Data preprocessing for machine learning using TensorFlow Transform

This article discusses how to use TensorFlow Transform (https://github.com/tensorflow/transform) (`tf.Transform`) to implement data preprocessing for machine learning (ML). `tf.Transform` is a library for TensorFlow that allows you to define both instance-level and full-pass data transformations through data preprocessing pipelines. These pipelines are efficiently executed with Apache Beam (https://beam.apache.org/) and they create as byproducts a TensorFlow graph to apply the same transformations during prediction as when the model is served.

This article goes through an end-to-end example, showing each step using Dataflow (/dataflow/docs) as a runner for Apache Beam. It assumes that you are familiar with BigQuery (/bigquery/docs), Dataflow, AI Platform (ML) Engine (/ml-engine/docs/tensorflow/technical-overview), and TensorFlow Estimator (https://www.tensorflow.org/guide/estimators) APIs.

Introduction

TensorFlow Transform (https://github.com/tensorflow/transform) (`tf.Transform`) is a library for preprocessing data with TensorFlow that is useful for transformations that require a full pass. The output of `tf.Transform` is exported as a TensorFlow graph, representing the instance-level transformation logic, with the statistics computed from full-pass transformations (as constants), to use for training and serving. Using the same graph for both training and serving can prevent skew, because the same transformations are applied in both stages. In addition, `tf.Transform` can run at scale on Dataflow in a batch processing pipeline to prepare the training data up front and improve training efficiency.

To learn more about the concepts of preprocessing types, challenges, and options on Google Cloud, see Data Preprocessing for Machine Learning with TensorFlow Transform - Part 1 (/solutions/machine-learning/data-preprocessing-for-ml-with-tf-transform-pt1).

Figure 1 shows a conceptual overview of `tf.Transform` behavior in preprocessing and transforming data for training and prediction.

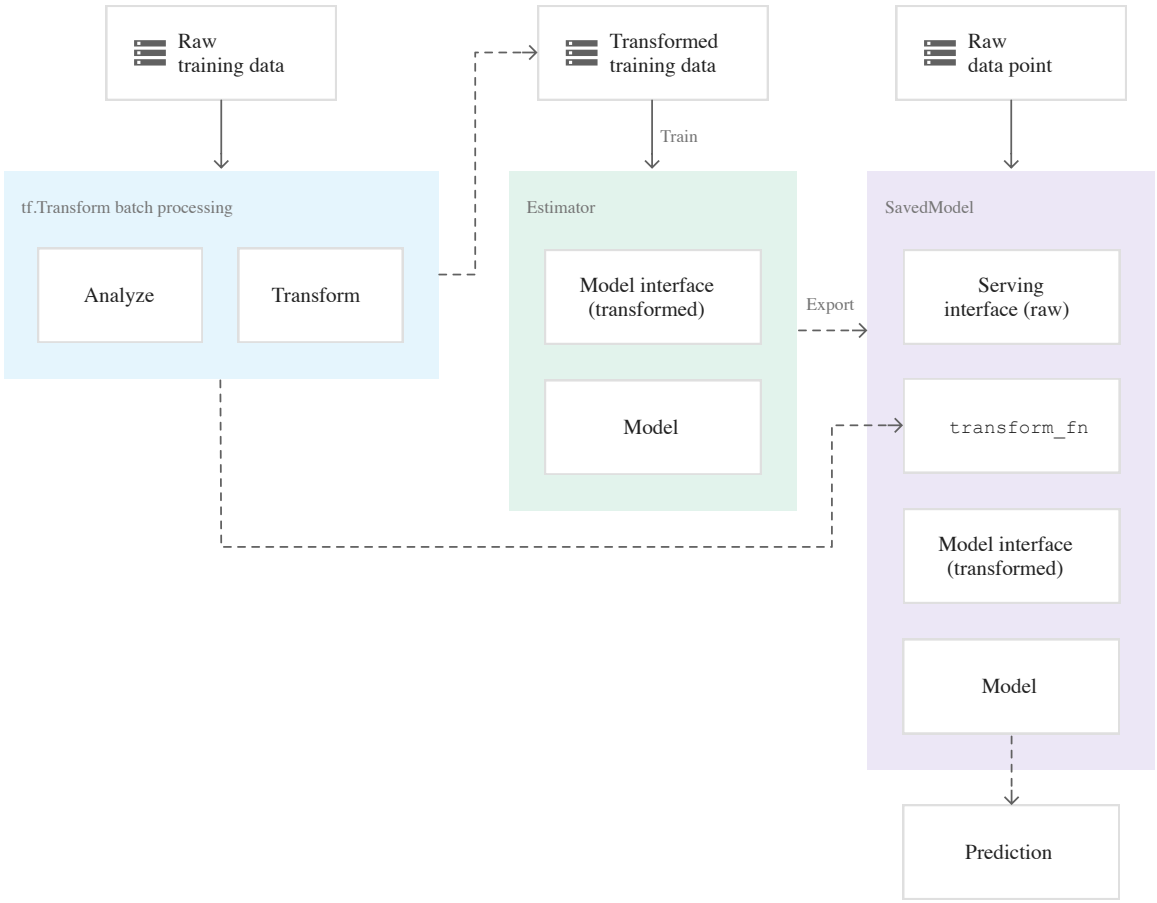

Figure 1. The behavior of `tf.Transform`

As discussed in Part 1 (/solutions/machine-learning/data-preprocessing-for-ml-with-tf-transform-pt1) , `tf.Transform` is used in the following way:

1. You preprocess the raw training data using transformations implemented in `tf.Transform` Apache Beam APIs, and running it at scale on Dataflow. The preprocessing occurs in two phases: analyze and transform.
2. During the analyze phase, the required statistics (for example, means and variances) for stateful transformations are computed with full-pass operations on the training data. During the transform phase, these computed statistics are used to process the training data (for example, z-score normalization of numerical features) with instance-level operations.
3. The `tf.Transform` preprocessing pipeline produces two main outputs, the transformed training data and a `transform_fn` function.
    - The transformed training data is used for training the model, where the model interface expects transformed features.
    - The `transform_fn` function consists of the transformation logic as a TensorFlow graph, in which the transformations are instance-level operations,

and in which the statistics computed in the full-pass transformations are constants.

