Data preprocessing for machine learning: options and recommendations

This two-part article explores the topic of data engineering and feature engineering for machine learning (ML). This first part discusses best practices of preprocessing data in a machine learning pipeline on Google Cloud. The article focuses on using TensorFlow and the open source TensorFlow Transform (https://github.com/tensorflow/transform) (`tf.Transform`) library to prepare data, train the model, and serve the model for prediction. This part highlights the challenges of preprocessing data for machine learning, and illustrates the options and scenarios for performing data transformation on Google Cloud effectively.

This article assumes that you are familiar with BigQuery (/bigquery/docs), Dataflow (/dataflow/docs), AI Platform (ML) Engine (/ml-engine/docs/tensorflow/technical-overview), and the TensorFlow Estimator (https://www.tensorflow.org/guide/estimators) API.

The second article provides a step-by-step tutorial of how to implement a `tf.Transform` pipeline. For details, see Data Preprocessing for Machine Learning on Google Cloud using tf.Transform - Part 2 (/solutions/machine-learning/data-preprocessing-for-ml-with-tf-transform-pt2).

Introduction

> *Coming up with features is difficult, time-consuming, requires expert knowledge. "Applied machine learning" is basically feature engineering.*
>
> —Andrew Ng, Stanford University (source (http://ai.stanford.edu/~ang/slides/DeepLearning-Mar2013.pptx))

Machine learning (ML) helps in automatically finding complex and potentially useful patterns in data. These patterns are condensed in an ML model that can then be used on new data points—a process called *making predictions* or *performing inference*.

Building an ML model is a multistep process. Each step presents its own technical and conceptual challenges. In this two-part set, we focus on the process of selecting, transforming, and augmenting the source data to create powerful predictive signals to the target (response) variable (in supervised learning tasks). These operations combine domain knowledge with data science techniques. They are the essence of feature engineering (https://developers.google.com/machine-learning/glossary/#feature_engineering).

The size of training datasets for real-world ML models can easily reach or surpass the terabyte (TB) mark. Hence, you need large-scale data processing frameworks in order to process these datasets efficiently and distributedly. Moreover, when you use an ML model for making predictions, you have to apply the same transformations that you used for the training data on the new data points, so that the live dataset is presented to the ML model the way that the model expects.

The first article discusses these challenges for different levels of granularity of feature engineering operations: instance-level, full-pass, and time-window aggregations. In addition, the article illustrates the options and scenarios for performing data transformation for ML on Google Cloud effectively.

The article also provides an overview of TensorFlow Transform (https://github.com/tensorflow/transform) (`tf.Transform`), a library for TensorFlow that allows you to define both instance-level and full-pass data transformation through data preprocessing pipelines. These pipelines are executed with Apache Beam (https://beam.apache.org/), and as by-products they create artifacts that let you apply the same transformations during prediction as when the model is served.

Preprocessing data for machine learning

This section introduces data preprocessing operations and stages of data readiness. It also discusses the types of the preprocessing operations and their granularity.

Data engineering compared to feature engineering

Preprocessing the data for ML involves both data engineering and feature engineering. Data engineering is the process of converting *raw data* into *prepared data*. Feature engineering then tunes the prepared data to create the features expected by the ML model. These terms have specific meanings, as outlined in the following list:

Raw data (or just data)

> This refers to the data in its source form, without any prior preparation for ML. Note that in this context, the data might be in its raw form (in a data lake) or in a transformed form (in a data warehouse). Transformed data in a data warehouse might have been converted from its original raw form to be used for analytics, but in this context it means that the data was not prepared specifically for your ML task. In addition, data sent from streaming systems that *eventually* call ML models for predictions is considered to be data in its raw form.

Prepared data

> This refers to the dataset in the form ready for your ML task. Data sources have been parsed, joined, and put into a tabular form. Data has been aggregated and summarized to the right granularity—for example, each row in the dataset represents a unique customer, and each column represents summary information for the customer, like the total spent in the last six weeks. In the case of supervised learning tasks, the target feature is present. Irrelevant columns have been dropped, and invalid records have been filtered out.

Engineered features

> This refers to the dataset with the tuned features expected by the model—that is, performing certain ML-specific operations on the columns in the prepared dataset, and creating new features for your model during training and prediction, as described later under Preprocessing operations (#preprocessing_operations). Examples include scaling numerical columns to a value between 0 and 1, clipping values, and one-hot-encoding categorical features.

Figure 1 illustrates the steps involved.

