ĐẠI HỌC QUỐC GIA TPHCM

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

Big data: Tensorflow Extended Research

Môn học: Nhập môn Dữ liệu lớn

Giảng viên hướng dẫn:

- Phạm Trọng Nghĩa
- Nguyễn Bảo Long
- Lê Nhựt Nam

Sinh viên thực hiện:

- 20127588. Nguyễn Tấn Phát
- 20127383. Lê Ngọc Tường
- 20127484. Nguyễn Tư Duy
- 20127374. Nguyễn Đức Trường

Ngày 12 tháng 4 năm 2023

Contents

1 Introduction

Deploying production-ready Machine Learning (ML) pipelines is not quite a straightforward operation, in contrast to traditional software engineering. Data ingestion, validation, pre-processing, model training, and post-training tasks are just a few examples of the many jobs that can be included in a ML pipeline. For the most part, data scientists or ML engineers eventually develop a lot of boilerplate code and glue code to execute these jobs. Since these glue codes are frequently fragile, pipeline problems result.

ML pipelines can get incredibly complex over time and require a lot of overhead to maintain job relationships. There are drawbacks to writing one's own boiler and glue codes. To adapt to any modifications in the input data format, many manual interventions are required. Platform changes may necessitate a complete restructuring of glue codes because they are platform dependent. The capabilities of distributed systems may not be fully utilized by specialized glue routines when working on massive datasets for computer vision or NLP projects.

TensorFlow Extended (TFX) is a Google-production-scale ML platform based on TensorFlow. TFX assists in reducing boilerplate code required for each task and in streamlining pipeline definitions. For orchestration technologies like Apache Airflow, Apache Beam, and Kubeflow Pipelines, TFX provides libraries to carry out various pipeline activities as well as glue code to connect the various components.

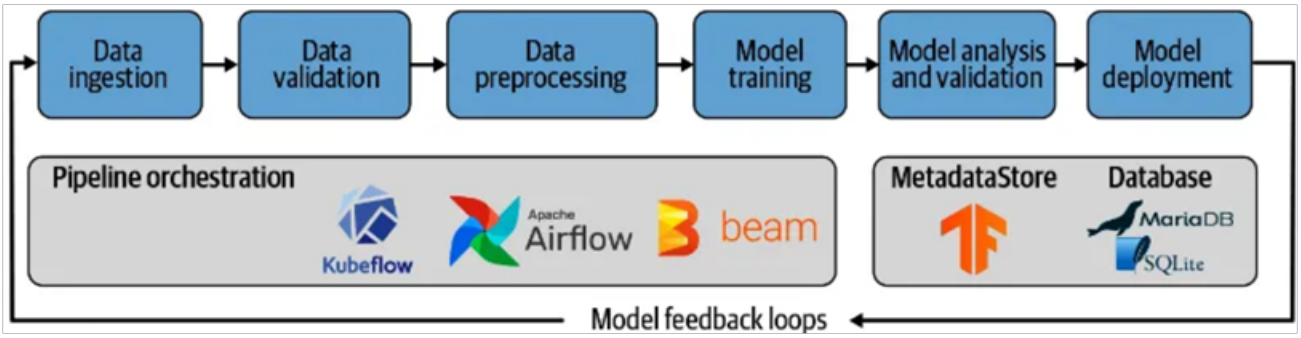

Figure 1: ML pipeline architecture

Over the years, Google has built TFX with assistance from a sizable developer and research community. Since its initial release in 2017, the platform has developed to include more elements and functionality. To enable the deployment of ML models at scale in production contexts and to offer a complete set of tools for organizing the end-to-end ML lifecycle have been two of the main driving forces behind the development of TFX.

Organizations from several sectors, including technology, banking, and the healthcare industry, have adopted TFX significantly in recent years. The platform is well-suited for a number of use cases, from developing recommendation systems to detecting fraud, thanks to its flexibility and scalability.

Figure 2: TFX logo

2 TFX Pipelines

Machine learning (ML) includes many steps which make organizing these quite challenging. Applying MLOps can help to overcome this and the ML workflow can be implemented as a TFX Pipeline so that:

- Your ML process can be automated then you can deploy, test and retrain your model on a regular basis.
- Make use of distributed computational resources while processing massive datasets and workloads.
- Running a pipeline with several sets of hyperparameters can accelerate experimentation.
- Check for anomalies in training data and eliminate training-serving skew.
- Create ML pipelines that involve deep model performance analysis and validation of freshly trained models to ensure reliability and performance.

2.1 Understanding TFX Pipelines

An ML workflow can be implemented portably using a TFX pipeline, which can be used with a variety of orchestrators - a system where you can execute pipeline runs, such as Apache Airflow, Apache Beam, and Kubeflow Pipelines. Component instances and input parameters make up a pipeline.

As outputs, component instances create artifacts, and as inputs, they often rely on artifacts created by upstream component instances. By building a directed acyclic graph of the artifact dependencies, the execution order for component instances is determined.

Suppose a pipeline that performs the following:

- Ingests data directly from a proprietary system using a custom component.
- Computes statistics for the training data using the **StatisticsGen** standard component.
- Generates a data schema using the **SchemaGen** standard component.
- Inspect the training data for abnormalities using the **ExampleValidator** standard component.

- Performs feature engineering on the dataset using the **Transform** standard component.
- Trains a model using the **Trainer** standard component.
- Evaluates the trained model using the **Evaluator** component.
- If the model succeeds in being evaluated, the pipeline uses a custom component to queue the trained model for deployment to a private deployment system.

