Data Quality in ML Pipelines

BEAM SUMMIT NYC 2025

beam X TensorFlow Extended >> Bad Data

Agenda

- Defining the problem of Data Quality
- What can we do about it?
- What have we done about it?
- What still remains?

Data Quality: An Analogy

To have a smooth and safe trip, all travelers must have:

- Identifying documents
- Boarding Pass with group
- Go through security
- Stay with their traveling party

Implementation by Analogy

Physical World	**Machine Learning**
A trip	ML Pipeline
Passport	Example Identifier + value of feature
Boarding Group	Split
Population Demographics	Statistics (The Shadow)
Description of traveling party	Schema
Traveling Party	Example
Luggage Tag	DQ Features
Security Check	The Prism
Check in	Transformed Features
Population	Entire Dataset (The Sun)
Sub-population	Subset (The beam)

Data Quality Challenges in ML Pipelines

- The data is in the pipeline. We are outside the pipeline.
- The data can be very large and messy
- Variety of formats to deal with at different stages
- Hard to see connection between data and its effect on models
- Good data is hard to find: > 85% of the effort/code is not actually machine learning, it is data processing

How Beam can help

Beam speaks a thousand formats. No data is outside of Beam's reach.

Beam + TFX reduce the surface area of skills required to do professional grade machine learning.

Beam, with its ability to execute user defined functions (UDFs) on behalf of the user, can reduce the burden of data processing at scale while abstracting the complexity away.

TFX, with its component architecture, can manage the end-to-end trip, using Beam wherever distributed computation is appropriate.

ML Metadata helps us open up the pipeline, without spilling a drop of data. Squeaky clean!

How Beam Can Help with Data Quality for ML

Step	How Beam Helps	Specifics
Pre-Ingestion	Determine Schema for Data	Schema Generator
Data Splitting & Identification	Deterministic Data Splitting at Scale	PartitionDoFn
Data Ingestion	Encoding Data at scale	ExampleGen (Standard)
Data Profile	Statistics	StatisticsGen (Standard)
Data Exploration	Produce JSON from TFRecord	JSONSampler
Non-graph intended feature injection	Apply Arbitrary Python UDFs	PreTransform
Filter	Remove examples which do not meet DQ requirements	FilterUDF

Schema Generation

Schema aware PCollections simplify data processing and quality greatly.

Computing a schema, using all of the data, can be computationally difficult

With beam python sdk, you can process each element as a string, and use functions such as yaml.safe_load(element) to determine type of element

You can then compute a rough schema that you can tune

```
(2, [('str', 100001)])
(3, [('float', 97924), ('int', 2076), ('str', 1)])
(5, [('NoneType', 1), ('str', 1), ('int', 2082), ('float', 97917)])
(1, [('str', 1), ('int', 16213), ('float', 83787)])
(0, [('datetime.datetime', 100000), ('str', 1)])
(7, [('str', 1), ('int', 100000)])
(4, [('float', 97935), ('str', 1), ('int', 2065)])
(6, [('str', 1), ('NoneType', 1), ('int', 2065), ('float', 97934)])
(3, {'name': ['pickup_longitude'], 'schema': [('float', False)]})
(6, {'name': ['dropoff_latitude'], 'schema': [('float', True)]})
(7, {'name': ['passenger_count'], 'schema': [('int', False)]})
(0, {'name': ['key'], 'schema': [('datetime.datetime', False)]})
(1, {'name': ['fare_amount'], 'schema': [('float', False)]})
(5, {'name': ['dropoff_longitude'], 'schema': [('float', True)]})
(2, {'name': ['pickup_datetime'], 'schema': [('str', False)]})
(4, {'name': ['pickup_latitude'], 'schema': [('float', False)]})
from typing import NamedTuple, Optional
import datetime
class MyRecord(NamedTuple):
    key: datetime.datetime
    fare_amount: float
    pickup_datetime: str
    pickup_longitude: float
    pickup_latitude: float
    dropoff_longitude: Optional[float]
    dropoff_longitude_missing: float
    dropoff_latitude: Optional[float]
    dropoff_latitude_missing: float
    passenger_count: int
```

Data Splitting & Identification

For non time series data, you can do deterministic splitting using hashing algorithms if applicable.

For temporal data, you can use spans in ExampleGen in TFX

Or, you can use beam.Partition

You can use a DoFn if you want to add in other information, such as a deterministic UUID, missing feature indicators, split information

Ideally, the uuid integrates split information.

