



School of
**Computing and
Information Systems**

CS611-Machine Learning Engineering

Project Report

Chicago Taxi Fare Price Prediction

Section G1 – Group 4

Team Members:

Nguyen Thi Ngoc Anh

Nguyen Ha Quan

Vu Viet Linh

Yan Li Cheng

Table of contents

List of figures and tables.....	2
Section 1: Project Introduction	3
1.1 Project Background.....	3
1.2 Project Objective.....	3
Section 2: Dataset & Exploration.....	4
2.1. Dataset	4
2.1.1 Dataset Overview.....	4
2.1.2. Data Ingestion & Cleaning.....	4
2.2. Feature Engineering.....	5
2.2.1. Trip Start Timestamp	5
2.2.2. Pick-up & Drop-off information	6
Section 3: Pipeline Design and overview	6
3.1 Intro to TFX and Vertex A.I.....	6
3.2 TFX Pipeline theory	7
3.3 TFX Interactive Context for prototyping	7
3.4 TFX Components upstream.....	9
3.5 TFX Components Downstream	9
Section 4: Model Deployment & Monitoring	12
4. 1 Model Deployment	12
4.2 Model Monitoring.....	12
Section 5: CI/CD Process.....	13
5.1 Trigger from GitHub	13
5.2 Trigger from Cloud Scheduler	14
Section 6: Metadata Management.....	15
Section 7: Summary, Challenges & Future Work.....	15
7.1 Summary	15
7.2 challenges	16
7.3 Future Work	16
Appendix.....	18
Item 1: Zoomed view of final pipeline	18
Item 2: Unit test sample:	19
Item 3: Our repository	19
Item 4: Test Pipeline with TFX transform component.....	19

List of figures and tables

Figure 1. Project scope and objective	4
Figure 2. Table of features	4
Figure 3. Big Query Details & Results	5
Figure 4. Data distribution and graphs	5
Figure 5. ML Pipeline key steps	6
Figure 6. Completed pipeline.....	6
Figure 7. Lineage graph of an example artifact	7
Figure 8. Dataflow between components and Meta Datastore	7
Figure 9. Time taken for resources to be provision for a single component	7
Figure 10. Code sample for Interactive context.....	8
Figure 11. Running a component Interactively.....	8
Figure 12. Results from running component Interactively	8
Figure 13. Code sample for setting Hparams search space	10
Figure 14. Code Sample for creating a FNN model	10
Figure 15. Code sample for resolver.....	11
Figure 16. Code sample for Evaluator	11
Figure 17. GCP console showing the request and prediction	12
Figure 18. Model Monitoring - Input feature distribution monitoring	13
Figure 19. CI/CD pipeline from GitHub.....	13
Figure 20. Build Summary from Cloud build.....	14
Figure 21. Continuous Deployment with Cloud Scheduler	14
Figure 22. Cloud Scheduler config	14
Figure 23. Cloud Function Config	14
Figure 24. Metadata UI.....	15
Figure 25. Slices of Mean Absolute Error across days	15
Figure 26. Error Logs.....	16
Figure 27. Diagram of model with serving signature	17
Figure 28. Diagram of TFX Bigquery Example Gen	17

Section 1: Project Introduction

1.1 Project Background

The introduction of ride hailing and ridesharing services resulted in great competition for Chicago taxi companies. To further add to their woes, the emergence of covid in the year 2019 lead to an increase in remote working arrangements which reduced the demand for taxi services.¹

To help the level the playing field for local taxi companies, it should be possible to create a machine learning model that learns from the data collected from the cumulative years of all Chicago taxi trips. The model, when deployed as an application, will help taxis shift to meterless payment by providing users with an upfront price estimation feature based on the starting location and ending location.²

The trained model can have the potential to be a business planning tool for the local taxi companies.

Another interesting caveat worth mentioning is that due to new data privacy laws, the starting and ending location coordinates are now masked. This means that location coordinates are not precise. Instead, nearby coordinates are grouped to a single point to protect the rider privacy. The same applies for pickup and drop-off time (nearest 15 mins).³

While previous pricing models are trained on the precise information, our group seeks to train a model based on the masked information. The results can give us an insight to whether a useful machine learning model can be trained while protecting user privacy.

1.2 Project Objective

To achieve our business objective, our group will train a simple multilayer feedforward neural network based on masked user information from Chicago Taxi Public Dataset hosted on bigquery. The resulting model will be used to predict the estimate cost of a trip while protecting user privacy since masked information are used as inputs.

To help build the infrastructure to accommodate our model, our team will leverage on the ML Ops with Google Cloud Platform's Vertex A.I to build and deploy our Machine Learning Pipeline.

ML Ops will be suitable for our business case since our team is intending to deploy a service that feeds on large amounts of frequently updated data. Such service also needs to be trained and scaled reliably. Moreover, our choice platform should also be capable of deploying our model while allowing us to manage training artifacts effectively.

The backbone of the pipeline will be the Tensorflow Extended SDK (TFX) which is compatible with Vertex A.I. TFX offers a suite of components which supports handling large tabular datasets which fits the needs of our project greatly.

¹ <https://www.nbcchicago.com/news/local/chicago-taxi-industry-hit-hard-amid-pandemic-ride-share-trip-increases/2502335/>

² <https://wgntv.com/news/chicago-news/upfront-pricing-app-curb-helps-chicago-taxis-go-meterless/>

³ <https://digital.chicago.gov/index.php/chicago-taxi-data-released/>

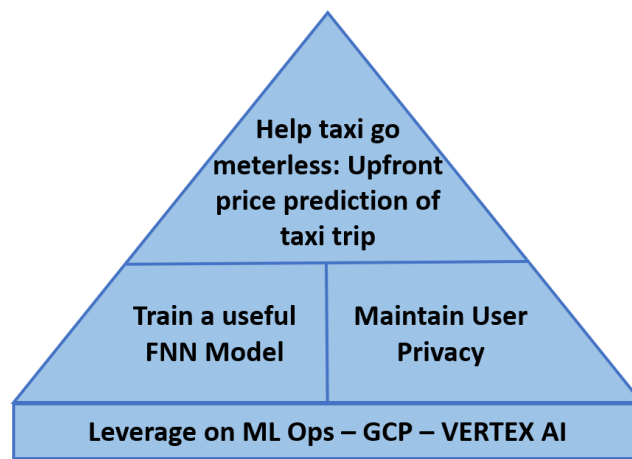


Figure 1. Project scope and objective

Section 2: Dataset & Exploration

2.1. Dataset

The dataset used in this project is Chicago Taxi Trips. This data is available in Google Big Query and have been collected since 2013, and there are around 200 millions of taxi trip records being updated every 10 to 20 days. The record contains relevant information of a taxi trip and most importantly is the taxi fare.

