anomalies in your datasets.

```
### START CODE HERE
# Instantiate ExampleValidator with the statistics and schema from the previous steps
example_validator = ExampleValidator(
    statistics=statistics_gen.outputs['statistics'],
    schema=schema_gen.outputs['schema'])

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

▼ **ExecutionResult** at 0x7ff54459f5e0

.execution_id 11

.component  **ExampleValidator** at 0x7ff544566700

.component.inputs

['statistics'] ▶ Channel of type 'ExampleStatistics' (1 artifact) at 0x7ff545845be0

['schema'] ▶ Channel of type 'Schema' (1 artifact) at 0x7ff544566a30

As with the previous steps, you can visualize the anomalies as a table.

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

Artifact at metro_traffic_pipeline/ExampleValidator/anomalies/11

'train' split:

No anomalies found.

'eval' split:

No anomalies found.

If there are anomalies detected, you should examine how you should handle it. For example, you can relax distribution constraints or modify the domain of some features. You've had some practice with this last week when you used TFDV and you can also do that here.

For this particular case, there should be no anomalies detected and we can proceed to the next step.

2.5 - Transform

In this section, you will use the [Transform](#) component to perform feature engineering.

`Transform` will take as input the data from `ExampleGen`, the schema from `SchemaGen`, as well as a module containing the preprocessing function.

The component expects an external module for your Transform code so you need to use the magic command `%% writefile` to save the file to disk. We have defined a few constants that group the data's attributes according to the transforms you will perform later. This file will also be saved locally.

```
# Set the constants module filename
_traffic_constants_module_file = 'traffic_constants.py'
```

```

%%writefile {_traffic_constants_module_file}

# Features to be scaled to the z-score
DENSE_FLOAT_FEATURE_KEYS = ['temp', 'snow_1h']

# Features to bucketize
BUCKET_FEATURE_KEYS = ['rain_1h']

# Number of buckets used by tf.transform for encoding each feature.
FEATURE_BUCKET_COUNT = {'rain_1h': 3}

# Feature to scale from 0 to 1
RANGE_FEATURE_KEYS = ['clouds_all']

# Number of vocabulary terms used for encoding VOCAB_FEATURES by tf.transform
VOCAB_SIZE = 1000

# Count of out-of-vocab buckets in which unrecognized VOCAB_FEATURES are hashed.
OOV_SIZE = 10

# Features with string data types that will be converted to indices
VOCAB_FEATURE_KEYS = [
    'holiday',
    'weather_main',
    'weather_description'
]

# Features with int data type that will be kept as is
CATEGORICAL_FEATURE_KEYS = [
    'hour', 'day', 'day_of_week', 'month'
]

# Feature to predict
VOLUME_KEY = 'traffic_volume'

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

```

Overwriting traffic_constants.py

Exercise 6

Next, you will fill out the transform module. As mentioned, this will also be saved to disk. Specifically, you will complete the `preprocessing_fn` which defines the transformations. See the code comments for instructions and refer to the [tft module documentation](#) to look up which function to use for a given group of keys.

For the label (i.e. `VOLUME_KEY`), you will transform it to indicate if it is greater than the mean of the entire dataset. For the transform to work, you will need to convert a [SparseTensor](#) to a dense one. We've provided a `_fill_in_missing()` helper function for you to use.

```

# Set the transform module filename
_traffic_transform_module_file = 'traffic_transform.py'

```

```

%%writefile {_traffic_transform_module_file}

import tensorflow as tf
import tensorflow_transform as tft

import traffic_constants

# Unpack the contents of the constants module
_DENSE_FLOAT_FEATURE_KEYS = traffic_constants.DENSE_FLOAT_FEATURE_KEYS
_RANGE_FEATURE_KEYS = traffic_constants.RANGE_FEATURE_KEYS
_VOCAB_FEATURE_KEYS = traffic_constants.VOCAB_FEATURE_KEYS
_VOCAB_SIZE = traffic_constants.VOCAB_SIZE
_OOV_SIZE = traffic_constants.OOV_SIZE
_CATEGORICAL_FEATURE_KEYS = traffic_constants.CATEGORICAL_FEATURE_KEYS
_BUCKET_FEATURE_KEYS = traffic_constants.BUCKET_FEATURE_KEYS
_FEATURE_BUCKET_COUNT = traffic_constants.FEATURE_BUCKET_COUNT
_VOLUME_KEY = traffic_constants.VOLUME_KEY
_transformed_name = traffic_constants.transformed_name

