Simplifying ML Workflows with Apache Beam & TensorFlow Extended

Tyler Akidau
@takidau

Software Engineer at Google
Apache Beam PMC

Apache Beam

Portable data-processing pipelines

Example pipelines

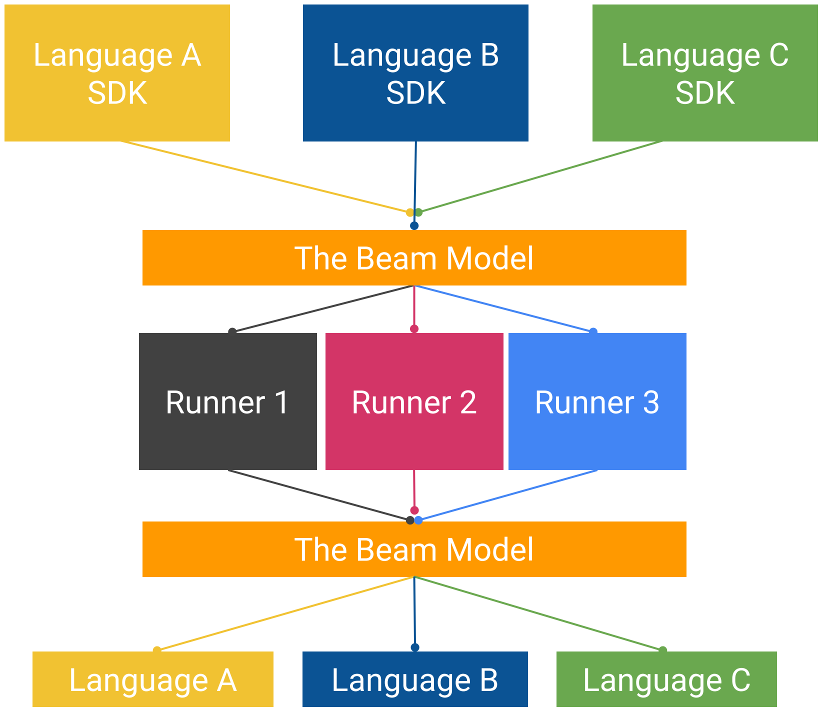
Python

```
input | WindowInto(FixedWindows(3600),
                   trigger=AfterWatermark())
      | Sum.PerKey()
      | Write(BigQuerySink(...))
```

Java

```
input
  .apply(Window.into(FixedWindows.of(...))
    .triggering(
    AfterWatermark.pastEndOfWindow()))
  .apply(Sum.integersPerKey())
  .apply(BigQueryIO.Write.to(...))
```

Cross-language Portability Framework



Python compatible runners

Direct runner (local machine):	**Now**
Google Cloud Dataflow:	**Now**
Apache Flink:	**Q2-Q3**
Apache Spark:	**Q3-Q4**

TensorFlow Extended

End-to-end machine learning in production

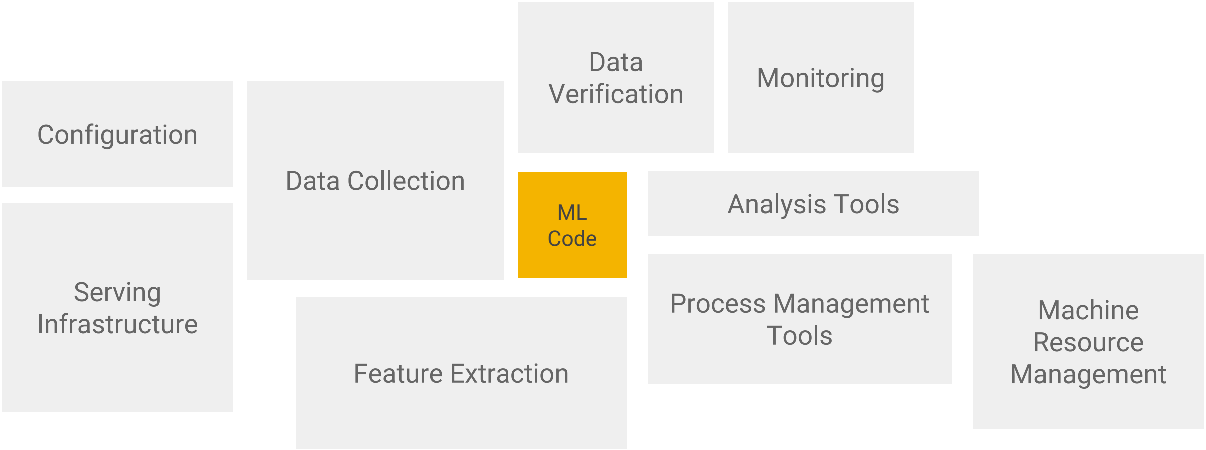
“Doing ML in production is hard.”

-Everyone who has ever tried

Because, in addition to the actual ML...

ML
Code

...you have to worry about so much more.



Source: Sculley et al.: Hidden Technical Debt in Machine Learning Systems

In this talk, I will...

In this talk, I will...

Show you how to apply transformations...

TensorFlow Transform

In this talk, we will...

Show you how to apply transformations...
... consistently between Training and Serving

TensorFlow Transform

TensorFlow Estimators

TensorFlow Serving

In this talk, we will...

Introduce something new...

TensorFlow Transform

TensorFlow Estimators

TensorFlow Model Analysis

TensorFlow Serving

TensorFlow Transform

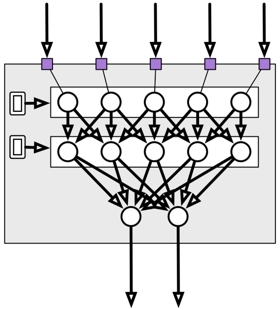
Consistent In-Graph Transformations in Training and Serving