Here we do beam based approximate quantiles unless user provides split point. Use a compiled language like Go to determine split point under a minute.

```
split_point_object = MyRecordWithUUID(
    key=datetime.datetime(
        2012, 6, 2, 20, 43, tzinfo=datetime.timezone.utc),
    fare_amount=15.7, pickup_datetime='2012-06-01 14:23:00 UTC',
    pickup_longitude=-73.988975, pickup_latitude=40.750348,
    dropoff_longitude=-73.96391, dropoff_longitude_missing=0.0,
    dropoff_latitude=40.799752, dropoff_latitude_missing=0.0,
    passenger_count=1, internal_sequence=1338560580,
    uuid='85bb164805a8658d5e548513143aaea4')
data_split = (
    parsed_timestamped_pcollection
    | "PartitionData" >> beam.ParDo(
        PartitionDoFn(fixed_threshold=split_point_object),
        threshold_from_side_input=split_threshold_side_input_view
        ).with_outputs('train', 'eval')
    )
```

```
def _generate_uuid(data_dict: dict, keys_for_hashing: list) -> str:
    hasher = hashlib.sha1()
    for k in sorted(keys_for_hashing):
        field_val_str = str(data_dict.get(k, ''))
        hasher.update(field_val_str.encode('utf-8'))
    return uuid.UUID(bytes=hasher.digest()[:16]).hex
```

```
example_uuid = _generate_uuid(converted_dict,
                              original_keys_in_obj)
return obj(**converted_dict), example_uuid
```

```
real    13m14.998s
user    242m20.128s
sys     1m29.149s
```

Data Exploration: The Shadow

```
display_statistics(oldest_statistics)
```

Sort by
Feature order ▾ ☐ Reverse order Feature search (regex enabled)

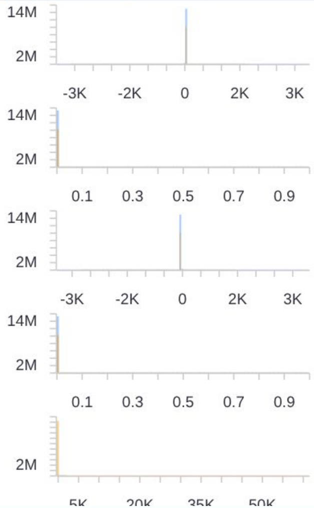
Features: ☑ int(2) ☑ float(7) ☑ string(4)

☑ lhs_statistics ☑ rhs_statistics

Numeric Features (9)

Chart to show: Standard ▾ ☐ log ☐ expand ☐ percentages

	count	missing	mean	std dev	zeros	min	median	max
dropoff_latitude								
lhs	15.3M	0%	39.95	10.97	1.96%	-3,488.08	40.75	3,400.39
rhs	10.2M	0%	39.88	6.88	1.8%	-3,114.34	40.75	1,651.55
dropoff_latitude_missing								
lhs	15.3M	0%	0	0	100%	0	0	1
rhs	10.2M	0%	0	0	100%	0	0	1
dropoff_longitude								
lhs	15.3M	0%	-72.51	14.08	1.97%	-3,414.13	-73.98	3,457.62
rhs	10.2M	0%	-72.5	10.7	1.8%	-2,771.29	-73.98	1,428.74
dropoff_longitude_missing								
lhs	15.3M	0%	0	0	100%	0	0	1
rhs	10.2M	0%	0	0	100%	0	0	1
fare_amount								
lhs	15.3M	0%	10.38	8.67	0%	-76	7.7	2,010.9
rhs	10.2M	0%	12.78	22.27	0.01%	-300	9.5	61.6k



Data Exploration: The Beam

The statistics are the shadow, and the talk is about data quality "in" ML pipelines. We have to go inside the pipeline.

This means going from tf.train.Examples to JSON or similar, and possibly sampling.