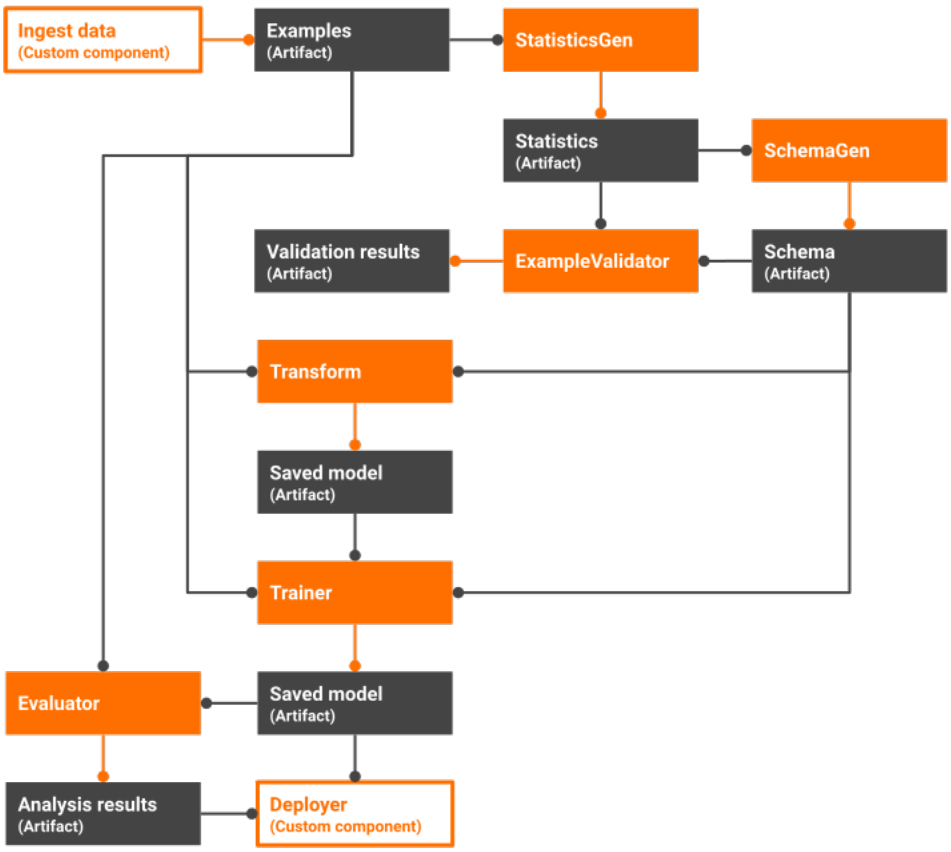

Figure 3: TFX Pipeline graph

Based on the artifact dependencies analysis, an orchestrator executes:

- The data ingestion, **StatisticsGen**, **SchemaGen** component instances in order.
- As the **ExampleValidator** and **Transform** components share input artifact dependencies and are independent of one another's output, they can operate in parallel.
- The **Trainer**, **Evaluator**, and custom deployer component instances execute one after the other when the **Transform** component is finished.

2.2 Building a TFX Pipeline

TFX pipelines are defined using the Pipeline class, following this example:

```
pipeline.Pipeline(
    pipeline_name=pipeline-name,
    pipeline_root=pipeline-root,
    components=components,
    enable_cache=enable-cache,
    metadata_connection_config=metadata-connection-config,
)
```

Listing 1: A simple definition of a TFX pipeline

List of arguments:

- *pipeline_name*: The name of this pipeline. Unexpected behaviour may occur if a pipeline name is reused, because of that it must be unique.
- *pipeline_root*: The pipeline's outputs' root path. The whole path to a directory that your orchestrator has read and write access to must be the root path. The pipeline root is used by TFX during runtime to create output pathways for component artifacts. This directory may be local or located on a distributed file system that is supported, such as HDFS or Google Cloud Storage.
- components: A list of component instances that form this pipeline's workflow.
- *enable_cache* (optional): a boolean value indicating whether or not caching is being used by this pipeline to accelerate pipeline execution.
- *metadata_connection_config* (optional): An ML Metadata connection configuration.

To make pipeline deployment easier, instead of building a custom pipeline, TFX provides a prebuilt set of pipeline definitions and can be customized for specific needs:

1. Using TFX CLIs display a list of available templates and select a template to create a copy

```
tfx template list
```

Listing 2: Display a list of templates

```
tfx template copy --model=template --pipeline_name=pipeline-name \
--destination_path=destination-path
```

Listing 3: Select a template

   - *template*: the name of the template.
   - $pipeline_name$: the name of the pipeline.
   - $destination_path$: the path where the template is copied into.

   Some directories and files will be copied to the pipeline's root directory, which contain pipeline's definition, configuration details, etc.

2. Create a pipeline using LocalDagRunner

```
tfx pipeline create --pipeline_path local_runner.py
```

```
tfx run create --pipeline_name pipeline_name
```

Listing 4: Create a new run instance for the pipeline

Some directories will be added to the pipeline, which contain ML metada and outputs of the pipeline.

3. Next is customize the pipeline to fit ones' requirements, like:
   - Changing pipeline parameters..
   - Adding or removing components.
   - Replacing the data input source.
   - Changing a component's customization function or its configuration in the pipeline.

3 Major components and main functionalities

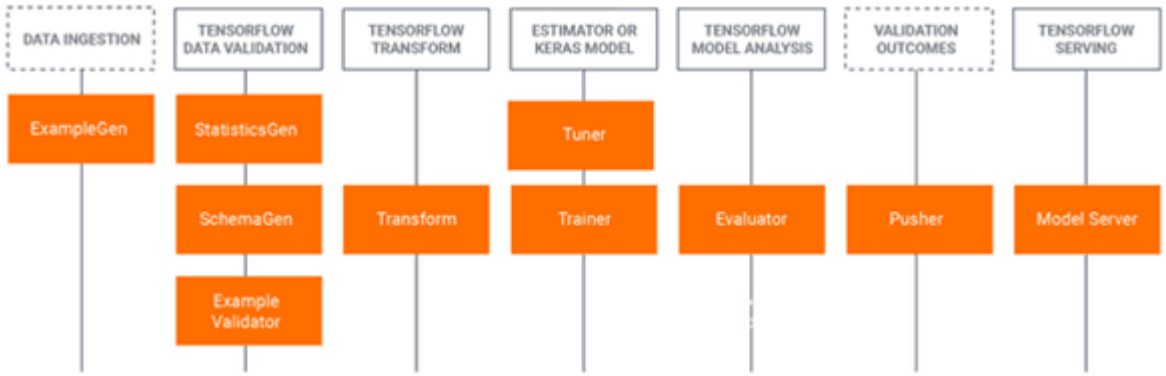

Figure 4: TFX Components

TFX is designed to support large-scale machine learning application deployment. It is capable of performing many tasks on a very large scale of data, such as data preprocessing, model training, and model serving. However, with the scale of data increasing, deploying machine learning applications becomes more complex and time-consuming. To address this challenge, TFX has developed several important components to simplify the process. Below is a detailed description for each component.