Typical ML Pipeline

During training

data

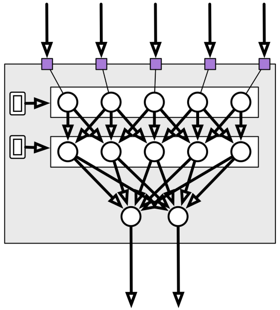
batch processing



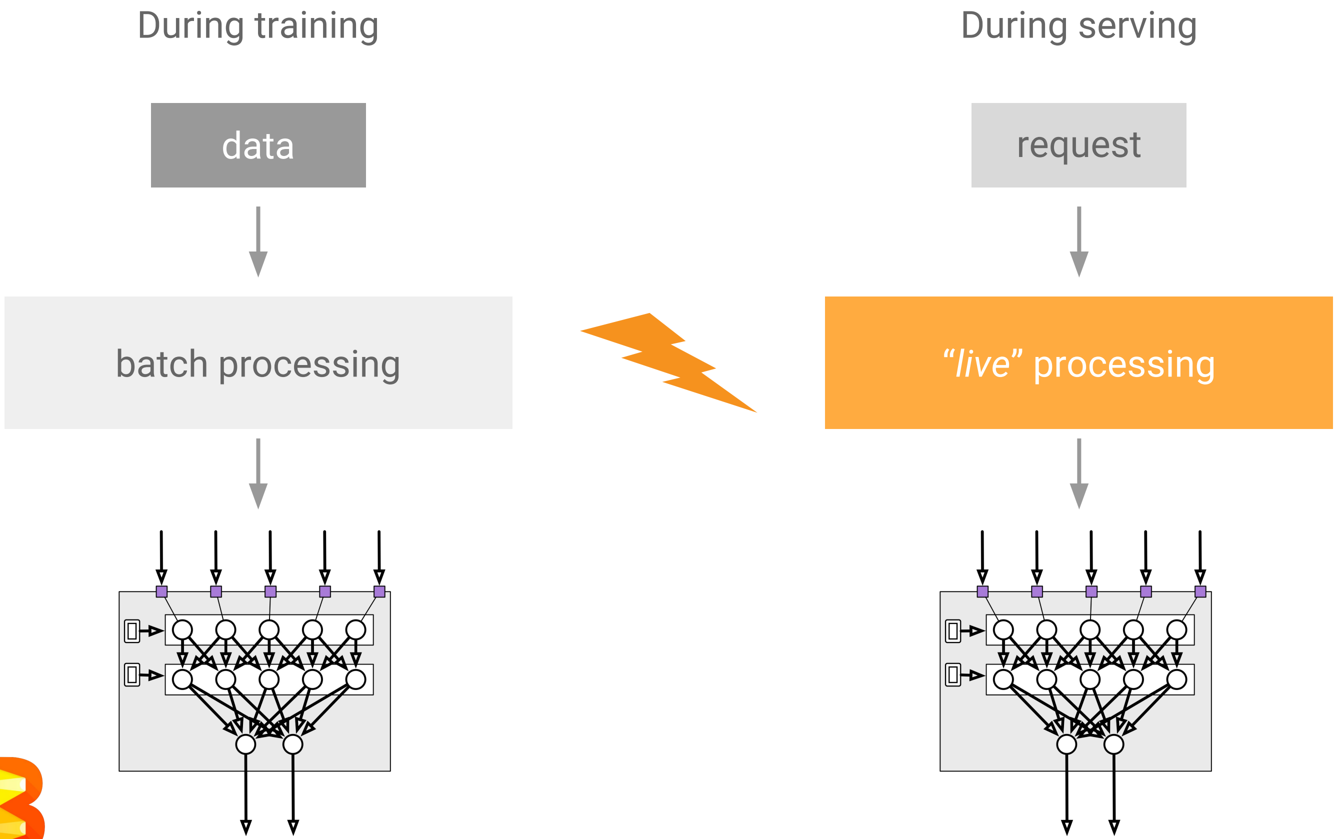
During serving

request

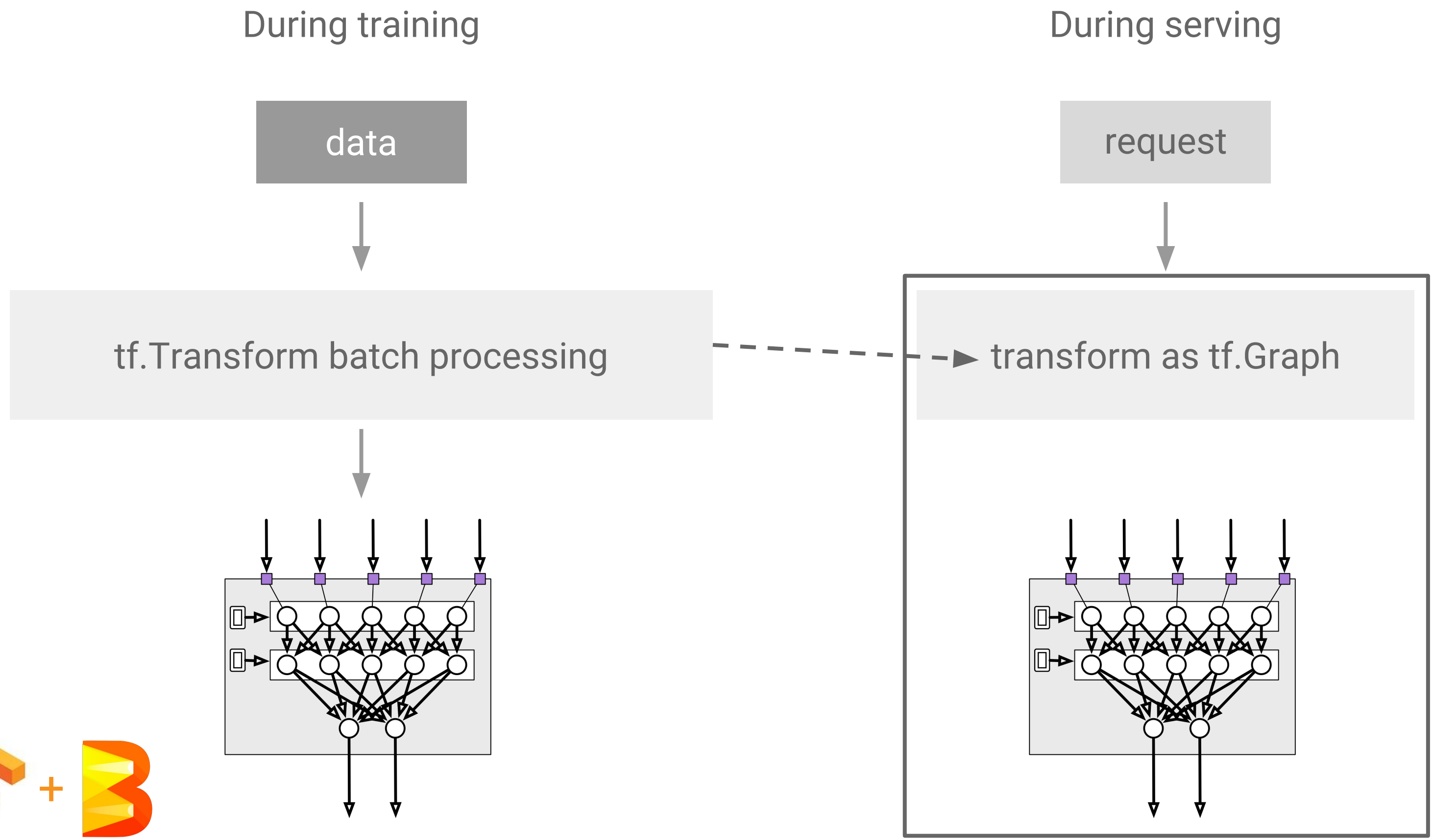
"live" processing



Typical ML Pipeline



TensorFlow Transform



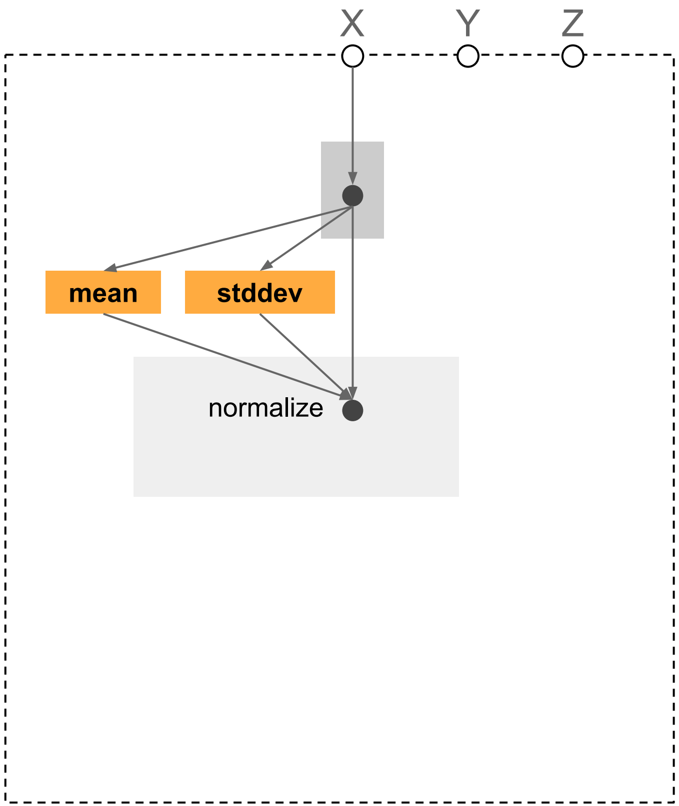
Defining a preprocessing function in TF Transform

```
def preprocessing_fn(inputs):
    x = inputs['X']
    ...
    return {
        "A": tft.bucketize(
                tft.normalize(x) * y),
        "B": tensorflow_fn(y, z),
        "C": tft.ngrams(z)
    }
```

Many operations available for dealing with text and numeric features, user can define their own.

Defining a preprocessing function in TF Transform

