Advanced Feature Engineering in BQML

Learning Objectives

1. Evaluate the model
2. Extract temporal features, feature cross temporal features
3. Apply ML.FEATURE_CROSS to categorical features
4. Create a Euclidian feature column, feature cross coordinate features
5. Apply the BUCKETIZE function, TRANSFORM clause, L2 Regularization

Introduction

In this lab, we utilize feature engineering to improve the prediction of the fare amount for a taxi ride in New York City. We will use BigQuery ML to build a taxifare prediction model, using feature engineering to improve and create a final model. By continuing the utilization of feature engineering to improve the prediction of the fare amount for a taxi ride in New York City by reducing the RMSE.

In this Notebook, we perform a feature cross using BigQuery's ML.FEATURE_CROSS, derive coordinate features, feature cross coordinate features, clean up the code, apply the BUCKETIZE function, the TRANSFORM clause, L2 Regularization, and evaluate model performance throughout the process.

Each learning objective will correspond to a **#TODO** in the student lab notebook -- try to complete that notebook first before reviewing this solution notebook.

Set up environment variables and load necessary libraries

In [ ]:

```
# Install the Google Cloud BigQuery
!pip install --user google-cloud-bigquery==1.25.0
```

```
Collecting google-cloud-bigquery==1.25.0
Downloading https://files.pythonhosted.org/packages/48/6d/e8f5e5cd05ee968682d389cec3fdbc
cb920f1f8302464a46ef87b7b8fdad/google_cloud_bigquery-1.25.0-py2.py3-none-any.whl (169kB)
|████████████████████████████████| 174kB 3.2MB/s eta 0:00:01
Requirement already satisfied: google-cloud-core<2.0dev,>=1.1.0 in /usr/local/lib/python
3.5/dist-packages (from google-cloud-bigquery==1.25.0) (1.2.0)
Requirement already satisfied: google-resumable-media<0.6dev,>=0.5.0 in /usr/local/lib/p
ython3.5/dist-packages (from google-cloud-bigquery==1.25.0) (0.5.0)
Requirement already satisfied: google-auth<2.0dev,>=1.9.0 in /usr/local/lib/python3.5/di
st-packages (from google-cloud-bigquery==1.25.0) (1.10.1)
Requirement already satisfied: protobuf>=3.6.0 in /usr/local/lib/python3.5/dist-packages
(from google-cloud-bigquery==1.25.0) (3.11.2)
Requirement already satisfied: google-api-core<2.0dev,>=1.15.0 in /usr/local/lib/python
3.5/dist-packages (from google-cloud-bigquery==1.25.0) (1.16.0)
Requirement already satisfied: six<2.0.0dev,>=1.13.0 in /usr/local/lib/python3.5/dist-pa
ckages (from google-cloud-bigquery==1.25.0) (1.14.0)
Requirement already satisfied: setuptools>=40.3.0 in /usr/local/lib/python3.5/dist-packa
ges (from google-auth<2.0dev,>=1.9.0->google-cloud-bigquery==1.25.0) (45.0.0)
Requirement already satisfied: cachetools<5.0,>=2.0.0 in /usr/local/lib/python3.5/dist-p
```

```
ackages (from google-auth<2.0dev,>=1.9.0->google-cloud-bigquery==1.25.0) (4.0.0)
Requirement already satisfied: rsa<4.1,>=3.1.4 in /usr/local/lib/python3.5/dist-packages
(from google-auth<2.0dev,>=1.9.0->google-cloud-bigquery==1.25.0) (4.0)
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3.5/dist-pa
ckages (from google-auth<2.0dev,>=1.9.0->google-cloud-bigquery==1.25.0) (0.2.8)
Requirement already satisfied: pytz in /usr/local/lib/python3.5/dist-packages (from goog
le-api-core<2.0dev,>=1.15.0->google-cloud-bigquery==1.25.0) (2019.3)
Requirement already satisfied: requests<3.0.0dev,>=2.18.0 in /usr/local/lib/python3.5/di
st-packages (from google-api-core<2.0dev,>=1.15.0->google-cloud-bigquery==1.25.0) (2.22.
0)
Requirement already satisfied: googleapis-common-protos<2.0dev,>=1.6.0 in /usr/local/li
b/python3.5/dist-packages (from google-api-core<2.0dev,>=1.15.0->google-cloud-bigquery==
1.25.0) (1.51.0)
Requirement already satisfied: pyasn1>=0.1.3 in /usr/local/lib/python3.5/dist-packages
(from rsa<4.1,>=3.1.4->google-auth<2.0dev,>=1.9.0->google-cloud-bigquery==1.25.0) (0.4.
8)
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/local/li
b/python3.5/dist-packages (from requests<3.0.0dev,>=2.18.0->google-api-core<2.0dev,>=1.1
5.0->google-cloud-bigquery==1.25.0) (1.24.2)
Requirement already satisfied: idna<2.9,>=2.5 in /usr/local/lib/python3.5/dist-packages
(from requests<3.0.0dev,>=2.18.0->google-api-core<2.0dev,>=1.15.0->google-cloud-bigquery
==1.25.0) (2.8)
Requirement already satisfied: chardet<3.1.0,>=3.0.2 in /usr/local/lib/python3.5/dist-pa
ckages (from requests<3.0.0dev,>=2.18.0->google-api-core<2.0dev,>=1.15.0->google-cloud-b
igquery==1.25.0) (3.0.4)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.5/dist-packa
ges (from requests<3.0.0dev,>=2.18.0->google-api-core<2.0dev,>=1.15.0->google-cloud-bigq
uery==1.25.0) (2019.11.28)
Installing collected packages: google-cloud-bigquery
Successfully installed google-cloud-bigquery-1.25.0 google-resumable-media-0.5.1
WARNING: You are using pip version 19.3.1; however, version 20.2.3 is available.
You should consider upgrading via the 'pip install --upgrade pip' command.
```

Note: Restart your kernel to use updated packages.