2.1.1 Dataset Overview

The Chicago taxi trip data structure includes numerical, categorical, geo-location as well as datetime features.

- Numerical columns: Taxi Fare, Trip Duration (in seconds), Trip Distance (Miles), and Extra Charges, Tips
- Categorical columns: Taxi Company, Payment Type, Pick-up Community Area and Drop-off community area
- Geo Location: start longitude, start latitude, end longitude and end latitude
- Datetime columns: Pick-up timestamp and drop-off timestamp

Numerical	Categorical	Geo Location	Date Time
Fare	Taxi Company	Start - Long	Pickup Time
Trip Seconds	Payment Type	Start - Lat	Dropoff Time
Trip Miles	Pickup Comm	End - Long	
Extra Charges	Drop Comm	End - Lat	

Figure 2. Table of features

Since the goal of this project is to predict the taxi fare price at the beginning of the trip, so such columns like Trip Duration (in seconds), Trip Distance (Miles), and Extra Charges, Tips and drop-off timestamp will not be available at the start. Therefore, those columns will be excluded from the entire project

2.1.2. Data Ingestion & Cleaning

The dataset will be ingested directly from the source, which is Google Big Query. Below figure is the snapshot of the SQL code used to query necessarily columns for this project. On top of that, the SQL code is also in charge of performing the initial data cleaning such as filtering NULL columns and formatting the datetime.

```
#get from an earlier generated table
FROM(
SELECT trip_total, pickup_latitude, pickup_longitude, dropoff_latitude, dropoff_longitude, ST_Distance(ST_GeogPoint(pickup_longitude,
pickup_latitude), ST_GeogPoint(dropoff_longitude, dropoff_latitude)) AS euclidean, FORMAT_DATE('%B', trip_start_timestamp) AS month,
pickup_community_area, dropoff_community_area,

FORMAT_DATE('%A', trip_start_timestamp) AS day,
FROM `bigquery-public-data.chicago_taxi_trips.taxi_trips`
WHERE trip_start_timestamp IS NOT NULL
AND trip_seconds IS NOT NULL
AND trip_miles IS NOT NULL
AND pickup_community_area IS NOT NULL
AND dropoff_community_area IS NOT NULL
AND fare IS NOT NULL
AND trip_total IS NOT NULL
AND pickup_latitude IS NOT NULL
AND pickup_longitude IS NOT NULL
AND dropoff_latitude IS NOT NULL
AND dropoff_longitude IS NOT NULL
AND pickup_location IS NOT NULL
AND dropoff_location IS NOT NULL
AND EXTRACT(YEAR FROM trip_start_timestamp) = 2021
AND trip_total < 100.0
AND trip_miles > 0
```

Query results						
JOB INFORMATION		RESULTS		JSON	EXECUTION DETAILS	
Row	extras	trip_total	payment_type	company	pickup_latitude	pickup_longitude
1	0.0	4.75	Cash	Flash Cab	41.968069	-87.721559063
2	0.0	5.75	Cash	Flash Cab	41.968069	-87.721559063
3	1.0	8.75	Cash	Flash Cab	41.968069	-87.721559063
4	0.0	3.25	Cash	Flash Cab	41.968069	-87.721559063
5	1.5	6.25	Cash	Flash Cab	41.968069	-87.721559063
6	21.5	28.46	Credit Card	Flash Cab	41.968069	-87.721559063

Figure 3. Big Query Details & Results

2.2. Feature Engineering

The feature engineering will be performed directly either from the Big Query or via the TFX Transform component as part of the pipeline. Since the main objective of this project focuses on bringing up the end-to-end machine learning pipeline on Google Cloud Platform, so the feature engineering is less emphasised and only focuses on trip start timestamp & pickup, drop-off related columns.

2.2.1. Trip Start Timestamp

trip_start_timestamp
2021-12-15 09:00:00

There are quite a number of aspects that can be explored from the trip start timestamp column. Firstly, the year information is extracted and we will train the model on yearly basis data as the data distribution is varying from year to year. In fact, the data distribution varies on different weeks, months or quarters, but for the simplicity, we will make the assumption that the data distribution is varying from year to year.

The month data is binned out and create a new quarter (season) column, and based on the data exploratory analysis (EDA) for the year of 2021, the total taxi trips for each quarter are really different as shown on the below figure.

The date will be map to the day of week information as new column since the taxi demand will be varying depend on the day of the week, and the EDA proves that, the taxi demands are more on the weekday and reaching the peak at Friday, and reducing over the weekend. Similarly, the hour information is also quite useful when predicting the taxi fare price, therefore hour is created as a standalone column.

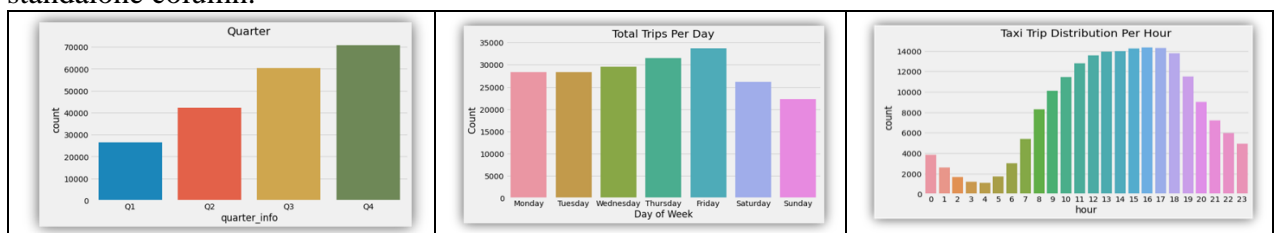


Figure 4. Data distribution and graphs

2.2.2. Pick-up & Drop-off information

The pick-up & drop-off information contains latitude and longitude information as well as the community area. The pick-up & drop-off latitude and longitude is used to calculate the Euclidean Distance between the start and the end location. This might not be the most accurate measuring, but since for the simplicity only. On the other hand, the pick-up & drop-off community area is binned out as Zone (Side) feature as the Chicago “Side” map.

Section 3: Pipeline Design and overview

3.1 Intro to TFX and Vertex A.I

The core feature of ML Ops is a running pipeline that carries out the following key tasks:

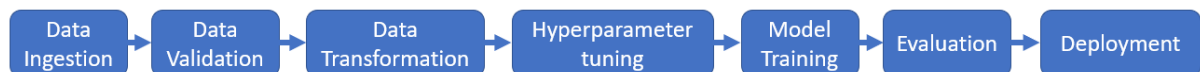


Figure 5. ML Pipeline key steps

Tensorflow Extended SDK (TFX) to help us build our pipeline.



TFX offers a useful suite of pre-built components which performs a series of functions that eventually leads to the deployment of a trained FNN model on an endpoint.