3.1 ExampleGen

ExampleGen [13] is a key component of data processing and data preparation for the next steps in the model deployment and training process.

ExampleGen provides APIs to read and process data from a variety of sources, including CSV, JSON, TFRecord, and BigQuery files. It can process data from multiple sources and convert them into a standard format (such as one or more TFRecord files) for inclusion in subsequent TFX components, such as **StatisticsGen**, **SchemaGen**, and **Trainer**.

Basically, **ExampleGen** takes data samples from the data source, does the initial analysis (like

counting the number of samples, unique values in columns, etc.) for downstream TFX components. **ExampleGen** can be configured to split the data into training and test sets, or split the data the other ways.

List of arguments (CsvExampleGen [10] in this case):

- *input_base*: A remote directory where the CSV files are kept.
- *input_config*: An *example_gen_pb2.Input* instance, providing input configuration. The files under *input_base* will be handled as one split if unset.
- *output_config*: An *example_gen_pb2.Output* instance, providing output configuration. If left unset, "train" and "eval" splits with a size of 2:1 will be used.
- *range_config*: An optional *range_config_pb2.RangeConfig* instance, defining the span values to take into account's range. Driver will by default look for the most recent span with no limitations if it is unset.

Component outputs includes:x

- *examples*: Channel of type *standard_artifacts.Examples* for output "train" and "eval" examples.

```
from tfx.components import CsvExampleGen
from tfx.proto import example_gen_pb2
from tfx.orchestration.experimental.interactive.interactive_context import
    InteractiveContext

# Create an InterativeContext to run TFX components interactively
context = InteractiveContext()

# Define splits configuration
output = example_gen_pb2.Output(
    split_config = example_gen_pb2.SplitConfig(splits = [
        example_gen_pb2.SplitConfig.Split(name = 'train', hash_buckets = 7), #
    train = 70%
        example_gen_pb2.SplitConfig.Split(name = 'eval', hash_buckets = 3) #
    eval = 30%
    ])
)

example_gen = CsvExampleGen(input_base = '/content/drive/MyDrive/BD',
    output_config = output)

context.run(example_gen)
```

Listing 5: A simple definition of ExampleGen

3.2 StatisticsGen

StatisticsGen [7] is a component that utilize the TensorFlow Data Validation (TFDV) library. It computes statistics from the dataset, which can be used for visualization and validation. The input that **StatisticsGen** uses is the dataset ingested by **ExampleGen**.

The **StatisticsGen** component can be used to displays different feature types, the number of values in each, the proportion of missing data, the number of unique features, and much more. Checking the distribution of each feature to make sure whether covariate balance is achieved. It is also very important to compare between train split and evaluation split to make sure the model can generalize to new data. **StatisticsGen** uses Apache Beam and approximate algorithms to scale to large datasets.

List of arguments [11]:

- *examples*: A BaseChannel of ExamplesPath type, produced by the **ExampleGen** component. This must contain "train" and "eval" splits.
- *schema* (optional): A Schema channel that will be used to automatically set the statistic options supplied to TFDV.
- *stats_options* (optional): Optional TFDV behavior can be set by using the StatsOptions instance. *stats_options.schema* will be used in place of the schema channel input if it is set.
- *exclude_splits* (optional): Names of splits where statistics and samples shouldn't be produced. When *exclude_splits* is set to None, there are no splits excluded by default.

Component outputs includes:

- *statistics*: Channel of type *standard_artifacts.ExampleStatistics* for each split's statistics, which are shown in the input examples.

```
from tfx.components import StatisticsGen

# Definition using ExampleGen's output
statistics_gen = StatisticsGen(
    examples = example_gen.outputs['examples'])

context.run(statistics_gen)
```

Listing 6: A simple definition of StatisticGen

3.3 SchemaGen

The SchemaGen[15] is component that is used to automatically generate a schema for input data. The schema describes the structure and data types of the input data, as well as any constraints or expectations on the data and this is used to ensure that data is consistent and valid throughout the machine learning pipeline.

Consumes and Emits:

SchemaGen component consumes statistics computed over the input data by the StatisticsGen component. Specifically, SchemaGen requires the output statistics artifact from StatisticsGen as input, which contains summary statistics of the input data such as min, max, mean, and variance of each feature.

SchemaGen then generates a data schema proto based on the computed statistics, which defines the expected data types, shapes, and constraints of each feature in the input data. The schema

is emitted as a standard output artifact, that is pbtxt file which similar to json file. The schema consumed by downstream components such as the ExampleValidator and Transform components.

```
...
feature {
  name: "age"
  value_count {
    min: 1
    max: 1
  }
  type: FLOAT
  presence {
    min_fraction: 1
    min_count: 1
  }
}
feature {
  name: "capital-gain"
  value_count {
    min: 1
    max: 1
  }
  type: FLOAT
  presence {
    min_fraction: 1
    min_count: 1
  }
}
...
```

Listing 7: Example of a Schema File

In addition, the schema is also used by the following libraries:

- TensorFlow Data Validation: the schema is used for validate the data, check the completeness of the data and provide information about the data.
- TensorFlow Transform: TFT uses schema for preprocessing.
- TensorFlow Model Analysis: TFMA uses schema for model evaluation across different slices of data.

Using the SchemaGen Component

For the initial schema generation: A SchemaGen pipeline component is typically very easy to deploy and requires little customization. Typical code looks like this:

```
from tft.components import SchemaGen

schema_gen = tfx.components.SchemaGen(
    statistics=stats_gen.outputs['statistics'])

context.run(schema_gen)
# to view schema file:
```

```
context.show(schema_gen.outputs['schema'])
```

Listing 8: Code sample to use SchemaGen

3.4 ExampleValidator

The ExampleValidator[14] Component is an important component in the data checking and validation process. It identifies anomalies in training and serving data and it can detect different classes of anomalies in the data so determine whether they meet specific data standards defined in the schema or not.

For example it can:

- Perform validity checks by comparing data statistics against a schema that
- Codifies expectations of the user.
- Detect training-serving skew by comparing training and serving data.
- Detect data drift by looking at a series of data. perform custom validations using a SQL-based configuration.