4. When the model is exported after training, the export process attaches the `transform_fn` graph to the exported `SavedModel`. During serving for prediction, the model serving interface expects data points in raw format. The `transform_fn`, which is now part of the model, applies all of the preprocessing logic on the incoming data point. It also uses the stored constants, like the mean and variance, to normalize a numeric feature as instance-level operations during prediction.
5. The `transform_fn` converts the raw data point into the transformed format, which is expected by the model interface in order to produce prediction.

Jupyter notebooks for this solution

There are two Jupyter notebooks to show the implementation example, which are both in a GitHub repository:

- Notebook #1 (https://github.com/GoogleCloudPlatform/tf-estimator-tutorials/blob/master/00_Miscellaneous/tf_transform/tft-01%20-%20Babyweight%20preprocessing%20with%20tf.Transform.ipynb) covers the data preprocessing part. Details are provided in the Implementing the Apache Beam pipeline (#implementing_the_apache_beam_pipeline) section that follows.
- Notebook #2 (https://github.com/GoogleCloudPlatform/tf-estimator-tutorials/blob/master/00_Miscellaneous/tf_transform/tft-02%20-%20Babyweight%20Estimation%20with%20Transformed%20Data.ipynb) covers the model training part. Details are provided in the Implementing the TensorFlow model (#implementing_the_tensorflow_model) section later.

Implementing the Apache Beam pipeline

This section uses a practical example to describe how to use `tf.Transform` to preprocess data. This article uses the Natality dataset, which is used to predict baby weights based on various inputs. The data is stored in the public natality (https://bigquery.cloud.google.com/table/bigquery-public-data:samples.natality?pli=1&tab=details) table in BigQuery.

In the example, Dataflow runs the `tf.Transform` pipeline at scale to prepare the data and produce the `transform_fn` artifacts. The following code runs the pipeline. Later sections in this document describe the functions that perform each step in the pipeline. The overall pipeline steps are as follows:

1. Read training data from BigQuery.
2. Analyze and transform training data using `tf.Transform`.
3. Write transformed training data to Cloud Storage as `tfRecords`.
4. Read evaluation data from BigQuery.
5. Transform evaluation data using the `transform_fn` produced by step 2.
6. Write transformed training data to Cloud Storage as `tfrecords`.
7. Write transformation artifacts to Cloud Storage for creating and exporting the model.

The following listing shows the Python code for the overall pipeline. The sections that follow provide explanations and code listings for each step.

```
def run_transformation_pipeline(args):

    pipeline_options = beam.pipeline.PipelineOptions(flags=[], **args)

    runner = args['runner']
    data_size = args['data_size']
    transformed_data_location = args['transformed_data_location']
    transform_artefact_location = args['transform_artefact_location']
    temporary_dir = args['temporary_dir']
    debug = args['debug']

    # Instantiate the pipeline
    with beam.Pipeline(runner, options=pipeline_options) as pipeline:
        with impl.Context(temporary_dir):

            # Preprocess train data
            step = 'train'
            # Read raw train data from BigQuery
            raw_train_dataset = read_from_bq(pipeline, step, data_size)
            # Analyze and transform raw_train_dataset
            transformed_train_dataset, transform_fn = analyze_and_transform(ra
            # Write transformed train data to sink as tfrecords
            write_tfrecords(transformed_train_dataset, transformed_data_locat

            # Preprocess evaluation data
            step = 'eval'
```

```
            # Read raw eval data from BigQuery
            raw_eval_dataset = read_from_bq(pipeline, step, data_size)
            # Transform eval data based on produced transform_fn
            transformed_eval_dataset = transform(raw_eval_dataset, transform_
            # Write transformed eval data to sink as tfrecords
            write_tfrecords(transformed_eval_dataset, transformed_data_locatio

            # Write transformation artefacts
            write_transform_artefacts(transform_fn, transform_artefact_locatio

            # (Optional) for debugging, write transformed data as text
            step = 'debug'
            # Wwrite transformed train data as text if debug enabled
            if debug == True:
            write_text(transformed_train_dataset, transformed_data_location, s
```

Step 1: Read raw training data from BigQuery

The first step is to read the raw training data from BigQuery using the **`read_from_bq`** method. This method returns a raw_dataset object extracted from BigQuery. You pass a **`data_size`** value and a value for **`step`** (which can be **`train`** or **`eval`**). The BigQuery source query is constructed using the **`get_source_query`** method.

```
def read_from_bq(pipeline, step, data_size):

    source_query = get_source_query(step, data_size)
    raw_data = (
        pipeline
        | '{} - Read Data from BigQuery'.format(step) >> beam.io.Read(
                            beam.io.BigQuerySource(query=source_query, use_sta
        | '{} - Clean up Data'.format(step) >> beam.Map(prep_bq_row)
    )

    raw_metadata = create_raw_metadata()
    raw_dataset = (raw_data, raw_metadata)
    return raw_dataset
```

Before you perform the **`tf.Transform`** preprocessing, you might need to perform typical Apache Beam–based processing, including Map, Filter, Group, and Window. In the example, the code cleanses the records read from BigQuery using **`beam.Map(prep_bq_row)`**, where **`prep_bq_row`** is a custom method. This custom method converts the numeric code for a categorical feature into human-readable labels.

In addition, to use `tf.Transform` to analyze and transform the `raw_data` object extracted from BigQuery, you need to create a `raw_dataset` object, which is a tuple of (`raw_data`, `raw_metadata`). The `raw_metadata` object is created using `create_raw_metadata` method, as follows:

```
CATEGORICAL_FEATURE_NAMES = ['is_male', 'mother_race']
NUMERIC_FEATURE_NAMES = ['mother_age', 'plurality', 'gestation_weeks']
TARGET_FEATURE_NAME = 'weight_pounds'
KEY_COLUMN = 'key'

def create_raw_metadata():

    raw_data_schema = {}

    # key feature schema
    raw_data_schema[KEY_COLUMN]= dataset_schema.ColumnSchema(
        tf.float32, [], dataset_schema.FixedColumnRepresentation())

    # target feature schema
    raw_data_schema[TARGET_FEATURE_NAME]= dataset_schema.ColumnSchema(
        tf.float32, [], dataset_schema.FixedColumnRepresentation())

    # categorical feature schema
    raw_data_schema.update({ column_name : dataset_schema.ColumnSchema(
        tf.string, [], dataset_schema.FixedColumnRepresentation())
              for column_name in CATEGORICAL_FEATURE_NAMES})

    # numerical feature schema
    raw_data_schema.update({ column_name : dataset_schema.ColumnSchema(
        tf.float32, [], dataset_schema.FixedColumnRepresentation())
              for column_name in NUMERIC_FEATURE_NAMES})

    # create dataset_metadata given raw_data_schema
    raw_metadata = dataset_metadata.DatasetMetadata(
        dataset_schema.Schema(raw_data_schema))

    return raw_metadata
```

This `raw_metadata` object is also used by the `seving_input_receiver_fn` to create the exported model serving interface, which expects data points in raw format as shown in Figure 1. Exporting the model for serving predictions is discussed later under Exporting the model for serving prediction (#exporting_the_model_for_serving_prediction).