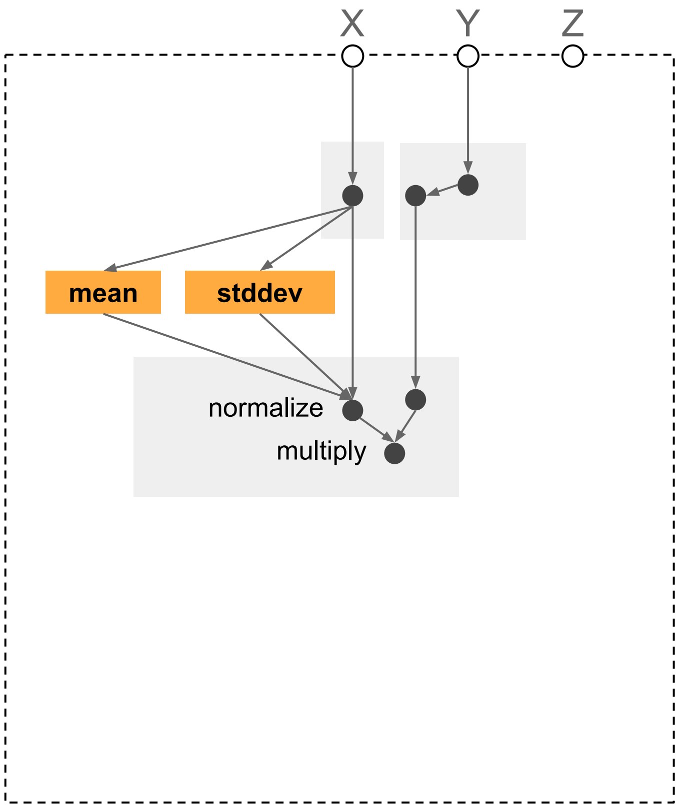
```
def preprocessing_fn(inputs):
    x = inputs['X']
    ...
    return {
        "A": tft.bucketize(
                 tft.normalize(x) * y),
        "B": tensorflow_fn(y, z),
        "C": tft.ngrams(z)
    }
```



Many operations available for dealing with text and numeric features, user can define their own.

Defining a preprocessing function in TF Transform

```
def preprocessing_fn(inputs):
    x = inputs['X']
    ...
    return {
        "A": tft.bucketize(
                tft.normalize(x) * y),
        "B": tensorflow_fn(y, z),
        "C": tft.ngrams(z)
    }
```

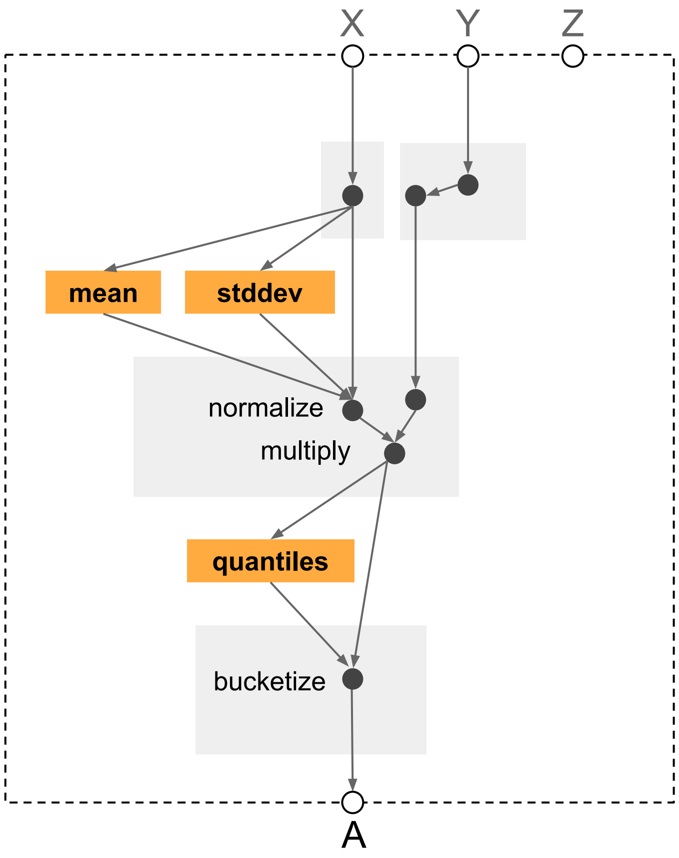


Many operations available for dealing with text and numeric features, user can define their own.