Kindly ignore the deprecation warnings and incompatibility errors related to google-cloud-storage.

In [ ]:

```
%%bash

export PROJECT=$(gcloud config list project --format "value(core.project)")
echo "Your current GCP Project Name is: "$PROJECT
```

```
Your current GCP Project Name is: munn-sandbox
```

The source dataset

Our dataset is hosted in BigQuery. The taxi fare data is a publically available dataset, meaning anyone with a GCP account has access. Click here to access the dataset.

The Taxi Fare dataset is relatively large at 55 million training rows, but simple to understand, with only six features. The fare_amount is the target, the continuous value we'll train a model to predict.

Create a BigQuery Dataset

A BigQuery dataset is a container for tables, views, and models built with BigQuery ML. Let's create one called **feat_eng** if we have not already done so in an earlier lab. We'll do the same for a GCS bucket for our project too.

In [ ]:
```
%%bash

# Create a BigQuery dataset for feat_eng if it doesn't exist
datasetexists=$(bq ls -d | grep -w feat_eng)

if [ -n "$datasetexists" ]; then
    echo -e "BigQuery dataset already exists, let's not recreate it."

else
    echo "Creating BigQuery dataset titled: feat_eng"

    bq --location=US mk --dataset \
        --description 'Taxi Fare' \
        $PROJECT:feat_eng
   echo "\nHere are your current datasets:"
   bq ls
fi
```

```
Creating BigQuery dataset titled: feat_eng
Dataset 'munn-sandbox:feat_eng' successfully created.
\nHere are your current datasets:
    datasetId
 ---------------
  feat_eng
```

Create the training data table

Since there is already a publicly available dataset, we can simply create the training data table using this raw input data. Note the WHERE clause in the below query: This clause allows us to TRAIN a portion of the data (e.g. one hundred thousand rows versus one million rows), which keeps your query costs down. If you need a refresher on using MOD() for repeatable splits see this post.

- Note: The dataset in the create table code below is the one created previously, e.g. "feat_eng". The table name is "feateng_training_data". Run the query to create the table.

In [ ]:
```
%%bigquery

# Creating the table in our dataset.
CREATE OR REPLACE TABLE
  feat_eng.feateng_training_data AS
SELECT
  (tolls_amount + fare_amount) AS fare_amount,
  passenger_count*1.0 AS passengers,
  pickup_datetime,
  pickup_longitude AS pickuplon,
  pickup_latitude AS pickuplat,
  dropoff_longitude AS dropofflon,
  dropoff_latitude AS dropofflat
FROM
  `nyc-tlc.yellow.trips`
WHERE
  MOD(ABS(FARM_FINGERPRINT(CAST(pickup_datetime AS STRING))), 10000) = 1
  AND fare_amount >= 2.5
  AND passenger_count > 0
  AND pickup_longitude > -78
  AND pickup_longitude < -70
  AND dropoff_longitude > -78
  AND dropoff_longitude < -70
  AND pickup_latitude > 37
  AND pickup_latitude < 45
  AND dropoff_latitude > 37
  AND dropoff_latitude < 45
```

Out[ ]: —

Verify table creation

Verify that you created the dataset.

In [ ]:

```
%%bigquery

# LIMIT 0 is a free query; this allows us to check that the table exists.
SELECT
*
FROM
  feat_eng.feateng_training_data
LIMIT
  0
```

Out[ ]:

fare_amount	passengers	pickup_datetime	pickuplon	pickuplat	dropofflon	dropofflat

Baseline Model: Create the baseline model

Next, you create a linear regression baseline model with no feature engineering. Recall that a model in BigQuery ML represents what an ML system has learned from the training data. A baseline model is a solution to a problem without applying any machine learning techniques.

When creating a BQML model, you must specify the model type (in our case linear regression) and the input label (fare_amount). Note also that we are using the training data table as the data source.

Now we create the SQL statement to create the baseline model.

In [ ]:

```
%%bigquery

# Creating the baseline model
CREATE OR REPLACE MODEL
  feat_eng.baseline_model OPTIONS (model_type='linear_reg',
    input_label_cols=['fare_amount']) AS
SELECT
  fare_amount,
  passengers,
  pickup_datetime,
  pickuplon,
  pickuplat,
  dropofflon,
  dropofflat
FROM
  feat_eng.feateng_training_data
```

Out[ ]: —

Note, the query takes several minutes to complete. After the first iteration is complete, your model (baseline_model) appears in the navigation panel of the BigQuery web UI. Because the query uses a CREATE MODEL statement to create a model, you do not see query results.

You can observe the model as it's being trained by viewing the Model stats tab in the BigQuery web UI. As soon as the first iteration completes, the tab is updated. The stats continue to update as each iteration completes.

Once the training is done, visit the BigQuery Cloud Console and look at the model that has been trained. Then, come back to this notebook.

Evaluate the baseline model

Note that BigQuery automatically split the data we gave it, and trained on only a part of the data and used the rest for evaluation. After creating your model, you evaluate the performance of the regressor using the ML.EVALUATE function. The ML.EVALUATE function evaluates the predicted values against the actual data.

NOTE: The results are also displayed in the BigQuery Cloud Console under the **Evaluation** tab.

Review the learning and eval statistics for the baseline_model.

In [ ]:
```
%%bigquery

# Eval statistics on the held out data.
SELECT
  *,
  SQRT(loss) AS rmse
FROM
  ML.TRAINING_INFO(MODEL feat_eng.baseline_model)
```

Out[ ]:

	training_run	iteration	loss	eval_loss	learning_rate	duration_ms	rmse
0	0	0	74.43591	68.880408	None	12426	8.627625

Tip: Make sure to delete the "# TODO 1" when you are about to run the cell.

