Exercise1: Time Series Forecasting with BigQuery ML

Objective: Learn to perform time series forecasting using BigQuery ML. This exercise involves visualizing data in Looker Studio, creating a time series model using the ARIMA_PLUS algorithm in BigQuery ML, and understanding how to interpret the results.

Preparation: Create a new Dataset in BiogQuery:

- **Dataset Creation**:
  - Name: `bqml`
  - Region: US

1.1 Visualize Data in Looker Studio:

SQL Query:

```
SELECT
  PARSE_TIMESTAMP("%Y%m%d", date) AS parsed_date,
  SUM(totals.visits) AS total_visits
FROM
  `bigquery-public-data.google_analytics_sample.ga_sessions_*`
GROUP BY date
```

- **Explanation**:
  - `PARSE_TIMESTAMP("%Y%m%d", date)`: Converts the date from a string in the format "YYYYMMDD" to a timestamp.
  - `SUM(totals.visits)`: Aggregates the total number of website visits for each date.
  - `GROUP BY date`: Organizes the data by each unique date.

Visualization Task: Create a line chart in Looker Studio using `parsed_date` as the x-axis and `total_visits` as the y-axis to visualize the trend of website visits over time.

1.2 Visualize Forecasted Values in Looker Studio (Without Decomposition):

Train the model:

```
CREATE OR REPLACE MODEL `bqml.ga_arima_model`
OPTIONS
  (model_type = 'ARIMA_PLUS',
```

```
    time_series_timestamp_col = 'parsed_date',
    time_series_data_col = 'total_visits',
    auto_arima = TRUE,
    data_frequency = 'AUTO_FREQUENCY',
    decompose_time_series = TRUE
  ) AS
SELECT
  PARSE_TIMESTAMP("%Y%m%d", date) AS parsed_date,
  SUM(totals.visits) AS total_visits
FROM
  `bigquery-public-data.google_analytics_sample.ga_sessions_*`
GROUP BY date
```

SQL Query with Comments:

```
-- Selecting historical and forecasted data for visualization
SELECT
  history_timestamp AS timestamp,  -- Historical timestamps
  history_value,                   -- Historical data values
  NULL AS forecast_value,          -- Placeholder for forecast values (NULL
   ↪  for historical data)
  NULL AS prediction_interval_lower_bound, -- Placeholder for lower bound of
   ↪  prediction interval
  NULL AS prediction_interval_upper_bound  -- Placeholder for upper bound of
   ↪  prediction interval
FROM
  (
    -- Subquery to get historical data
    SELECT
      PARSE_TIMESTAMP("%Y%m%d", date) AS history_timestamp, -- Parsing date
 ↪  to timestamp
      SUM(totals.visits) AS history_value                  -- Summing total
       ↪  visits per day
    FROM
      `bigquery-public-data.google_analytics_sample.ga_sessions_*` -- From
 ↪  public analytics data
    GROUP BY date
    ORDER BY date ASC  -- Ordered chronologically
  )
UNION ALL
-- Adding forecasted data to the selection
SELECT
  forecast_timestamp AS timestamp,   -- Forecasted timestamps
```

```
  NULL AS history_value,             -- Placeholder for historical data (NULL
   ↪  for forecasted data)
  forecast_value,                     -- Forecasted data values
  prediction_interval_lower_bound,    -- Lower bound of prediction interval
  prediction_interval_upper_bound     -- Upper bound of prediction interval
FROM
  ML.FORECAST(MODEL `bqml.ga_arima_model`, -- Using the ARIMA model for
 ↪  forecasting
              STRUCT(30 AS horizon, 0.8 AS confidence_level)) -- Forecasting
 ↪  parameters
```

Detailed Explanation:

- The query's main purpose is to combine historical data with forecasted data for visualization. This combination allows for a seamless transition from observed to predicted values in the chart.
- **First Part of the Query (Historical Data):**
  - `PARSE_TIMESTAMP("%Y%m%d", date) AS history_timestamp`: Converts dates from string format to timestamp format. This standardization is essential for time series analysis, ensuring consistency in date and time representation.
  - `SUM(totals.visits) AS history_value`: Sums the total number of visits for each date, representing the actual observed data.
  - The data is grouped by date and ordered in ascending order, providing a chronological sequence of historical data points.
- **Second Part of the Query (Forecasted Data):**
  - This part uses the `ML.FORECAST` function to generate future values based on the `ga_arima_model`.
  - `forecast_timestamp AS timestamp`: Represents the timestamp for each forecasted point.
  - `forecast_value`: The predicted number of visits for the forecasted dates.
  - `prediction_interval_lower_bound` and `prediction_interval_upper_bound`: These columns provide the lower and upper bounds of the prediction interval, indicating the range within which the actual values are expected to fall with a given confidence level (80% in this case).
- **UNION ALL**: This operator is used to append the forecasted data to the historical data. The structure of both parts of the query is aligned so that historical and forecasted values can be combined seamlessly.

- **Visualization Task**:
  - Create a line chart in Looker Studio.
  - Use the `timestamp` column for the x-axis.
  - Display the following columns as different series in the chart:
    * `history_value`: Represents the historical data points.
    * `forecast_value`: Shows the predicted values for future dates.
    * `prediction_interval_lower_bound` and `prediction_interval_upper_bound`: Illustrate the range of the prediction intervals, offering a visual representation of the forecast's uncertainty.
  - This chart will provide a clear visual distinction between historical data and forecasted values, along with the confidence intervals, enabling a comprehensive understanding of past trends and future predictions.

1.3 Visualize Forecasted Values in Looker Studio (With Decomposition):

SQL Query:

```
#standardSQL
SELECT
  *
FROM
  ML.EXPLAIN_FORECAST(MODEL `bqml_tutorial.ga_arima_model`,
                      STRUCT(30 AS horizon, 0.8 AS confidence_level))
```

- **Visualization Task**:
  - Create a line chart in Looker Studio.
  - In the **Date Range Dimension** section, select `time_series_timestamp` (Date).
  - In the **Dimension** section, select `time_series_timestamp` (Date).
  - In the **Metric** section, remove the default metric `Record Count`, and add:
    * `time_series_data`
    * `prediction_interval_lower_bound`
    * `prediction_interval_upper_bound`
    * `trend`
    * `seasonal_period_weekly`
    * `step_changes`
  - This allows you to visualize the forecasted values and their components, like trends and seasonality.

Exercise 2: Classification with BigQuery ML

A common problem in machine learning is to classify data into one of two types, known as labels. For example, a retailer may want to predict whether a given customer will purchase a new product, based on other information about that customer. In that case, the two labels might be "will buy" and "won't buy." You can construct your dataset such that one column represents the label. The data you can use to train such a binary logistic regression model include the customer's location, their previous purchases, the customer's reported preferences, and so on.

2.1 Inspect the Data:

SQL Query:

```
SELECT
  age,
  workclass,
  marital_status,
  education_num,
  occupation,
  hours_per_week,
  income_bracket
FROM
  `bigquery-public-data.ml_datasets.census_adult_income`
LIMIT
  100;
```

- **Observations**:
  - The `income_bracket` column has two values: `<=50K` or `>50K`.
  - The `education` and `education_num` columns represent the same data differently.
  - The `functional_weight` column's values appear unrelated to `income_bracket`.

The query results show that the income_bracket column in the census_adult_income table has only one of two values: <=50K or >50K. It also shows that the columns education and education_num in the census_adult_income table express the same data in different formats. The functional_weight column is the number of individuals that the Census Organizations believes a particular row represents; the values of this column appear unrelated to the value of income_bracket for a particular row.