When you execute the cell in the notebook that defines this method, the content of the `raw_metadata.schema` is displayed. It includes the following columns:

- `mother_race` (`tf.string`)
- `weight_pounds` (`tf.float32`)
- `gestation_weaks` (`tf.float32`)
- `key` (`tf.float32`)
- `is_male` (`tf.string`)
- `mother_age` (`tf.string`)

Step 2: Transform raw training data using preprocess_fn

Imagine that you want to apply typical preprocessing transformations to the input raw features of the training data in order to prepare it for ML. These transformations include both full-pass and instance-level operations. Specifically, you are going to perform the transformations listed in the following table.

Input feature	Transformation	Stats needed	Type	Output feature
`weight_pound`	None	None	NA	`weight_pound`
`mother_age`	Normalize	mean, var	Full-pass	`mother_age_normalized`
`mother_age`	Equal size bucketization	quantiles	Full-pass	`mother_age_bucketized`
`mother_age`	Compute the log	None	Instance-level	`mother_age_log`
`plurality`	Indicate if it is single or multiple babies	None	Instance-level	`is_multiple`
`is_multiple`	Convert nominal values to numerical index	vocab	Full-pass	`is_multiple_index`
`gestation_weeks`	Scale between 0 and 1	min, max	Full-pass	`gestation_weeks_scaled`
`mother_race`	Convert nominal values to numerical index	vocab	Full-pass	`mother_race_index`
`is_male`	Convert nominal values to numerical index	vocab	Full-pass	`is_male_index`

These transformations are implemented in a `preprocess_fn` method, which expects a dictionary of tensors (`input_features`) and returns a dictionary of processed features (`output_features`).

The following code shows the implementation of the `preprocess_fn` method, using the `tf.Transform` full-pass transformation APIs (prefixed with `tft.`), and TensorFlow (prefixed with `tf.`) instance-level operations:

```
def preprocess_fn(input_features):

    output_features = {}

    # target feature
    output_features['weight_pounds'] = input_features['weight_pounds']

    # normalization
    output_features['mother_age_normalized'] = tft.scale_to_z_score(input_fea
    output_features['gestation_weeks_normalized'] = tft.scale_to_z_score(inpu

    # bucketization based on quantiles
    output_features['mother_age_bucketized'] = tft.bucketise(input_features['r

    # you can compute new features based on custom formulas
    output_features['mother_age_log'] = tf.log(input_features['mother_age'])

    # or create flags/indicators
    is_multiple = tf.as_string(input_features['plurality'] > tf.constant(1.0)

    # convert categorical features to indexed vocab
    output_features['mother_race_index'] = tft.compute_and_apply_vocabulary(ir
        vocab_filename='mother_race')
    output_features['is_male_index'] = tft.compute_and_apply_vocabulary(input_
        vocab_filename='is_male')
    output_features['is_multiple_index'] = tft.compute_and_apply_vocabulary(is
        vocab_filename='is_male')
    return output_features
```

The `tf.Transform` framework (https://github.com/tensorflow/transform) has several other transformations in addition to those in the preceding example, including those listed in the following table.

Transformation	Applies to	Description
`scale_by_min_max`	Numeric features	Scales a numerical column into the range [`output_min`, `output_max`]
`scale_to_0_1`	Numeric features	Returns a column which is the input column scaled to have range [0,1]

Transformation	Applies to	Description
`scale_to_z_score`	Numeric features	Returns a standardized column with mean 0 and variance 1
`tfidf`	Text features	Maps the terms in x to their term frequency * inverse document frequency
`compute_and_apply_vocabulary`	Categorical features	Generates a vocabulary for a categorical feature and maps it to an integer with this vocab
`ngrams`	Text features	Creates a `SparseTensor` of n-grams
`hash_strings`	Categorical features	Hashes strings into buckets
`pca`	Numeric features	Computes PCA on the dataset using biased covariance
`bucketize`	Numeric features	Returns an equal-sized (quantiles-based) bucketized column, with a bucket index assigned to each input

In order to apply the transformations implemented in the `preprocess_fn` method to the `raw_train_dataset` object produced in the previous step of the pipeline, you use the `AnalyzeAndTransformDataset` method. This method expects the `raw_dataset` object as input, applies the `preprocess_fn` method, and produces the `transformed_dataset` object and the `transform_fn`. The following code illustrates this processing.

```
def analyze_and_transform(raw_dataset, step):

    transformed_dataset, transform_fn = (
        raw_dataset
        | '{} - Analyze & Transform'.format(step) >> impl.AnalyzeAndTransforml
    )

    return transformed_dataset, transform_fn
```

The transformations are applied on the raw data in two phases: the analyze phase and the transform phase. Figure 5 later in this document shows how the `AnalyzeAndTransformDataset` is decomposed to `AnalyzeDataset` and `TransformDataset`.

The analyze phase

In the analyze phase, the raw training data is analyzed in a full-pass process to compute the statistics needed for the transformations. This includes computing the mean, variance, minimum, maximum, quantiles, and vocabulary. The analyze process expects a raw dataset (raw data plus raw metadata), and it produces two outputs:

- `transform_fn`. A function that contains the computed stats from the analyze phase and the transformation logic (which uses the stats) as instance-level operations. As discussed later in Step 7 (#save_transform_fn), the `transform_fn` is saved to be attached to the model `serving_input_fn`. This makes it possible to apply the same transformation to the online prediction data points.
- `transform_metadata`. An object that describes the expected schema of the data after transformation.

The analyze phase is illustrated in Figure 2.