Defining a preprocessing function in TF Transform

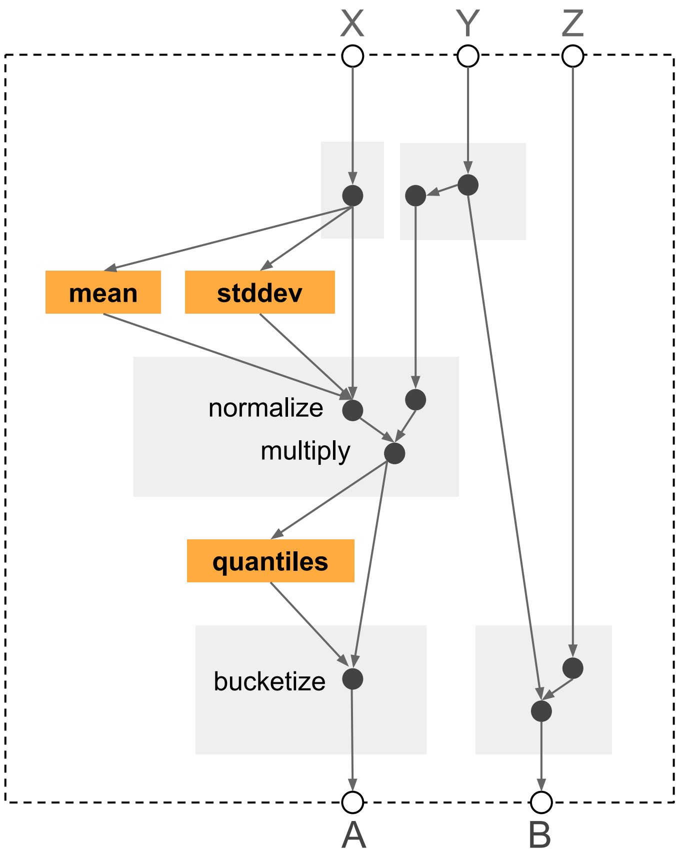
```
def preprocessing_fn(inputs):
    x = inputs['X']
    ...
    return {
        "A": tft.bucketize(
                 tft.normalize(x) * y),
        "B": tensorflow_fn(y, z),
        "C": tft.ngrams(z)
    }
```

Many operations available for dealing with text and numeric features, user can define their own.



Defining a preprocessing function in TF Transform

```
def preprocessing_fn(inputs):
    x = inputs['X']
    ...
    return {
        "A": tft.bucketize(
                tft.normalize(x) * y),
        "B": tensorflow_fn(y, z),
        "C": tft.ngrams(z)
    }
```

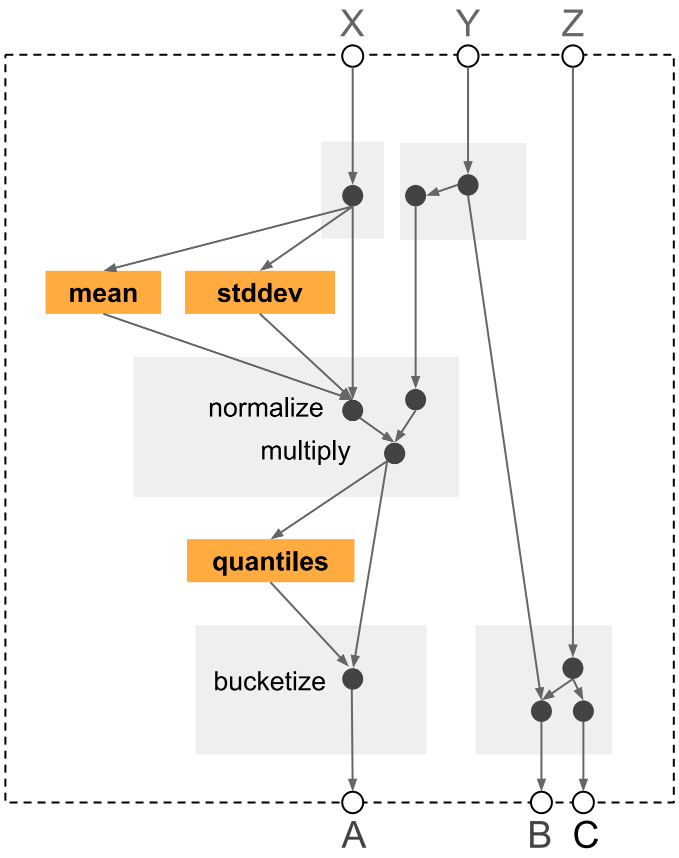


Many operations available for dealing with text and numeric features, user can define their own.

Defining a preprocessing function in TF Transform

```
def preprocessing_fn(inputs):
    x = inputs['X']
    ...
    return {
        "A": tft.bucketize(
                tft.normalize(x) * y),
        "B": tensorflow_fn(y, z),
        "C": tft.ngrams(z)
    }
```

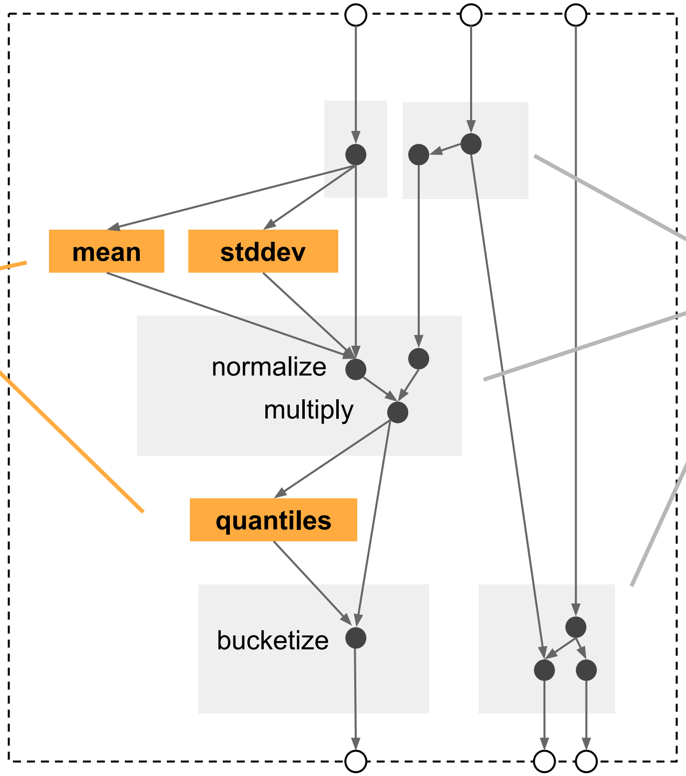
Many operations available for dealing with text and numeric features, user can define their own.