We can use the schema computed in the TFX pipeline to create a NamedTuple object dynamically, and plug in to Schema Aware PCollections. This is the heart of the Executor in JSONSampler

The json versions come from tf.train.Examples, and are fully tracked from an ML Metadata perspective

```
json_sampler = JSONSampler(examples=example_gen.outputs['examples'],
                           schema=schema_gen.outputs['schema'],
                           statistics=statistics_gen.outputs['statistics'],
                           sample_percent=0.25)

local_component_list = [
    example_gen,
    statistics_gen,
    schema_gen,
    example_validator,
    json_sampler,
]
```

```
from typing import NamedTuple
def NamedTupleGenerator(class_name, schema):
    name_type_tuples = [(f.name, f.type) for f in schema.feature]
    proto_dictionary = {}
    for feature_name, feature_type_int in name_type_tuples:
        if feature_type_int == 3:
            proto_dictionary[feature_name] = 1.0
        elif feature_type_int == 2:
            proto_dictionary[feature_name] = 1
        elif feature_type_int == 1:
            proto_dictionary[feature_name] = ""
        else:
            raise NotImplementedError
    TupleClass = NamedTuple(class_name, [(k, type(v)) for k, v in proto_dictionary.items()])
    return TupleClass
NTObject = NamedTupleGenerator("NTObject", schema)
beam.coders.registry.register_coder(NTObject, beam.coders.RowCoder)
```

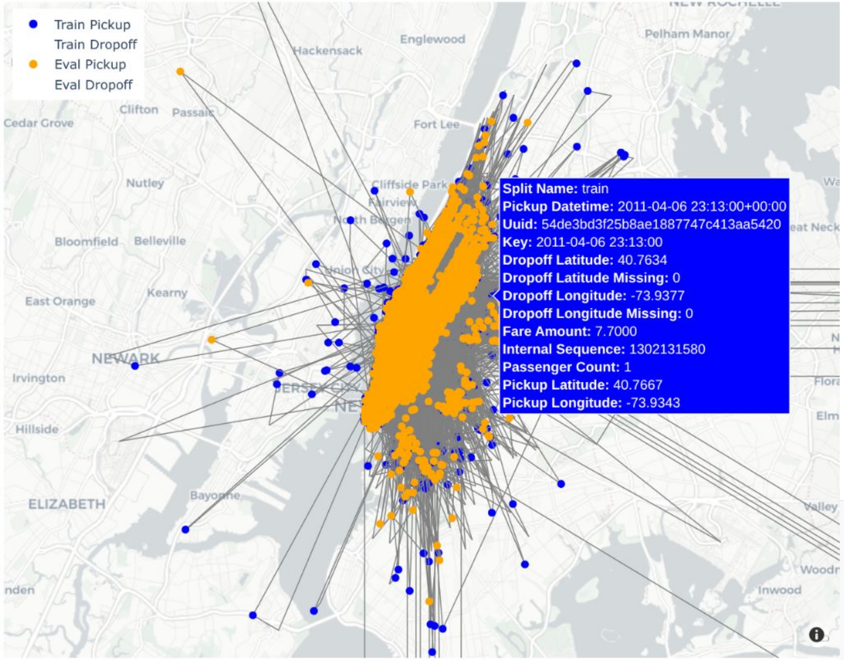
```
[utM] In [21]: [j.uri for j in store.get_artifacts_by_type('JSONSampler')]
Out[21]:
['/home/pdodeja/experiments/fareamount_original/pipeline/JSONSampler/json_sampler/5',
 '/home/pdodeja/experiments/fareamount_original/pipeline/JSONSampler/json_sampler/11',
 '/home/pdodeja/experiments/fareamount_original/pipeline/JSONSampler/json_sampler/18']
```

Data Exploration: The Beam and the Shadow

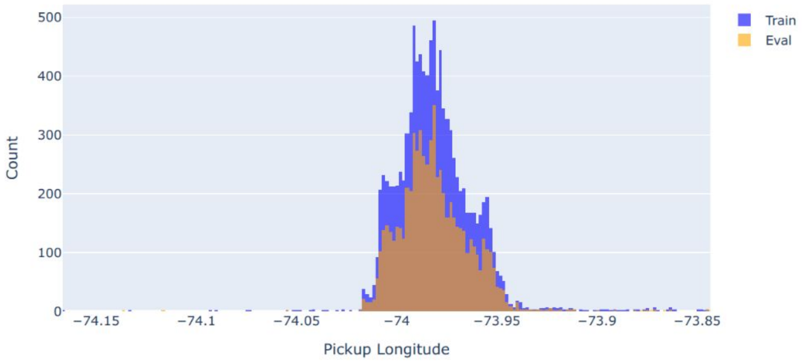
Histogram Feature: Pickup Longitude
Highlight Data Quality Issue: None
Histogram Bins (193): Auto
Y-Axis Scale: ☐ Log Scale
Selection History: ⬅ Previous | Next ➡