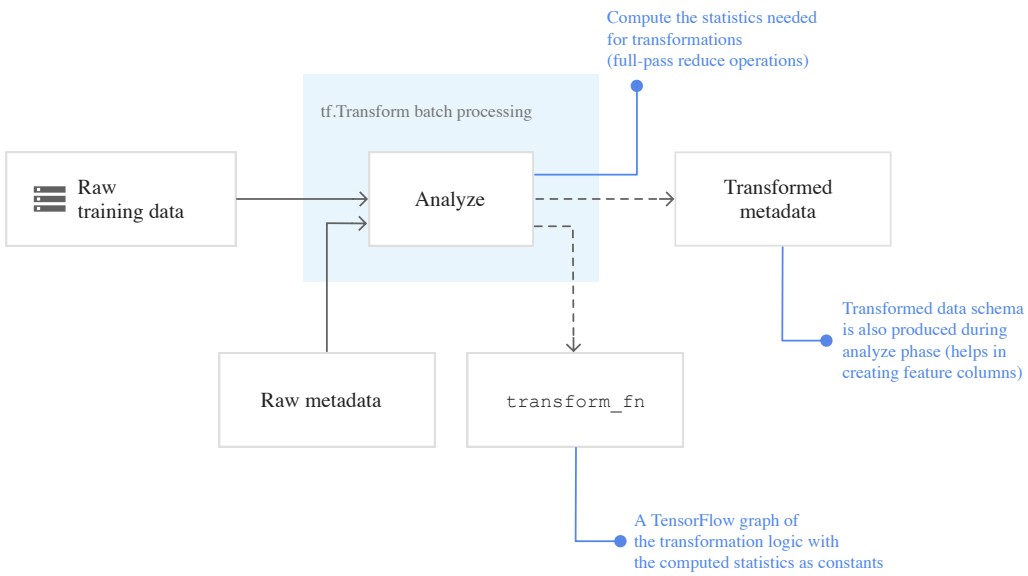

Figure 2. The `tf.Transform` analyze phase

The `tf.Transform` analyzers (https://github.com/tensorflow/transform/blob/master/tensorflow_transform/beam/analyzer_impls.py) include `min`, `max`, `sum`, `size`, `mean`, `var`, `covariance`, `quantiles`, `vocabulary`, and `pca`.

The transform phase

In the transform phase, the `transform_fn` produced by the analyze phase is used to transform the raw training data in an instance-level process in order to produce the

transformed training data. The transformed training data is paired with the transformed metadata (produced by the analyze phase) to produce the `transformed_train_dataset`.

The transform phase is illustrated in Figure 3.

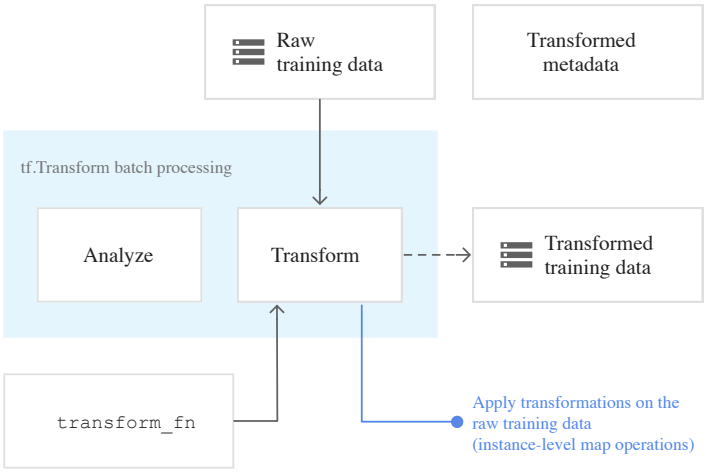

Figure 3. The `tf.Transform` transform phase

To preprocess the features, you call the required `tensorflow_transform` (imported as `tft` in the code) transformations in your implementation of `preprocess_fn`. For example, when you call `tft.scale_to_z_score`, `tf.Transform` translates this function call into mean and variance analyzers, computes the stats, and then applies these stats to normalize the numeric feature. This is all done automatically by calling the following method:

```
AnalyzeAndTransformDataset(preprocess_fn)
```

The `transformed_metadata.schema` entity produced by this call will include the following columns:

- `gestation_weeks_normalixed` (`tf.float32`)
- `is_multiple_index` (`tf.int64`, `_is_categorical: True`)
- `mother_race_index'` (`tf.int64`, `_is_categorical: True`)
- `is_male_index` (`tf.int64`, `_is_categorical: True`)
- `mother_age_log` (`tf.float32`)
- `mother_age_bucketized` (`tf.int64`, `_is_categorical: True`)
- `mother_age_normalized` (`tf.float32`)
- `weight_pounds` (`tf.float32`)

Step 3: Write transformed training data

After the training data is preprocessed with the `tf.Transform` `preprocess_fn` through the analyze and transform phases, you can write the data to a sink to be used for training the TensorFlow model. When you run the Apache Beam pipeline using Dataflow, the sink is Cloud Storage. Otherwise, the sink is the local disk. Although you can write the data as a CSV file of fixed-width formatted files, the recommended file format for TensorFlow datasets is the TFRecord (https://www.tensorflow.org/guide/datasets) format. This is a simple record-oriented binary format that consists of `tf.train.Example` (https://www.tensorflow.org/api_docs/python/tf/train/Example) protocol buffer messages.

Each `tf.train.Example` record contains one or more features. These are converted into tensors when they are fed to the model for training. The following code writes the transformed dataset to TFRecord files in the specified location.

```
def write_tfrecords(dataset, location, step):

    transformed_data, transformed_metadata = transformed_dataset
    (
        transformed_data
        | '{} - Write Transformed Data'.format(step) >> beam.io.tfrecordio.Wr
            file_path_prefix=os.path.join(location,'{}-'.format(step)),
            file_name_suffix=".tfrecords",
            coder=example_proto_coder.ExampleProtoCoder(transformed_metadata.
    )
```

Step 4, 5, and 6: Read, transform, and write evaluation data

After you transform the training data and produce the `transform_fn` function, you can use the function to transform the evaluation data. First, you read and cleanse the evaluation data from BigQuery using the `read_from_bq`methods described earlier in Step 1: Read raw training data from BigQuery (#read_raw_training_data), and passing `eval` for the `step` parameter.