In [ ]:
```
%%bigquery
# TODO 1

# Use `ML.EVALUATE` function to evaluate model metrics.
SELECT
  *
FROM
  ML.EVALUATE(MODEL feat_eng.baseline_model)
```

Out[ ]:

	mean_absolute_error	mean_squared_error	mean_squared_log_error	median_absolute_error	r2_score
0	5.213479	68.880408	0.258098	3.794535	0.226065

NOTE: Because you performed a linear regression, the results include the following columns:

- mean_absolute_error
- mean_squared_error
- mean_squared_log_error
- median_absolute_error
- r2_score
- explained_variance

Resource for an explanation of the Regression Metrics.

Mean squared error (MSE) - Measures the difference between the values our model predicted using the test set and the actual values. You can also think of it as the distance between your regression (best fit) line and the predicted values.

Root mean squared error (RMSE) - The primary evaluation metric for this ML problem is the root mean-squared error. RMSE measures the difference between the predictions of a model, and the observed values. A large RMSE is equivalent to a large average error, so smaller values of RMSE are better. One nice property of RMSE is that the error is given in the units being measured, so you can tell very directly how incorrect the model might be on unseen data.

R2: An important metric in the evaluation results is the R2 score. The R2 score is a statistical measure that determines if the linear regression predictions approximate the actual data. Zero (0) indicates that the model explains none of the variability of the response data around the mean. One (1) indicates that the model explains all the variability of the response data around the mean.

Next, we write a SQL query to take the SQRT() of the mean squared error as your loss metric for evaluation for the benchmark_model.

In [ ]:
```
%%bigquery

# Use `ML.EVALUATE` function to evaluate model metrics.
SELECT
  SQRT(mean_squared_error) AS rmse
FROM
  ML.EVALUATE(MODEL feat_eng.baseline_model)
```

Out[ ]:
```
       rmse
0  8.299422
```

Model 1: EXTRACT dayofweek from the pickup_datetime feature.

- As you recall, dayofweek is an enum representing the 7 days of the week. This factory allows the enum to be obtained from the int value. The int value follows the ISO-8601 standard, from 1 (Monday) to 7 (Sunday).

- If you were to extract the dayofweek from pickup_datetime using BigQuery SQL, the datatype returned would be integer.

Next, we create a model titled "model_1" from the benchmark model and extract out the DayofWeek.

Tip: Make sure to delete the "# TODO 2" when you are about to run the cell.

In [ ]:

```
%%bigquery

# Creating the model from benchmark model and extract the Days of Week.
CREATE OR REPLACE MODEL
  feat_eng.model_1 OPTIONS (model_type='linear_reg',
    input_label_cols=['fare_amount']) AS
SELECT
  fare_amount,
  passengers,
  pickup_datetime,
# TODO 2
  EXTRACT(DAYOFWEEK
  FROM
    pickup_datetime) AS dayofweek,
  pickuplon,
  pickuplat,
  dropofflon,
  dropofflat
FROM
  feat_eng.feateng_training_data
```

Out[ ]: —

Once the training is done, visit the BigQuery Cloud Console and look at the model that has been trained. Then, come back to this notebook.

Next, two distinct SQL statements show the TRAINING and EVALUATION metrics of model_1.

In [ ]:

```
%%bigquery

# Use `ML.TRAINING_INFO` function which allows you to see information about the trainin
SELECT
  *,
  SQRT(loss) AS rmse
FROM
  ML.TRAINING_INFO(MODEL feat_eng.model_1)
```

Out[ ]:

	training_run	iteration	loss	eval_loss	learning_rate	duration_ms	rmse
0	0	0	72.440724	88.953232	None	17251	8.511212

In [ ]:

```
%%bigquery

# Use `ML.EVALUATE` function to evaluate model metrics.
SELECT
  *
FROM
  ML.EVALUATE(MODEL feat_eng.model_1)
```

Out[ ]:

mean_absolute_error	mean_squared_error	mean_squared_log_error	median_absolute_error	r2_score

	mean_absolute_error	mean_squared_error	mean_squared_log_error	median_absolute_error	r2_score
0	5.287076	88.953232	0.260932	3.713439	0.08481

Here we run a SQL query to take the SQRT() of the mean squared error as your loss metric for evaluation for the benchmark_model.

In [ ]:
```
%%bigquery

# Use `ML.EVALUATE` function to evaluate model metrics.
SELECT
  SQRT(mean_squared_error) AS rmse
FROM
  ML.EVALUATE(MODEL feat_eng.model_1)
```

Out[ ]:

	rmse
0	9.431502

Model 2: EXTRACT hourofday from the pickup_datetime feature

As you recall, **pickup_datetime** is stored as a TIMESTAMP, where the Timestamp format is retrieved in the standard output format – year-month-day hour:minute:second (e.g. 2016-01-01 23:59:59). Hourofday returns the integer number representing the hour number of the given date.

Hourofday is best thought of as a discrete ordinal variable (and not a categorical feature), as the hours can be ranked (e.g. there is a natural ordering of the values). Hourofday has an added characteristic of being cyclic, since 12am follows 11pm and precedes 1am.

Next, we create a model titled "model_2" and EXTRACT the hourofday from the pickup_datetime feature to improve our model's rmse.