To develop our pipeline, our team chose to use Vertex A.I to prototype, experiment and develop our pipeline.

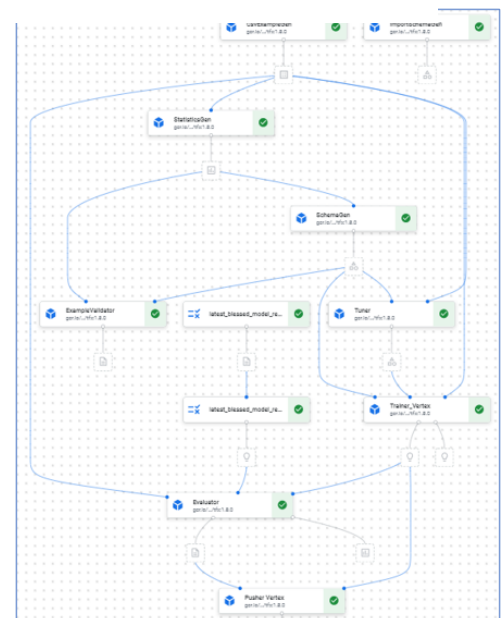


Figure 6. Completed pipeline



The vertex A.I Platform is an upgrade of the current A.I Platform by GCP. With vertex A.I, it is no longer necessary to spin up a Kubernetes compute instance to run the pipeline. Instead, all these are handled in the back end by vertex A.I.

Every single component is treated as a runnable container and the required compute resource is provisioning by Vertex A.I in the backend with the need for user configuration.

In addition to managed computational resource, Vertex A.I also offers our team a useful metadata store which keeps track of the Artifacts generated by TFX components during the pipeline run. Clicking on the metadata further provides data about the lineage of each artifact as shown below.

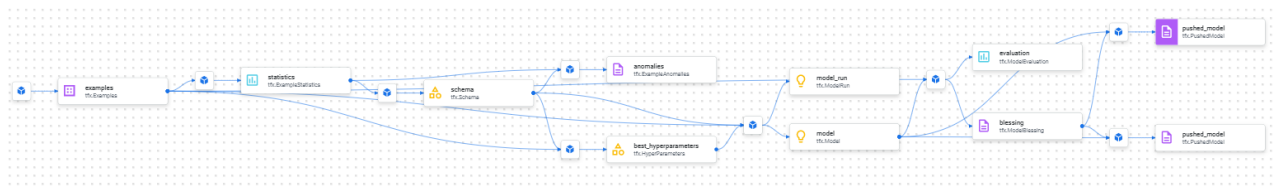


Figure 7. Lineage graph of an example artifact

3.2 TFX Pipeline theory

A TFX pipeline is made up of several TFX components. Each of these components are executed independently and produces outputs which are consumed by other TFX components. The outputs created by each component are first stored in the TFX metadata storage and then accessed by other components with an URI address.

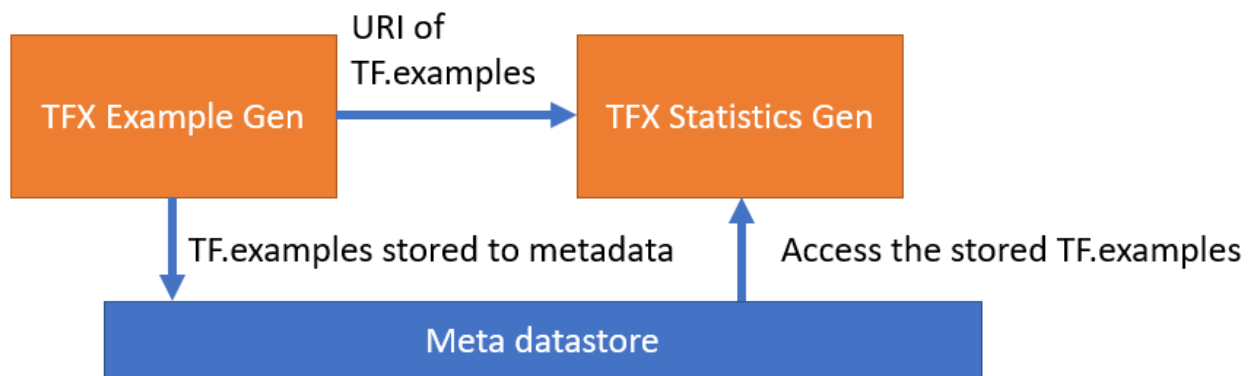


Figure 8. Dataflow between components and Meta Datastore

3.3 TFX Interactive Context for prototyping

One of the biggest challenge our team faced when prototyping the pipeline is the job submission and waiting for the pipeline run to finish and then debug any issues that may happen.

Often this process takes a very long time since for each component, the computational resources are provisioned independently. The time taken to provision varies but generally takes considerable amounts of time as shown by the logs below:

Logs Showing 212 log entries		Severity	Filter	Filter logs	
		Default			
No older entries found matching current filter.					
	2022-06-25T17:44:27.574778214Z	Waiting for job to be provisioned.			
	2022-06-25T17:44:27.807944254Z	Waiting for training program to start.			
	2022-06-25T17:44:28.792277123Z	Job is preparing.			
	2022-06-25T17:44:29.160363282Z	Opening GCS connection...			
	2022-06-25T17:44:29.160565217Z	Mounting file system "gcsfuse"...			

Figure 9. Time taken for resources to be provision for a single component

Fortunately, one of the useful features of TFX is that it offers users to execute the pipeline components locally within the notebook. This feature proved invaluable to our team and contributed greatly to the ease of experimentation and saved a lot of precious time.

```
context = InteractiveContext(  
    pipeline_name=PIPELINE_NAME,  
    pipeline_root=PIPELINE_ROOT,  
    metadata_connection_config=None)
```

Figure 10. Code sample for Interactive context

Once the interactive context is configured, our team members can run an individual component and get the output right away. Should there be an error, we can see the output right away and make the necessary corrections.

```
context.run(example_gen)
```

```
INFO:absl:Running driver for CsvExampleGen  
INFO:absl:MetadataStore with DB connection initialized  
INFO:absl:select span and version = (0, None)  
INFO:absl:latest span and version = (0, None)  
INFO:absl:Running executor for CsvExampleGen  
INFO:absl:Generating examples.  
WARNING:apache_beam.runners.interactive.interactive_environment:Dependencies required for Interactive Beam PCollections, please use: `pip install apache-beam[interactive]` to install necessary dependencies to enable all data visualization
```


Figure 11. Running a component Interactively

A typical execution result is as follow:

```
INFO:absl:Processing input csv data gs://licheng-test-06/datas/chicago-vertex-pipelines/* to TFExample.  
WARNING:root:Make sure that locally built Python SDK docker image has Python 3.7 interpreter.  
WARNING:apache_beam.io.tfrecordio:Couldn't find python-snappy so the implementation of _TFRecordUtil._make_instance will not be available.  
INFO:absl:Examples generated.  
INFO:absl:Running publisher for CsvExampleGen  
INFO:absl:MetadataStore with DB connection initialized
```

▼ **ExecutionResult** at 0x7fa08d991bd0

.execution_id 1

.component  **CsvExampleGen** at 0x7fa08d914bd0

.component.inputs {}

.component.outputs


['examples']  **Channel** of type 'Examples' (1 artifact) at 0x7fa08d914e90

Figure 12. Results from running component Interactively

The availability of the TFX interactive context made the pipeline prototyping process a much smoother one.