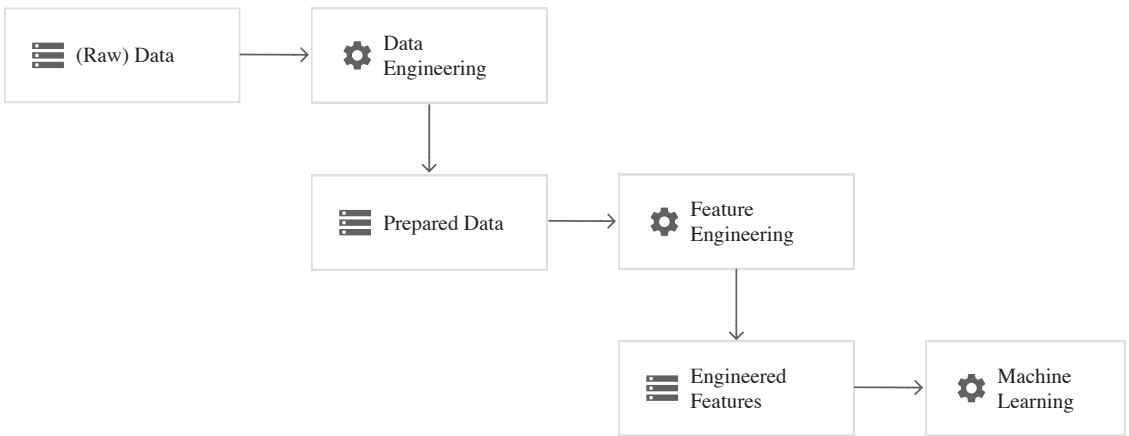

Figure 1. The flow of data from raw data to prepared data to engineered features to machine learning

In practice, data from the same source is often at different stages of readiness. For example, a field from a table in your data warehouse could be used directly as an engineered feature. At the same time, another field in the same table might need to go through transformations before becoming an engineered feature. Similarly, data engineering and feature engineering operations might be combined in the same data preprocessing step.

Preprocessing operations

Data preprocessing includes various operations. Each operation aims to help machine learning build better predictive models. The details of these preprocessing operations are outside the scope of this article, but for your reference, some of the operations are briefly discussed in this section.

For structured data, data preprocessing operations include the following:

- **Data cleansing**. Removing or correcting records with corrupted or invalid values from raw data, as well as removing records that are missing a large number of columns.
- **Instances selection and partitioning**. Selecting data points from the input dataset to create training, evaluation (validation), and test sets (https://wikipedia.org/wiki/Training,_test,_and_validation_sets). This process includes techniques for repeatable random sampling, minority classes oversampling, and stratified partitioning.
- **Feature tuning**. Improving the quality of a feature for ML, which includes scaling and normalizing numeric values, imputing missing values, clipping outliers, and adjusting values with skewed distributions.
- **Representation transformation**. Converting a numeric feature to a categorical feature (through bucketization), and converting categorical features to a numeric representation (through one-hot encoding (https://developers.google.com/machine-learning/glossary/#one-hot_encoding), learning with counts (https://blogs.technet.microsoft.com/machinelearning/2015/02/17/big-learning-made-easy-with-counts/), sparse feature embeddings, and so on). Some models work only with numeric or categorical features, while others can handle mixed type features. Even when models handle both types, they can benefit from different representation (numeric and categorical) of the same feature.
- **Feature extraction**. Reducing the number of features by creating lower-dimension, more powerful data representations using techniques such as PCA (https://wikipedia.org/wiki/Principal_component_analysis), embedding (https://developers.google.com/machine-learning/glossary/#embeddings) extraction, and hashing (https://medium.com/value-stream-design/introducing-one-of-the-best-hacks-in-machine-learning-the-hashing-trick-bf6a9c8af18f) .
- **Feature selection**. Selecting a subset of the input features for training the model, and ignoring the irrelevant or redundant ones, using filter or wrapper methods

(https://wikipedia.org/wiki/Feature_selection). This can also involve simply dropping features if the features are missing a large number of values.

- **Feature construction**. Creating new features either by using typical techniques, such as polynomial expansion (https://wikipedia.org/wiki/Polynomial_expansion) (by using univariate mathematical functions) or feature crossing (https://developers.google.com/machine-learning/glossary/#feature_cross) (to capture feature interactions). Features can also be constructed by using business logic from the domain of the ML use case.

When you work with unstructured data (for example, images, audio, or text documents), deep learning has gotten rid of the domain knowledge-based feature engineering by folding it into the model architecture. A convolutional layer (https://developers.google.com/machine-learning/glossary/#convolutional_layer) is an automatic feature preprocessor. Naturally, constructing the right model architecture requires some empirical knowledge of the data. In addition, some amount of preprocessing is needed, such as the following:

- For text documents, stemming and lemmatization, (https://nlp.stanford.edu/IR-book/html/htmledition/stemming-and-lemmatization-1.html) TF-IDF (https://wikipedia.org/wiki/Tf%E2%80%93idf) calculation, and n-gram (https://wikipedia.org/wiki/N-gram) extraction, embedding lookup.
- For images, clipping, resizing, cropping, Gaussian blur, and canary filters.
- Transfer learning (https://developers.google.com/machine-learning/glossary/#transfer_learning), in which you are treating all-but-last layers of the fully trained model as a feature engineering step. This applies to all types of data, including text and images.

Preprocessing granularity

This section discusses the granularity of types of data transformations. It shows why this perspective is critical when preparing new data points for predictions using transformations that are applied on training data.

Preprocessing and transformation operations can be categorized as follows, based on operation granularity:

- **Instance-level transformations during training and prediction**. These are straightforward transformations, where only values from the same instance (data point) are needed for the transformation. For example, this might include clipping the

value of a feature to some threshold, polynomially expanding another feature, multiplying two features, or comparing two features to create a Boolean flag.

These transformations have to be applied identically during training and prediction, because the model will be trained on the transformed features, not on the raw input values. If the data is not transformed identically, the model behaves poorly because it is presented with data that has a distribution of values that it was not trained with. For more information, see the discussion of training/serving skew in the preprocessing challenges (#preprocessing_challenges) section later in this document.