In [ ]:
```
%%bigquery

# Creating the model from benchmark model and extract the hours of day.
CREATE OR REPLACE MODEL
  feat_eng.model_2 OPTIONS (model_type='linear_reg',
    input_label_cols=['fare_amount']) AS
SELECT
  fare_amount,
  passengers,
  #pickup_datetime,
  EXTRACT(DAYOFWEEK
  FROM
    pickup_datetime) AS dayofweek,
  EXTRACT(HOUR
  FROM
    pickup_datetime) AS hourofday,
  pickuplon,
  pickuplat,
  dropofflon,
  dropofflat
```

```
FROM
  `feat_eng.feateng_training_data`
```

Out[ ]: —

In [ ]:

```
%%bigquery

# Use `ML.EVALUATE` function to evaluate model metrics.
SELECT
  *
FROM
  ML.EVALUATE(MODEL feat_eng.model_2)
```

Out[ ]:

	mean_absolute_error	mean_squared_error	mean_squared_log_error	median_absolute_error	r2_score
0	5.256421	70.709518	0.262399	3.895416	0.236668

In [ ]:

```
%%bigquery

# Use `ML.EVALUATE` function to evaluate model metrics.
SELECT
  SQRT(mean_squared_error) AS rmse
FROM
  ML.EVALUATE(MODEL feat_eng.model_2)
```

Out[ ]:

	rmse
0	8.408895

Model 3: Feature cross dayofweek and hourofday using CONCAT

First, let's allow the model to learn traffic patterns by creating a new feature that combines the time of day and day of week (this is called a feature cross.

Note: BQML by default assumes that numbers are numeric features, and strings are categorical features. We need to convert both the dayofweek and hourofday features to strings because the model (Neural Network) will automatically treat any integer as a numerical value rather than a categorical value. Thus, if not cast as a string, the dayofweek feature will be interpreted as numeric values (e.g. 1,2,3,4,5,6,7) and hour ofday will also be interpreted as numeric values (e.g. the day begins at midnight, 00:00, and the last minute of the day begins at 23:59 and ends at 24:00). As such, there is no way to distinguish the "feature cross" of hourofday and dayofweek "numerically". Casting the dayofweek and hourofday as strings ensures that each element will be treated like a label and will get its own coefficient associated with it.

Create the SQL statement to feature cross the dayofweek and hourofday using the CONCAT function. Name the model "model_3"

Tip: Make sure to delete the "# TODO 3" when you are about to run the cell.

In [ ]:

```
%%bigquery

# Using `CONCAT` function to feature cross the dayofweek and hourofday
CREATE OR REPLACE MODEL
  feat_eng.model_3 OPTIONS (model_type='linear_reg',
    input_label_cols=['fare_amount']) AS
# TODO 3
SELECT
  fare_amount,
  passengers,
  #pickup_datetime,
  #EXTRACT(DAYOFWEEK FROM pickup_datetime) AS dayofweek,
  #EXTRACT(HOUR FROM pickup_datetime) AS hourofday,
  CONCAT(CAST(EXTRACT(DAYOFWEEK
      FROM
        pickup_datetime) AS STRING), CAST(EXTRACT(HOUR
      FROM
        pickup_datetime) AS STRING)) AS hourofday,
  pickuplon,
  pickuplat,
  dropofflon,
  dropofflat
FROM
  `feat_eng.feateng_training_data`
```

Out[ ]: —

In [ ]:

```
%%bigquery

# Use `ML.EVALUATE` function to evaluate model metrics.
SELECT
  *
FROM
  ML.EVALUATE(MODEL feat_eng.model_3)
```

Out[ ]:

	mean_absolute_error	mean_squared_error	mean_squared_log_error	median_absolute_error	r2_score
0	5.265321	69.356068	0.267477	3.887588	0.220371

In [ ]:

```
%%bigquery

# Use `ML.EVALUATE` function to evaluate model metrics.
SELECT
  SQRT(mean_squared_error) AS rmse
FROM
  ML.EVALUATE(MODEL feat_eng.model_3)
```

Out[ ]:

	rmse
0	8.328029

Model 4: Apply the ML.FEATURE_CROSS clause to categorical features

BigQuery ML now has ML.FEATURE_CROSS, a pre-processing clause that performs a feature cross.

- ML.FEATURE_CROSS generates a STRUCT feature with all combinations of crossed categorical features, except for 1-degree items (the original features) and self-crossing items.
- Syntax: ML.FEATURE_CROSS(STRUCT(features), degree)
- The feature parameter is a categorical features separated by comma to be crossed. The maximum number of input features is 10. An unnamed feature is not allowed in features. Duplicates are not allowed in features.
- Degree(optional): The highest degree of all combinations. Degree should be in the range of [1, 4]. Default to 2.

Output: The function outputs a STRUCT of all combinations except for 1-degree items (the original features) and self-crossing items, with field names as concatenation of original feature names and values as the concatenation of the column string values.

Here, we examine the components of ML.Feature_Cross. Note that the next cell contains errors, please correct the cell before continuing or you will get errors.

In [ ]:
```
%%bigquery

# Using the `ML.FEATURE_CROSS` clause
CREATE OR REPLACE MODEL feat_eng.model_4
OPTIONS
  (model_type='linear_reg',
    input_label_cols=['fare_amount'])
AS
SELECT
  fare_amount,
  passengers,
  #pickup_datetime,
  #EXTRACT(DAYOFWEEK FROM pickup_datetime) AS dayofweek,
  #EXTRACT(HOUR FROM pickup_datetime) AS hourofday,
  #CONCAT(CAST(EXTRACT(DAYOFWEEK FROM pickup_datetime) AS STRING),
        #CAST(EXTRACT(HOUR FROM pickup_datetime) AS STRING)) AS hourofday,
 ML.FEATURE_CROSS(STRUCT(CAST(EXTRACT(DAYOFWEEK FROM pickup_datetime) AS STRING) AS day
  CAST(EXTRACT(HOUR FROM pickup_datetime) AS STRING) AS hourofday)) AS day_hr,
  pickuplon,
  pickuplat,
  dropofflon,
  dropofflat

FROM `feat_eng.feateng_training_data`
```

Out[ ]: —

Next, two distinct SQL statements show the TRAINING and EVALUATION metrics of model_1.