2.2 Select Training and Evaluation/Testing Data:

Next, you select the data used to train your logistic regression model. In this tutorial, you predict census respondent income based on the following attributes:

```
CREATE OR REPLACE VIEW
  `bqml.input_view` AS
SELECT
  age,
  workclass,
  marital_status,
  education_num,
  occupation,
  hours_per_week,
  income_bracket,
  CASE
    WHEN MOD(functional_weight, 10) < 8 THEN 'training'
    WHEN MOD(functional_weight, 10) = 8 THEN 'evaluation'
    WHEN MOD(functional_weight, 10) = 9 THEN 'prediction'
  END AS dataframe
FROM
  `bigquery-public-data.ml_datasets.census_adult_income`
```

- **Attributes for Prediction**:
  - Age
  - Type of work performed
  - Marital status
  - Level of education
  - Occupation
  - Hours worked per week

2.3 Create a Logistic Regression Model:

You can create and train a logistic regression model using the CREATE MODEL statement with the option 'LOGISTIC_REG'. The following query uses a CREATE MODEL statement to train a new binary logistic regression model on the view from the previous query.

SQL Query: `sql CREATE OR REPLACE MODEL `bqml.census_model` OPTIONS ( model_type='LOGISTIC_REG', auto_class_weights=TRUE, input_label_cols=['income_bracket'] ) AS SELECT * EXCEPT(dataframe) FROM `bqml.input_view` WHERE dataframe = 'training'`

Query Details: - The `CREATE MODEL` statement trains a logistic regression model. - `LOGISTIC_REG` specifies the model type. - `auto_class_weights=TRUE` balances class labels in training data.

The CREATE MODEL statement trains a model using the training data in the SELECT statement.

The OPTIONS clause specifies the model type and training options. Here, the LOGISTIC_REG option specifies a logistic regression model type. It is not necessary to specify a binary logistic regression model versus a multiclass logistic regression model: BigQuery can determine which to train based on the number of unique values in the label column.

The input_label_cols option specifies which column in the SELECT statement to use as the label column. Here, the label column is income_bracket, so the model will learn which of the two values of income_bracket is most likely based on the other values present in each row.

The 'auto_class_weights=TRUE' option balances the class labels in the training data. By default, the training data is unweighted. If the labels in the training data are imbalanced, the model may learn to predict the most popular class of labels more heavily. In this case, most of the respondents in the dataset are in the lower income bracket. This may lead to a model that predicts the lower income bracket too heavily. Class weights balance the class labels by calculating the weights for each class in inverse proportion to the frequency of that class.

The SELECT statement queries the view from Step 2. This view contains only the columns containing feature data for training the model. The WHERE clause filters the rows in input_view so that only those rows belonging to the training dataframe are included in the training data.

2.4 Evaluate the Model:

After training your model, you evaluate the performance of the classifier using the ML.EVALUATE function. The ML.EVALUATE function evaluates the predicted values against the actual data.

SQL Query: `sql SELECT * FROM ML.EVALUATE (MODEL `bqml.census_model`, ( SELECT * FROM `bqml.input_view` WHERE dataframe = 'evaluation' ) )`

Query details The ML.EVALUATE function takes the model trained in Step 3 and evaluation data returned by a SELECT subquery. The function returns a single row of statistics about the model. This query uses data from input_view as evaluation data. The WHERE clause filters the input data so that the subquery contains only rows in the evaluation dataframe.

Because you performed a logistic regression, the results include the following columns: - **Evaluation Metrics**: - Precision - Recall - Accuracy - F1 score - Log loss - ROC AUC

1. Precision

- **What it Measures**: Precision measures how accurate your model's predictions are. In other words, of all the instances the model predicted as positive (e.g., predicting that a customer will buy a product), how many were actually positive.
- **Analogy**: Imagine you have a basket of fruits, and your task is to pick out all the apples. Precision would be the proportion of actual apples in the fruits you picked, assuming you might have mistakenly picked some oranges too.

2. Recall

- **What it Measures**: Recall measures the ability of your model to find all the relevant cases within a dataset. For the positive cases, it's the proportion that your model correctly identified.
- **Analogy**: Continuing with the fruit basket analogy, recall would be the proportion of apples you successfully picked out of all the apples in the basket, regardless of whether you picked some oranges too.

3. Accuracy

- **What it Measures**: Accuracy measures the proportion of predictions your model got right – both positive and negative predictions.
- **Analogy**: If you have to sort apples and oranges, accuracy tells you the proportion of fruits you classified correctly as either apples or oranges.