Analyzers

Reduce (full pass)

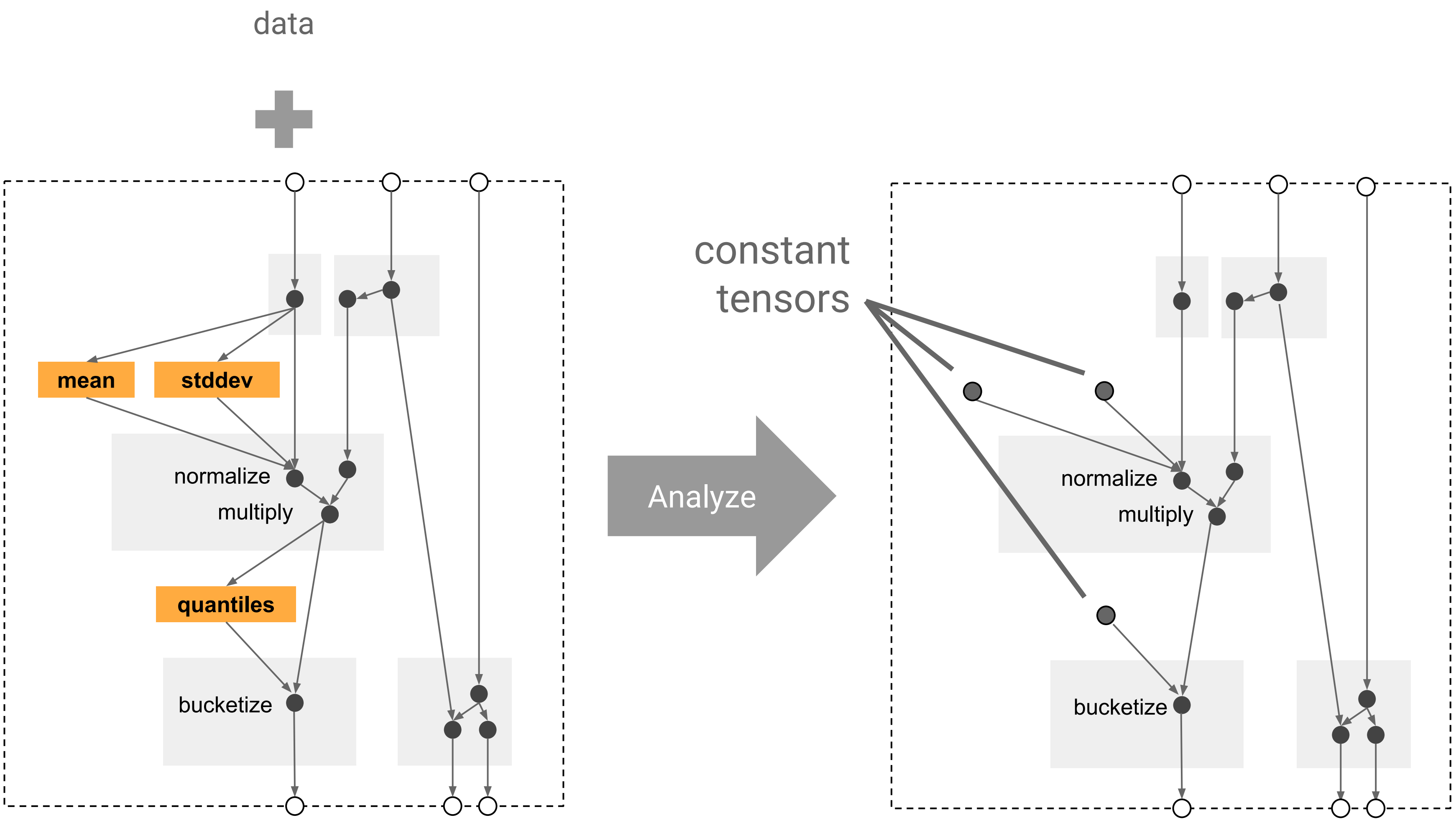
Implemented as a distributed data pipeline



Transforms

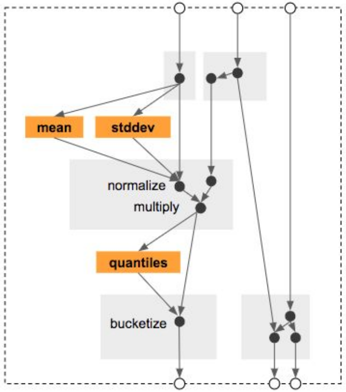
Instance-to-instance (don’t change batch dimension)

Pure TensorFlow


data
mean
stddev
normalize
multiply
quantiles
bucketize
Analyze
constant
tensors
normalize
multiply
bucketize

What can be done with TF Transform?

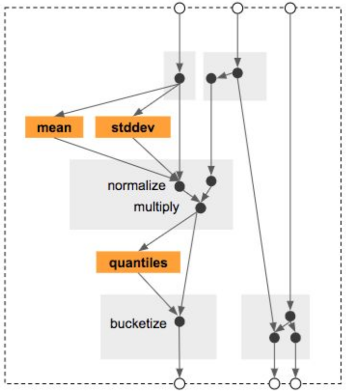
tf.Transform batch processing



Pretty much anything.

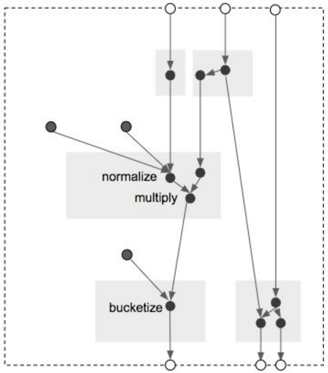
What can be done with TF Transform?

tf.Transform batch processing



Pretty much anything.