In [ ]:
```
%%bigquery

# Use `ML.EVALUATE` function to evaluate model metrics.
SELECT
  *
```

```
FROM
  ML.EVALUATE(MODEL feat_eng.model_4)
```

Out[ ]:

	mean_absolute_error	mean_squared_error	mean_squared_log_error	median_absolute_error	r2_score
0	5.500029	93.260577	0.27445	3.808678	0.080263

In [ ]:

```
%%bigquery

# Use `ML.EVALUATE` function to evaluate model metrics.
SELECT
  SQRT(mean_squared_error) AS rmse
FROM
  ML.EVALUATE(MODEL feat_eng.model_4)
```

Out[ ]:

	rmse
0	9.657152

Sliding down the slope toward a loss minimum (reduced taxi fare)!

- Our fourth model above gives us an RMSE of 9.65 for estimating fares. Recall our heuristic benchmark was 8.29. This may be the result of feature crossing. Let's apply more feature engineering techniques to see if we can't get this loss metric lower!

Model 5: Feature cross coordinate features to create a Euclidean feature

Pickup coordinate:

- pickup_longitude AS pickuplon
- pickup_latitude AS pickuplat

Dropoff coordinate:

- #dropoff_longitude AS dropofflon
- #dropoff_latitude AS dropofflat

Coordinate Features:

- The pick-up and drop-off longitude and latitude data are crucial to predicting the fare amount as fare amounts in NYC taxis are largely determined by the distance traveled. As such, we need to teach the model the Euclidean distance between the pick-up and drop-off points.

- Recall that latitude and longitude allows us to specify any location on Earth using a set of coordinates. In our training data set, we restricted our data points to only pickups and drop offs within NYC. New York city has an approximate longitude range of -74.05 to -73.75 and a latitude range of 40.63 to 40.85.

- The dataset contains information regarding the pickup and drop off coordinates. However, there is no information regarding the distance between the pickup and drop off points. Therefore, we create a new feature that calculates the distance between each pair of pickup and drop off points. We can do this using the Euclidean Distance, which is the straight-line distance between any two coordinate points.

- We need to convert those coordinates into a single column of a spatial data type. We will use the ST_DISTANCE and the ST_GEOGPOINT functions.

- ST_DISTANCE: ST_DISTANCE(geography_1, geography_2). Returns the shortest distance in meters between two non-empty GEOGRAPHYs (e.g. between two spatial objects).

- ST_GEOGPOINT: ST_GEOGPOINT(longitude, latitude). Creates a GEOGRAPHY with a single point. ST_GEOGPOINT creates a point from the specified FLOAT64 longitude and latitude parameters and returns that point in a GEOGRAPHY value.

Next we convert the feature coordinates into a single column of a spatial data type. Use the The ST_Distance and the ST.GeogPoint functions.

SAMPLE CODE: ST_Distance(ST_GeogPoint(value1,value2), ST_GeogPoint(value3, value4)) AS euclidean

Tip: Make sure to delete the "# TODO 4" when you are about to run the cell.

In [ ]:
```
%%bigquery

# Convert the feature coordinates into a single column of a `spatial` data type
CREATE OR REPLACE MODEL
  feat_eng.model_5 OPTIONS (model_type='linear_reg',
    input_label_cols=['fare_amount']) AS
SELECT
  fare_amount,
  passengers,
  #pickup_datetime,
  #EXTRACT(DAYOFWEEK FROM pickup_datetime) AS dayofweek,
  #EXTRACT(HOUR FROM pickup_datetime) AS hourofday,
  #CONCAT(CAST(EXTRACT(DAYOFWEEK FROM pickup_datetime) AS STRING),
    #CAST(EXTRACT(HOUR FROM pickup_datetime) AS STRING)) AS hourofday,
  ML.FEATURE_CROSS(STRUCT(CAST(EXTRACT(DAYOFWEEK
        FROM
          pickup_datetime) AS STRING) AS dayofweek,
      CAST(EXTRACT(HOUR
        FROM
          pickup_datetime) AS STRING) AS hourofday)) AS day_hr,
  #pickuplon,
  #pickuplat,
  #dropofflon,
  #dropofflat,
  # TODO 4
  ST_Distance(ST_GeogPoint(pickuplon,
      pickuplat),
    ST_GeogPoint(dropofflon,
      dropofflat)) AS euclidean
FROM
  `feat_eng.feateng_training_data`
```

Out[ ]: —

Next, two distinct SQL statements show metrics for model_5.

In [ ]:

```
%%bigquery

# Use `ML.EVALUATE` function to evaluate model metrics.
SELECT
  *
FROM
  ML.EVALUATE(MODEL feat_eng.model_5)
```

Out[ ]:

	mean_absolute_error	mean_squared_error	mean_squared_log_error	median_absolute_error	r2_score
0	3.121311	31.229718	0.106923	2.21774	0.661492

In [ ]:

```
%%bigquery

# Use `ML.EVALUATE` function to evaluate model metrics.
SELECT
  SQRT(mean_squared_error) AS rmse
FROM
  ML.EVALUATE(MODEL feat_eng.model_5)
```

Out[ ]:

	rmse
0	5.588356

Model 6: Feature cross pick-up and drop-off locations features

In this section, we feature cross the pick-up and drop-off locations so that the model can learn pick-up-drop-off pairs that will require tolls.

This step takes the geographic point corresponding to the pickup point and grids to a 0.1-degree-latitude/longitude grid (approximately 8km x 11km in New York—we should experiment with finer resolution grids as well). Then, it concatenates the pickup and dropoff grid points to learn “corrections” beyond the Euclidean distance associated with pairs of pickup and dropoff locations.

Because the lat and lon by themselves don't have meaning, but only in conjunction, it may be useful to treat the fields as a pair instead of just using them as numeric values. However, lat and lon are continuous numbers, so we have to discretize them first. That's what SnapToGrid does.

- ST_SNAPTOGRID: ST_SNAPTOGRID(geography_expression, grid_size). Returns the input GEOGRAPHY, where each vertex has been snapped to a longitude/latitude grid. The grid size is determined by the grid_size parameter which is given in degrees.