The following section will explain each component in detail and explain our group's rationale for the design of the pipeline and why each component was chosen.

3.4 TFX Components upstream

The components mentioned in this section deals mainly with data ingestion & validation.

TFX Example Gen

This component serves as the entry point to our pipeline. After data cleaning and feature engineering using bigquery, our team placed the generated csv data in cloud storage where it is accessed by the above component and parsed into TF.example records format. The component also does an automatic default train/eval split of 2:1 ratio on the data. Users can further specify additional split requirements where necessary.

Input: raw csv data

Output: TF.example records

TFX Statistic Gen

Once the data is parsed into TF.example format, the TFX Statistics Gen component calculates the statistics and distribution of the train and eval dataset. The resulting statistics info is later useful for data validation and generating the schema. Users can view the generated statistics with the Tensorflow Data Validation Library.

Input: TF.example records

Output: Statistics

TFX Schema Gen

This component is capable of inferring the schema using the generated data statistics and the parsed TF.examples. Schema plays an important role in maintaining the data integrity and validation because it specifies clearly the datatype of each feature as well as the domain.

For subsequent runs, users can edit and save the schema and import it to the pipeline for usage by replacing TFX schema Gen with TFX Import Schema Gen component.

Input: Statistics from TFX Statistic Gen & TF.examples from TFX Example Gen

Output: Schema

TFX Example Validator

This component plays an important role to validate the data before it is trained by the downstream components. By comparing the statistics and generated/imported schema, the component can identify anomalies such as train-eval distribution mismatch, missing values, values out of domain. By checking the report generated, users can ensure that the input data to the pipeline is of good quality.

Input: Statistics from TFX Statistic Gen & Schema from TFX Schema Gen

Output: Anomaly report

TFX Transform component

For our project, our team opted to conduct the feature engineering and data transformation in the bigquery directly. This step can also be done with the TFX Transform Component, the resulting transform graph can be added as a wrapper on top of trained model for ease of usage. This can be further explored in future work.

3.5 TFX Components Downstream

The components mentioned in this section deals with the training, evaluation, and deployment of the model.

TFX Tuner Component

Before our pipeline commences the training of the multilayer FNN, we first need to determine the best set of parameters to train the model. This is accomplished with the TFX Tuner component. For our project, we specified search grid with random search algorithm:

```
hp.Choice('learning_rate', [1e-1, 1e-2, 1e-3, 1e-4], default=1e-2)
hp.Int('n_layers', min_value=1, max_value=3, step=1, default=1)
with hp.conditional_scope('n_layers', 1):
    hp.Int('n_units_1', min_value=8, max_value=128, step=8, default=16)
with hp.conditional_scope('n_layers', 2):
    hp.Int('n_units_1', min_value=8, max_value=128, step=8, default=16)
    hp.Int('n_units_2', min_value=8, max_value=128, step=8, default=16)
with hp.conditional_scope('n_layers', 3):
    hp.Int('n_units_1', min_value=8, max_value=128, step=8, default=16)
    hp.Int('n_units_2', min_value=8, max_value=128, step=8, default=16)
    hp.Int('n_units_3', min_value=8, max_value=128, step=8, default=16)
return hp
```

Figure 13. Code sample for setting Hparams search space

The above grid is specified within the script `trainer_tune.py` in our repository.

There is an additional extension that allows the TFX component to submit a job to Vertex Vizioner tuning. This allows the user to take advantage of additional computational hardware in Vertex A.I. This can be included in future work.

Input: TFX Schema Gen & TF.examples from TFX Example Gen

Output: Best Hyper parameters

TFX Vertex Trainer Component

With the best Hyper Parameters from TFX Tuner, we can then start the training for our model. For this, we used the TFX Vertex Trainer Component extension that submits a training job to vertex A.I. To start the training, the component requires an additional trainer script where the user specifies how the model should be built as well as other conditions. A code snippet is as follow:

```
feature_keys = [f.name for f in schema.feature if f.name != _LABEL_KEY]
inputs = [keras.layers.Input(shape=(1,), name=f) for f in feature_keys]
output = keras.layers.concatenate(inputs)

print('START BUILD NOW . . .')
# build the remaining layers based on the hparams
for n in range(int(hparams.get('n_layers'))):
    print('LAYER:', n)
    print('WITH NEURONS:', hparams.get('n_units_' + str(n + 1)))
    output = tf.keras.layers.Dense(units=hparams.get('n_units_' + str(n + 1)), activation='relu')(output)

# link to the output layer
output = tf.keras.layers.Dense(1, activation='linear')(output)

# make model
model = tf.keras.Model(inputs=inputs, outputs=output)
```

Figure 14. Code Sample for creating a FNN model

The full script `trainer_vertex.py` for the above component can be found in our repository.

Since our model is trained with Vertex Trainer, we can also take advantage of the additional computational resources. This can be explored in future work.

Input: TFX Schema Gen & TF.examples from TFX Example Gen & TFX Tuner best HParams

Output: Trained Model

TFX Model Resolver Component

The TFX model resolver plays a big role in evaluating the resulting trained model. It does a search in the metadata store and automatically located the latest “Blessed” model for comparison against the recently trained model by the pipeline. By definition, a “Blessed” model is a model that is deemed fit for deployment according to the user specified evaluation metrics. The code snippet is as follow:

```
#####
# 08 resolver to find the latest blessed model
# if the latest blessed model doesn not exist, the component will ignore and auto bless current model
# NEW: RESOLVER Get the latest blessed model for Evaluator.
#####
model_resolver = tfx.dsl.Resolver(
    strategy_class=tfx.dsl.experimental.LatestBlessedModelStrategy,
    model=tfx.dsl.Channel(type=tfx.types.standard_artifacts.Model),
    model_blessing=tfx.dsl.Channel(
        type=tfx.types.standard_artifacts.ModelBlessing)).with_id(
        'latest_blessed_model_resolver')
```

Figure 15. Code sample for resolver

The full code can be found in pipeline.py at our group repository.