Serving Graph



Anything that can be expressed as a TensorFlow Graph

Some common use-cases...

Scale to ...

`tft.scale_to_z_score`

`...`

Bag of Words / N-Grams

`tf.string_split`

`tft.ngrams`

`tft.string_to_int`

Bucketization

`tft.quantiles`

`tft.apply_buckets`

Feature Crosses

`tf.string_join`

`tft.string_to_int`

Some common use-cases...

Scale to ...

`tft.scale_to_z_score`

`...`

Bag of Words / N-Grams

`tf.string_split`

`tft.ngrams`

`tft.string_to_int`

Bucketization

`tft.quantiles`

`tft.apply_buckets`

Feature Crosses

`tf.string_join`

`tft.string_to_int`

Apply another TensorFlow Model

`tft.apply_saved_model`

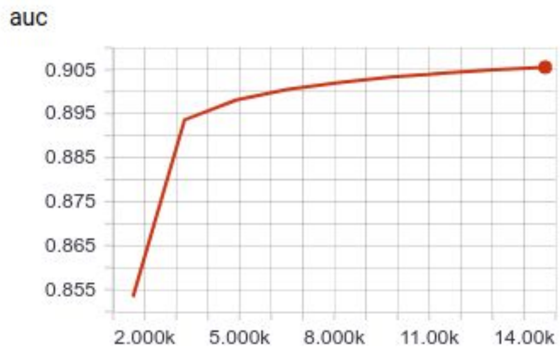
github.com/tensorflow/transform

Introducing...

TensorFlow Model Analysis

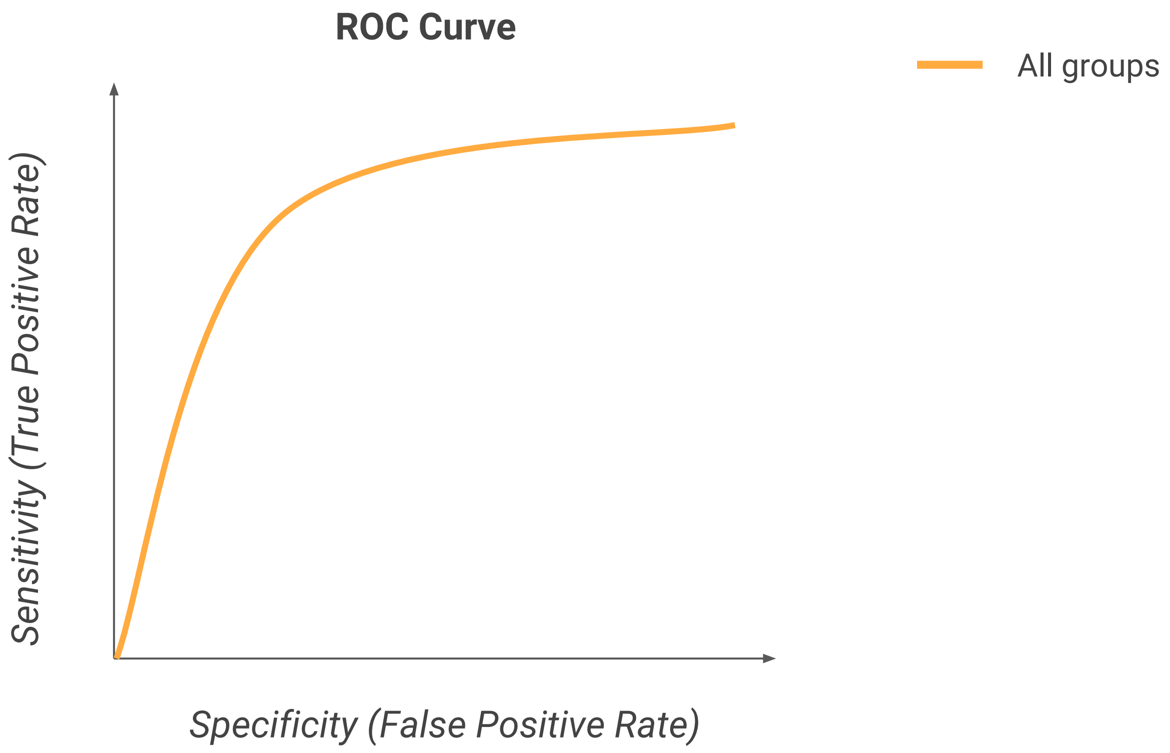
Scaleable, sliced, and full-pass metrics

Let’s Talk about Metrics...