REMINDER: The ST_GEOGPOINT creates a GEOGRAPHY with a single point. ST_GEOGPOINT creates a point from the specified FLOAT64 longitude and latitude parameters and returns that point in a

GEOGRAPHY value. The ST_Distance function returns the minimum distance between two spatial objects. It also returns meters for geographies and SRID units for geometrics.

The following SQL statement is incorrect. Modify the code to feature cross the pick-up and drop-off locations features.

In [ ]:

```
%%bigquery

# Use feature cross for the pick-up and drop-off locations features.
CREATE OR REPLACE MODEL
  feat_eng.model_6 OPTIONS (model_type='linear_reg',
    input_label_cols=['fare_amount']) AS
SELECT
  fare_amount,
  passengers,
  #pickup_datetime,
  #EXTRACT(DAYOFWEEK FROM pickup_datetime) AS dayofweek,
  #EXTRACT(HOUR FROM pickup_datetime) AS hourofday,
  ML.FEATURE_CROSS(STRUCT(CAST(EXTRACT(DAYOFWEEK
        FROM
          pickup_datetime) AS STRING) AS dayofweek,
      CAST(EXTRACT(HOUR
        FROM
          pickup_datetime) AS STRING) AS hourofday)) AS day_hr,
  #pickuplon,
  #pickuplat,
  #dropofflon,
  #dropofflat,
  ST_Distance(ST_GeogPoint(pickuplon,
      pickuplat),
    ST_GeogPoint(dropofflon,
      dropofflat)) AS euclidean,
  CONCAT(ST_AsText(ST_SnapToGrid(ST_GeogPoint(pickuplon,
          pickuplat),
        0.01)), ST_AsText(ST_SnapToGrid(ST_GeogPoint(dropofflon,
          dropofflat),
        0.01))) AS pickup_and_dropoff
FROM
  `feat_eng.feateng_training_data`
```

Out[ ]: —

Next, we evaluate model_6.

In [ ]:

```
%%bigquery

# Use `ML.EVALUATE` function to evaluate model metrics.
SELECT
  *
FROM
  ML.EVALUATE(MODEL feat_eng.model_6)
```

Out[ ]:

	mean_absolute_error	mean_squared_error	mean_squared_log_error	median_absolute_error	r2_score
0	2.678752	34.885692	0.088598	1.485065	0.640979

In [ ]:
```
%%bigquery

# Use `ML.EVALUATE` function to evaluate model metrics.
SELECT
  SQRT(mean_squared_error) AS rmse
FROM
  ML.EVALUATE(MODEL feat_eng.model_6)
```

Out[ ]:
```
      rmse
0  5.906411
```

Code Clean Up

Exercise: Clean up the code to see where we are

Remove all the commented statements in the SQL statement. We should now have a total of five input features for our model.

1. fare_amount
2. passengers
3. day_hr
4. euclidean
5. pickup_and_dropoff

In [ ]:
```
%%bigquery

CREATE OR REPLACE MODEL
  feat_eng.model_6 OPTIONS (model_type='linear_reg',
    input_label_cols=['fare_amount']) AS
SELECT
  fare_amount,
  passengers,
# Use `ML.FEATURE_CROSS` function is a formed by multiplying two or more features.
  ML.FEATURE_CROSS(STRUCT(CAST(EXTRACT(DAYOFWEEK
        FROM
          pickup_datetime) AS STRING) AS dayofweek,
      CAST(EXTRACT(HOUR
        FROM
          pickup_datetime) AS STRING) AS hourofday)) AS day_hr,
  ST_Distance(ST_GeogPoint(pickuplon,
      pickuplat),
    ST_GeogPoint(dropofflon,
      dropofflat)) AS euclidean,
  CONCAT(ST_AsText(ST_SnapToGrid(ST_GeogPoint(pickuplon,
          pickuplat),
        0.01)), ST_AsText(ST_SnapToGrid(ST_GeogPoint(dropofflon,
          dropofflat),
        0.01))) AS pickup_and_dropoff
FROM
  `feat_eng.feateng_training_data`
```

Out[ ]: —

BQML's Pre-processing functions:

Here are some of the preprocessing functions in BigQuery ML:

- ML.FEATURE_CROSS(STRUCT(features)) does a feature cross of all the combinations
- ML.POLYNOMIAL_EXPAND(STRUCT(features), degree) creates x, x^2, x^3, etc.
- ML.BUCKETIZE(f, split_points) where split_points is an array

Model 7: Apply the BUCKETIZE Function

BUCKETIZE

Bucketize is a pre-processing function that creates "buckets" (e.g bins) - e.g. it bucketizes a continuous numerical feature into a string feature with bucket names as the value.

- ML.BUCKETIZE(feature, split_points)
- feature: A numerical column.
- split_points: Array of numerical points to split the continuous values in feature into buckets. With n split points (s1, s2 … sn), there will be n+1 buckets generated.
- Output: The function outputs a STRING for each row, which is the bucket name. bucket*name is in the format of bin*, where bucket_number starts from 1.
- Currently, our model uses the ST_GeogPoint function to derive the pickup and dropoff feature. In this lab, we use the BUCKETIZE function to create the pickup and dropoff feature.

Next, apply the BUCKETIZE function to model_7 and run the query.

