Hands on Tensorflow Data Validation

Vincent Teyssier Follow

Sep 13, 2018 · 8 min read

Google just released their new piece for an end to end big data platform, TFDV! One of the big pain in data science is to handle data quality issue, ie

data validation. Let's see how google delivers on this first version and how useful this new library is.

1. Installation

Quite a standard installation process via pip, however make sure you have pre-installed a few dependencies to be sure it compiles without issues. In my case, on Ubuntu 16.04, I was missing python-dev and python-snappy for faster performance. Bazel is also good to have if you start to use TFX libraries.

```
sudo apt-get install python-dev python-snappy
pip install tensorflow-data-validation
```

This TFDV being a rather new library, the documentation is pretty…. inexistant, so I strongly advise you to clone the repository and look at what parameters each function you use are accepting.

2. Loading data and generating statistics:

No rocket science here, we first import the library, and the first step in your dataset analysis/validation is to generated_statistics on your dataset. I will load a csv here and specify a separator, but you can also load a TFRecords instead. For convenience here is both function definition headers:

```
import tensorflow_data_validation as tfdv


def generate_statistics_from_csv(
data_location,
column_names = None,
delimiter = ',',
output_path = None,
stats_options = stats_api.StatsOptions(),
pipeline_options = None,):


...


def generate_statistics_from_tfrecord(
data_location,
output_path = None,
stats_options = stats_api.StatsOptions(),
pipeline_options = None,):
```

Let's load and generate our stats:

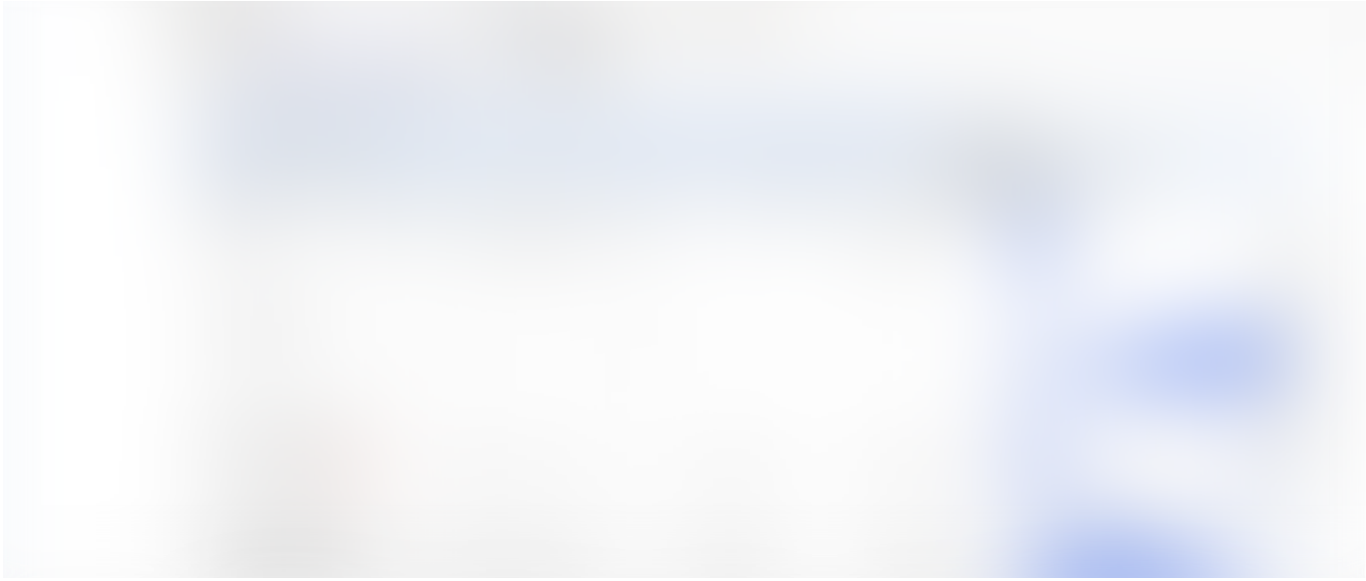
```
BASE_DIR = os.getcwd()
DATA_DIR = os.path.join(BASE_DIR, 'data')
TRAIN_DATA = os.path.join(DATA_DIR, 'train.csv')


train_stats = tfdv.generate_statistics_from_csv(TRAIN_DATA,
delimiter=';')
```

First thing to notice is that the operation is quite memory intensive. My dataset was only a 86Mb csv file, but the RAM usage went up by nearly 1Gb, so make sure you have a lot of RAM available!