Output: The latest “Blessed” model

TFX Evaluator Component

The evaluator component is one of the most crucial components in our pipeline as it is the final gatekeeper before the model is deployed to the endpoint. Our newly generated model is first compared against the baseline model brought in by the TFX Model Resolver. The criterion for evaluation is:

```
# Eval component
accuracy_threshold = tfma.MetricThreshold(
    value_threshold=tfma.GenericValueThreshold(lower_bound={'value':1.0},upper_bound={'value':3.5}),
    change_threshold=tfma.GenericChangeThreshold(absolute={'value':0.6},direction=tfma.MetricDirection.LOWER_IS_BETTER))

metrics_specs = tfma.MetricsSpec(
    metrics = [
        tfma.MetricConfig(class_name='MeanAbsoluteError',
            threshold=accuracy_threshold),
```

Figure 16. Code sample for Evaluator

Change_threshold: compares the mean absolute error between baseline model and new model. Pass if change is within acceptable levels.

Value threshold: Checks if the mean absolute error of the new mode meets the minimum deployment criteria. Pass if within bounds.

Only when both condition passes will the model receive the blessing and be allowed to deploy.

Input: Baseline model from TFX Model Resolver & Model from TFX Vertex Trainer & TF.example from TFX Example Gen

Output: Model Blessing

TFX Vertex A.I Pusher component

For our pusher component, our group chose to use the TFX Vertex A.I push component over the original TFX Pusher. The pusher does a few important things:

1. Locates and uploads the blessed model from the metadata store
2. Automatically creates an endpoint
3. Deploys the model on the endpoint

This component is straightforward and easy to utilise for our project.

Input: Trained model from TFX Vertex Trainer & Model Blessing from TFX Evaluator

Output: Endpoint

Section 4: Model Deployment & Monitoring

4.1 Model Deployment

Once the Pipeline finished running its process, the final component Pusher will push newly trained models (provided that they pass the evaluation) onto endpoints, where they are available for users to send requests to and receive predictions and corresponding responses.

Request can be sent as a batch of samples (eg., CSV files) or a single instance in JSON format, as described in the example of PUT request using POSTMAN below

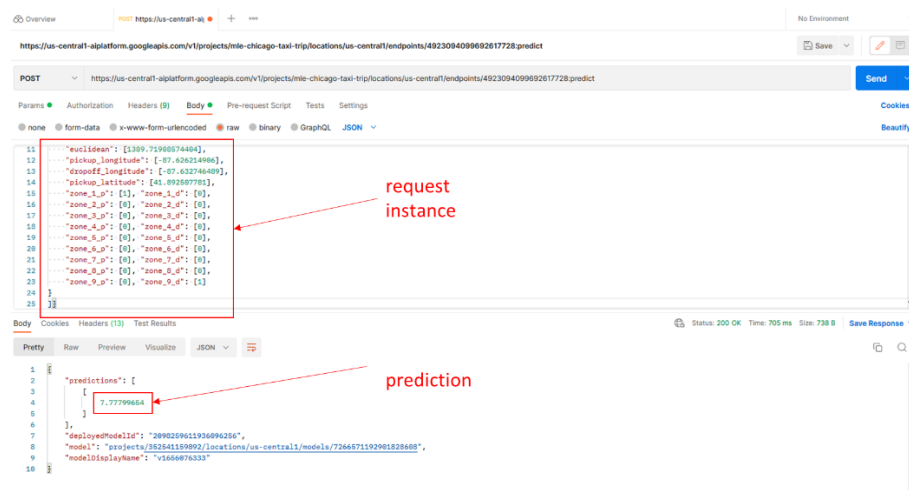


Figure 17. GCP console showing the request and prediction

Request traffic can be split into different models for multiple purposes (canary testing, A/B testing...). GCP helps record the traffic over time and request events on logs.

4.2 Model Monitoring

Model Monitoring is implemented in this project using the monitoring capability provided by GCP. Observations are applied on the serving-predict table in BigQuery while the training dataset for reference is set to be the 200,000 samples from the 2021 dataset. Several skew and drift thresholds are placed on the pickup and drop-off coordination and on multiple features to monitor the distributions between newly coming data and the training dataset.

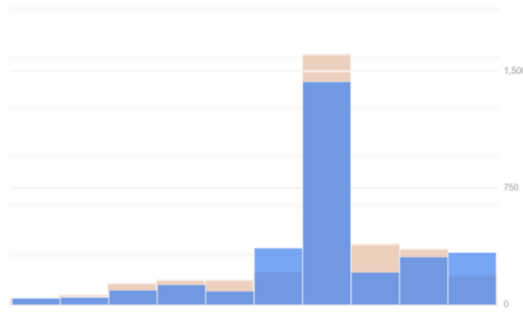


Figure 18. Model Monitoring - Input feature distribution monitoring

Above is an example of 2021(orange) and 2022(blue) distribution over the pickup_latitude feature. Through various experiments, despite the declining amount of taxi rides from 2021 to 2022, there is no significant (> 30%) difference between data distributions of multiple features. By adjusting the threshold lower, several features are able to trigger the alert and send corresponding emails to project members.

Section 5: CI/CD Process

5.1 Trigger from GitHub

When someone push their change in source code to GitHub Repo, GitHub triggered a post-commit hook to Cloud Build, then Cloud Build built the image and pushed it to Artifact Registry while re-deploy TFX pipeline in Vertex AI. While the pipeline ran in Vertex AI pipeline, it pulled the image from Artifact Registry, and resulted to an endpoint with updated model.

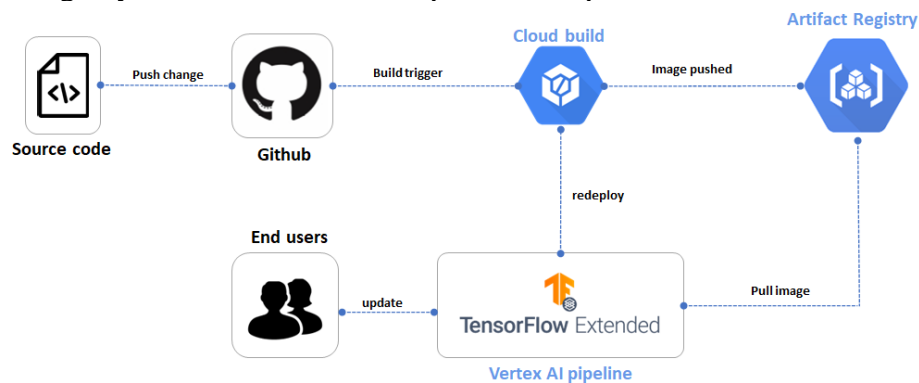


Figure 19. CI/CD pipeline from GitHub

Cloud Build Summary:

There are 8 step to run the pipeline:

- Clone repository from GitHub
- Build TFX image: image contain the pipeline
- Build CI/CD image: image to execute the pipeline
- Unit test for model: simple unit test is executed to validate configuration set correctly
- Upload trainer file to GCS
- Compile the pipeline: to pipeline.json file
- Save pipeline file to GCS

- Run the pipeline using template saved in GCS

Build details		REBUILD	COPY URL	LEARN
Successful: 8015fd66		Trigger	Source	Branch
Started on 25 Jun 2022, 11:38:55		test trigger	pkato2022/mile-taxi-fare-prediction	main/untitled
				Commit: 23f76314
Steps	Duration	BUILD LOG	EXECUTION DETAILS	BUILD ARTIFACTS
Build summary	00:29:43	<input type="checkbox"/> Wrap lines <input type="checkbox"/> Show newest entries first <input type="checkbox"/> T <input type="checkbox"/> L		
10 Steps				
0: Clone Repository	00:00:00	clone --single-branch --branch main/untitled https://github.com/pkato2022/...		
1: Build TFX Image	00:05:59	build -t us-central1-docker.pkg.dev/curious-ivy-350600/mile-pipeline/tfx-...		
2: Build cloud image	00:04:04	build -t us-central1-docker.pkg.dev/curious-ivy-350600/mile-pipeline/cloud-...		
3: Unit Test Model	00:00:01	pytest -s src/tests/model_tests.py		
4: Upload transform to GCS	00:00:03	cp src/pipeline/transform.py gs://curious-ivy-350600/pipeline_m...		
5: Upload tuner to GCS	00:00:01	cp src/pipeline/trainer_tune.py gs://curious-ivy-350600/pipeline_module...		
6: Upload vertex to GCS	00:00:01	cp src/pipeline/trainer_vertex.py gs://curious-ivy-350600/pipeline_module...		
7: Compile Pipeline	00:00:17	python build/utis.py --mode compile-pipeline --pipeline-name chicago-ve...		
8: Upload Pipeline to GCS	00:00:01	cp ./chicago-vertex-pipelines_pipeline.json gs://curious-ivy-350600/pipel...		
9: Run Pipeline	00:09:25	python build/utis.py --mode run-pipeline --pipeline-name chicago-vertex-...		

Figure 20. Build Summary from Cloud build

5.2 Trigger from Cloud Scheduler

As our data is updated every 20-25 days, we set up a Cloud Scheduler to re-execute the pipeline monthly. Every month, Cloud Scheduler trigger Cloud Function to re-execute pipeline with new data added. The pipeline re-run in Vertex AI and endpoint with new trained model is updated to end-user.

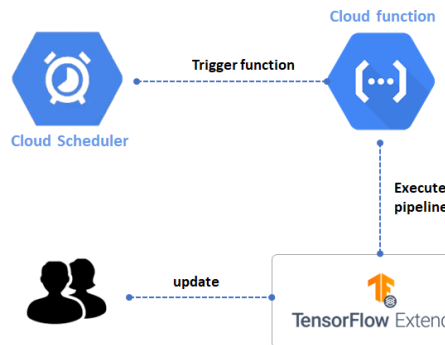


Figure 21. Continuous Deployment with Cloud Scheduler

Cloud Scheduler config is set in here: (This config set to run every 20th day of the month at 9:00 am SGT)

<input checked="" type="checkbox"/>	Name ↑	Region	State	Description	Frequency	Target	Last run	Last run result	Next run	Actions
<input checked="" type="checkbox"/>	mile-chicago-taxi-trip-scheduler	us-central1	Enabled		0 9 20 * * (Asia/Singapore)	URL: https://us-central1-mile-chicago-taxi-trip.cloudfunctions.net/pipeline-trigger	26 Jun 2022, 17:20:28	Success	20 Jul 2022, 09:00:00	⋮

Figure 22. Cloud Scheduler config

Cloud Function config is set in here. It contain the script to re-execute the pipeline

Cloud Functions

Functions

CREATE FUNCTION

REFRESH

er Filter functions

Environment	Name ↑	Region	Trigger	Runtime	Memory allocated	Executed function	Last deployed	Authentication ?	Actions
1st gen	pipeline-trigger-1	us-central1	HTTP	Python 3.7	256 MB	trigger-pipeline-via-cloudfunction	26 Jun 2022, 17:17:37		⋮

Figure 23. Cloud Function Config

Section 6: Metadata Management

The metadata for every pipeline run is stored in the Google Cloud Storage, and we can use different TFX modules in order to visualize the metadata. For example, the Feature Statistics (Train & Eval) Visualization using TensorFlow Data Validation (TFDV), Training Epoch Info using TensorBoard and Model Result Metrics Visualization using Tensorflow Model Analysis (TFMA) like below figures.

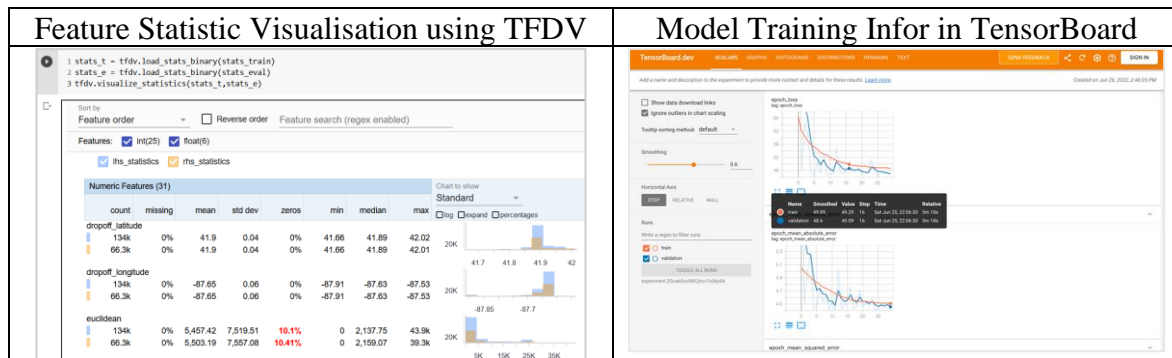


Figure 24. Metadata UI

Model Result Metrics can be viewed using TFMA and even the result can be viewed as a slice of feature as below example that the model mean absolute error is sliced in terms of day of week. From this figure, Tuesday and Saturday have the highest mean absolute error in comparison with the other days.