Consumes and Emits:

ExampleValidator uses TensorFlow Data Validation (TFDV) to read the schema from the SchemaGen component and data statistics computed by the StatisticsGen, then it identifies any anomalies in the example data by comparing and finally generates a report of errors, omissions, and contrasts in the input data. The errors may include missing data, duplicate data, outlier values, or other errors related to data integrity and data format. The inferred schema codifies properties which the input data is expected to satisfy, and can be modified by the developer.

Using the ExampleValidator Component

An ExampleValidator pipeline component is typically very easy to deploy and requires little customization. Typical code looks like this:

```
from tft.components import ExampleValidator

validate_stats = ExampleValidator(
      statistics=statistics_gen.outputs['statistics'],
      schema=schema_gen.outputs['schema']
      )

context.run(validate_stats)
```

Listing 9: Code sample to use ExampleValidator

3.5 Transform

The Transform component [16] is responsible for preprocessing raw data into a format that is suitable for training machine learning models. It takes raw data as input and applies a series of transformations to convert the data into a feature representation that can be used by machine learning models. It supports common preprocessing tasks such as feature scaling, feature normalization,

one-hot encoding, and feature selection. And it also provides functionality for handling missing values and creating new features from existing features.

The Transform component uses Apache Beam, a distributed data processing framework, to scale to large datasets and can be run on a variety of platforms, including on-premises clusters, cloud platforms such as Google Cloud, and standalone machines.

Configuring a Transform Component
Configuring a Transform Component in a TFX pipeline involves defining a set of parameters that control the behavior of the component. These parameters specify details such as the input and output data locations, the data schema, the list of transformations to apply, and the data statistics to compute.

Here are some of the key parameters that can be configured in the Transform component:

1. *input_data*: This parameter specifies the location of the input data that needs to be transformed. It could be a file or a directory containing multiple files.

2. *output_data*: This parameter specifies the location where the transformed data should be written to. The transformed data can be stored in a different format, such as Apache Parquet, to improve efficiency and compatibility.

3. *schema*: The schema parameter defines the data type and structure of the input data, such as column names, data types, and default values. The schema is used to ensure that the input data is compatible with the expected data format.

4. *module_file*: The module_file parameter specifies the location of the Python module containing the transformation logic, such as a TensorFlow graph or a scikit-learn pipeline.

5. *transformed_metadata_path*: This parameter specifies the location of the metadata file that describes the transformed data. The metadata file includes statistics such as mean and standard deviation that are used to normalize the input data.

6. *analyzer_cache_file_path*: This parameter specifies the location of the cache file that stores the intermediate results of the transformation process, such as computed statistics and vocabulary.

7. *transformed_feature_name*: This parameter specifies the name of the feature column that contains the transformed data.

Here is an example configuration for the Transform component:

```
from tfx.components import Transform
...
# Define the transform component
transform = Transform(
    input_data=input_data,
    output_data=output_data,
    schema=schema,
    module_file='/path/to/transform_module.py',
    transformed_metadata_path='/path/to/transformed_metadata',
    analyzer_cache_file_path='/path/to/analyzer_cache',
```

```
    transformed_feature_name='transformed_features',
    parameters=transform_pb2.Transform.ExampleSplitTransformParameters(
        feature_splits=['features', 'labels'])
)

# Add the transform component to the TFX pipeline
pipeline = Pipeline(
    pipeline_name='my_pipeline',
    pipeline_root='/path/to/pipeline/root',
    components=[transform]
)
```

Listing 10: Example configuration for the Transform component

3.6 Trainer

The Trainer Component [8] is an important component of the TFX Pipeline, used to define and train machine learning models. It is used to build models from data that have been preprocessed and transformed by other TFX components such as ExampleGen and Transform. The trainer can use any machine learning framework, such as TensorFlow or scikit-learn, to train the model.

Component

- module_file: A Python file containing the code for the model and related functions. This file defines the model_fn function, which takes input training data and returns an EstimatorSpec describing the defined model.
- examples: Input data prepared for training. This can be the output of the Transform component or manually processed data.
- transform_graph: A representation of the TensorFlow Graph transformation created by the Transform component. The Trainer component uses this representation to apply input transformations before training the model.
- schema: Describes the structure of the input data, including names, types, and constraints.
- transformed_examples: Input data that has been transformed by the input transformations from the Transform component. The Trainer component uses this data to train the model.
- train_args: Parameters to configure the training process, including the number of epochs, batch size, etc.
- eval_args: Parameters to configure the evaluation process, including the number of batches in each evaluation, etc.
- model_exports: Exported files of the trained model.

```
from tfx.components import Trainer
trainer = Trainer(
    module_file=module_file,
    examples=transform.outputs['transformed_examples'],
```

```
    transform_graph=transform.outputs['transform_graph'],
    train_args=trainer_pb2.TrainArgs(num_steps=10000),
    eval_args=trainer_pb2.EvalArgs(num_steps=5000))
...
```

Listing 11: Typical pipeline DSL code

After training, Trainer will output files storing the trained model, including checkpoint files and SavedModel.

```
from tfx.components.trainer.fn_args_utils import FnArgs

def run_fn(fn_args: FnArgs) -> None:
  """Build the TF model and train it."""
  model = _build_keras_model()
  model.fit(...)
  # Save model to fn_args.serving_model_dir.
  model.save(fn_args.serving_model_dir, ...)
```

Listing 12: Fit and save model example

3.7 Evaluator

Evaluator [12] has two functions. Analysis for performs deep analysis on the training results (use TensorFlow Model Analysis library, which in turn use Apache Beam for scalable processing.) And validation for Validate exported models ensuring model can be pushed to production. Evaluate trained model with pre-defined metrics and slices and can visualize with library. Validation functions can be enabled or disabled. If validation is enable, evaluator compares new models against a baseline by evaluating both models on an eval dataset and computing their performance on metrics. If the new model's metrics meet developer-specified criteria, this model is marked as good for *Pusher*.

Information for setup:

- Metrics and slices.
- Model to compare against and thresholds (if validation is enable).
- Validation will be performed against all of the metrics and slices

Consumes and Emits

- Consumes: eval split (*ExampleGen*), trained model (*Trainer*), blessed model (if validation is enable).
- Emits: analysis results (*ML Metadata*), validation results (*ML Metadata*, if validation is enable).