4. F1 Score

- **What it Measures**: The F1 Score is a balance between Precision and Recall. It's a way to combine both metrics into a single number. This is especially useful if you need a balance between finding as many positives as possible (Recall) and ensuring those positives are accurate (Precision).
- **Analogy**: Think of F1 Score as a grading system that considers both your ability to pick apples (precision) and your ability not to miss any apples (recall).

5. Log Loss

- **What it Measures**: Log loss measures the uncertainty of your model's predictions. It penalizes false classifications by considering how far off the predictions are, not just whether they're right

or wrong.

- **Analogy**: Imagine you're guessing the weight of suitcases. Log loss doesn't just care if you guessed them as heavy or light, but it also considers how far off your guess was from their actual weight.

6. ROC AUC

- **ROC (Receiver Operating Characteristic) AUC (Area Under the Curve)**: This metric measures the ability of a model to distinguish between classes. A higher AUC means the model is better at predicting 0s as 0s and 1s as 1s.
- **Analogy**: If you were betting on horses, ROC AUC would measure how good you are at predicting the winning horse versus the losing horse. A higher score means you're better at distinguishing winners from losers.

In summary, these metrics are tools to evaluate how well your ML model performs, each focusing on a different aspect of its predictions. Precision and recall focus on positive predictions, accuracy looks at overall correctness, F1 Score combines precision and recall, log loss penalizes confident wrong predictions, and ROC AUC measures the model's ability to distinguish between classes.

2.5 Make Predictions:

After you train your model, you can use it to predict the income bracket of a person based on the values of the other columns in the row. The ML.PREDICT function returns the predicted value for each row in the input table.

To identify the income bracket to which a particular respondent belongs, use the ML.PREDICT function. The following query predicts the income bracket of every respondent in the prediction dataframe.

SQL Query:

```
SELECT
  *
FROM
  ML.PREDICT (MODEL `bqml.census_model`,
    (
    SELECT
      *
    FROM
      `bqml.input_view`
    WHERE
      dataframe = 'prediction'
```

```
    )
  )
```

Query details The ML.PREDICT function predicts results using your model and the data from input_view, filtered to include only rows in the 'prediction' dataframe. The top-most SELECT statement retrieves the output of the ML.PREDICT function.

2.6 Explain Predictions with AI Methods:

To understand why your model is generating these prediction results, you can use the ML.EXPLAIN_PREDICT function.

ML.EXPLAIN_PREDICT is an extended version of ML.PREDICT. ML.EXPLAIN_PREDICT not only outputs prediction results, but also outputs additional columns to explain the prediction results. So in practice, you only need to run ML.EXPLAIN_PREDICT while skipping running ML.PREDICT.

SQL Query:

```
SELECT
  *
FROM
  ML.EXPLAIN_PREDICT(MODEL `bqml.census_model`,
    (
    SELECT
      *
    FROM
      `bqml.input_view`
    WHERE
      dataframe = 'evaluation'),
    STRUCT(3 as top_k_features))
```

2.7 Global Model Explanation:

To know which features are the most important to determine the income bracket in general, you can use the ML.GLOBAL_EXPLAIN function. In order to use ML.GLOBAL_EXPLAIN, the model must be retrained with the option ENABLE_GLOBAL_EXPLAIN=TRUE. Rerun the training query with this option using the following query:

SQL Query for Retraining:

```
CREATE OR REPLACE MODEL `bqml.census_model`
OPTIONS
  ( model_type='LOGISTIC_REG',
    auto_class_weights=TRUE,
```

```
    enable_global_explain=TRUE,
    input_label_cols=['income_bracket']
  ) AS
SELECT
  * EXCEPT(dataframe)
FROM
  `bqml.input_view`
WHERE
  dataframe = 'training'
```

SQL Query for Global Explanation:

```
SELECT
  *
FROM
  ML.GLOBAL_EXPLAIN(MODEL `bqml.census_model`)
```