▶ **Custom Data Quality Filters (Notepad)**

Trip Locations



Distribution of Pickup Longitude



Capture Valid Ranges from Histogram

Statistics for Pickup Longitude

	Train Split		Eval Split
Count	10326.0000	**Count**	6374.0000

Data Exploration: Designing the Prism

We inject data quality indicators, _dq features, by interacting with statistics & sample data

▼ **Custom Data Quality Filters (Notepad)**

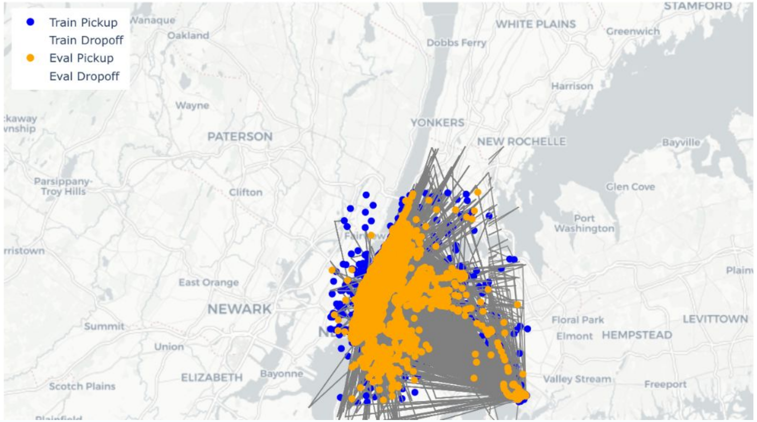
Get Python Equivalent

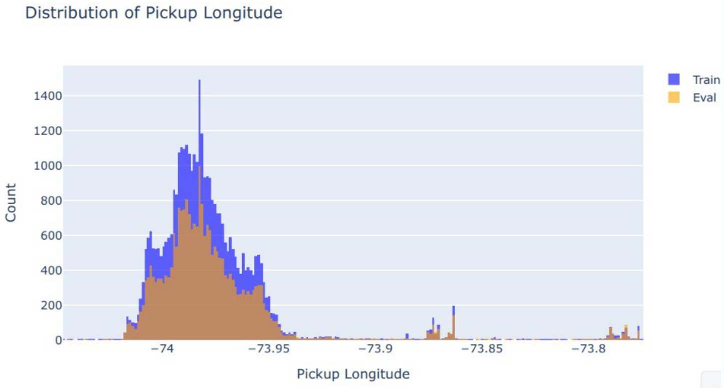
```
new_element['dropoff_latitude_dq'] = int(40.58639144897461 <= element['dropoff_latitude'] <= 40.9129753112793)
new_element['dropoff_longitude_dq'] = int(-74.0776596069336 <= element['dropoff_longitude'] <= -73.7736587524414)
new_element['fare_amount_dq'] = int(2.5 <= element['fare_amount'] <= 101)
new_element['pickup_latitude_dq'] = int(40.637603759765625 <= element['pickup_latitude'] <= 40.858280181884766)
new_element['pickup_longitude_dq'] = int(-74.0463638305664 <= element['pickup_longitude'] <= -73.77411651611328)
```

Copy

Dropoff Latitude	40.58639144897461	40.9129753112793	☑ Enable Filter
Dropoff Longitude	-74.0776596069336	-73.7736587524414	☑ Enable Filter
Fare Amount	2.5	101	☑ Enable Filter
Internal Sequence	Min	Max	☐ Enable Filter
Passenger Count	1	6	☐ Enable Filter
Pickup Latitude	40.637603759765625	40.858280181884766	☑ Enable Filter
Pickup Longitude	-74.0463638305664	-73.77411651611328	☑ Enable Filter