Here example for add information to *Evaluator* for analysis and validation.

```
eval_config = tfma.EvalConfig(
    model_specs=[tfma.ModelSpec(...)],
    metrics_specs=[
        tfma.MetricsSpec(metrics=[
            tfma.MetricConfig(...),
```

```
            tfma.MetricConfig(...),
            ...
            ])
    ],
    slicing_specs=[
        tfma.SlicingSpec()
        tfma.SlicingSpec(...)
        ])
```

Listing 13: Add information for Evaluator

Need to use Resolver to specify the latest blessed model will be used as the baseline.

```
model_resolver = Resolver(
      strategy_class=dsl.experimental.LatestBlessedModelStrategy,
      model=Channel(type=Model),
      model_blessing=Channel(type=ModelBlessing)
).with_id('latest_blessed_model_resolver')
```

Listing 14: Define Resolver

And final, implement define Evaluator with above configure.

```
model_analyzer = Evaluator(
      examples=examples_gen.outputs['examples'],
      model=trainer.outputs['model'],
      baseline_model=model_resolver.outputs['model'],
      eval_config=eval_config)
```

Listing 15: Define Evaluator

The following is an example of how to load the results of *Evaluator* and visualize it.

```
import tensorflow_model_analysis as tfma

output_path = evaluator.outputs['evaluation'].get()[0].uri
eval_result = tfma.load_eval_result(output_path)
tfma.view.render_slicing_metrics(eval_result)
validation_result = tfma.load_validation_result(output_path)
```

Listing 16: Load the results of Evaluator

3.8 Pusher

During model training or re-training, *Pusher* [6] relies on blessings from *Evaluator* and *InfraValidator* (option) to decide whether to push the model to deployment target.

Consumes and Emits

- Consumes: trained model (SavedModel format), model is blessed (*Evaluator*, *InfraValidator*).
- Emits: the same SavedModel (*ML Metadata*).

The following is an example of how to using *Pusher*.

```

pusher = Pusher(
  model=trainer.outputs['model'],
```

```
  model_blessing=evaluator.outputs['blessing'],
  infra_blessing=infra_validator.outputs['blessing'],
  push_destination=tfx.proto.PushDestination(
    filesystem=tfx.proto.PushDestination.Filesystem(base_directory=
   serving_model_dir)
  )
)
```

Listing 17: Using Pusher

4 TFX Libraries

TensorFlow Extended provides a set of open-source libraries that are designed to help developers build, train, and deploy machine learning models at scale. These libraries include a range of tools and components that are specifically designed for large-scale machine learning applications. By leveraging TFX's libraries, developers can easily preprocess data, train models, and deploy them to production. Below are detailed descriptions of key libraries in TFX.

4.1 Data Validation

TensorFlow Data Validation [1] (TFDV) is a library used to explore and validate machine learning data. It's made to be very scalable and integrate well with TensorFlow and TensorFlow Extended (TFX). TFDV can help in detecting problems in your data such as missing data, data anomalies, labels get misidentified as features, values that have unexpected range, etc. TFDV can also help you with feature engineering in order to make the feature sets more effective. With TFDV you can spot out which features are useful, which are redundant, features with such a wide range of scale that may cause slow learning, etc.

4.1.1 Setup

1. Install the TensorFlow Data Validation package using pip:

```
pip install tensorflow-data-validation
```

Listing 18: Install TFDV

2. Import TFDV in your Python code:

```
import tensorflow_data_validation as tfdv
```

Listing 19: Import TFDV

3. Load data and generating statistics on your dataset using a CSV file or a TFRecord instead.
4. Once the statistics have been generated, visualize it either using Facets or using the built in TFDV visualization function.
5. Create your schema that explains the common features of your data, such as column datatypes, data existence or absence, and the expected range of values.

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

Listing 20: Infer a schema

4.1.2 Some main functions and methods

1. *generate_statistics_from_csv*: Computes the data statistics and returns the result data statistics proto from CSV files. *generate_statistics_from_dataframe* for data in dataframe format, *generate_statistics_from_tfrecord* for TFRecord files.
2. *visualize_statistics*: Visualizes the input statistics using Facets.
3. *infer_schema*: Infers schema from the input statistics.
4. *validate_statistics*: Validates the input statistics against the provided input schema.
5. *get_feature*: Get a feature from the schema.
6. *get_domain*: Get the domain associated with the input feature from the schema.
7. *set_domain*: Sets the domain for the input feature in the schema.

For example, the below code fragment infers a schema from statistics computed from train split and validates the statistics of eval split to check for anomalies:

```
# Read and split the input data
df = pd.read_csv('data.csv')
train_df, eval_df = train_test_split(df, test_size=0.2, shuffle=False)

# Compute statistics for 2 splits
train_stats = tfdv.generate_statistics_from_dataframe(train_df)
eval_stats = tfdv.generate_statistics_from_dataframe(eval_df)

# Infer the schema and validate the eval split statistics
schema = tfdv.infer_schema(statistics=train_stats)
anomalies = tfdv.validate_statistics(statistics=eval_stats, schema=schema)
```

4.2 Transform

TensorFlow Transform(TFT)[17] provides functions to describe the transformations to be applied to data. When a function of TFT is called, this function is then converted into a TensorFlow graph that can be executed efficiently on large datasets using Apache Beam or similar calculation tools. TFT is particularly useful for data that requires a full-pass over the entire dataset, such as normalizing an input value by mean and standard deviation or converting strings to integers by generating a vocabulary over all input values.

TFT integrates seamlessly with other components of the TensorFlow Extended (TFX) platform, such as ExampleGen for ingesting data, SchemaGen for inferring a schema for your data, and Trainer for training machine learning models. This is necessary to build end-to-end machine learning pipelines.

4.2.1 Module, Class and Function

A module in TFT library is a collection of class, function, and variable definitions organized in a meaningful way. Following are the main modules in TFT[9]:

- tft.coders: provides classes for encoding and decoding data. These classes help convert data between different formats, such as from tabular data to tensor data and vice versa.
- tft.experimental contains experimental features that are not yet part of the stable API.
- tft: tft is the main module of the TensorFlow Transform library. It provides a set of functions and classes for preprocessing data before training a machine learning model.

Since performing a transform usually only requires using the functions and classes in the tft module, here we will only present the functions and classes in this module.

Main Classes

- class DatasetMetadata: Metadata about a dataset used for the "instance dict" format.
- class TFTransformOutput: A wrapper around the output of the tf.Transform.
- class TransformFeaturesLayer: A Keras layer for applying a tf.Transform output to input layers.