Second, you use the following code to transform the raw evaluation dataset (`raw_dataset`) to the expected transformed format (`transformed_dataset`), as shown in the following code:

```
def transform(raw_dataset, transform_fn, step):
    transformed_dataset = (
```

```
        (raw_dataset, transform_fn)
        | '{} - Transform'.format(step) >> impl.TransformDataset()
        )
    return transformed_dataset
```

When you transform the evaluation data, only instance-level operations apply, using both the logic in the `transform_fn` and the statistics computed from the analyze phase in the training data. In other words, you don't analyze the evaluation data in a full-pass fashion to compute new statistics, like the mean and the variance for z-score normalization of numeric features in evaluation data. Instead, you use the computed statistics from the training data to transform the evaluation data in an instance-level fashion.

Therefore, you use `impl.AnalyzeAndTransform` in the context of training data to compute the statistics and transform the data. At the same time, you use `impl.TransformDataset` in the context of transforming evaluation data to only transform the data using the statistics computed on the training data.

Third, you write the data to a sink (Cloud Storage or local disk, depending on the runner) as TFRecord files for evaluating the TensorFlow model during the training process. To do this, you use the `write_tfrecords` method discussed under Step 3: Write transformed training data (#step_3_write_transformed_training_data). Figure 4 shows how the `transform_fn` produced in the analyze phase of the training data is used to transform the evaluation data.

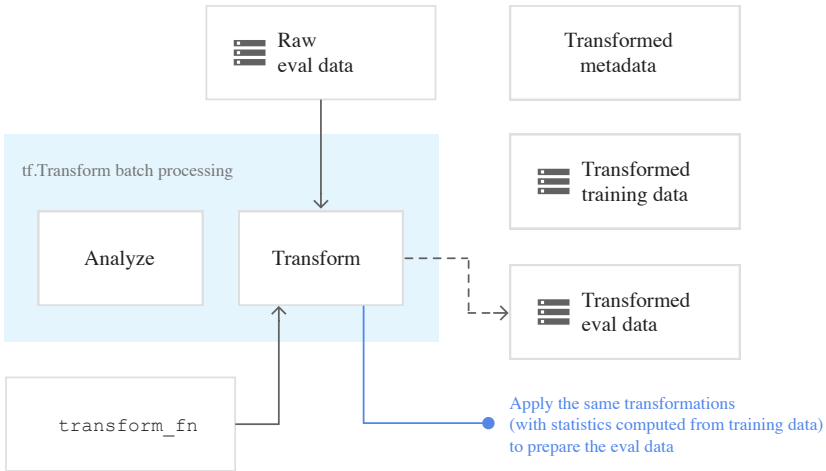

Figure 4. Transforming evaluation data using `transform_fn`

Step 7: Save transform_fn

A final step in a `tf.Transform` preprocessing pipeline is to store the `transform_fn`, which includes artifacts produced by the analyze phase on the training data. The code for storing the `transform_fn` is shown in the following `write_transform_artifacts` method:

```
def write_transform_artefacts(transform_fn, location):

    (
        transform_fn
        | 'Write Transform Artifacts' >> transform_fn_io.WriteTransformFn(loca
    )
```

These artifacts will be used later for model training and exporting for serving. The following artifacts are also produced, as shown later in Figure 6:

- `saved_model.pb`. This represents the TensorFlow graph that includes the transformation logic, which is to be attached to the model serving interface to transform the raw data points to the transformed format.
- `variables`. This includes the statistics computed during the analyze phase of the training data, and is used in the transformation logic in `saved_model.pb`.
- `assets`. This includes vocabulary files, one for each categorical feature processed with `compute_and_apply_vocabulary` method, to be used during serving to convert an input raw nominal value to a numerical index.
- `transformed_metadata`. This includes the `schema.json` file that describes the schema of the transformed data.

Running the pipeline in Dataflow

To run the `tf.Transform` pipeline in Dataflow, you execute the code in the notebook (https://github.com/GoogleCloudPlatform/tf-estimator-tutorials/blob/master/00_Miscellaneous/tf_transform/tft-01%20-%20Babyweight%20preprocessing%20with%20tf.Transform.ipynb) step by step. For details on how you can run the notebook in Datalab (/datalab/docs), refer to the Machine Learning with Structured Data (/solutions/machine-learning/ml-on-structured-data-analysis-prep-1) tutorial.

Make sure that you make the following changes in the first code cell of the notebook:

- Set the `PROJECT` variable to your project name.
- Set the `BUCKET` variable to your bucket name.
- Set the `REGION` variable to the bucket regions.

- Set the `LOCAL_RUN` variable to `False` in order to run your pipeline on Dialogflow.

Figure 5 shows the Dataflow execution graph of the `tf.Transform` pipeline described in the example.