Trip Locations





Data PreTransform: Implementing the Prism

PreTransform takes in examples, a schema, and a module file. The module file contains our data quality contracts we got from the last step. It transforms the examples to native python equivalents using the schema, applies the provided function, and re-packs them back to TFRecords. We place StatisticsGen, SchemaGen, and JSONSampler to further verify data quality downstream. Because it can do arbitrary python functions, we could have created a polygon and evaluated data quality geographically (e.g. avoid slivers of the river that may not be possible using ranges)

```
pre_transform_module_file = os.path.join(REPO_PROJECT_ROOT, 'pre_transform_module.py')
pre_transform = PreTransform(
    examples=examples_cached_pre_transform.outputs['examples'],
    schema=import_schema_gen.outputs['schema'],
    module_file=pre_transform_module_file,
)
```

```
def transform_dict(element):
    new_element = {}
    for k in sorted(element.keys()):
        new_element[k] = element[k]
    new_element['dropoff_latitude_dq'] = int(
        40.57697296142578 <= element['dropoff_latitude'] <= 40.95429611206055)
    new_element['dropoff_longitude_dq'] = int(
        -74.04652404785156 <= element['dropoff_longitude'] <= -73.76942443847656)
    new_element['passenger_count_dq'] = int(
        0.7438825448613378 <= element['passenger_count'] <= 6)
    new_element['pickup_latitude_dq'] = int(
        40.61164093017578 <= element['pickup_latitude'] <= 40.90285110473633)
    new_element['pickup_longitude_dq'] = int(
        -74.0501480102539 <= element['pickup_longitude'] <= -73.77628326416016)
    return new_element
```

```
stats_options = tfdv.StatsOptions(label_feature=LABEL_KEY, num_histogram_buckets=NUM_HISTOGRAM_BUCKETS)
statistics_gen = StatisticsGen(examples=pre_transform.outputs['output_examples'], stats_options=stats_options)
schema_gen = SchemaGen(statistics=statistics_gen.outputs['statistics'], infer_feature_shape=True)
json_sampler = JSONSampler(examples=pre_transform.outputs['output_examples'], schema=schema_gen.outputs['schema'], statistics=statistics_gen.outputs['statistics'], sample_percent=25)   ■ Line too long (182
local_component_list = [
    examples_cached_pre_transform,
    import_schema_gen,
    pre_transform,
    statistics_gen,
    schema_gen,
    json_sampler,
]
```