- **Full-pass transformations during training, but instance-level transformations during prediction**. In this scenario, transformations are stateful, because they have to use some precomputed statistics to perform the transformation. During training, you need to analyze the whole body of training data to compute quantities such as minimum, maximum, mean, and variance for transforming training data, evaluation data, and new data at prediction time.

  For example, to normalize a numeric feature for training, you need to compute its mean (μ) and its standard deviation (σ) across the whole of the training data. This is called a *full-pass* (or *analyze*) operation. Then when you serve the model for prediction, the value of a new data point has to be normalized to avoid training/serving skew. Thus, μ and σ values that are computed during training are used to adjust the feature value, which is a simple *instance-level* operation:

  $$value_{scaled} = (value_{raw} - \mu) \div \sigma$$

  Transformations that fall into these categories are:

  - MinMax scaling numerical features using *min* and *max* computed from the training dataset.
  - Standard scaling (z-score normalization) numerical features using μ and σ computed on the training dataset.
  - Bucketizing numerical features using quantiles.
  - Imputing missing values using the median (numerical features) or the mode (categorical features).
  - Converting strings (nominal values) to integers (indexes) by extracting all of the distinct values (vocabulary) of an input categorical feature.
  - Counting the occurrence of a term (feature value) in all of the documents (instances) to calculate for TF-IDF.

- Computing the PCA of the input features to project the data into a lower dimensional space (with linearly dependent features).

You should use only the training data to compute statistics like μ, σ, *min*, *max*, and so on. If you add the test and evaluation data for these operations, you are leaking information (https://towardsdatascience.com/data-leakage-in-machine-learning-10bdd3eec742) from the evaluation and test data to train the model. Doing so affects the reliability of the test and evaluation results. In addition, you use the same statistics computed from the training data to transform the test and evaluation data so that you apply the consistent transformation to all datasets.

- **Window aggregations during training and prediction**. This approach involves creating a feature by summarizing real-time values over time. That is, the instances to aggregate are defined through temporal window clauses. For example, imagine that you want to train a model that estimates the taxi trip time based on the traffic metrics for the route in the last 5 minutes, in the last 10 minutes, in the last 30 minutes, and at other intervals. Another example is predicting the failure of an engine part based on the moving average of temperature and vibration values computed over the last 3 minutes. Although these aggregations can be prepared offline for training, they have to be computed in real-time from a data stream during serving.

  More precisely, when you are preparing training data, if the aggregated value is not in the raw data, it is created during the data engineering phase. The raw data is usually stored in a database with format of `(entity, timestamp, value)`. In the previous examples, `entity` would be the route segment identifier for the taxi routes and the engine part identifier for the engine failure. You can use windowing operations to compute `(entity, time_index, aggregated_value_over_time_window)` and use the aggregation features as an input for your model training.

  However, when the model for real-time (online) prediction is being served, the model expects features derived from the aggregated values as an input. Thus, you can use a stream-processing technology like Apache Beam to compute the aggregations on the fly from the real-time data points streamed into your system. You can also perform additional feature engineering (tuning) to these aggregations before training and prediction.

Machine learning pipeline on Google Cloud

This section discusses the building blocks of a typical end-to-end pipeline to train and serve TensorFlow ML models on Google Cloud using managed services. It also discusses where

you can implement different categories of the data preprocessing operations, as well as common challenges you might face when you implement such transformations. Later, you will see how the TensorFlow Transform library helps address these challenges.

High-level architecture

Figure 2 shows a high-level architecture of a typical ML pipeline for training and serving TensorFlow models. The labels A, B, and C in the diagram refer to the different places in the pipeline where data preprocessing can take place. Details about these steps are discussed in the following section.