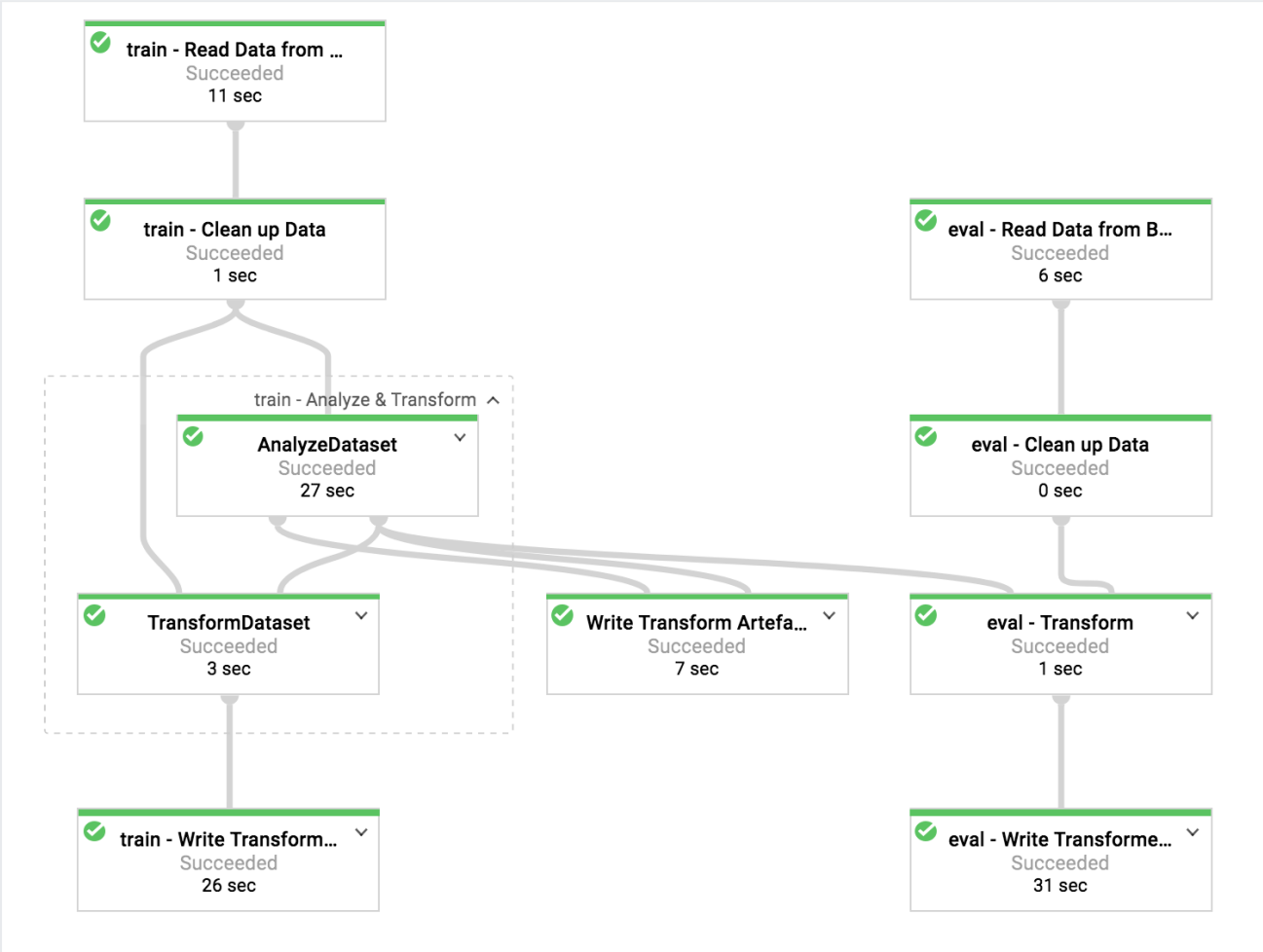

Figure 5. Dataflow execution graph of the `tf.Transform` pipeline

After you execute the Dataflow pipeline to preprocess the training and evaluation data, you can explore the produced objects in Cloud Storage by executing the last cell in the notebook. The transformed training and evaluation data in TFRecord format at the following location:

```
gs://[YOUR_BUCKET_NAME]/tft_babyweight/transformed
```

The transform artifacts are produced at the following location:

```
gs://[YOUR_BUCKET_NAME]/tft_babyweight/transform
```

Figure 6 is a listing of the output of the pipeline, showing the produced data objects and artifacts.

```
transformed data:
gs://ksalama-gcs-cloudml/babyweight_tft/transformed/eval-00000-of-00001.tfrecords
gs://ksalama-gcs-cloudml/babyweight_tft/transformed/train-00000-of-00003.tfrecords
gs://ksalama-gcs-cloudml/babyweight_tft/transformed/train-00001-of-00003.tfrecords
gs://ksalama-gcs-cloudml/babyweight_tft/transformed/train-00002-of-00003.tfrecords

transformed metadata:
gs://ksalama-gcs-cloudml/babyweight_tft/transform/transformed_metadata/
gs://ksalama-gcs-cloudml/babyweight_tft/transform/transformed_metadata/v1-json/

transform artefact:
gs://ksalama-gcs-cloudml/babyweight_tft/transform/transform_fn/
gs://ksalama-gcs-cloudml/babyweight_tft/transform/transform_fn/saved_model.pb
gs://ksalama-gcs-cloudml/babyweight_tft/transform/transform_fn/assets/
gs://ksalama-gcs-cloudml/babyweight_tft/transform/transform_fn/variables/

transform assets:
gs://ksalama-gcs-cloudml/babyweight_tft/transform/transform_fn/assets/
gs://ksalama-gcs-cloudml/babyweight_tft/transform/transform_fn/assets/is_male
gs://ksalama-gcs-cloudml/babyweight_tft/transform/transform_fn/assets/is_multiple
gs://ksalama-gcs-cloudml/babyweight_tft/transform/transform_fn/assets/mother_race
```

Figure 6. Listing of the data objects and artifacts produced as output by the `tf.Transform` pipeline

Implementing the TensorFlow model

In the baby-weight estimation example, a TensorFlow model is implemented using DNNLinearCombinedRegressor (https://www.tensorflow.org/api_docs/python/tf/estimator/DNNLinearCombinedRegressor). The code for creating, training, evaluating, and exporting the model is a notebook (https://github.com/GoogleCloudPlatform/tf-estimator-tutorials/blob/master/00_Miscellaneous/tf_transform/tft-02%20-%20Babyweight%20Estimation%20with%20Transformed%20Data.ipynb) on the GitHub repository. The model uses the data and artifacts produced by the `tf.Transform` preprocessing pipeline explained earlier.

The steps for creating the model are as follows:

1. Load the `transform_metadata` object.
2. Create an `input_fn` to read and parse the training and evaluation data using the `transform_metadata` object.
3. Create feature columns using the `transform_metadata` object.
4. Create the `DNNLinearCombinedRegressor` estimator with the `feature_columns`.
5. Train and evaluate the estimator.
6. Export the estimator by defining a `serving_input_fn` with a `transform_fn` attached to it.

7. Inspect the exported model using the saved_model_cli (https://www.tensorflow.org/guide/saved_model) tool.
8. Use the exported model for prediction.

The goal of this article is not to build the model. Therefore, it does not discuss in detail how the model was built or trained. However, in this section, the article shows how the `transform_metadata` object, which is produced by the `tf.Transform` process, is used to create the feature columns of the model. The article also shows how the `transform_fn`, which is also produced by `tf.Transform` process, is used in the `serving_input_fn` when the model is exported for serving.

Using the generated transform artifacts in model training

When you train the TensorFlow model, you use the transformed `train` and `eval` objects produced in the previous data processing step. These objects are stored as sharded TFRecord files. The `transformed_metadata` object generated in the previous step can be useful in the following places:

- Parsing the data (`tf.train.Example` objects) to feed into the model for training and evaluation.
- Creating the feature columns in a dynamic, metadata-driven fashion.

Parsing tf.train.Example data

Because you read TFRecord files to feed the model with training and evaluation data, you need to parse each `tf.train.Example` object in the files to create a dictionary of features (tensors). This makes sure that the features are mapped to the model input layer using the feature columns, which act as the model training/evaluation interface. You use the `transformed_metadata` object generated from the previous step. This takes two steps.