Data PreTransform Scaling: Parallel Beams

Surprisingly, although there are no matrix or gradient operations, PreTransform processes allocate memory on the GPUs, and show a lot of parallelism (44 pids in nvidia-smi pmon). This likely has to do with the unpacking and packing of tf.train.Examples by the workers using beam. These workers execute work on both GPUs and use all CPU cores. Twenty five million examples processed in ~57 minutes, ~0.438 million examples/minute -> 7.3K data quality feature injections per second. It does as many in a few seconds as we are capable of visualizing. 20x wall clock performance on DirectRunner.

```
top - 04:48:37 up 10:56, 14 users,  load average: 25.05, 23.85, 15.98
Tasks: 1095 total,   1 running, 1094 sleeping,   0 stopped,   0 zombie
%Cpu(s): 98.2 us,  0.7 sy,  0.0 ni,  0.3 id,  0.0 wa,  0.8 hi,  0.0 si,  0.0 st
MiB Mem : 128611.8 total,  40625.1 free,  63230.3 used,  53502.1 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.  65381.4 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
 675519 pdodeja   20   0   20.5g   1.0g 556748 S  99.3   0.8  14:57.14 python
 675543 pdodeja   20   0   20.5g   1.0g 556596 S  99.3   0.8  14:57.18 python
 675553 pdodeja   20   0   20.5g   1.0g 556128 S  99.3   0.8  14:56.39 python
 675580 pdodeja   20   0   20.5g   1.0g 560728 S  99.3   0.8  14:55.25 python
 675511 pdodeja   20   0   20.5g   1.0g 554596 S  99.0   0.8  14:57.29 python
 675524 pdodeja   20   0   20.5g   1.0g 554316 S  99.0   0.8  14:56.46 python
 675536 pdodeja   20   0   20.5g   1.0g 556196 S  99.0   0.8  14:55.02 python
 675547 pdodeja   20   0   20.5g   1.0g 561828 S  99.0   0.8  14:55.23 python
 675557 pdodeja   20   0   20.5g   1.0g 560984 S  99.0   0.8  14:58.91 python
 675560 pdodeja   20   0   20.5g   1.0g 560620 S  99.0   0.8  14:56.07 python
 675562 pdodeja   20   0   20.5g   1.0g 560512 S  99.0   0.8  14:55.12 python
 675574 pdodeja   20   0   20.5g   1.0g 558852 S  99.0   0.8  14:56.32 python
 675522 pdodeja   20   0   20.5g   1.0g 560104 S  98.7   0.8  14:58.78 python
 675528 pdodeja   20   0   20.5g   1.0g 562192 S  98.7   0.8  14:54.60 python
 675566 pdodeja   20   0   20.5g   1.0g 557240 S  98.7   0.8  14:56.82 python
 675506 pdodeja   20   0   20.5g   1.0g 561092 S  98.3   0.8  14:55.95 python
 675509 pdodeja   20   0   20.5g   1.0g 557816 S  98.0   0.8  14:56.74 python
 675571 pdodeja   20   0   20.5g   1.0g 561460 S  98.0   0.8  14:59.85 python
 675545 pdodeja   20   0   20.5g   1.0g 557004 S  97.7   0.8  14:54.98 python
 675531 pdodeja   20   0   20.5g   1.0g 558444 S  97.4   0.8  14:58.34 python
 675540 pdodeja   20   0   20.5g   1.0g 561308 S  95.4   0.8  14:55.05 python
 675577 pdodeja   20   0   20.5g   1.0g 559480 S  95.0   0.8  14:55.16 python
 675533 pdodeja   20   0   20.5g   1.0g 557108 S  93.7   0.8  14:59.11 python
 675516 pdodeja   20   0   20.5g   1.0g 561992 S  93.4   0.8  14:55.12 python
 675260 pdodeja   20   0 4299804   1.0g 357524 S   9.9   0.8   1:44.09 python
```

```
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 570.144                Driver Version: 570.144        CUDA Version: 12.8     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 3060        Off |   00000000:01:00.0 Off |                  N/A |
| 36%   36C    P8              7W /  170W |    2718MiB /  12288MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
|   1  NVIDIA GeForce RTX 3090        Off |   00000000:05:00.0 Off |                  N/A |
|  0%   34C    P8              9W /  350W |    6300MiB /  24576MiB |      0%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI              PID   Type   Process name                        GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
|    0   N/A  N/A            1863      G   /usr/libexec/Xorg                         11MiB |
|    0   N/A  N/A            3579    C+G   ...c/gnome-remote-desktop-daemon         104MiB |
|    0   N/A  N/A          675506      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675509      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675511      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675516      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675519      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675522      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675524      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675528      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675531      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675533      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675536      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675540      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675543      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675545      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675547      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675553      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675557      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675560      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675562      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675566      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675571      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675574      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675577      C   ...nv/versions/tfx115/bin/python         102MiB |
|    0   N/A  N/A          675580      C   ...nv/versions/tfx115/bin/python         102MiB |
|    1   N/A  N/A          675506      C   ...nv/versions/tfx115/bin/python         256MiB |
|    1   N/A  N/A          675509      C   ...nv/versions/tfx115/bin/python         256MiB |
```

```
real    56m33.018s
user    1170m32.700s
sys     5m11.067s
```

Data Filtration: Beams Filtered through Prism

FilterUDF takes in examples, a schema, statistics (currently unused), and a module file. We construct our _dq indicators from the schema, which results in only data that passes all of our data quality checks to be passed through to the next stage.

```
filter_gen = FilterUDF(
    examples=examples_cached_pre_transform.outputs['examples'],
    schema=import_schema_gen.outputs['schema'],
    statistics=statistics_cached.outputs['result'],
    module_file=filter_module_file,
)
```

```
def filterfn(schema_dict, example):
    features = tf.io.parse_single_example(example, schema_dict.feature_spec)
    filter_features = [f for f in schema_dict.feature_spec.keys() if '_dq' in f]
    good_data = True
    for feature in filter_features:
        good_data = good_data and (features[feature].numpy()[0] > 0)
    return good_data
```

```
local_component_list = [
    examples_cached_pre_transform,
    statistics_cached,
    filter_gen,
    import_schema_gen,
    statistics_gen_filter,
    schema_gen_filter,
    json_sampler,
]
```

```
real    35m34.102s
user    689m10.120s
sys     4m29.607s
```