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

    ### START CODE HERE

    # Scale these features to the z-score.
    for key in _DENSE_FLOAT_FEATURE_KEYS:
        # Scale these features to the z-score.
        outputs[_transformed_name(key)] = tft.scale_to_z_score(inputs[key])

    # Scale these feature/s from 0 to 1
    for key in _RANGE_FEATURE_KEYS:
        outputs[_transformed_name(key)] = tft.scale_to_0_1(inputs[key])

    # Transform the strings into indices
    # hint: use the VOCAB_SIZE and OOV_SIZE to define the top_k and num_oov parameters
    for key in _VOCAB_FEATURE_KEYS:
        outputs[_transformed_name(key)] = tft.compute_and_apply_vocabulary(inputs[key],
                                                                           top_k=_VOCAB_SIZE)

    # Bucketize the feature
    for key in _BUCKET_FEATURE_KEYS:
        outputs[_transformed_name(key)] = tft.bucketize(inputs[key],
                                                         _FEATURE_BUCKET_COUNT[key])

    # Keep as is. No tft function needed.
    for key in _CATEGORICAL_FEATURE_KEYS:
        outputs[_transformed_name(key)] = tft.compute_and_apply_vocabulary(inputs[key])

    # Use `tf.cast` to cast the label key to float32 and fill in the missing values.
    traffic_volume = _fill_in_missing(tf.cast(inputs['traffic_volume'], tf.float32))

    # Create a feature that shows if the traffic volume is greater than the mean and cast to an int
    outputs[_transformed_name(_VOLUME_KEY)] = tf.cast(

        # Use `tf.greater` to check if the traffic volume in a row is greater than the mean of the entire traffic
        tf.greater(traffic_volume, tft.mean(tf.cast(inputs[_VOLUME_KEY], tf.float32))),

        tf.int64)

    ### END CODE HERE
    return outputs

def _fill_in_missing(x):
    """Replace missing values in a SparseTensor and convert to a dense tensor.
    Fills in missing values of `x` with `1` or `0`, and converts to a dense tensor.
    Args:

```

```

        x: A `SparseTensor` of rank 2. Its dense shape should have size at most 1
           in the second dimension.
    Returns:
        A rank 1 tensor where missing values of `x` have been filled in.
    """
    default_value = ' ' if x.dtype == tf.string else 0

    # ignore tf warning messages
    tf.get_logger().setLevel('ERROR')

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

    # Run the component.
    # The `enable_cache` flag is disabled in case you need to update your transform module file.
    context.run(transform, enable_cache=False)
    ### END CODE HERE

```

```

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

```

▼ ExecutionResult at 0x7ff560510bb0

.execution_id 20

.component  **Transform** at 0x7ff544595940


.component.inputs

['examples']  **Channel** of type '**Examples**' (1 artifact) at 0x7ff4ac8d42e0

['schema']  **Channel** of type '**Schema**' (1 artifact) at 0x7ff544566a30

.component.outputs

['transform_graph']  **Channel** of type '**TransformGraph**' (1 artifact) at 0x7ff5445957f0

['transformed_examples']  **Channel** of type '**Examples**' (1 artifact) at 0x7ff53bd61400

['updated_analyzer_cache']  **Channel** of type '**TransformCache**' (1 artifact) at 0x7ff53bd61520

You should see the output cell by `InteractiveContext` above and see the three artifacts in `.component.outputs` :

- `transform_graph` is the graph that performs the preprocessing operations. This will be included during training and serving to ensure

consistent transformations of incoming data.

- `transformed_examples` points to the preprocessed training and evaluation data.
- `updated_analyzer_cache` are stored calculations from previous runs.

The `transform_graph` artifact URI should point to a directory containing:

- The `metadata` subdirectory containing the schema of the original data.
- The `transformed_metadata` subdirectory containing the schema of the preprocessed data.
- The `transform_fn` subdirectory containing the actual preprocessing graph.

Again, for grading purposes, we inserted an `except` and `else` below to handle checking the output outside the notebook environment.