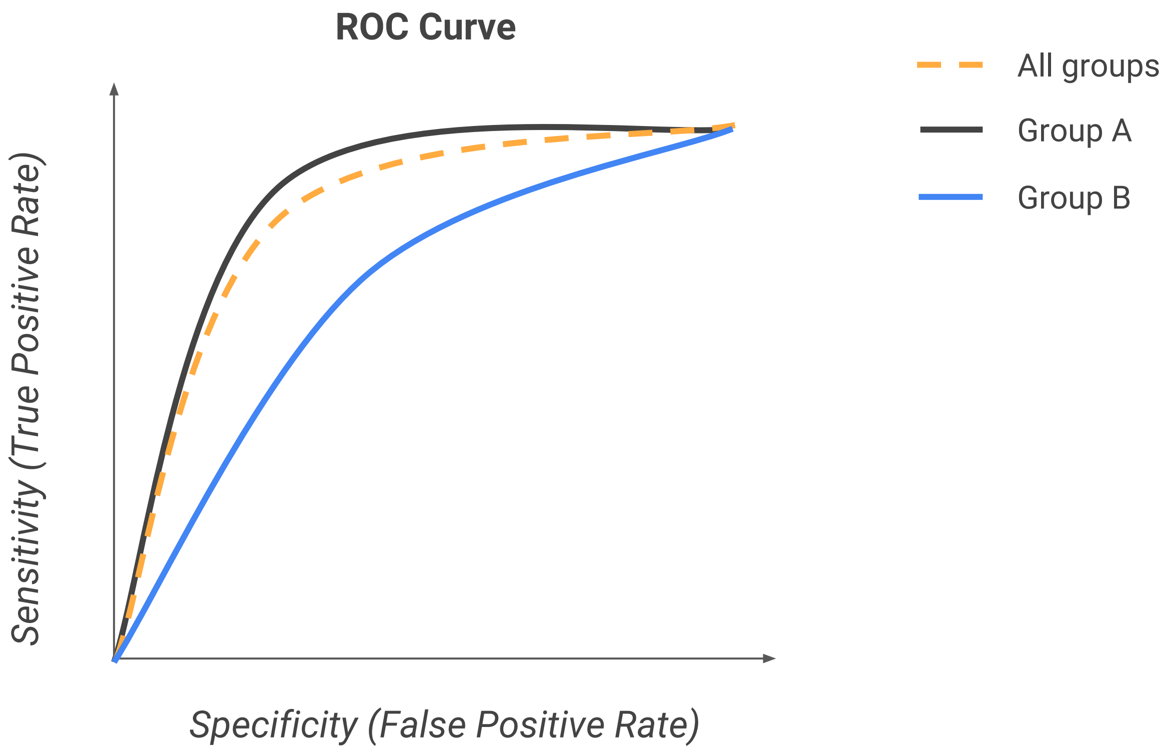


- How accurate?
- Converged model?
- What about my TB sized eval set?
- Slices / subsets?
- Across model versions?

ML Fairness: analyzing model mistakes by subgroup

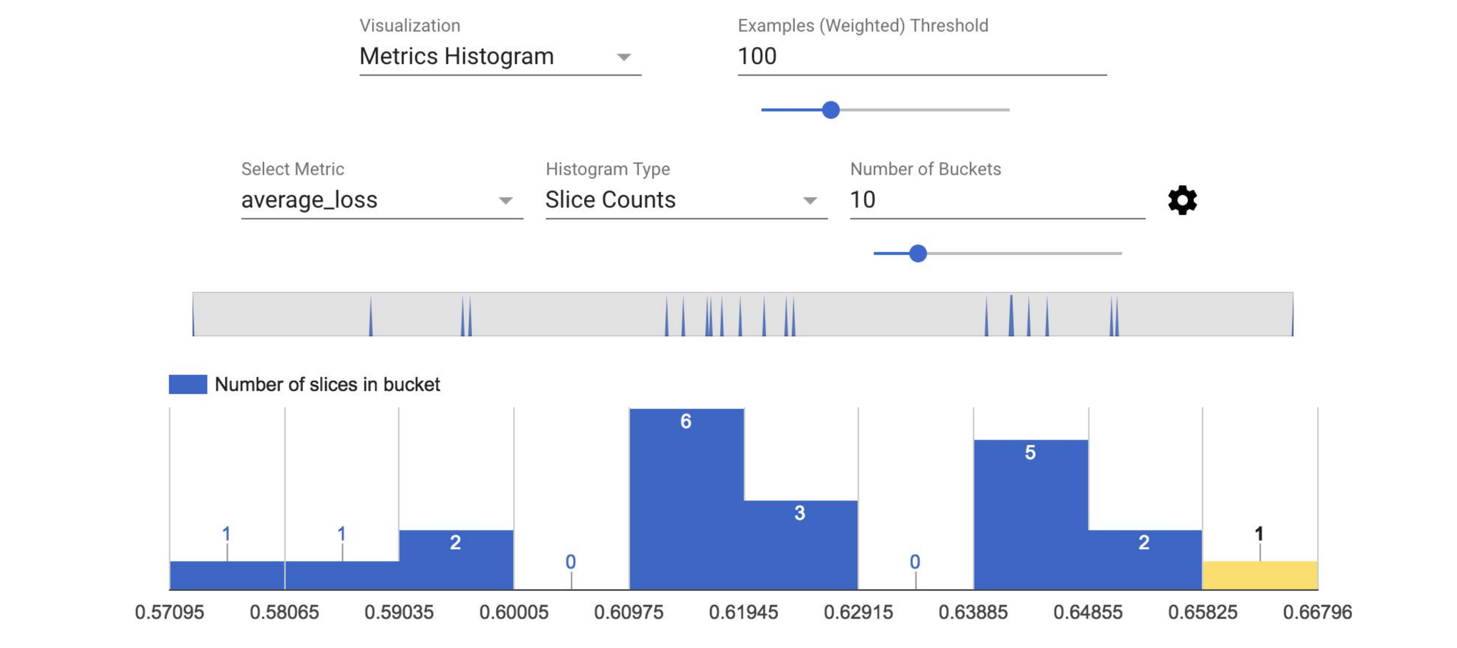


ML Fairness: analyzing model mistakes by subgroup



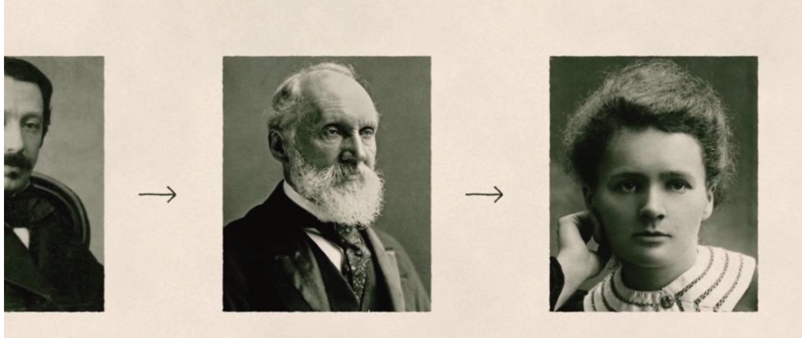
Learn more at ml-fairness.com

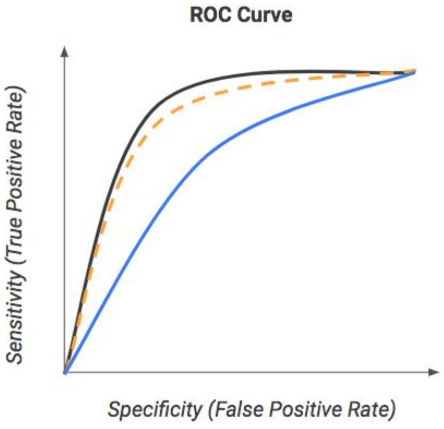
ML Fairness: understand the failure modes of your models



feature	accuracy	accuracy_baseline	auc	auc_precision_recall	average_loss
trip_start_hour:7	0.60976	0.63415	0.57236	0.44873	0.667
trip_start_hour:2	0.66102	0.63559	0.58527	0.47002	0.652
trip_start_hour:18	0.64241	0.60443	0.64457	0.52275	0.651

ML Fairness: Learn More





ml-fairness.com

Measuring and Mitigating Unintended Bias in Text Classification

Lucas Dixon ldixon@google.com
John Li jetpack@google.com
Jeffrey Sorensen sorenj@google.com
Nithum Thain nthain@google.com
Lucy
Jigsaw

Abstract

We introduce and illustrate a new approach to measuring and mitigating unintended bias in machine learning models. Our definition of unintended bias is parameterized by a test set and a subset of input features. We illustrate how this can be used to evaluate text classifiers using a synthetic test set and a public corpus of comments annotated for toxicity from Wikipedia Talk pages. We also demonstrate how imbalances in training data can lead to unintended bias in the resulting models, and therefore potentially unfair applications. We use a set of common demographic identity terms as the subset of input features on which we measure bias. This technique permits analysis in the common scenario where demographic information on authors and readers is unavailable, so that bias mitigation must focus on the content of the text itself. The mitigation method we introduce is an unsupervised approach based on balancing the training dataset. We demonstrate that this approach reduces the unintended bias without compromising overall model quality.