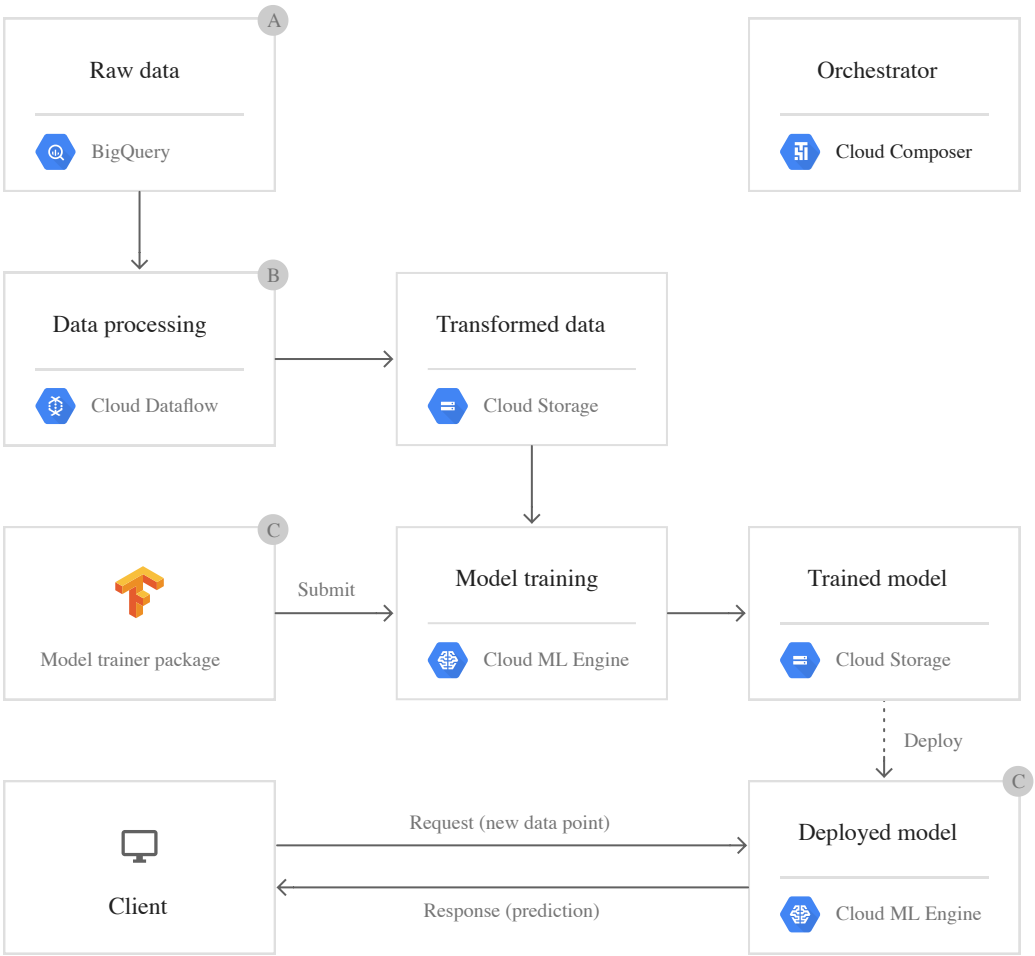

Figure 2. High-level architecture for ML training and serving on Google Cloud

The pipeline consists of the following steps:

1. After being imported, the raw data is stored in BigQuery (or in Cloud Storage, in the case of images, docs, audio, video, and so on).
2. Data engineering (preparation) and feature engineering are executed at scale using Dataflow. This produces ML-ready training, evaluation, and test sets that are stored in Cloud Storage. Ideally, these datasets are stored as files, which is the optimized format for TensorFlow computations.

3. A TensorFlow model trainer package (/ml-engine/docs/tensorflow/packaging-trainer) is submitted to AI Platform, which uses the preprocessed data from the previous steps to train the model. The output of this step is a trained TensorFlow SavedModel (https://www.tensorflow.org/guide/saved_model) that is exported to Cloud Storage.
4. The trained TensorFlow model is deployed to AI Platform as a microservice that has a REST API so that it can be used for online predictions. The same model can be used also for batch prediction jobs.
5. After the model is deployed as a REST API, client apps and internal systems can invoke this API by sending requests with some data points, and receiving responses from the model with predictions.
6. For orchestrating and automating this pipeline, Cloud Composer (/composer/docs) can be used as a scheduler to invoke the data preparation, model training, and model deployment steps.

Where to do preprocessing

As shown by the labels A, B, and C in Figure 2, there are three options where data preprocessing operations can take place: BigQuery, Dataflow, and TensorFlow.

Option A: BigQuery

Typically, the logic for the following operations is implemented in BigQuery:

- Sampling: randomly selecting a subset from the data.
- Filtering: removing irrelevant or invalid instances.
- Partitioning: splitting the data to produce training, evaluation, and test sets.

BigQuery SQL scripts can be used as a source query for the Dataflow preprocessing pipeline. This is the data processing step in Figure 2. For example, imagine that a system will be used in Canada, but the data warehouse has transactions from around the world. Filtering to get Canadian-only training data is best done in BigQuery.

It is possible to implement instance-level transformations, stateful full-pass transformations, and window aggregations feature transformations in a BigQuery SQL script to prepare the training data. However, this is not recommended.

If you are deploying the model for online predictions, you have to replicate the SQL preprocessing operations on the raw data points that you generated from other systems and that will be deployed model's API. In other words, you need to implement the logic

twice. The first time is in SQL to preprocess training data in BigQuery. The other time is in the logic of the app that consumes the model to preprocess online data points for prediction. For example, if your client app is written in Java, you need to reimplement the logic in Java. This can introduce errors due to implementation discrepancies. (See the discussion of training/serving skew in the preprocessing challenges (#preprocessing_challenges) section later in this document.)

In addition, maintaining two different implementations is extra overhead. Whenever you change the logic in SQL to preprocess the training data, you need to change the Java implementation accordingly to preprocess data at serving time.

If you are using your model only for batch prediction (scoring) using AI Platform batch prediction (/ml-engine/docs/tensorflow/batch-predict), and if your data for scoring is sourced from BigQuery, it is feasible to implement these preprocessing operations as part of the BigQuery SQL script. In that case, you can use the same preprocessing SQL script to prepare both training and scoring data. This implies that you are using auxiliary tables to store quantities needed by stateful transformations, such as means and variances to scale numerical features. It also means increased complexity in the SQL scripts, and intricate dependency between training and the scoring SQL scripts.