```
try:
    # Get the uri of the transform graph
    transform_graph_uri = transform.outputs['transform_graph'].get()[0].uri

except IndexError:
    print("context.run() was no-op")
    transform_path = './metro_traffic_pipeline/Transform/transformed_examples'
    dir_id = os.listdir(transform_path)[0]
    transform_graph_uri = f'{transform_path}/{dir_id}'

else:
    # List the subdirectories under the uri
    os.listdir(transform_graph_uri)
```

Lastly, you can also take a look at a few of the transformed examples.

```
try:
    # Get the URI of the output artifact representing the transformed examples
    train_uri = os.path.join(transform.outputs['transformed_examples'].get()[0].uri, 'train')

except IndexError:
    print("context.run() was no-op")
    train_uri = os.path.join(transform_graph_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="GZIP")
```

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

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

```
{'features': {'feature': {'clouds_all_xf': {'floatList': {'value': [0.39999998]}},
                          'day_of_week_xf': {'int64List': {'value': ['2']}},
                          'day_xf': {'int64List': {'value': ['26']}},
                          'holiday_xf': {'int64List': {'value': ['0']}},
                          'hour_xf': {'int64List': {'value': ['5']}},
                          'month_xf': {'int64List': {'value': ['11']}},
                          'rain_1h_xf': {'int64List': {'value': ['2']}},
                          'snow_1h_xf': {'floatList': {'value': [-0.027424417]}},
                          'temp_xf': {'floatList': {'value': [0.53368527]}},
                          'traffic_volume_xf': {'int64List': {'value': ['1']}},
                          'weather_description_xf': {'int64List': {'value': ['4']}},
                          'weather_main_xf': {'int64List': {'value': ['0']}}}},
{'features': {'feature': {'clouds_all_xf': {'floatList': {'value': [0.75]}},
                          'day_of_week_xf': {'int64List': {'value': ['2']}},
                          'day_xf': {'int64List': {'value': ['26']}},
                          'holiday_xf': {'int64List': {'value': ['0']}},
                          'hour_xf': {'int64List': {'value': ['1']}},
                          'month_xf': {'int64List': {'value': ['11']}},
                          'rain_1h_xf': {'int64List': {'value': ['2']}},
                          'snow_1h_xf': {'floatList': {'value': [-0.027424417]}},
                          'temp_xf': {'floatList': {'value': [0.6156978]}},
                          'traffic_volume_xf': {'int64List': {'value': ['1']}},
                          'weather_description_xf': {'int64List': {'value': ['3']}},
                          'weather_main_xf': {'int64List': {'value': ['0']}}}},
{'features': {'feature': {'clouds_all_xf': {'floatList': {'value': [0.9]}},
                          'day_of_week_xf': {'int64List': {'value': ['2']}},
                          'day_xf': {'int64List': {'value': ['26']}},
                          'holiday_xf': {'int64List': {'value': ['0']}},
                          'hour_xf': {'int64List': {'value': ['16']}},
                          'month_xf': {'int64List': {'value': ['11']}},
                          'rain_1h_xf': {'int64List': {'value': ['2']}},
                          'snow_1h_xf': {'floatList': {'value': [-0.027424417]}}},
```

```
'temp_xf': {'floatList': {'value': [0.6324043]}},  
'traffic_volume_xf': {'int64List': {'value': ['1']}},  
'weather_description_xf': {'int64List': {'value': ['2']}},  
'weather main xf': {'int64List': {'value': ['0']}}}]
```

Congratulations on completing this week's assignment! You've just demonstrated how to build a data pipeline and do feature engineering. You will build upon these concepts in the next weeks where you will deal with more complex datasets and also access the metadata store. Keep up the good work!