A CONSTRAINTS IN THE DATA SCHEMA

```
message Feature {
  ...

  // Limits the distribution drift between training
  // and serving data.
  FeatureComparator skew_comparator;

  // Limits the distribution drift between two
  // consecutive batches of data.
  FeatureComparator drift_comparator;
}
```

Figure 8. Extensions to the `Feature` message of `Schema` to check for distribution drifts.

We explain some of these feature level characteristics below using an instance of the schema shown in Figure 3:

Feature type: One of the key invariants of a feature is its data type. For example, a change in the data type from integer to string can easily cause the trainer to fail and is therefore considered a serious *anomaly*. Our schema allows specification of feature types as `INT`, `FLOAT`, and `BYTES`, which are the allowed types in the `tf.train.Example` (tfe) format. In Figure 3, the feature "event" is marked as of type `BYTES`. Note that features may have richer semantic types, which we capture in a different part of the schema (explained later).

Feature presence: While some features are expected to be present in all examples, others may only be expected in a fraction of the examples. The `FeaturePresence` field can be used to specify this expectation of presence. It allows specification of a lower limit on the fraction of examples that the feature must be present. For instance, the property `presence: {min_fraction: 1}` for the "event" feature in Figure 3 indicates that this feature is expected to be present in all examples.

Feature value count: Features can be single valued or lists. Furthermore, for features that are lists, they may or may not all be of the same length. These value counts are important to determine how the values can be encoded into the low-level tensor representation. The `ValueCount` field in the schema can be used to express such properties. In the example in Figure 3, the feature "event" is indicated to be a scalar as expressed using the `min` and `max` values set to 1.

Feature domains: While some features may not have a restricted domain (for example, a feature for "user queries"), many features assume values only from a limited domain. Furthermore, there may be related features that assume values from the same domain. For instance, it makes sense for two features like "apps_installed" and "apps_used" to be drawn from the same set of values. Our schema allows specification of domains both at the level of individual features as well as at the level of the schema. The named domains at the level of schema can be shared by all relevant features. Currently, our schema only supports shared domains for features with string domains.

A domain can also encode the semantic type of the data, which can be different than the raw type captured by the `TYPE` field. For instance, a bytes features may use the values "TRUE" or "FALSE" to essentially encode a boolean feature. Another example is an integer feature encoding categorical ids (e.g., enum values). Yet another example is a bytes feature that encodes numbers (e.g., values of the sort "123"). These patterns are fairly common in production and reflect common practices in translating structured data into the flat `tf.train.Example` format. These semantic properties are important for both data validation and understanding, and so we allow them to be marked explicitly in the domain construct of each feature.

Feature life cycle: The feature set used in a machine learning pipeline keeps evolving. For instance, initially a feature may be introduced only for experimentation. After sufficient trials, it may be promoted to a *beta* stage before finally getting upgraded to be a *production* feature. The gravity of anomalies in the features at different stages is different. Our schema allows tagging of features with the stage of life cycle that they are currently in. The current set of stages supported are `UNKNOWN_STAGE`, `PLANNED`, `ALPHA`, `BETA`, `PRODUCTION`, `DEPRECATED`, and `DEBUG_ONLY`

Figure 2 only shows only a fragment of the constraints that can be expressed by our schema. For instance, our schema can encode how groups of features can encode logical sequences (e.g., the sequence of queries issued by a user where each query can be described with a set of features), or can express constraints on the distribution of values over the feature's domain. We will cover some of these extensions in Section 4, but we omit a full presentation of the schema in the interest of space.

B ADDITIONAL CASE STUDIES

Product B This product employs ML to optimize the collection of customer feedback within apps. The team set up a pipeline for end-to-end ML training and serving and was able to diagnose several data errors: (a) serving data had more features than the training data, (b) one feature had unexpected string values over a large percentage of the examples in training, and (c) another feature had values that were present in the serving data, but never in the training data. The first two anomalies pointed to bugs in the data-generation code for serving and training, respectively. However, the show-stopper was the last anomaly since it resulted in significant training-serving skew that ultimately affected model quality. By inspecting the offending feature

the team was able to diagnose that the string feature was taking lower-case values in the serving data and camel-case values in the training data. This is a typical bug that results from having separate code paths for training- and serving-data generation. The fix was again easy (the team added a lower-casing transform during training) and resulted in a measurable improvement in model quality.

Product C This product generates lists that are "similar" to a remarketing seed list in the hope of reaching out to the most relevant audience outside of the seed list. The team was migrating their machine learning infrastructure from one machine learning platform to another. During this migration, validation detected that a significant fraction of training data had missing features, which prompted the team to investigate further and discover a bug in the data generation pipeline.