Introduction

With the recent proliferation of the use of machine learning for a wide variety of tasks, researchers have identified unfairness in ML models as one of the growing concerns in the field. Many ML models are built from human-generated data, and human biases can easily result in a skewed distribution in the training data. ML practitioners must be proactive in recognizing and counteracting these biases, otherwise our models and products risk perpetuating unfairness by performing better for some users than for others.

Recent research in fairness in machine learning proposes

Mitigating Unwanted Biases with Adversarial Learning

Brian Hu Zhang, Stanford University, Stanford, CA, bhz@stanford.edu
Blake Lemoine, Google, Mountain View, CA, lemoine@google.com
Margaret Mitchell, Google, Mountain View, CA, mmitchellai@google.com

93v1 [cs.LG] 22 Jan 2018

Abstract

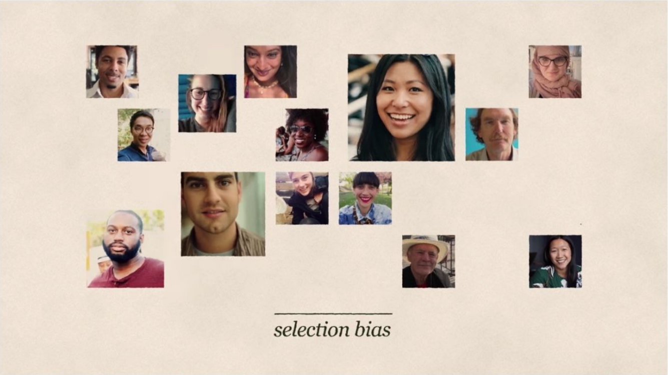
Machine learning is a tool for building models that accurately represent input training data. When undesired biases concerning demographic groups are in the training data, well-trained models will reflect those biases. We present a framework for mitigating such biases by including a variable for the group of interest and simultaneously learning a predictor and an adversary. The input to the network X, here text or census data, produces a prediction Y, such as an analogy completion or income bracket, while the adversary tries to model a protected variable Z, here gender or zip code.

The objective is to maximize the predictors ability to predict Y while minimizing the adversary's ability to predict Z. Applied to analogy completion, this method results in accurate predictions that exhibit less evidence of stereotyping Z. When applied to a classification task using the UCI Adult (Census) Dataset, it results in a predictive model that does not lose much accuracy while achieving very close to equality of odds (Hardt, et al., 2016). The method is flexible and applicable to multiple definitions of fairness as well as a wide range of gradient-based learning models, including both regression

incorporated into a loss function in order to mitigate disproportional outcomes in the system's output predictions regarding a protected demographic, such as sex.

In this paper, we examine these fairness measures in the context of *adversarial debiasing*. We consider supervised deep learning tasks in which the task is to predict an output variable Y given an input variable X, while remaining unbiased with respect to some variable Z. We refer to Z as the *protected variable*. For these learning systems, the predictor $\hat{Y} = f(X)$ can be constructed as (input, output, protected) tuples (X, Y, Z). The predictor $f(X)$ is usually given access to the protected variable Z, though this is not strictly necessary. This construction allows the determination of which types of bias are considered undesirable for a particular application to be chosen through the specification of the protected variable.

We speak to the concept of *mitigating bias* using the known term *debiasing*[1], following definitions provided by Hardt et al. (2016) and refined by Beutel et al. (2017).



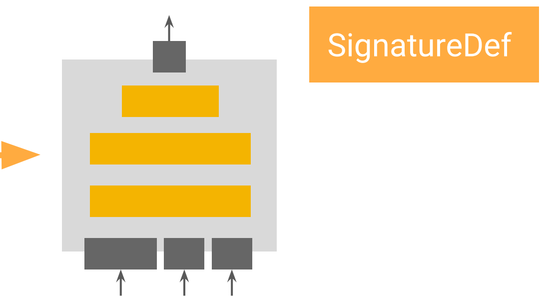
How does it work?

```
...
estimator = DNNLinearCombinedClassifier(...)
estimator.train(...)

estimator.export_savedmodel(
  serving_input_receiver_fn=serving_input_fn)

tfma.export.export_eval_savedmodel (
  estimator=estimator,
  eval_input_receiver_fn=eval_input_fn)
...
```

Inference Graph (SavedModel)



How does it work?

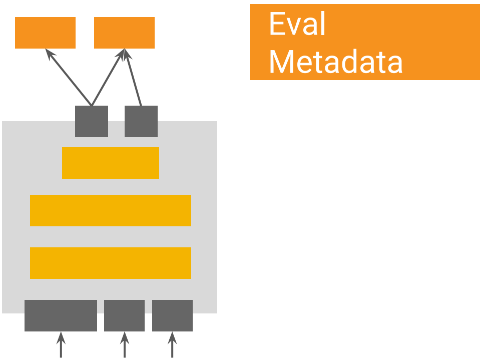
```
...
estimator = DNNLinearCombinedClassifier(...)
estimator.train(...)

estimator.export_savedmodel(
  serving_input_receiver_fn=serving_input_fn)

tfma.export.export_eval_savedmodel (
  estimator=estimator,
  eval_input_receiver_fn=eval_input_fn)
...
```

Inference Graph (SavedModel)

Eval Graph (SavedModel)



github.com/tensorflow/model-analysis

Summary

Apache Beam: Data-processing framework the runs locally and scales to massive data, in the Cloud (now) and soon on-premise via Flink (Q2-Q3) and Spark (Q3-Q4). Powers large-scale data processing in the TF libraries below.

tf.Transform: Consistent in-graph transformations in training and serving.

tf.ModelAnalysis: Scalable, sliced, and full-pass metrics.