Exploring Filtered Data

Histogram Feature: Fare Amount Dq

Highlight Data Quality Issue: None

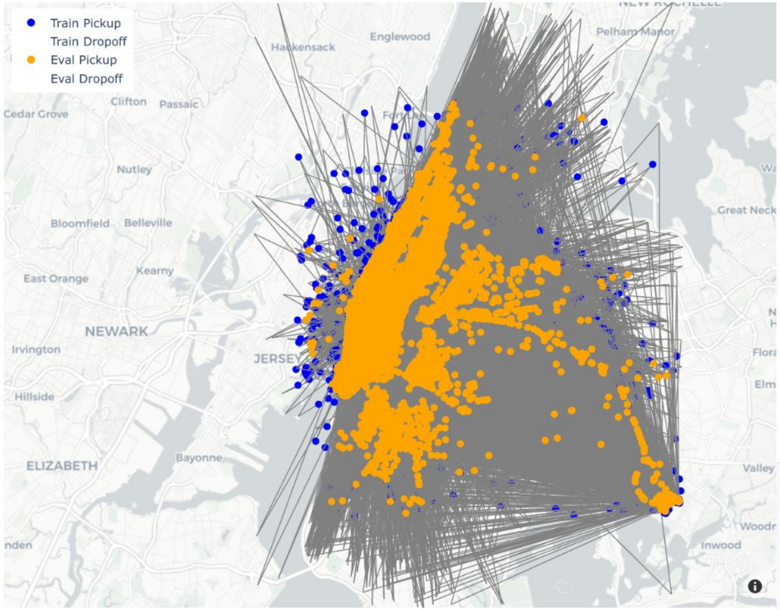
Histogram Bins (20): Auto

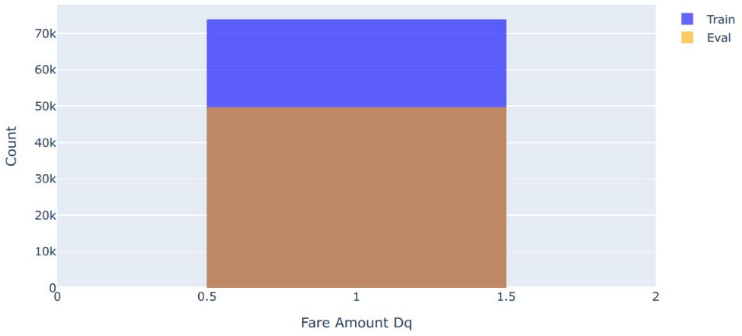
Y-Axis Scale: Log Scale

Selection History: Previous | Next

▶ **Custom Data Quality Filters (Notepad)**

Trip Locations





Capture Valid Ranges from Histogram

Statistics for Fare Amount Dq

	Train Split		Eval Split
Count	74022.0000	**Count**	49766.0000
Min	1	**Min**	1
Max	1	**Max**	1
Mean	1.0000	**Mean**	1.0000

Downstream Pipeline Components

```
real    58m22.517s
user    174m0.728s
sys     8m54.994s
```

Now that our data is filtered, we can enrich (feature engineer) it, with tf.Transform.

We can place our JSONSampler to receive JSON versions of TFTransform'ed Examples.

As features are sometimes higher dimensional, we would need to enhance our conversion process (currently only python primitives).

Since we have UUIDs and split information, we can have fine grained tracing of the impact of each example on the training process (e.g. Analyze BulkInferrer protocol buffers, re-inject back into visualization after sampling).

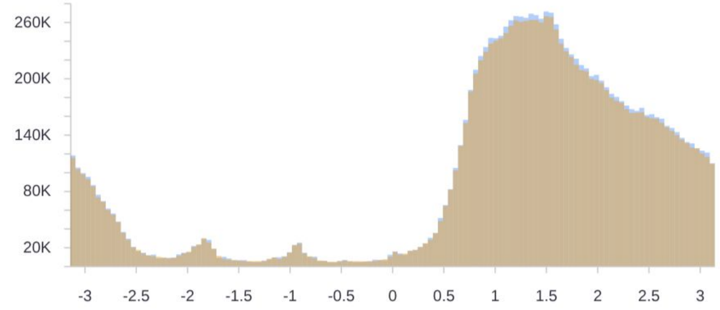
Models trained on the filtered data have lower validation loss (both train and val are filtered) than those on unfiltered data (no filtering on either split)

ML Metadata is what allows us to open up the pipeline, and re-seal it back together. An end-to-end hermetically sealed pipeline is not optimized for data quality, it is optimized for scaling and reproducibility.