The following code snippets show hypothetical examples of BigQuery SQL for data preparation for training and prediction. In the first script (the training script), the mean and the standard deviation for fields `f1` and `f2` are stored in the `stats` table. Then you join the `training_data` table with the `stats` table to transform your training data.

```
# computing stats for normalizing numerical features: f1 and f2
CREATE OR REPLACE TABLE stats AS
(
  SELECT
    AVG(f1) AS f1_mean,
    STDDEV(f1) AS f1_stdv,
    AVG(f2) AS f2_mean,
    STDDEV(f2) f2_stdv
 FROM
    my_dataset.training_data
)
# extracting data for training
SELECT
  (data.f1 - stats.f1_mean)/stats.f1_stdv AS f1_NORMALIZED,
  (data.f2 - stats.f2_mean)/stats.f2_stdv AS f2_NORMALIZED,
  POWER((data.f1 - stats.f1_mean)/stats.f1_stdv,2) AS f1_NORMALIZED_SQUARED,
  POWER((data.f1 - stats.f1_mean)/stats.f1_stdv,2) AS f2_NORMALIZED_SQUARED,
  ((data.f1 - stats.f1_mean)/stats.f1_stdv) * ((data.f2 - stats.f2_mean)/stats
FROM
```

```
  my_dataset.training_data AS data
CROSS JOIN
  stats
WHERE #filtering data for training
  data.entry_year = 2018
AND
  data.entry_location = 'Canada'
```

The following listing shows the prediction script. In this script, the `stats` table is joined with the `prediction_data` table to prepare the data for scoring. Similarly, other instance-level transforms are replicated.

```
#extracting data for prediction
SELECT
  (data.f1 - stats.f1_mean)/stats.f1_stdv AS f1_NORMALIZED,
  (data.f2 - stats.f2_mean)/stats.f2_stdv AS f2_NORMALIZED,
  POWER((data.f1 - stats.f1_mean)/stats.f1_stdv,2) AS f1_NORMALIZED_SQUARED,
  POWER((data.f1 - stats.f1_mean)/stats.f1_stdv,2) AS f2_NORMALIZED_SQUARED,
  ((data.f1 - stats.f1_mean)/stats.f1_stdv) * ((data.f2 - stats.f2_mean)/stats
FROM
  my_dataset.prediction_data AS data
CROSS JOIN
  stats
```

Option B: Dataflow

As shown in Figure 2, you can implement computationally expensive preprocessing operations in Apache Beam, and run them at scale using Dataflow, which is a fully managed autoscaling service for batch and stream data processing.

Dataflow can perform instance-level transformations, stateful full-pass transformations, and window-aggregation feature transformations. In particular, if your ML models expect an input feature like `total_number_of_clicks_last_90sec`, Apache Beam windowing functions (https://beam.apache.org/documentation/programming-guide/#windowing) can compute these features based on aggregating the values of time windows of real-time (streaming) events data (for example, clicks). In the earlier discussion of granularity of transformations (#preprocessing_granularity), this was referred to as "Window aggregations during training and serving."

Figure 3 illustrates the role of Dataflow in processing stream data for near real-time predictions. In essence, events (data points) are ingested into Pub/Sub (/pubsub/docs).

Dataflow consumes these data points, computes features based on aggregates over time, and calls the deployed ML model API for predictions. Predictions are then sent to an outbound Pub/Sub queue. From there, the predictions can be consumed by downstream (monitoring or control) systems or pushed back (for example, as notifications) to the original requesting client. Another option is to store the predictions in a low-latency data store like Cloud Bigtable (/bigtable/docs) for real-time fetching. Cloud Bigtable can also be used to accumulate and store these real-time aggregations so they can be looked up when needed for prediction. Figure 3 illustrates this scenario.

The same Apache Beam implementation can be used to batch-process training data that comes from an offline datastore like BigQuery and stream-process real-time data for serving online predictions.