First, you load the `transformed_metadata` object generated and saved in the previous preprocessing step, as described in the Save transform_fn section (#save_transform_fn):

```
transformed_metadata = metadata_io.read_metadata(
    os.path.join(TRANSFORM_ARTEFACTS_DIR,"transformed_metadata"))
```

You then convert the `transformed_metadata` object to a `feature_spec` object, and use it in the `tfrecords_input_fn`:

```
def tfrecords_input_fn(files_name_pattern, transformed_metadata,
        mode=tf.estimator.ModeKeys.EVAL,
        num_epochs=1,
        batch_size=500):

    dataset = tf.contrib.data.make_batched_features_dataset(
        file_pattern=files_name_pattern,
        batch_size=batch_size,
        features=transformed_metadata.schema.as_feature_spec(),
        reader=tf.data.TFRecordDataset,
        num_epochs=num_epochs,
        shuffle=True if mode == tf.estimator.ModeKeys.TRAIN else False,
        shuffle_buffer_size=1+(batch_size*2),
        prefetch_buffer_size=1
    )

    iterator = dataset.make_one_shot_iterator()
    features = iterator.get_next()
    target = features.pop(TARGET_FEATURE_NAME)
    return features, target
```

Creating the feature columns

The pipeline produces the** **`transformed_metadata` object that describes the schema of the transformed data expected by the model for training and evaluation. You can use this metadata to create the feature columns dynamically, without specifying each one by name. This dynamic, metadata-driven approach for creating feature columns is helpful when you have hundreds of features. The following code shows how to create feature columns using the metadata.

```
def create_wide_and_deep_feature_columns(transformed_metadata, hparams):

    deep_feature_columns = []
    wide_feature_columns = []

    column_schemas = transformed_metadata.schema.column_schemas

    for feature_name in column_schemas:
        if feature_name == TARGET_FEATURE_NAME:
            continue

        # creating numerical features
        column_schema = column_schemas[feature_name]
```

```
        if isinstance(column_schema._domain, dataset_schema.FloatDomain):
            deep_feature_columns.append(tf.feature_column.numeric_column(featu

       # creating categorical features with identity
        elif isinstance(column_schema._domain, dataset_schema.IntDomain):
            if column_schema._domain._is_categorical==True:
                wide_feature_columns.append(
                    tf.feature_column.categorical_column_with_identity(
                        feature_name,
                        num_buckets=column_schema._domain._max_value+1)
                )
            else:
                deep_feature_columns.append(tf.feature_column.numeric_column(

    if hparams.extend_feature_columns==True:
        mother_race_X_mother_age_bucketized = tf.feature_column.crossed_colum
            ['mother_age_bucketized', 'mother_race_index'],  55)

        wide_feature_columns.append(mother_race_X_mother_age_bucketized)

        mother_race_X_mother_age_bucketized_embedded = tf.feature_column.embe
            mother_race_X_mother_age_bucketized, hparams.embed_dimensions)
        deep_feature_columns.append(mother_race_X_mother_age_bucketized_embed

    return wide_feature_columns, deep_feature_columns
```

The code creates a **`tf.feature_column.numeric_column`** column if the column schema is **`FloatDomain`**. On the other hand, the code creates a **`tf.feature_column.categorical_column_with_identity`** column if the column schema is **`IntDomain`** and the **`_is_categorical`** property is **`True`**.

In addition, you can create extended feature columns, as discussed under Option C (/solutions/machine-learning/data-preprocessing-for-ml-with-tf-transform-pt1#option_c_tensorflow) in the "Where to do preprocessing" section of Part 1 (/solutions/machine-learning/data-preprocessing-for-ml-with-tf-transform-pt1#where_to_do_preprocessing). In the example used for this set of articles, a new feature is created, **`mother_race_X_mother_age_bucketized`**, by crossing the **`mother_race`** and **`mother_age_bucketized`** features using **`tf.feature_column.crossed_column`**. Moreover, low-dimensional, dense representation of this crossed feature is created using the **`tf.feature_column.embedding_column`** feature column.

Figure 7 shows the transformed data and how the transformed metadata is used to define and train a Tensorflow estimator.

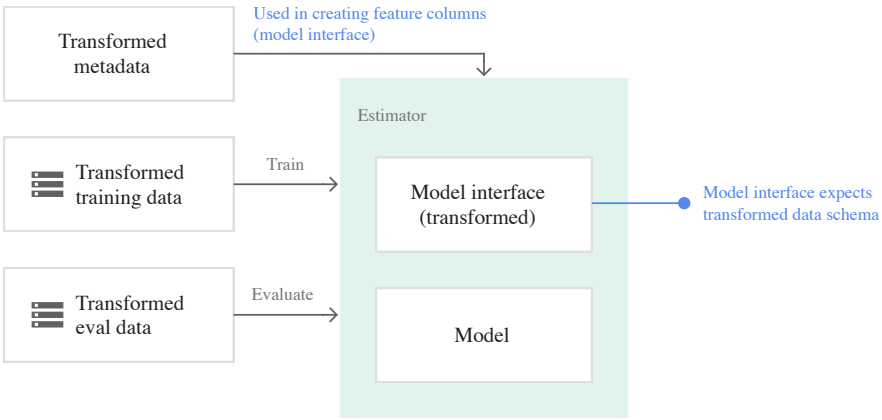

Figure 7. Training the TensorFlow model with the transformed data

Exporting the model for serving prediction

After you train the TensorFlow model (estimator), you export the estimator as a `SavedModel` object, so that it can serve new data points for prediction. When you export the model, you have to define its interface—that is, the input features schema that is expected during serving. This input features schema is defined in a `serving_input_fn`, as shown in the following code:

```
def serving_input_fn():

    from tensorflow_transform.saved import saved_transform_io

    # get the feature_spec of raw data
    raw_metadata = create_raw_metadata()

    # create receiver placeholders to the raw input features
    raw_input_features = raw_metadata.schema.as_batched_placeholders()
    raw_input_features.pop(TARGET_FEATURE_NAME)
    raw_input_features.pop(KEY_COLUMN)

    # apply transform_fn on raw features
    _, transformed_features = (
    saved_transform_io.partially_apply_saved_transform(
        os.path.join(TRANSFORM_ARTEFACTS_DIR,transform_fn_io.TRANSFORM_FN_DIR
    raw_input_features)
    )

    return tf.estimator.export.ServingInputReceiver(
        transformed_features, raw_input_features)

export_dir = os.path.join(model_dir, 'export')
```

```
if tf.gfile.Exists(export_dir):
    tf.gfile.DeleteRecursively(export_dir)

estimator.export_savedmodel(
    export_dir_base=export_dir,
    serving_input_receiver_fn=serving_input_fn
)
```

During serving, the model expects the data points in their raw form (that is, raw features before transformations). Thus, `serving_input_fn` uses the `raw_metadata` object to create receiver placeholders for the raw features. However, as discussed earlier, the trained model expects the data points in the transformed schema. Therefore, after receiving the raw features, you apply the saved `transform_fn` to the `raw_input_features` objects, received using the placeholders, to convert them into the `transformed_features` that are expected by the model interface. Figure 8 illustrates the final step of exporting a model for serving.