Transformed Statistics

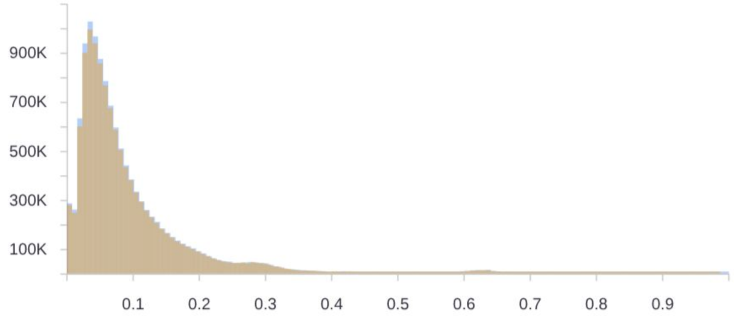
level_3_atan2

	12.1M	0%	1.21	1.5	0%	-3.14	1.49	3.14
	11.8M	0%	1.21	1.5	0%	-3.14	1.49	3.14



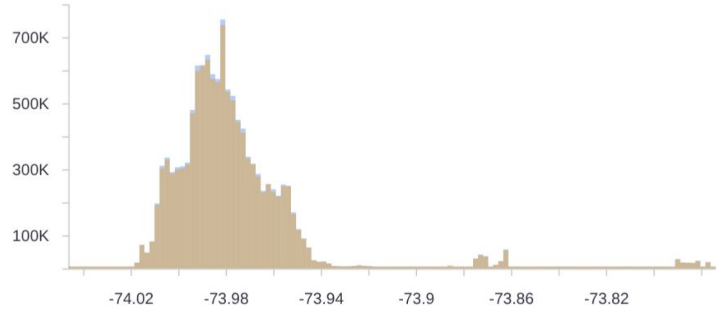
level_2_haverstein_distance_scaled

	12.1M	0%	0.1	0.1	1.22%	0	0.07	1
	11.8M	0%	0.1	0.1	1.22%	0	0.07	0.99



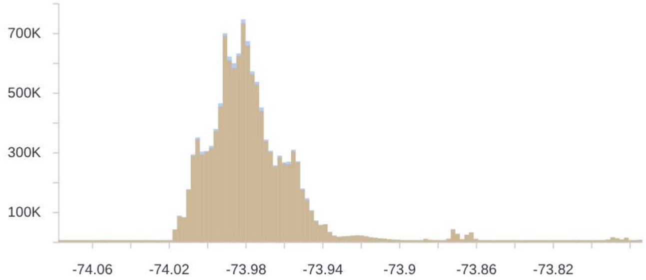
level_0_pickup_longitude

	12.1M	0%	-73.98	0.03	0%	-74.05	-73.98	-73.77
	11.8M	0%	-73.98	0.03	0%	-74.05	-73.98	-73.77



level_0_dropoff_longitude

	12.1M	0%	-73.97	0.03	0%	-74.08	-73.98	-73.77
	11.8M	0%	-73.97	0.03	0%	-74.08	-73.98	-73.77



How can we make it better?

Step	How Beam Helps	How we can improve?
Pre-Ingestion	Determine Schema for Data	Utility code for schema generation, integrate uuids better Use more scalable runner than DirectRunner (overall)
Data Splitting & Identification	Deterministic Data Splitting at Scale	Manage split point in metadata, splits in data Generate UUID post-split Component for temporal data (not just managing split points, also sequences/time series)
Data Exploration	Produce JSON from TFRecord	Multi-dimensional support (temporal, geo) Figure out scaling visualization (e.g. infinite zoom using “James Webb” feature.)
Non-graph intended feature injection	Apply Arbitrary Python UDFs	Support polygon generation from visualization, auto generate UDF
Filter	Remove examples which do not meet DQ requirements	Store filtered data as a managed artifact, filter individual examples
Post Transform	Evaluator, BulkInference	Integrate output into ML loop

Demo (Time/Conditions Permitting)

Capabilities:

How can we assign passports to our data? - Windowing/UUID

How can we attach luggage tags? - Inject Data Quality Indicators

How can we filter bad data at scale? - FilterUDFs

How can we manage the boarding process? - Data Quality Post Transform (Future)

How can we manage, and possibly avoid, turbulence? - BulkInference/Evaluator Integration (Future)

Call to action

- If you know Beam, you are more than halfway there for large scale machine learning.
- Follow https://github.com/tensorflow/tfx to keep up with the improvements in large scale ML.
- Try to understand TFX component architecture for a Beam based component. Three parts: component, with children component spec, and executor. I can provide book recommendations.
- Help me get these components into TFX!

Pritam Dodeja, ML Engineer
@ Intuitive.cloud

QUESTIONS?

LinkedIn: https://www.linkedin.com/in/pritam-dodeja/

Github: https://github.com/pritamdodeja