Functions in tft: tft library provides many functions to use for transform and processing before training, can be grouped into different categories based on their functionality:

- Scaling functions: These functions scale numerical data to a specific range. Examples include tft.scale_to_0_1, tft.scale_by_min_max, and tft.scale_by_min_max_per_key. These functions take in a numerical tensor and additional arguments specifying the range to scale to. They return a tensor of the same shape where the values have been scaled to the specified range.
- Bucketizing functions: These functions bucketize numerical data into discrete ranges. Examples include tft.bucketize and tft.bucketize_per_key. These functions take in a numerical tensor and additional arguments specifying the number of buckets or the bucket boundaries. They return an integer tensor where the values have been bucketized into discrete ranges.
- Mapping functions: These functions map categorical data to a vocabulary. Examples include tft.compute_and_apply_vocabulary and tft.apply_vocabulary. These functions take in a categorical tensor and additional arguments specifying the vocabulary to map to. They return an integer tensor where the values have been mapped to the specified vocabulary.
- Analyzers: These functions compute full-pass statistics over the entire dataset. Examples include tft.min, tft.max, and tft.mean. These functions take in one or more tensors and compute full-pass statistics over the entire dataset. They return a constant tensor representing the computed statistic.

These functions are usually called in function **preprocessing_fn** to preprocess input columns into transformed columns. Here is an example of using tft.scale_to_0_1 function to transform numeric features and using tft.compute_and_apply_vocabulary to transform category features.

```
1     def preprocessing_fn(inputs):
2       """Preprocess input columns into transformed columns."""
3       outputs = []
4       # Scale numeric columns to have range [0, 1].
```

```
    for key in NUMERIC_FEATURE_KEYS:
      outputs[key] = tft.scale_to_0_1(inputs[key])
    # Generates a vocabulary for x and maps it to an integer with this vocab.
    for key in _VOCAB_FEATURE_KEYS:
      outputs[_transformed_name(
          key)] = tft.compute_and_apply_vocabulary(
              _fill_in_missing(inputs[key]),
              top_k=_VOCAB_SIZE,
              num_oov_buckets=_OOV_SIZE)

      return outputs
```

Listing 21: preprocssing_fn function example

4.3 Model Analysis

Model Analysis [17] allows users to analyze the performance of machine learning models and gain insights into their behavior. It supports a variety of metrics, including accuracy, precision, recall, F1 score, and AUC-ROC curve. These metrics can be computed on different slices of the data, such as by label, feature, or time, to provide more granular insights into model performance.

Model Analysis also supports visualizations for exploring model behavior, such as confusion matrices, calibration plots, and partial dependence plots. These visualizations can help identify areas where the model may be making mistakes or where it may be overfitting or underfitting the data.

4.3.1 Setup

1. Install the TensorFlow Model Analysis package using pip:

```
pip install tensorflow-model-analysis
```

Listing 22: Install TFMA

2. Import the necessary packages in your Python code:

```
import tensorflow_model_analysis as tfma
```

Listing 23: Import TFMA

3. Prepare data for evaluation. This may involve preprocessing data, splitting it into training and evaluation sets, and converting it into the appropriate format.

4. Train TensorFlow model using prepared data.

5. Evaluate trained model using TFMA. Calling the $tfma.run\_model\_analysis$ function and passing in the necessary parameters, such as the location of the trained model, the location of evaluation data, and the evaluation metrics that want to use.

4.3.2 Tensorflow Model Analysis: Metrics and Plots

TFMA supports a range of metrics and plots [18] for model evaluation, including:

- Standard Keras metrics, which are available in the $tf.keras.metrics$ module.

- Standard TFMA metrics and plots, which are available in the $tfma.metrics$ module.
- Custom Keras metrics that are derived from $tf.keras.metrics.Metric$.
- Custom TFMA metrics that are derived from $tfma.metrics.Metric$, and which can be combined using custom beam combiners or derived from other metrics.

Regression Metrics: are used to evaluate models that predict continuous numerical values, such as housing prices or stock prices. Common regression metrics include mean absolute error and mean squared error.

Binary Classification Metrics: are used to evaluate models that make binary predictions, such as whether a customer will churn or not. Common binary classification metrics include accuracy, precision, recall, and area under the ROC curve.

...

4.3.3 TensorFlow Model Analysis: Visualizations

TFMA provides a range of visualization tools [3] that help users gain insights into their model's performance and identify areas for improvement. Some of the key visualizations include:

1. Slice-Based Metrics: TFMA allows users to slice their data and evaluate their model's performance on specific subsets of their data. This can be done using custom slicing functions or by selecting pre-defined slices like time or geographic regions. The results of these analyses can be visualized using heatmaps, bar charts, or line graphs.
2. Confusion Matrices: Confusion matrices are a common tool for visualizing the performance of classification models. TFMA provides functions for generating confusion matrices for binary and multi-class classification models
3. ROC Curves: ROC curves are another common tool for evaluating classification models. TFMA provides functions for generating ROC curves for binary and multi-class classification models.
4. Fairness Indicators: TFMA includes tools for evaluating and visualizing model fairness, using metrics like demographic parity and equalized odds.
5. Training and Validation Curves: TFMA can be used to visualize how a model's performance changes over time as it is trained on larger and larger datasets.

All of these visualizations can be customized and combined to provide a comprehensive picture of a model's performance, and to help users identify areas for improvement.

4.3.4 Tensorflow Model Analysis Model: Validations

TFMA provides a range of model validation tools [2] that help users ensure that their models are performing correctly and not overfitting the data. Some of the key model validation functions include:

- Cross-validation: TFMA provides functions for performing k-fold cross-validation, which involves dividing the data into k subsets and training the model on k-1 of those subsets, using the remaining subset for testing. This allows users to evaluate the model's performance on multiple subsets of the data, reducing the risk of overfitting.
- Bootstrapping: Bootstrapping involves randomly sampling the data with replacement to generate multiple "bootstrap samples," which are then used to train and evaluate the model. This can be useful for assessing the stability of the model's performance and identifying potential sources of bias.
- Baseline Comparisons: TFMA allows users to compare their model's performance to baseline models, such as a model that always predicts the mean value of the target variable. This can help users identify whether their model is performing better than chance, and how much room there is for improvement.
- Metric Thresholding: Some models require that certain performance metrics meet a certain threshold before they can be considered "valid." TFMA provides functions for setting these thresholds and evaluating whether the model meets them.
- Hyperparameter Tuning: TFMA can be used in conjunction with hyperparameter tuning libraries like TensorFlow's Tuner to identify the best hyperparameters for the model, and ensure that the model is not overfitting the data.