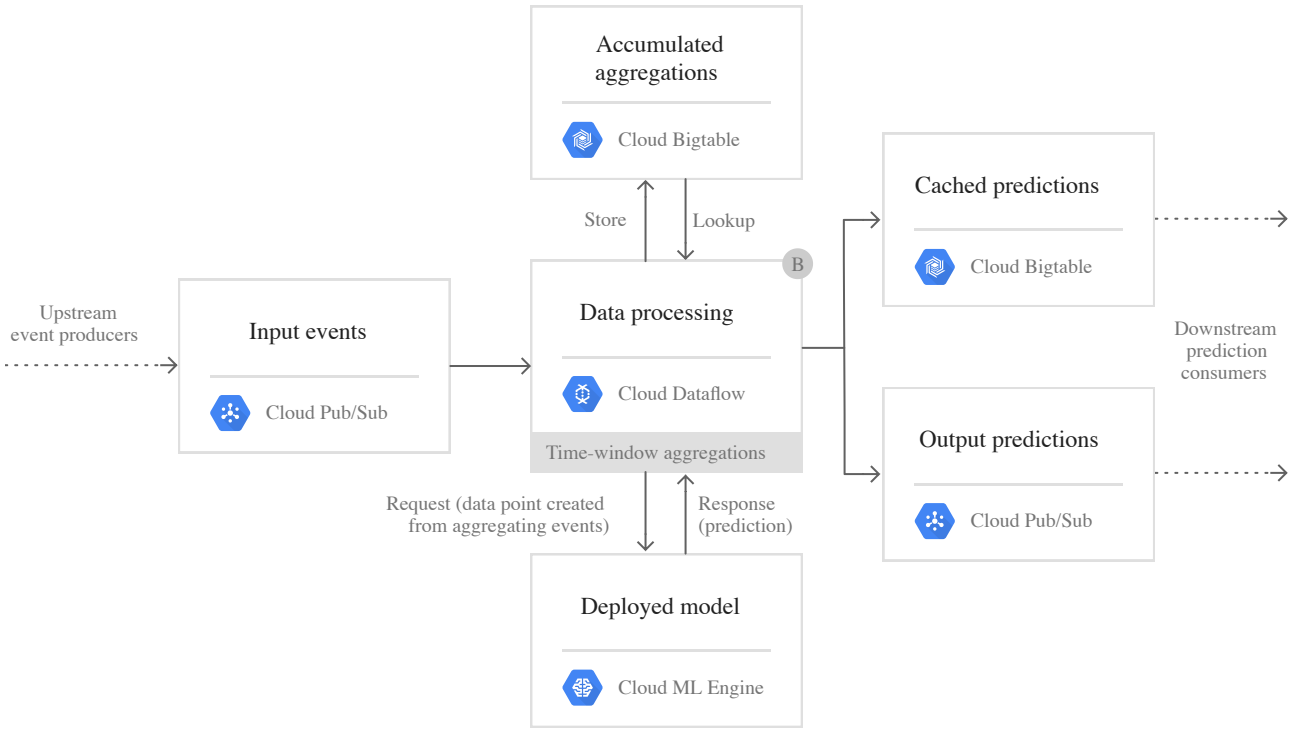

Figure 3. High-level architecture using stream data for prediction in Dataflow

In other typical architectures, the client app directly calls the deployed model API for online predictions (as shown earlier in Figure 2). In that case, if preprocessing operations are implemented in Dataflow to prepare the training data, these operations are not applied to the prediction data going directly to the model. Thus, transformations like these should be an integral part of the model during serving for online predictions.

Option C: TensorFlow

As you can see in Figure 2, you can implement data preprocessing and transformation operations in the TensorFlow model itself. As shown in the figure, the preprocessing you implement for training the TensorFlow model becomes an integral part of the model when

the model is exported and deployed for predictions. TensorFlow transformations can be accomplished in one of the following ways:

- Extending your base feature_columns (https://www.tensorflow.org/api_docs/python/tf/feature_column) (using `crossed_column`, `embedding_column`, `bucketized_column`, and so on).
- Implementing all of the instance-level transformation logic in a function that you call in all three input functions (https://www.tensorflow.org/guide/datasets_for_estimators): `train_input_fn`, `eval_input_fn`, and `serving_input_fn`.
- If you are creating custom estimators (https://www.tensorflow.org/guide/custom_estimators), putting the code in the `model_fn` function.

If you use the same transformation logic code in the `serving_input_fn` (https://www.tensorflow.org/guide/saved_model#savedmodels_from_estimators) function, which defines the serving interface of your `SavedModel` for online prediction, it assures that the same transformations used for preparing training data will be applied on new prediction data points during serving.

However, as highlighted earlier, full-pass transformations cannot be implemented in your TensorFlow model.

Preprocessing challenges

The following are the key challenges faced when implementing data preprocessing:

- **Training-serving skew**. Training-serving skew (https://developers.google.com/machine-learning/guides/rules-of-ml/#training-serving_skew) refers to a difference between effectiveness (predictive performance) during training and during serving. This skew can be caused by a discrepancy between how you handle data in the training and the serving pipelines. For example, if your model is trained on a logarithmically transformed feature, but it is presented with the raw feature during serving, the prediction output might not be accurate.

  Handling instance-level transformations can be straightforward if they become part of the model itself, as described earlier under Option C: TensorFlow (#option_c_tensorflow). In that case, the model serving interface expects raw data, while the model internally transforms this data before computing the output. The same transformations are applied on the raw training and prediction data points.

- **Full-pass transformations**. Implementing transformations in your TensorFlow model cannot be used with transformations like scaling and normalization. In transformations like these, some statistics (for example, *max* and *min* values to scale numeric features) must be computed on the training data beforehand, as described earlier under Option B: Dataflow (#option_b_cloud_dataflow). The values are then stored somewhere to be used during model serving for prediction as instance-level transformations to transform the new raw data points, which avoids training-serving skew.
- **Preparing the data up front for better training efficiency**. Implementing instance-level transformations as part of the model can affect the efficiency of the training process. Imagine that you have raw training data with 1000 features, and you apply a mix of instance-level transformations to generate 10,000 features. If you implement these transformation as part of your model, and if you then feed the model the raw training data, these 10,000 operations are applied *N* times on each instance, where *N* is the number of epochs. On top of that, if you are using accelerators (GPUs or TPUs), they sit idle while the CPU is performing those transformations, which is not an efficient use of your costly accelerators.

  Ideally, the training data is transformed before training using the technique described under Option B: Dataflow (#option_b_cloud_dataflow), where the 10,000 transformation operations are applied only once on each training instance. The transformed training data is then presented to the model. No further transformations are applied, and the accelerators are busy all of the time. In addition, using Dataflow helps preprocessing large amount of data at scale, using a fully managed service.