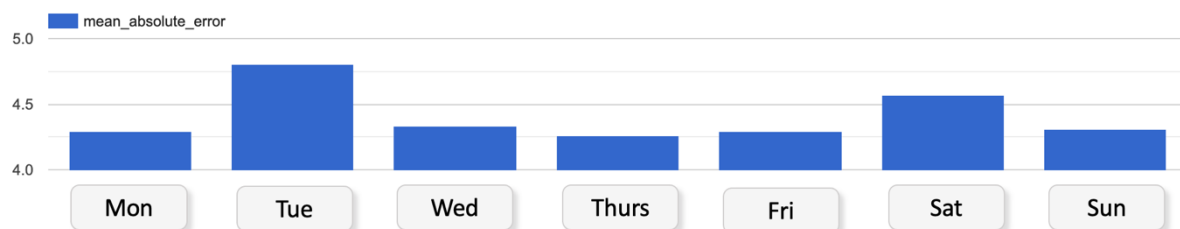


Figure 25. Slices of Mean Absolute Error across days

Section 7: Summary, Challenges & Future Work

7.1 Summary

By leveraging on ML Ops and the Vertex A.I platform, our group was able to access data from a public dataset stored on bigquery, perform the required feature engineering, train, and deploy a model using a single pipeline.

In the process of prototyping and deploying the pipeline, our team also learned to apply continuous integration methods by triggering automatic pipeline build when a code change occurs and running simple unit tests.

We also learned to apply continuous deployment where we have several models stored on Vertex A.I. By using traffic split, we can divert some of the prediction requests to the new model to test the performance.

Model monitoring further provided us the ability to record and study incoming data and their feature distributions. This proved to be useful especially for continuous training where we can identify distribution shifts in incoming data and trigger an email notification to the end user to suggest running the pipeline with latest data.

Our end result was a model that can predict the cost of a taxi trip with a mean absolute error between 3 – 4 dollars while trained on masked user data. This shows that it is still possible to produce a useful predictive model while protecting use privacy.

7.2 Challenges

Over the course of our work, our group faced many challenges as listed below:

- Challenging documentation, lots of trial and error.
- Understanding the error logs
- Limited metadata UI in Vertex A.I
- Limited interoperability with other SDK

In particular, the first 2 points proved to be highly difficult for us to overcome. The challenging documentation and examples meant that our team had to experiment and test out the behaviour of each component to fully understand how they work.

In addition, understanding the error logs proved to be a big challenge as well, since they are not easily interpreted as show below.

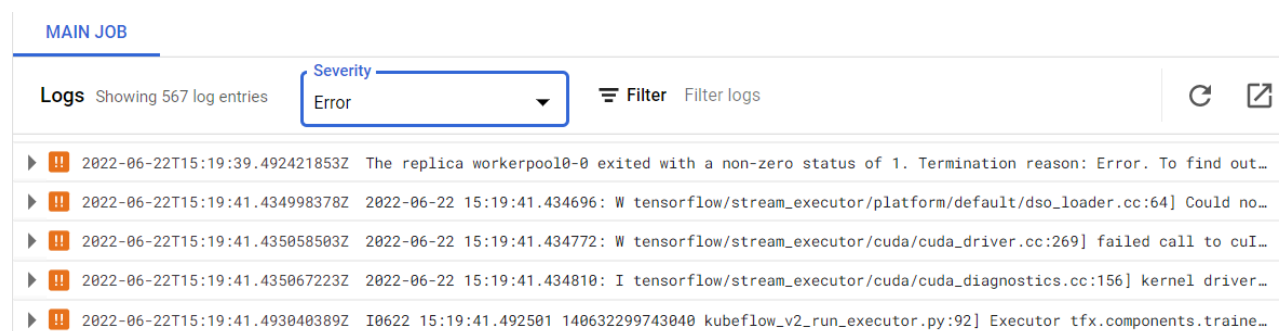


Figure 26. Error Logs

7.3 Future Work

Due to the vast capability of Vertex A.I and the TFX SDK, many areas have still been left unexplored. Our team has identified the following areas where further work can be done:

- Apply Vertex Vizier Tuning
- Create Cloud Datalab dashboard
- Incorporate TFX Transform component
- Incorporate the CSV Example Gen component
- TFX Infrastructure Validation component
- Make use of Vertex AI feature store
- Perform Vertex Model Monitoring while sending serialized TF.Example instances
- Additional feature engineering with Heuristics

Our team will elaborate on a few select points:

Incorporate TFX Transform component

By incorporating the TFX transform component, feature engineering and transformation can then be done directly within the pipeline instead of doing so at the big query. The TFX transformation component generates an additional transform graph which can then be used as a wrapper for the final model in the form of serving signatures. This way, users can provide raw feature values to the deployed model and the transformation layer will handle the data before it is sent to the model for prediction.

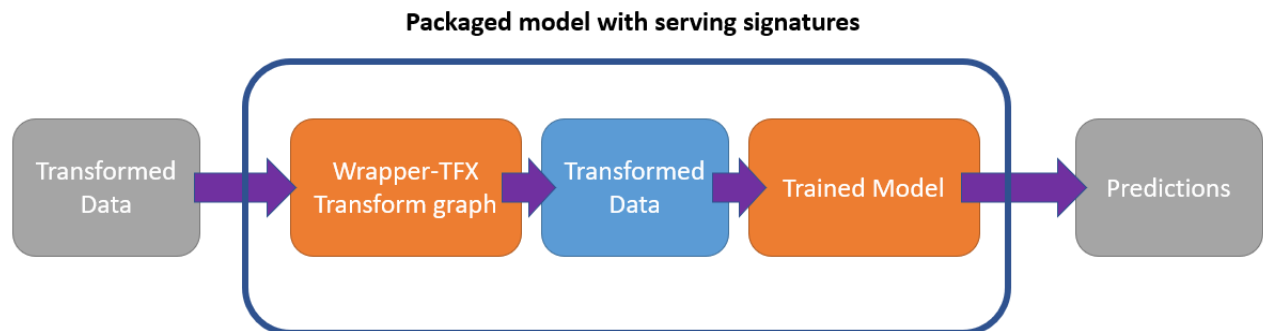


Figure 27. Diagram of model with serving signature

Incorporate TFX Bigquery Example Gen Component

Currently our team generates the raw csv files manually from Big Query. In the true automated pipeline, this needs to be automated. Fortunately, TFX offers the extension component TFX BigQuery Example Gen component.

The component works like the TFX CSV example Gen component with the exception that it runs a user provided query instead of taking in a user provided csv file. This can also provided the added benefit that our team now only need to store the input query as a metadata instead of store the actual csv itself.

