Hybrid Recommendations with the Movie Lens Dataset

Note: It is recommended that you complete the companion **als_bqml.ipynb** notebook before continuing with this **als_bqml_hybrid.ipynb** notebook. If you already have the movielens dataset and trained model you can skip the "Import the dataset and trained model" section.

Learning Objectives

1. Know extract user and product factors from a BigQuery Matrix Factorizarion Model
2. Know how to format inputs for a BigQuery Hybrid Recommendation Model

In [1]:
```
import os
import tensorflow as tf
PROJECT = "qwiklabs-gcp-02-618603922cbf" # REPLACE WITH YOUR PROJECT ID

# Do not change these
os.environ["PROJECT"] = PROJECT
os.environ["TFVERSION"] = '2.5'
```

Import the dataset and trained model

In the previous notebook, you imported 20 million movie recommendations and trained an ALS model with BigQuery ML

To save you the steps of having to do so again (if this is a new environment) you can run the below commands to copy over the clean data and trained model.

First create the BigQuery dataset and copy over the data

In [2]:
```
!bq mk movielens
```

```
BigQuery error in mk operation: Dataset 'qwiklabs-gcp-02-618603922cbf:movielens'
already exists.
```

Next, copy over the trained recommendation model. Note that if you're project is in the EU you will need to change the location from US to EU below. Note that as of the time of writing you cannot copy models across regions with `bq cp`.

In [7]:
```
%%bash
bq --location=US cp \
cloud-training-demos:movielens.recommender_16 \
movielens.recommender_16

bq --location=US cp \
```

```
cloud-training-demos:movielens.recommender_hybrid \
movielens.recommender_hybrid
```

```
Table 'cloud-training-demos:movielens.recommender_16' successfully copied to 'qw
iklabs-gcp-02-618603922cbf:movielens.recommender_16'
Table 'cloud-training-demos:movielens.recommender_hybrid' successfully copied to
'qwiklabs-gcp-02-618603922cbf:movielens.recommender_hybrid'
Waiting on bqjob_r70840d8f10e5306_0000017d00b6df14_1 ... (0s) Current status: DO
NE
Waiting on bqjob_r173b9b4e59e05efd_0000017d00b6e9d8_1 ... (0s) Current status: D
ONE
```

Next, ensure the model still works by invoking predictions for movie recommendations:

In [ ]:

```
%%bigquery --project $PROJECT
SELECT * FROM
ML.PREDICT(MODEL `movielens.recommender_16`, (
  SELECT
    movieId, title, 903 AS userId
  FROM movielens.movies, UNNEST(genres) g
  WHERE g = 'Comedy'
))
ORDER BY predicted_rating DESC
LIMIT 5
```

Incorporating user and movie information

The matrix factorization approach does not use any information about users or movies beyond what is available from the ratings matrix. However, we will often have user information (such as the city they live, their annual income, their annual expenditure, etc.) and we will almost always have more information about the products in our catalog. How do we incorporate this information in our recommendation model?

The answer lies in recognizing that the user factors and product factors that result from the matrix factorization approach end up being a concise representation of the information about users and products available from the ratings matrix. We can concatenate this information with other information we have available and train a regression model to predict the rating.

Obtaining user and product factors

We can get the user factors or product factors from ML.WEIGHTS. For example to get the product factors for movieId=96481 and user factors for userId=54192, we would do:

In [18]:

```
%%bigquery --project $PROJECT
SELECT
    processed_input,
    feature,
    TO_JSON_STRING(factor_weights) AS factor_weights,
    intercept
FROM ML.WEIGHTS(MODEL `movielens.recommender_16`)
WHERE
    (processed_input = 'movieId' AND feature = '96481')
    OR (processed_input = 'userId' AND feature = '54192')
```

Out[18]:

	processed_input	feature	factor_weights	intercept
0	movieId	96481	[{"factor":16,"weight":4.7488055154865094e-16}...	-1.398794
1	userId	54192	[{"factor":16,"weight":1.451774918706243},{"fa...	1.839885

Multiplying these weights and adding the intercept is how we get the predicted rating for this combination of movieId and userId in the matrix factorization approach.

These weights also serve as a low-dimensional representation of the movie and user behavior. We can create a regression model to predict the rating given the user factors, product factors, and any other information we know about our users and products.

Creating input features

The MovieLens dataset does not have any user information, and has very little information about the movies themselves. To illustrate the concept, therefore, let's create some synthetic information about users:

In [19]:

```
%%bigquery --project $PROJECT
CREATE OR REPLACE TABLE movielens.users AS
SELECT
    userId,
    RAND() * COUNT(rating) AS loyalty,
    CONCAT(SUBSTR(CAST(userId AS STRING), 0, 2)) AS postcode
FROM
  movielens.ratings
GROUP BY userId
```

Out[19]: —

Input features about users can be obtained by joining the user table with the ML weights and selecting all the user information and the user factors from the weights array.