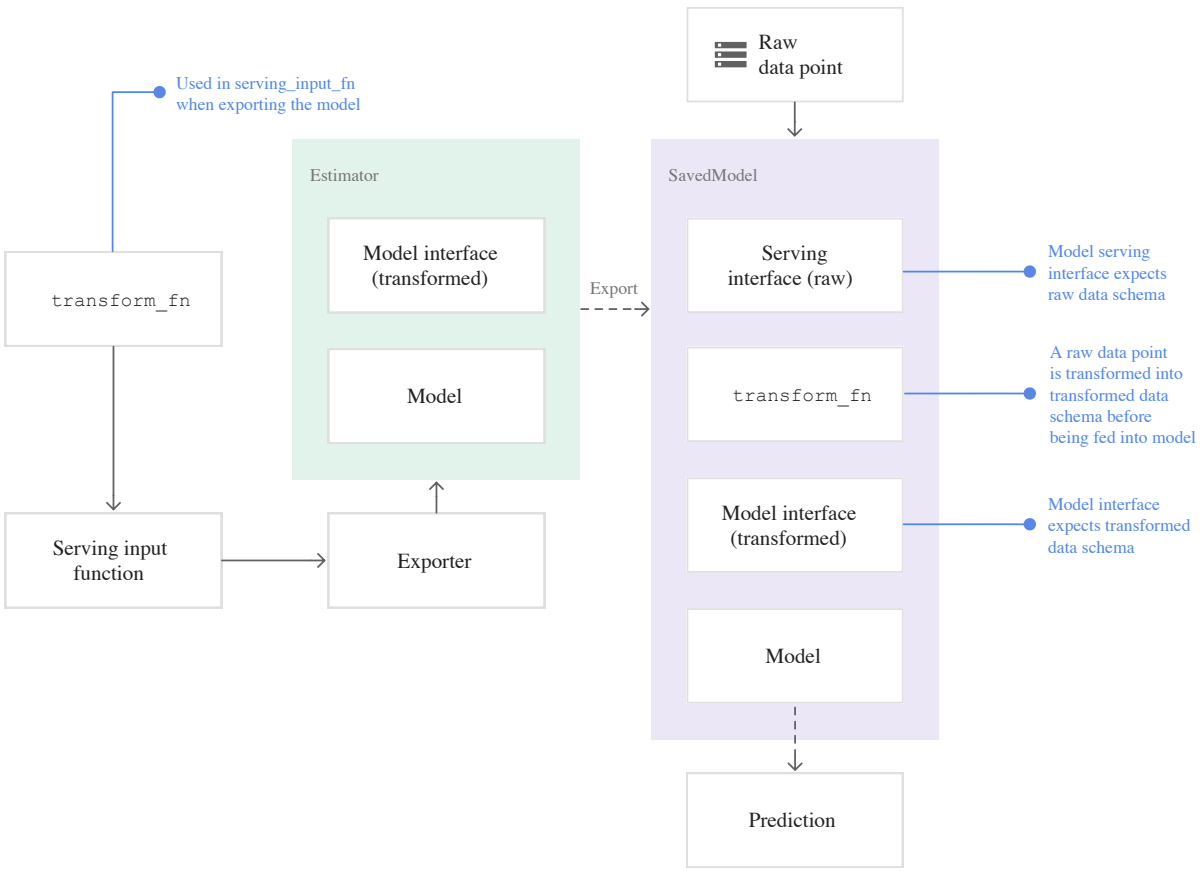

Figure 8. Exporting the model for serving with `transform_fn` attached

Train and use the model for predictions

You can train the model locally by running the cells of the notebook. For examples of how to package the code and train your model at scale using AI Platform, see the samples and guides in the Google Cloud cloudml-samples (https://github.com/GoogleCloudPlatform/cloudml-samples) GitHub repository.

When you inspect the exported `SavedModel` using the `saved_model_cli` tool, you see that the inputs of the `signature_def` include the raw features, as shown in Figure 9:

```
%%bash

saved_model_dir=${export_dir}/$(ls ${export_dir} | tail -n 1)
echo ${saved_model_dir}
ls ${saved_model_dir}
saved_model_cli show --dir=${saved_model_dir} --all
```

```
babyweight_tft/models/dnn_estimator/export/1535488965
assets
saved_model.pb
variables

MetaGraphDef with tag-set: 'serve' contains the following SignatureDefs:

signature_def['predict']:
  The given SavedModel SignatureDef contains the following input(s):
    inputs['gestation_weeks'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1)
        name: gestation_weeks:0
    inputs['is_male'] tensor_info:
        dtype: DT_STRING
        shape: (-1)
        name: is_male:0
    inputs['mother_age'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1)
        name: mother_age:0
    inputs['mother_race'] tensor_info:
        dtype: DT_STRING
        shape: (-1)
        name: mother_race:0
    inputs['plurality'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1)
        name: plurality:0
  The given SavedModel SignatureDef contains the following output(s):
    outputs['predictions'] tensor_info:
        dtype: DT_FLOAT
        shape: (-1, 1)
        name: add:0
  Method name is: tensorflow/serving/predict
```

Figure 9. The exported `SavedModel` interface

The last cell of the notebook shows you how to use the exported model for prediction. It is important to highlight that the input (sample) data point is in the raw schema. To learn about how to deploy the model as a microservice on AI Platform for online prediction, see the Deploying Models (/ml-engine/docs/tensorflow/deploying-models) documentation.

What's next