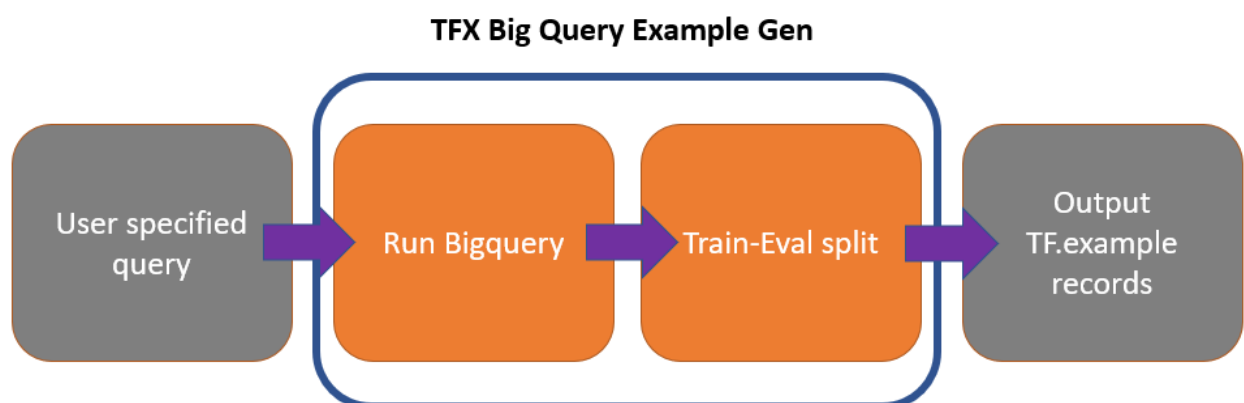
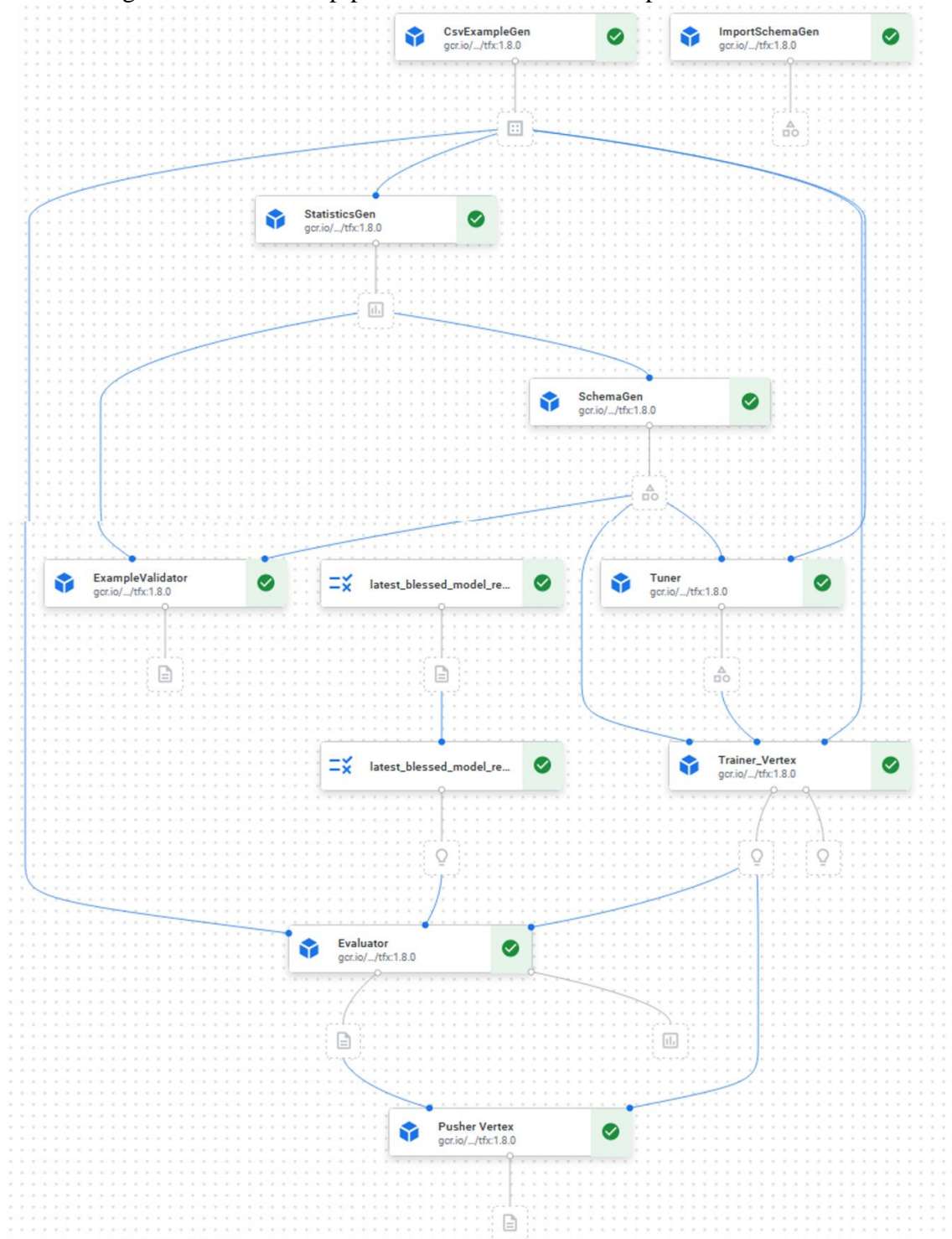


Figure 28. Diagram of TFX Bigquery Example Gen

Appendix

Item 1: Zoomed view of final pipeline

Below image details our final pipeline made with TFX components on Vertex A.I



Item 2: Unit test sample:

- Test model hyperparameter

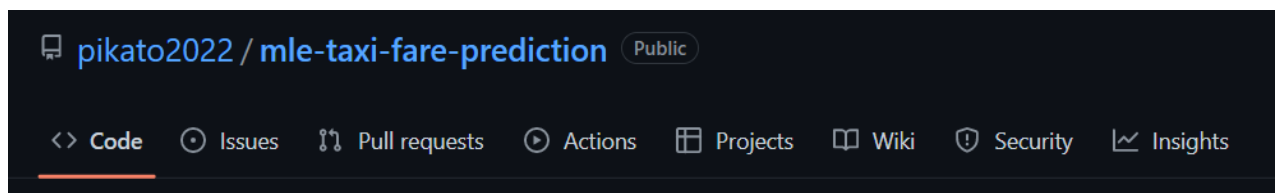
```
29 def test_hyperparams_defaults():
30     hyperparams = trainer_tune._get_hyperparameters()
31     assert set(hyperparams.keys()) == set(EXPECTED_HYPERPARAMS_KEYS)
```

- Test config set correctly

```
def test_config():
    project = os.getenv("GOOGLE_CLOUD_PROJECT")
    pipeline_name = os.getenv("PIPELINE_NAME")
    assert project, config.GOOGLE_CLOUD_PROJECT
    assert pipeline_name, config.PIPELINE_NAME
```

Item 3: Our repository

<https://github.com/pikato2022/mle-taxi-fare-prediction>



Item 4: Test Pipeline with TFX transform component

In the earlier iterations of our pipeline, our group also experimented with TFX transform component, but due to challenges with the serving signature, we opted to perform the transformation and feature engineering on Big Query instead.