In [ ]:
```
%%bigquery

# Use `BUCKETIZE` function
CREATE OR REPLACE MODEL
  feat_eng.model_7 OPTIONS (model_type='linear_reg',
    input_label_cols=['fare_amount']) AS
SELECT
  fare_amount,
  passengers,
  ST_Distance(ST_GeogPoint(pickuplon,
      pickuplat),
    ST_GeogPoint(dropofflon,
      dropofflat)) AS euclidean,
# Use `ML.FEATURE_CROSS` function is a formed by multiplying two or more features.
  ML.FEATURE_CROSS(STRUCT(CAST(EXTRACT(DAYOFWEEK
        FROM
          pickup_datetime) AS STRING) AS dayofweek,
      CAST(EXTRACT(HOUR
        FROM
          pickup_datetime) AS STRING) AS hourofday)) AS day_hr,
  CONCAT( ML.BUCKETIZE(pickuplon,
      GENERATE_ARRAY(-78, -70, 0.01)), ML.BUCKETIZE(pickuplat,
      GENERATE_ARRAY(37, 45, 0.01)), ML.BUCKETIZE(dropofflon,
      GENERATE_ARRAY(-78, -70, 0.01)), ML.BUCKETIZE(dropofflat,
      GENERATE_ARRAY(37, 45, 0.01)) ) AS pickup_and_dropoff
FROM
  `feat_eng.feateng_training_data`
```

Out[ ]: —

Next, we evaluate model_7.

In [ ]:

```
%%bigquery

# Use `ML.TRAINING_INFO` function which allows you to see information about the trainin
SELECT
  *,
  SQRT(loss) AS rmse
FROM
  ML.TRAINING_INFO(MODEL feat_eng.model_7)
```

Out[ ]:

	training_run	iteration	loss	eval_loss	learning_rate	duration_ms	rmse
0	0	1	20.798913	38.816795	0.4	4776	4.560583
1	0	0	71.421406	83.045042	0.2	3388	8.451119

In [ ]:

```
%%bigquery

# Use `ML.EVALUATE` function to evaluate model metrics.
SELECT
  *
FROM
  ML.EVALUATE(MODEL feat_eng.model_7)
```

Out[ ]:

	mean_absolute_error	mean_squared_error	mean_squared_log_error	median_absolute_error	r2_score
0	3.441163	38.816795	0.132281	2.554729	0.579521

In [ ]:

```
%%bigquery

# Use `ML.EVALUATE` function to evaluate model metrics.
SELECT
  SQRT(mean_squared_error) AS rmse
FROM
  ML.EVALUATE(MODEL feat_eng.model_7)
```

Out[ ]:

	rmse
0	6.230313

Final Model: Apply the TRANSFORM clause and L2 Regularization

Before we perform our prediction, we should encapsulate the entire feature set in a TRANSFORM clause. BigQuery ML now supports defining data transformations during model creation, which will be automatically applied during prediction and evaluation. This is done through the TRANSFORM clause in the existing CREATE MODEL statement. By using the TRANSFORM clause, user specified transforms during training will be automatically applied during model serving (prediction, evaluation, etc.)

In our case, we are using the TRANSFORM clause to separate out the raw input data from the TRANSFORMED features. The input columns of the TRANSFORM clause is the query_expr (AS SELECT part). The output columns of TRANSFORM from select_list are used in training. These transformed columns are post-processed with standardization for numerics and one-hot encoding for categorical variables by default.

The advantage of encapsulating features in the TRANSFORM clause is the client code doing the PREDICT doesn't change, e.g. our model improvement is transparent to client code. Note that the TRANSFORM clause MUST be placed after the CREATE statement.

L2 Regularization

Sometimes, the training RMSE is quite reasonable, but the evaluation RMSE illustrate more error. Given the severity of the delta between the EVALUATION RMSE and the TRAINING RMSE, it may be an indication of overfitting. When we do feature crosses, we run into the risk of overfitting (for example, when a particular day-hour combo doesn't have enough taxi rides).

Overfitting is a phenomenon that occurs when a machine learning or statistics model is tailored to a particular dataset and is unable to generalize to other datasets. This usually happens in complex models, like deep neural networks. Regularization is a process of introducing additional information in order to prevent overfitting.

Therefore, we will apply L2 Regularization to the final model. As a reminder, a regression model that uses the L1 regularization technique is called Lasso Regression while a regression model that uses the L2 Regularization technique is called Ridge Regression. The key difference between these two is the penalty term. Lasso shrinks the less important feature’s coefficient to zero, thus removing some features altogether. Ridge regression adds “squared magnitude” of coefficient as a penalty term to the loss function.

In other words, L1 limits the size of the coefficients. L1 can yield sparse models (i.e. models with few coefficients); Some coefficients can become zero and eliminated.

L2 regularization adds an L2 penalty equal to the square of the magnitude of coefficients. L2 will not yield sparse models and all coefficients are shrunk by the same factor (none are eliminated).

The regularization terms are ‘constraints’ by which an optimization algorithm must ‘adhere to’ when minimizing the loss function, apart from having to minimize the error between the true y and the predicted ŷ. This in turn reduces model complexity, making our model simpler. A simpler model can reduce the chances of overfitting.

Apply the TRANSFORM clause and L2 Regularization to the final model.

Tip: Make sure to delete the "# TODO 5" when you are about to run the cell.

In [ ]:

```
%%bigquery

# Use the `TRANSFORM` clause
CREATE OR REPLACE MODEL
  feat_eng.final_model
# TODO 5
    TRANSFORM(fare_amount,
    #SQRT( (pickuplon-dropofflon)*(pickuplon-dropofflon) + (pickuplat-dropofflat)*(pick
    ST_Distance(ST_GeogPoint(pickuplon,
        pickuplat),
      ST_GeogPoint(dropofflon,
        dropofflat)) AS euclidean,
# Use `ML.FEATURE_CROSS` function is a formed by multiplying two or more features.
    ML.FEATURE_CROSS(STRUCT(CAST(EXTRACT(DAYOFWEEK
          FROM
            pickup_datetime) AS STRING) AS dayofweek,
        CAST(EXTRACT(HOUR
          FROM
            pickup_datetime) AS STRING) AS hourofday)) AS day_hr,
    CONCAT( ML.BUCKETIZE(pickuplon,
        GENERATE_ARRAY(-78, -70, 0.01)), ML.BUCKETIZE(pickuplat,
        GENERATE_ARRAY(37, 45, 0.01)), ML.BUCKETIZE(dropofflon,
        GENERATE_ARRAY(-78, -70, 0.01)), ML.BUCKETIZE(dropofflat,
        GENERATE_ARRAY(37, 45, 0.01)) ) AS pickup_and_dropoff ) OPTIONS(input_label_col
    model_type='linear_reg',
    l2_reg=0.1) AS
SELECT
  *
FROM
  feat_eng.feateng_training_data
```

Out[ ]: —

Next, we evaluate the final model.