3. Visualizing the statistics:

Once the stats are generated, you have 2 options to vizualize them. Either you use **Facets Overview**, which can be tricky to install depending on your platform, or you can use the built in TFDV visualization function which provides exactly the same information as facets:

```
tfdv.visualize_statistics(stats)
```



It is pretty useful to spot immediately if you have high amount of missing data, high standard deviation, etc…. however I really like Facets for the Dive module which let's you explore in a very visual way how your dataset looks like.

So far nothing revolutionary…. but here comes the big deal…

4.Describe your dataset for future validation

The main feature of TFDV is the concept of "schema". This is basically a description of how your data look like so you match this description against

new coming data and validate them… or not.

It describes standard characteristics of your data such as column datatypes, presence/abscence of data, expected range of values.

You create your schema on a dataset that you consider as a reference dataset and can reuse it to validate other sets of the same structure.

One more interesting feature is that you can reuse this schema in TFTransform to automatically declare your dataset structure.

That's where things become easy… to do that, only one line of code is sufficient:

```
schema = tfdv.infer_schema(train_stats)
```

However, as simply as it looks, the Tensorflow team comes with a warning:

> *In general, TFDV uses conservative heuristics to infer stable data properties from the statistics in order to avoid overfitting the schema to the specific*

> *dataset. It is strongly advised to* ***review the inferred schema and refine it as needed****, to capture any domain knowledge about the data that TFDV's heuristics might have missed.*

To store your schema, TFDV uses the protobuf library, which is becoming a unified method to manipulate your static data (datastructure, transformation scheme, frozen models…).

If there is one point on which we may all agree, it is that no dataset is perfect. Domain knowledge is essential when it comes to validate your data, hence why an automatic tool is a kind of unicorn! To take this into account, TFDV comes with helper functions so you manually hard code rules that your dataset shouldn't derive from.
Each feature is coming with a set of property that you can access and modify. For example let's describe that we want feature f1 to be populated in at least 50% of the examples, this is achieved by this line:

```
tfdv.get_feature(schema, 'f1').presence.min_fraction = 0.5
```

Each feature is composed of the following components:
Feature name, Type, Presence, Valency, Domain

And each component then gets its own subset of sub-components. I won't list them here, but the best advice I can give you is to parse the schema protobuf and you will find something looking like that:

```
num_examples:1000000000000000
weighted_num_examples: 1000000000000000
features {
  name: "one_in_a_quadrillion"
  type:INT
  num_stats: {
    common_stats: {
      num_missing: 1
      num_non_missing: 1000000000000000
      min_num_values: 1
      max_num_values: 1
      weighted_common_stats {
        num_non_missing: 1000000000000000
        num_missing: 1
      }
    }
  min: 0.0
  max: 10.0
}
```

You can also simply display it in your notebook like this:

```
tfdv.display_schema(schema)
```

5. Validate newer data

Your “perfect” training set is now described, but you keep getting fresh data everyday to ETL, and look for a way to validate the new coming set, here is the core of TFDV. Using the previously described schema/domain…. it will parse your new set and report outliers, missing or wrong data.

I think the picture included in the tutorial is a perfect illustration of what a nice pipeline data validation chain should look like:

When you try to productify a data ingestion pipeline for your models, this is basically what you try to achieve.

Let’s get a new CSV file, load it, generate stats, and parse it for validation using the previously generated schema:

```
NEW_DATA = os.path.join(DATA_DIR, 'test.csv')


new_csv_stats = tfdv.generate_statistics_from_csv(NEW_DATA,
delimiter=';')

anomalies = tfdv.validate_statistics(statistics=new_csv_stats,
schema=schema)
```

You can then display these anomalies the following way:

```
tfdv.display_anomalies(anomalies)
```