You can see how preparing the training data up front can improve training efficiency. However, implementing the transformation logic outside of the model (the approaches described under Option A: BigQuery (#option_a_bigquery) or Option B: Dataflow (#option_b_cloud_dataflow)) does not resolve the issue of training-serving skew. This transformation logic would need to be implemented somewhere to be applied on new data points coming for prediction, because the model interface is now expecting transformed data. The TensorFlow Transform (`tf.Transform`) library can address this issue.

How tf.Transform works

The `tf.Transform` library is useful for transformations that require a full pass. The output of `tf.Transform` is exported as a TensorFlow graph that represents the instance-level transformation logic, as well as the statistic computed from full-pass transformations, 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 in a batch processing pipeline on Dataflow to prepare the training data up front and improve training efficiency.

The following diagram shows `tf.Transform` preprocessing and transforming data for training and prediction.

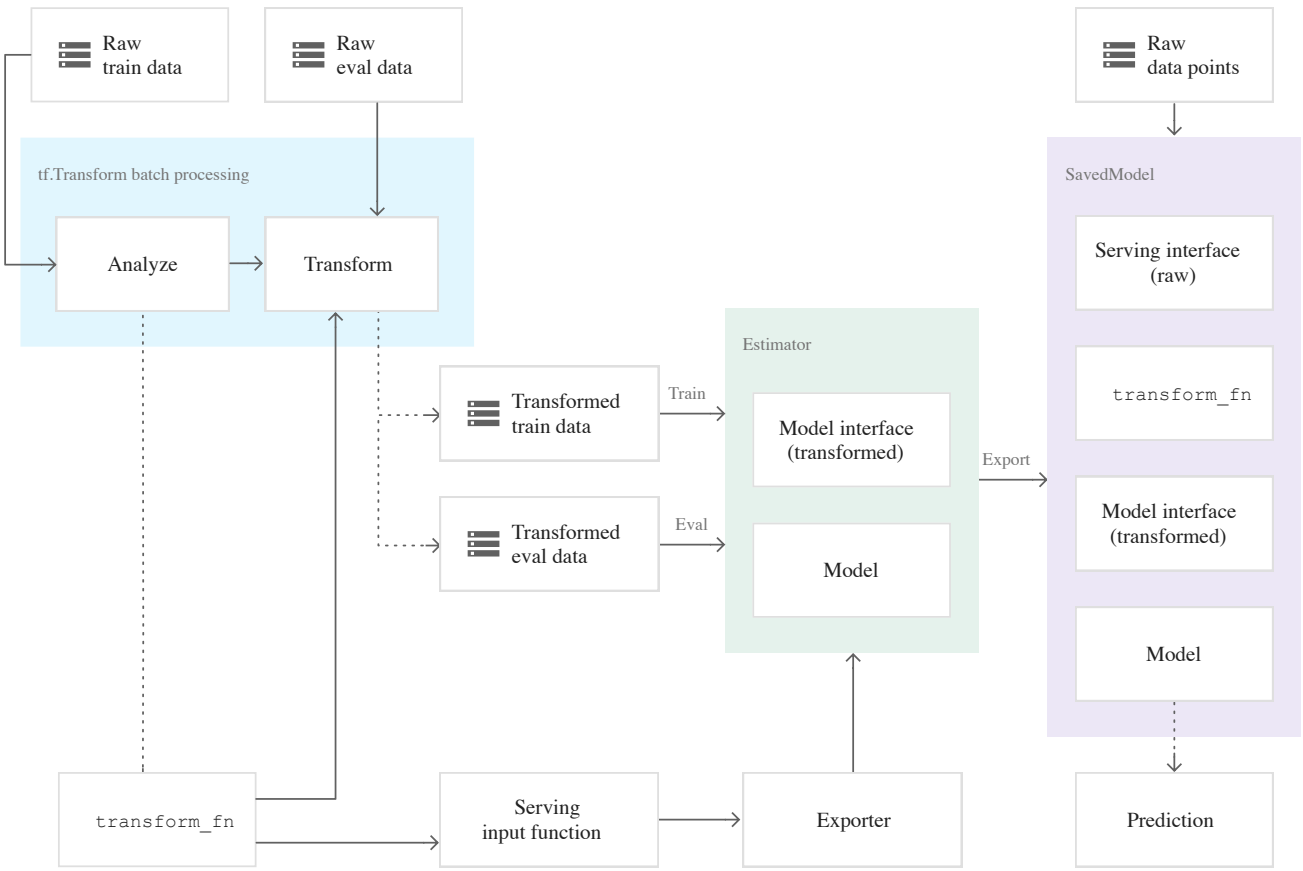

Figure 4. Behavior of `tf.Transform` for preprocessing and transforming data

Transforming training and evaluation data

You preprocess the raw training data using the transformation implemented in the `tf.Transform` Apache Beam APIs, and running it at scale on Dataflow. The preprocessing occurs in two phases:

- **Analyze phase**. During the analyze phase, the required statistics (like means, variances, and quantiles) for stateful transformations are computed on the training data with full-pass operations. This phase produces a set of transformation artifacts, including the `transform_fn`. The `transform_fn` is a TensorFlow graph that has the

transformation logic as instance-level operations, and includes the statistics computed in the analyze phase as constants.