In [ ]:

```
%%bigquery

# Use `ML.TRAINING_INFO` function which allows you to see information about the trainin
SELECT
  *,
  SQRT(loss) AS rmse
FROM
  ML.TRAINING_INFO(MODEL feat_eng.final_model)
```

Out[ ]:

	training_run	iteration	loss	eval_loss	learning_rate	duration_ms	rmse
0	0	3	11.543481	29.058845	0.4	4625	3.397570
1	0	2	14.003867	31.897211	0.8	5065	3.742174
2	0	1	21.037874	32.961365	0.4	4679	4.586706
3	0	0	69.922209	76.298004	0.2	3311	8.361950

In [ ]:

```
%%bigquery

# Use `ML.EVALUATE` function to evaluate model metrics.
SELECT
```

```
  *
FROM
  ML.EVALUATE(MODEL feat_eng.final_model)
```

Out[ ]:

	mean_absolute_error	mean_squared_error	mean_squared_log_error	median_absolute_error	r2_score
0	2.623061	29.058845	0.086064	1.567397	0.673497

In [ ]:

```
%%bigquery

# Use `ML.EVALUATE` function to evaluate model metrics.
SELECT
  SQRT(mean_squared_error) AS rmse
FROM
  ML.EVALUATE(MODEL feat_eng.final_model)
```

Out[ ]:

	rmse
0	5.390626

Predictive Model

Now that you have evaluated your model, the next step is to use it to predict an outcome. You use your model to predict the taxifare amount. The ML.PREDICT function is used to predict results using your model: feat_eng.final_model.

Since this is a regression model (predicting a continuous numerical value), the best way to see how it performed is to evaluate the difference between the value predicted by the model and the benchmark score. We can do this with an ML.PREDICT query.

Apply the ML.PREDICT function.

In [ ]:

```
%%bigquery

# Use `ML.PREDICT` function to predict outcomes using the model.
SELECT
  *
FROM
  ML.PREDICT(MODEL feat_eng.final_model,
    (
    SELECT
      -73.982683 AS pickuplon,
      40.742104 AS pickuplat,
      -73.983766 AS dropofflon,
      40.755174 AS dropofflat,
      3.0 AS passengers,
      TIMESTAMP('2019-06-03 04:21:29.769443 UTC') AS pickup_datetime ))
```

Out[ ]:

predicted_fare_amount	pickuplon	pickuplat	dropofflon	dropofflat	passengers	pickup_dateti

	predicted_fare_amount	pickuplon	pickuplat	dropofflon	dropofflat	passengers	pickup_dateti
0	7.516129	-73.982683	40.742104	-73.983766	40.755174	3.0	2019-06- 04:21:29.769443+00

Lab Summary:

Our ML problem: Develop a model to predict taxi fare based on distance -- from one point to another in New York City.

OPTIONAL Exercise: Create a RMSE summary table.

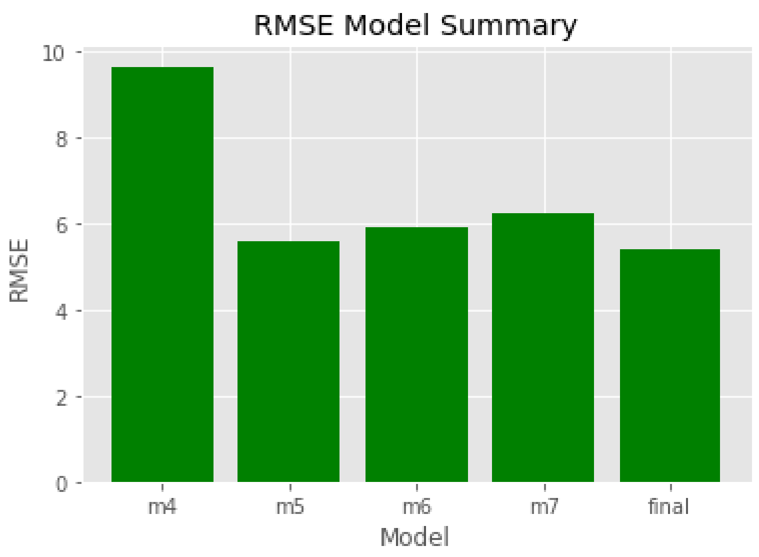
Create a RMSE summary table:

Solution Table

Model	Taxi Fare	Description
model_4	9.65	--Feature cross categorical features
model_5	5.58	--Create a Euclidian feature column
model_6	5.90	--Feature cross Geo-location features
model_7	6.23	--Apply the TRANSFORM Clause
final_model	5.39	--Apply L2 Regularization

RUN the cell to visualize a RMSE bar chart.

In [ ]:
```
# Visualize the RMSE chart
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('ggplot')

x = ['m4', 'm5', 'm6','m7', 'final']
RMSE = [9.65,5.58,5.90,6.23,5.39]

x_pos = [i for i, _ in enumerate(x)]

plt.bar(x_pos, RMSE, color='green')
plt.xlabel("Model")
plt.ylabel("RMSE")
plt.title("RMSE Model Summary")

plt.xticks(x_pos, x)

plt.show()
```


RMSE Model Summary
RMSE
Model
10
8
6
4
2
0
m4
m5
m6
m7
final