And you will get a list containing one or several of the following error messages describing which conditions the new dataset is not fullfilling knowing the expected conditions from the schema:

```
Integer larger than 1
BYTES type when expected INT type
BYTES type when expected STRING type
FLOAT type when expected INT type
FLOAT type when expected STRING type
INT type when expected STRING type
Integer smaller than 1
STRING type when expected INT type
Expected a string, but not the string seen
BYTES type when expected STRING type
FLOAT type when expected STRING type
INT type when expected STRING type
Invalid UTF8 string observed
Unexpected string values
The number of values in a given example is too large
The fraction of examples containing a feature is too small
The number of examples containing a feature is too small
The number of values in a given example is too small
No examples contain the value
The feature is present as an empty list
The feature is repeated in an example, but was expected to be a
```

```
singleton
There is a float value that is too high
The type is not FLOAT
There is a float value that is too low
The feature is supposed to be floats encoded as strings, but there is
a string that is not a float
The feature is supposed to be floats encoded as strings, but it was
some other type (INT, BYTES, FLOAT)
The type is completely unknown
There is an unexpectedly large integer
The type was supposed to be INT, but it was not.
The feature is supposed to be ints encoded as strings, but some
string was not an int.
The type was supposed to be STRING, but it was not.
There is an unexpectedly small integer
The feature is supposed to be ints encoded as strings, but it was
some other type (INT, BYTES, FLOAT)
Unknown type in stats proto
There are no stats for a column at all
There is a new column that is not in the schema.
Training serving skew issue
Expected STRING type, but it was FLOAT.
Expected STRING type, but it was INT.
Control data is missing (either training data or previous day).
Treatment data is missing (either scoring data or current day).
L infinity between treatment and control is high.
No examples in the span.
```

I have to say that I felt a bit disappointed by the result here…. I know we are only talking about data validation, but pointing at an error and not returning a subset containing these errors feels a bit like an unfinished work.

I also know that this feature is a quite expensive one, if you take yearly

license from packages like Talend or Alooma, pricing between 10k usd to 80k usd, you get a nice stream to handle your defects, but I believe TF will take this road sooner or later!

You still have a few more details in the column “Anomaly long description” so you should be fine with interpretability of what you have.

What is also nice is that if you believe the deviation from the schema is to be expected, you can amend the original schema easily:

```
tfdv.get_domain(schema, 'f2').value.append('new_unique_value')
```

6. Adapting to different dataset configurations:

You may want to validate your set using the schema definition, but depending on the context (in the tutorial example we take training and predicting, ie labels/no labels) you might have need to disregard some conditions. This is achievable the following way:

```
# All features are by default in both TRAINING and SERVING
environments.


schema.default_environment.append('TRAINING')
schema.default_environment.append('SERVING')

# Specify that labels column is not in SERVING environment.
tfdv.get_feature(schema,
'labels').not_in_environment.append('SERVING')

serving_anomalies_with_env = tfdv.validate_statistics(
        serving_stats, schema, environment='SERVING')
```

7. Comparing different sets in the visualization tool

One picture says it all:

You get the benefit of all the above in one comparative visualization. You are however limited to 2 sets:

```
tfdv.visualize_statistics(lhs_statistics=train_stats,
rhs_statistics=test_stats, lhs_name='TRAIN', rhs_name='TEST')
```

8. Re using your schema in TF Transform

As previously explained the schema you generate will avoid you to manually describe your features types. It can be loaded in TFTransform the following way:

```
feature_spec =
schema_utils.schema_as_feature_spec(schema).feature_spec

schema = dataset_schema.from_feature_spec(feature_spec)
```

9. Final note

I will skip the skew and drift comparator since they are an extension of the logic previously explained, and you may want to look deeper in the git for how you want to use these features in TFDV. Both are very useful!

To conclude, TFDV is exactly what it stands for, a data validation tool, nothing more, nothing less, that integrates perfectly with the Tensorflow ecosystem, providing more automation for TFTransform and completing the end-to-end framework that Google is trying to provide for machine learning practitioners. It still feels that defect handling is missing, but given the different libraries that have been offered to us to lighten our pain points, I believe this is coming soon! The next planned release by Google is Data Ingestion module, then the jobs management, orchestration, monitoring…as seen on top of the below picture.

You can find the full official TFDV tutorial there:

Get started with Tensorflow Data Validation | TFX | TensorFlow

In short, the schema describes the expectations for "correct" data and can thus be used to detect errors in the data...

www.tensorflow.org

Data Science TensorFlow Big Data Machine Learning AI