In [20]:

```
%%bigquery --project $PROJECT
WITH userFeatures AS (
  SELECT
     u.*,
     (SELECT ARRAY_AGG(weight) FROM UNNEST(factor_weights)) AS user_factors
  FROM movielens.users u
  JOIN ML.WEIGHTS(MODEL movielens.recommender_16) w
  ON processed_input = 'userId' AND feature = CAST(u.userId AS STRING)
)

SELECT * FROM userFeatures
LIMIT 5
```

Out[20]:

	userId	loyalty	postcode	user_factors
0	65536	12.023909	65	[5.2869248538055835, 0.5717508872411037, -8.53...
1	131072	18.668811	13	[9.205648021521588, 4.480600489131113, -6.9878...
2	256	5.854007	25	[-6.81069854746253, 2.445484088458461, -0.1672...

	userId	loyalty	postcode	user_factors
3	65792	37.295750	65	[-2.4468382816400025, 4.920656619448598, -5.54...
4	131328	28.484759	13	[-6.25690712503519, -3.8063811222966732, -1.11...

Similarly, we can get product features for the movies data, except that we have to decide how to handle the genre since a movie could have more than one genre. If we decide to create a separate training row for each genre, then we can construct the product features using.

In [21]:
```
%%bigquery --project $PROJECT
WITH productFeatures AS (
  SELECT
      p.* EXCEPT(genres),
      g, (SELECT ARRAY_AGG(weight) FROM UNNEST(factor_weights))
            AS product_factors
  FROM movielens.movies p, UNNEST(genres) g
  JOIN ML.WEIGHTS(MODEL movielens.recommender_16) w
  ON processed_input = 'movieId' AND feature = CAST(p.movieId AS STRING)
)

SELECT * FROM productFeatures
LIMIT 5
```

Out[21]:

	movieId	title	g	product_factors
0	6441	Battle of Britain (1969)	War	[-0.05378647448089614, 0.12521521613203146, 0....
1	117750	The Real Glory (1939)	War	[4.996003610813204e-16, -7.112366251504901e-16...
2	100574	Raid on Rommel (1971)	War	[-0.04352516797793908, 0.028588098837068587, -...
3	1450	Prisoner of the Mountains (Kavkazsky plennik) ...	War	[-0.0068831575882775786, -0.03107592153404104,...
4	118296	The Diary of an Unknown Soldier (1959)	War	[-2.4286128663675334e-16, 4.163336342344337e-1...

Combining these two WITH clauses and pulling in the rating corresponding the movieId-userId combination (if it exists in the ratings table), we can create the training dataset.

TODO 1: Combine the above two queries to get the user factors and product factor for each rating.

NOTE: The below cell will take approximately 4~5 minutes for the completion.

In [22]:
```
%%bigquery --project $PROJECT
CREATE OR REPLACE TABLE movielens.hybrid_dataset AS

    WITH userFeatures AS (
      SELECT
         u.*,
         (SELECT ARRAY_AGG(weight) FROM UNNEST(factor_weights))
            AS user_factors
      FROM movielens.users u
```

```
  JOIN ML.WEIGHTS(MODEL movielens.recommender_16) w
  ON processed_input = 'userId' AND feature = CAST(u.userId AS STRING)
),

productFeatures AS (
  SELECT
      p.* EXCEPT(genres),
      g, (SELECT ARRAY_AGG(weight) FROM UNNEST(factor_weights))
        AS product_factors
  FROM movielens.movies p, UNNEST(genres) g
  JOIN ML.WEIGHTS(MODEL movielens.recommender_16) w
  ON processed_input = 'movieId' AND feature = CAST(p.movieId AS STRING)
)

SELECT
    p.* EXCEPT(movieId),
    u.* EXCEPT(userId),
    rating
FROM productFeatures p, userFeatures u
JOIN movielens.ratings r
ON r.movieId = p.movieId AND r.userId = u.userId
```

Out[22]: —

One of the rows of this table looks like this:

In [23]:
```
%%bigquery --project $PROJECT
SELECT *
FROM movielens.hybrid_dataset
LIMIT 1
```

Out[23]:

	title	g	product_factors	loyalty	postcode	user_factors	rating
0	Traffic (2000)	Crime	[0.050302306705910786, 0.026067456450144123, 0...	799.017058	29	[5.420790935241976, 2.2204505040712856, -4.530...	2.0

Essentially, we have a couple of attributes about the movie, the product factors array corresponding to the movie, a couple of attributes about the user, and the user factors array corresponding to the user. These form the inputs to our "hybrid" recommendations model that builds off the matrix factorization model and adds in metadata about users and movies.