- **Transform phase**. During the transform phase, the `transform_fn` is applied to the raw training data, where the computed statistics are used to process the data records (for example, to scale numerical columns) in an instance-level fashion.

A two-phase approach like this manages to address the preprocessing challenge (#preprocessing_challenges) of performing full-pass transformations.

In order to preprocess the evaluation data, only instance-level operations are applied, using the logic in the `transform_fn`, as well as 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, such as μ and σ, to normalize 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.

The transformed training and evaluation data is prepared at scale, using Dataflow, before it is used to train the model. This batch data-preparation process addresses the preprocessing challenge (#preprocessing_challenges) of preparing the data up front to improve training efficiency. Note that the model internal interface expects transformed features, as shown in Figure 4.

Attaching transformations to the exported model

As noted, the `transform_fn` produced by the `tf.Transform` pipeline is stored as an exported TensorFlow graph, which consists of the transformation logic as instance-level operations, as well as all of the statistics computed in the full-pass transformations as graph constants. When the trained model is exported for serving, the `transform_fn` graph is attached to the `SavedModel` as part of its `serving_input_fn`.

While it is serving the model for prediction, the model serving interface expects data points in the raw format (that is, before any transformations). However, the model internal interface expects the data in the transformed format.

The `transform_fn`, which is now part of the model, applies all of the preprocessing logic on the incoming data point. It uses the stored constants (like μ and σ to normalize the numeric features) in the instance-level operation during prediction. Thus, the `transform_fn` converts the raw data point into the transformed format, which is what is expected by the model internal interface in order to produce prediction, as shown in Figure 4.

This resolves the preprocessing challenge (#preprocessing_challenges) of the training/serving skew, because the same logic (implementation) that is used to transform the training and

evaluation data is applied to transform the new data points during prediction serving.

Preprocessing options summary

The following table summarizes the data preprocessing options that were discussed in this article. The table is organized as follows:

- The rows represent the tools that you can use to implement your transformations.
- The columns represent the types of the transformation by granularity.
- The subcolumns (for example, **Batch Scoring**) represent a model-serving requirement.
- Each cell represents the recommendation of using the tool (row) to implement a transformation type (column) with respect to the serving requirement (subcolumn).

<table>
<tr><th></th><th colspan="2">Instance-level
(stateless transformations)</th><th colspan="2">Full-pass during training instance-level during serving (stateful transformations)</th><th colspan="2">Real-time (window) aggregations during training and serving (streaming transformations)</th></tr>
<tr><th></th><th>Batch scoring</th><th>Online prediction</th><th>Batch scoring</th><th>Online prediction</th><th>Batch scoring</th><th>Online prediction</th></tr>
<tr><td>BigQuery (SQL)</td><td>OK

The same transformation implementation is applied on data during training and batch scoring.</td><td>Possible to process training data but not recommended

Results in training/serving skew, because you process serving data using different tools.</td><td colspan="2">Possible to use statistics computed using BigQuery for instance-level batch/online transformations.

Not easy; you must maintain a stats store to be populated during training and used during prediction.</td><td>N/A

Aggregates like these computed based on real-time events.</td><td>Possible to process training data but not recommended.

Results in training/serving skew, because you process serving data using different tools.</td></tr>
<tr><td>Dataflow (Apache Beam)</td><td></td><td>OK if data at serving time comes from Pub/Sub to be consumed by Dataflow. Otherwise, results in training/serving skew.</td><td colspan="2">Possible to use statistics computed using Dataflow for instance-level batch/online transformations.

Not easy; you must maintain a stats store to be populated during training and used during prediction.</td><td></td><td>OK

The same Apache Beam transformation is applied on data during training (batch) and serving (stream).</td></tr>
</table>

Dataflow (Apache Beam + TFT)	**OK** The same transformation implementation is applied to data during training and batch scoring.	**Recommended** Avoids training/serving skew and prepares training data up front.	**Recommended** Transformation logic + computed statistics during training are stored as a `tf.Graph` that's attached to the exported model for serving.		
TensorFlow* (`input_fn` & `serving_input_fn`)	**Possible** but **not recommended** For training & prediction efficiency, it's better to prepare the training data up front.	**Possible** but not **recommended** For training efficiency, it's better to prepare the training data up front.	**Not Possible**		**Not Possible**

* Transformations such as crossing, embedding, and one-hot encoding should be performed declaratively as feature_columns.

What's next

- Read the second part of the article (/solutions/machine-learning/data-preprocessing-for-ml-with-tf-transform-pt2) for a tutorial for implementing a `tf.Transform` pipeline and running it using Dataflow.
- Learn about the Architecture of a Serverless Machine Learning Model (/solutions/architecture-of-a-serverless-ml-model).
- Learn about Comparing Machine Learning Models for Predictions in Dataflow Pipelines (/solutions/comparing-ml-model-predictions-using-cloud-dataflow-pipelines).
- Take the 5-course Coursera specialisation on Machine Learning with TensorFlow on Google Cloud (https://www.coursera.org/specializations/machine-learning-tensorflow-gcp).
- Learn about best practices for ML engineering in Rules of ML (https://developers.google.com/machine-learning/guides/rules-of-ml/).
- Try out other Google Cloud features for yourself. Have a look at our tutorials (/docs/tutorials).