All of these model validation tools help users ensure that their models are performing correctly and are not overfitting the data, reducing the risk of errors and ensuring that the model is useful for making accurate predictions.

4.4 Serving

TensorFlow Serving for production can: deploy new algorithms, but still keeping the same server architecture and APIs; integration with TensorFlow, but can be easily extended to serve other types of models; zero-downtime deployment; high Available; serving multiple models; performance for many requests, multi threading task; interface for gRPC and REST API, servable for many format (such as text, image, embedding,..), encapsulation separate from model, batching processing for server and client side; and well maintained.

4.4.1 Setup

1. Use tensorflow-model-server from APT or can install Docker [5] (from web) for serving.

```
apt-get update && apt-get install tensorflow-model-server
```

Listing 24: Install tensorflow-model-server

2. If use Docker, pull down a tensorflow serving image by:

```
docker pull tensorflow/serving
```

Listing 25: Pull tensorflow serving image

4.4.2 Serving model

Running a serving image

The serving images have the following properties:

- Port 8500 and 8501 exposed for gRPC and REST API.
- Name and base path to model (*MODEL_NAME* and *MODEL_BASE_PATH*).

```
tensorflow_model_server --port=8500 --rest_api_port=8501 \
  --model_name=${MODEL_NAME} --model_base_path=${MODEL_BASE_PATH}/${MODEL_NAME}
)
```

Listing 26: Using *tensorflow_model_server*

```
docker run -p 8500:8500 -p 8501:8501 \
  --mount type=bind,source=${MODEL_BASE_PATH}/${MODEL_NAME},target=/models/${
   MODEL_NAME} \
  -e MODEL_NAME=${MODEL_NAME} -t tensorflow/serving
)
```

Listing 27: Using *docker*

When running a serving imange with *docker*, needs:

- Open port on serving host.
- SavedModel.
- Name of model.

Query the model by curl [19] to predict API

```
curl -d '{"instances": ${INPUT}}' \
  -X POST http://localhost:8501/v1/models/${MODEL_NAME}:predict
```

Listing 28: Request to the lastest model

```
curl -d '{"instances": ${INPUT}}' \
  -X POST http://localhost:8501/v1/models/${MODEL_NAME}/versions/${VERSION}:
   predict
```

Listing 29: Request to the specified model

Creating own serving image

Add built model to serving image into the container by create own image:

- Run a serving image daemon.
- Copy SavedModel to the container.
- Commit the container.
- Stop daemon.

```
docker run -d --name serving_base tensorflow/serving
docker cp models/<my model> serving_base:/models/<my model>
docker commit --change "ENV MODEL_NAME <my model>" serving_base <my container>
docker kill serving_base
```

Listing 30: Creating own image

4.4.3 Architecture

Key Concepts

- Servables: the underlying objects that clients use to perform computation (for example, a lookup or inference).
- Loaders: manage a servable's life cycle, standardize the APIs for loading and unloading a servable.
- Sources: plugin modules that find and provide servables.
- Managers: handle the full lifecycle of Servables (loading, serving, unloading).
- Core: manages lifecycle and metrics of servables.

Tensorflow Servable Architecture [4]

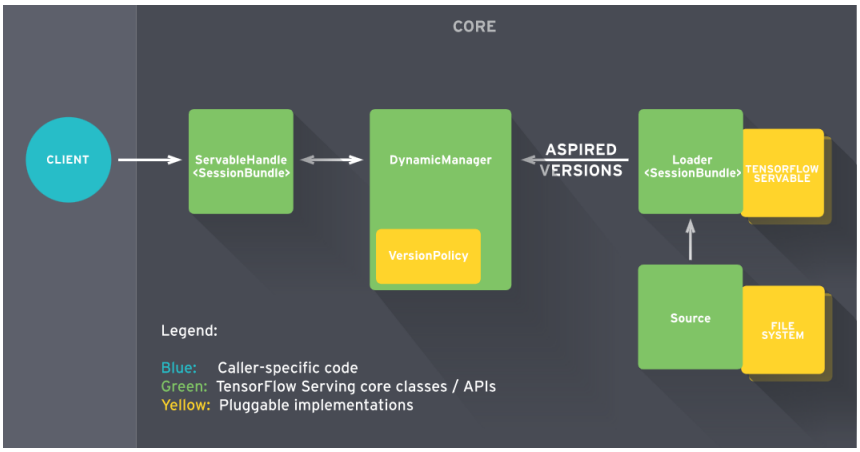

Figure 5: Life of a Servable

1. Source creates a Loader with specific version (Loader contains metadata).
2. Source callback the Aspired Version to Manager.
3. Manager use Version Policy to determine the next action (unload previous loaded version or load new version).
4. Manager determines if safe, it give resource and request load new version to Loader.
5. Client ask (specifying version or lastest version) to Manager via Servable.
6. Manager returns a handle for Servable.

5 Discussions and Conclusion

5.1 Application & Popularity

There are no specific statistics on the number of users of TFX. However, according to a survey by Statista [20], TensorFlow is one of the most used frameworks among developers globally in 2022 with 12.95% users, outperforms other machine learning frameworks. That TFX is one of the important and prominent platforms of Tensorflow, so it can be said that the number of users of TFX is also very large.

TFX has a wide range of applications in both academic and industry activities. In the academic field, TFX has been used to facilitate research in various areas such as natural language processing, computer vision, and speech recognition. For example, Google used TFX to build a system that could transcribe speech in real-time, which was used to improve accessibility for individuals who are deaf or hard of hearing. TFX has also been used by researchers to automate the process of data cleaning and preparation for machine learning models, which can save significant time and effort.

In the industry, TFX has been used by various companies to streamline their machine learning pipelines and improve the efficiency of their data processing. For example, Airbnb used TFX to build a machine learning platform that could automatically generate descriptions for their property listings, which improved the quality and consistency of their listings. TFX has also been used by Uber to improve their fraud detection system by automating the process of data preparation and feature engineering. In addition, many companies are using tfx like google, spotify, cocacola...

5.2 Other solutions

Platform	**Description**	**Pros**	**Cons**
TFX	Google's end-to-end platform helps build large-scale machine learning pinelines on Tensorflow.	Tightly integrates with TensorFlow and provides components for data processing, model training and evaluation, and model deployment.	There are some limitations when using on non-Google platforms.
Kubeflow Pipeline	Open source platform for building and deploying end-to-end machine learning workflows on Kubernetes.	Each step in the pipeline is isolated in its own container, improving the developer experience. Highly scalable and flexible due to relying on Kubernetes to manage all code execution, resource and network management.	To use it well requires experience. Works just fine on Kubernetes.

Apache Airflow	An open source platform for scheduling and monitoring workflows.	Allows you to build workflows as Python code and schedule them to run on a specific schedule or when an event occurs. There is a large library of tools.	The user interface may be unresponsive and unsuitable for special machine learning models.
ML flow	An open source platform for machine learning process lifecycle management.	Provides tools for test tracking, code packaging, and model deployment. Compatible with various machine learning platforms.	Limited model serving.

5.3 Conclusion

In conclusion, TFX is a powerful framework that offers a wide range of functionalities for building and deploying machine learning pipelines at scale. It provides a unified platform for managing and automating the entire ML workflow, from data preparation to model training and deployment. TFX allows data scientists and engineers to focus on the high-level aspects of machine learning, such as feature engineering and model selection, while abstracting away the low-level details of data preprocessing and model serving.

TFX has gained significant popularity in both academia and industry, and is being widely used by many organizations around the world (as Google, Intel,...). While there are other solutions available that offer similar functionalities, TFX stands out in its ability to integrate seamlessly with other TensorFlow-based tools and technologies, making it a preferred choice for many users. The rich set of features, ease of use, and scalability of TFX make it a compelling option for anyone looking to build and deploy ML pipelines efficiently and effectively

6 Group Schedule

The plan is made by Nguyen Tan Phat - 20127588. Teachers can see details in Trello

ID	Week 4 & 5	Week 6 & 7 & 8	Week 9 & 10 & 11
20127588	- Overview research - Install Tensorflow on laptop - Build basic model ML - Plan for the week	- Plan for the week - Research about TFX pipeline - Assign work to each member - Report: Transform component - Report: Trainer component	- Plan for the week - Report: Model Analysis Libraries - tfma - Slide for the first submission - Code: TFX Pipeline - Code: Model Analysis - Synthesize code, slides, reports to edit and submit
20127383	- Overview research. - Install TF, TFX on Colab.	- TFX pipeline Basics. - Research, report: Evaluator, Pusher components.	- Research, report: Serving library and deploy. - Code: demo for serving, TFX Pipeline. - Slides design: Components.
20127484	- Overview research. - Import Tensorflow and TFX on Google Colab.	- Watch some basic tutorials. - Research and write report about: ExampleGen component and StatisticsGen component.	- Research and write report about: TFX pipeline and TensorFlow Data Validation (TFDV) library. - Code a simple demo using some main functions of TFDV. - Make presentation slides (PowerPoint): TFX pipeline
20127374	- Overview research - Try TF, TFX on Google Colab	- Watch some tutorials - Research and write report about: SchemaGen and ExampleValidator	- Research and write report about: Transform(TFT) library - Code a simple demo using TFT library - Slides design: Conclusion.

References

[1] Google. TensorFlow Data Validation: Checking and analyzing your data. https://www.tensorflow.org/tfx/guide/tfdv, 2021.

[2] Google. Tensorflow Model Analysis Model Validations. https://www.tensorflow.org/tfx/model_analysis/model_validations, 2021.

[3] Google. TensorFlow Model Analysis Visualizations. https://www.tensorflow.org/tfx/model_analysis/visualizations, 2021.

[4] Google. TensorFlow Serving Architecture. https://www.tensorflow.org/tfx/serving/architecture, 2021.

[5] Google. TensorFlow Serving with Docker. https://www.tensorflow.org/tfx/serving/docker, 2021.

[6] Google. The Pusher TFX Pipeline Component. https://www.tensorflow.org/tfx/guide/pusher, 2021.

[7] Google. The StatisticsGen TFX Pipeline Component. https://www.tensorflow.org/tfx/guide/statsgen, 2021.

[8] Google. The Trainer TFX Pipeline Component. https://www.tensorflow.org/tfx/guide/trainer?hl=en, 2021.

[9] Google. Getting Started with TensorFlow Model Analysis. https://www.tensorflow.org/tfx/model_analysis/get_started, 2022.

[10] Google. tfx.v1.components.CsvExampleGen. https://www.tensorflow.org/tfx/api_docs/python/tfx/v1/components/CsvExampleGen, 2022.

[11] Google. tfx.v1.components.StatisticsGen. https://www.tensorflow.org/tfx/api_docs/python/tfx/v1/components/StatisticsGen, 2022.

[12] Google. The Evaluator TFX Pipeline Component. https://www.tensorflow.org/tfx/guide/evaluator, 2022.

[13] Google. The ExampleGen TFX Pipeline Component. https://www.tensorflow.org/tfx/guide/examplegen, 2022.

[14] Google. The ExampleValidator TFX Pipeline Component. https://www.tensorflow.org/tfx/guide/exampleval, 2022.

[15] Google. The SchemaGen TFX Pipeline Component. https://www.tensorflow.org/tfx/guide/schemagen, 2022.

[16] Google. The Transform TFX Pipeline Component. https://www.tensorflow.org/tfx/guide/transform?hl=en, 2022.

[17] Google. TensorFlow Model Analysis. https://www.tensorflow.org/tfx/tutorials/model_analysis/tfma_basic, 2023.

[18] Google. Tensorflow Model Analysis Metrics and Plots. https://www.tensorflow.org/tfx/model_analysis/metrics, 2023.

[19] Google. Train and serve a TensorFlow model with TensorFlow Serving. https://www.tensorflow.org/tfx/tutorials/serving/rest_simple, 2023.

[20] Statista. Most used libraries and frameworks among developers, worldwide, as of 2022. https://www.statista.com/statistics/793840/worldwide-developer-survey-most-used-frameworks/, 2022.