Training hybrid recommendation model

At the time of writing, BigQuery ML can not handle arrays as inputs to a regression model. Let's, therefore, define a function to convert arrays to a struct where the array elements are its fields:

In [24]:
```
%%bigquery --project $PROJECT
CREATE OR REPLACE FUNCTION movielens.arr_to_input_16_users(u ARRAY<FLOAT64>)
RETURNS
    STRUCT<
        u1 FLOAT64,
        u2 FLOAT64,
        u3 FLOAT64,
        u4 FLOAT64,
```

```
        u5 FLOAT64,
        u6 FLOAT64,
        u7 FLOAT64,
        u8 FLOAT64,
        u9 FLOAT64,
        u10 FLOAT64,
        u11 FLOAT64,
        u12 FLOAT64,
        u13 FLOAT64,
        u14 FLOAT64,
        u15 FLOAT64,
        u16 FLOAT64
    > AS (STRUCT(
        u[OFFSET(0)],
        u[OFFSET(1)],
        u[OFFSET(2)],
        u[OFFSET(3)],
        u[OFFSET(4)],
        u[OFFSET(5)],
        u[OFFSET(6)],
        u[OFFSET(7)],
        u[OFFSET(8)],
        u[OFFSET(9)],
        u[OFFSET(10)],
        u[OFFSET(11)],
        u[OFFSET(12)],
        u[OFFSET(13)],
        u[OFFSET(14)],
        u[OFFSET(15)]
));
```

Out[24]: —

which gives:

In [25]:
```
%%bigquery --project $PROJECT
SELECT movielens.arr_to_input_16_users(u).*
FROM (SELECT
    [0., 1., 2., 3., 4., 5., 6., 7., 8., 9., 10., 11., 12., 13., 14., 15.] AS u)
```

Out[25]:

	u1	u2	u3	u4	u5	u6	u7	u8	u9	u10	u11	u12	u13	u14	u15	u16
0	0.0	1.0	2.0	3.0	4.0	5.0	6.0	7.0	8.0	9.0	10.0	11.0	12.0	13.0	14.0	15.0

We can create a similar function named movielens.arr_to_input_16_products to convert the product factor array into named columns.

TODO 2: Create a function that returns named columns from a size 16 product factor array.

In [26]:
```
%%bigquery --project $PROJECT
CREATE OR REPLACE FUNCTION movielens.arr_to_input_16_products(p ARRAY<FLOAT64>)
RETURNS
    STRUCT<
        p1 FLOAT64,
        p2 FLOAT64,
        p3 FLOAT64,
        p4 FLOAT64,
```

```
        p5 FLOAT64,
        p6 FLOAT64,
        p7 FLOAT64,
        p8 FLOAT64,
        p9 FLOAT64,
        p10 FLOAT64,
        p11 FLOAT64,
        p12 FLOAT64,
        p13 FLOAT64,
        p14 FLOAT64,
        p15 FLOAT64,
        p16 FLOAT64
    > AS (STRUCT(
        p[OFFSET(0)],
        p[OFFSET(1)],
        p[OFFSET(2)],
        p[OFFSET(3)],
        p[OFFSET(4)],
        p[OFFSET(5)],
        p[OFFSET(6)],
        p[OFFSET(7)],
        p[OFFSET(8)],
        p[OFFSET(9)],
        p[OFFSET(10)],
        p[OFFSET(11)],
        p[OFFSET(12)],
        p[OFFSET(13)],
        p[OFFSET(14)],
        p[OFFSET(15)]
));
```

Out[26]: —

Then, we can tie together metadata about users and products with the user factors and product factors obtained from the matrix factorization approach to create a regression model to predict the rating:

NOTE: The below cell will take approximately 25~30 minutes for the completion.

In [ ]:

```
%%bigquery --project $PROJECT
CREATE OR REPLACE MODEL movielens.recommender_hybrid
OPTIONS(model_type='linear_reg', input_label_cols=['rating'])
AS

SELECT
  * EXCEPT(user_factors, product_factors),
    movielens.arr_to_input_16_users(user_factors).*,
    movielens.arr_to_input_16_products(product_factors).*
FROM
  movielens.hybrid_dataset
```

```
Executing query with job ID: 3ccc5208-b63e-479e-980f-2e472e0d65ba
Query executing: 1327.21s
```

There is no point looking at the evaluation metrics of this model because the user information we used to create the training dataset was fake (not the RAND() in the creation of the loyalty column) -- we did this exercise in order to demonstrate how it could be done. And of course, we could train a dnn_regressor model and optimize the hyperparameters if we want a more

sophisticated model. But if we are going to go that far, it might be better to consider using Auto ML tables.

In [ ]: