Cloud Data Engineering Lab Solutions (Questions 21-35)

Analyzed by Gemini

November 14, 2025

Q21: Databricks - Linear Regression

Lab Task: Train a Linear Regression model on housing price data in Databricks using Spark MLlib and evaluate RMSE.

Conceptual Basis

This task involves the standard MLlib pipeline. **Linear Regression** attempts to find a linear relationship between input features (e.g., square footage, bedrooms) and a continuous-valued label (e.g., price). The **VectorAssembler** is a crucial transformer that combines multiple feature columns into a single `features` vector, which is the format MLlib models require. **RMSE** (Root Mean Squared Error) is the standard metric for evaluating regression models.

Step-by-Step Solution (PySpark)

```python
from pyspark.ml.regression import LinearRegression
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.evaluation import RegressionEvaluator

# 1. Load Data
data = spark.read.csv("/path/to/housing.csv", header=True, inferSchema=True)

# 2. Assemble Features
# Combine all feature columns into one 'features' vector
assembler = VectorAssembler(
    inputCols=["sq_footage", "bedrooms", "bathrooms"],
    outputCol="features"
)
df_vector = assembler.transform(data)

# 3. Split Data
(train, test) = df_vector.randomSplit([0.8, 0.2])

# 4. Define and Train Model
lr = LinearRegression(featuresCol="features", labelCol="price")
lr_model = lr.fit(train)

# 5. Make Predictions
predictions = lr_model.transform(test)

# 6. Evaluate Model (RMSE)
evaluator = RegressionEvaluator(
    labelCol="price",
    predictionCol="prediction",
    metricName="rmse"
)
rmse = evaluator.evaluate(predictions)
print(f"Root Mean Squared Error (RMSE) on test data: {rmse}")
```

Listing 1: PySpark Linear Regression

Q22: Databricks - Logistic Regression with Scaling

Lab Task: Build a Logistic Regression model to predict bank loan approval using Vector Assembler and StandardScaler.

Conceptual Basis

Logistic Regression is a model for binary classification (e.g., approve/deny). It's sensitive to the scale of input features. **StandardScaler** is a feature transformer that scales each feature to have a unit standard deviation and/or zero mean. This prevents features with large ranges (like `Salary`) from dominating features with small ranges (like `Years_Employed`), leading to a more accurate model.

Step-by-Step Solution (PySpark)

```
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import VectorAssembler, StandardScaler

# 1. Load Data
data = spark.read.csv("/path/to/bank_loans.csv", header=True, inferSchema=True)

# 2. Assemble Features
assembler = VectorAssembler(
    inputCols=["income", "credit_score", "age"],
    outputCol="raw_features"
)
df_vector = assembler.transform(data)

# 3. Scale Features
# StandardScaler scales the 'raw_features' vector
scaler = StandardScaler(
    inputCol="raw_features",
    outputCol="scaled_features",
    withStd=True,
    withMean=True
)
scaler_model = scaler.fit(df_vector)
df_scaled = scaler_model.transform(df_vector)

# 4. Define and Train Model
# Note: We use 'scaled_features' as input
lr = LogisticRegression(featuresCol="scaled_features", labelCol="loan_approved")
lr_model = lr.fit(df_scaled)

print("Model trained successfully.")
```

Listing 2: PySpark Logistic Regression with StandardScaler

Q23: Databricks - Random Forest Classifier

Lab Task: Train a Random Forest Classifier on the Iris dataset for multiclass classification and evaluate F1-score.

Conceptual Basis

Random Forest is a powerful ensemble model that's effective for multiclass classification (like the Iris dataset, which has 3 species). It works by building many decision trees and averaging their predictions. The **F1-score** is a robust metric for classification that balances Precision and Recall, making it ideal for evaluating model performance.

Step-by-Step Solution (PySpark)

```
from pyspark.ml.classification import RandomForestClassifier
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
```

```
# 1. Load and Assemble (Iris data)
# Assume data is loaded with label (species) and features
assembler = VectorAssembler(
    inputCols=["sepal_length", "sepal_width", "petal_length", "petal_width"],
    outputCol="features"
)
df_vector = assembler.transform(iris_data)

# 2. Split Data
(train, test) = df_vector.randomSplit([0.7, 0.3])

# 3. Define and Train Model
rf = RandomForestClassifier(featuresCol="features", labelCol="species_index")
rf_model = rf.fit(train)

# 4. Make Predictions
predictions = rf_model.transform(test)

# 5. Evaluate Model (F1-Score)
evaluator = MulticlassClassificationEvaluator(
    labelCol="species_index",
    predictionCol="prediction",
    metricName="f1"  -- This is the key step
)
f1_score = evaluator.evaluate(predictions)
print(f"F1-Score: {f1_score}")
```

Listing 3: PySpark Random Forest and F1-Score

Q24: Databricks - Polynomial Regression

Lab Task: Implement Polynomial Regression on a dataset and compare results with Linear Regression in Databricks MLlib.

Conceptual Basis

Polynomial Regression is not a new model type, but rather a feature engineering technique. It creates non-linear relationships by adding polynomial terms (e.g., x^2, x^3) of the original features. You then fit a standard **Linear Regression** model to this expanded, non-linear feature set. This is done in MLlib using the **PolynomialExpansion** transformer.

Step-by-Step Solution (PySpark)

```
from pyspark.ml.feature import PolynomialExpansion
from pyspark.ml.regression import LinearRegression

# 1. Assemble Features (e.g., 'x')
assembler = VectorAssembler(inputCols=["x"], outputCol="features")
df_vector = assembler.transform(data)

# 2. Create Polynomial Features
poly = PolynomialExpansion(degree=3, inputCol="features", outputCol="poly_features")
df_poly = poly.transform(df_vector)

# df_poly now has 'features' (e.g., [x])
# and 'poly_features' (e.g., [x, x^2, x^3])

# 3. Train Polynomial Model
lr_poly = LinearRegression(featuresCol="poly_features", labelCol="y")
poly_model = lr_poly.fit(df_poly)

# 4. Train Linear Model (for comparison)
lr_linear = LinearRegression(featuresCol="features", labelCol="y")
linear_model = lr_linear.fit(df_poly) # Can use same df

# 5. Compare RMSE on test set (not shown for brevity)
```

```
# The poly_model will have a lower RMSE if the data has a curve.
```

Listing 4: PySpark Polynomial Regression

Q25: Databricks - Decision Tree Classifier

Lab Task: Perform Binary Classification on spam detection using DecisionTreeClassifier in Databricks.

Conceptual Basis

A **Decision Tree** is an intuitive model that learns a set of "if-then" rules to make a prediction. It's highly interpretable (you can visualize the tree) and works well for binary classification tasks like spam detection. The task assumes text features (e.g., word counts) have already been extracted and assembled into a `features` vector.

Step-by-Step Solution (PySpark)

```
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.evaluation import MulticlassClassificationEvaluator

# 1. Load Data
# Assume data is loaded with 'label' (0=not spam, 1=spam)
# and features (e.g., 'word_count', 'capital_count')
assembler = VectorAssembler(
    inputCols=["word_count", "capital_count", "special_char_count"],
    outputCol="features"
)
df_vector = assembler.transform(spam_data)

# 2. Split Data
(train, test) = df_vector.randomSplit([0.7, 0.3])

# 3. Define and Train Model
dt = DecisionTreeClassifier(featuresCol="features", labelCol="label")
dt_model = dt.fit(train)

# 4. Make Predictions
predictions = dt_model.transform(test)

# 5. Evaluate (e.g., Accuracy)
evaluator = MulticlassClassificationEvaluator(
    labelCol="label",
    predictionCol="prediction",
    metricName="accuracy"
)
accuracy = evaluator.evaluate(predictions)
print(f"Accuracy: {accuracy}")
```

Listing 5: PySpark Decision Tree for Spam Detection

Q26: Synapse - Create External Table

Lab Task: Create an external table in Synapse referencing data stored in ADLS Gen2. Query the table and validate schema.

Conceptual Basis

An **External Table** is a T-SQL concept (used in Serverless SQL Pools) that creates a permanent metadata pointer to data files in a data lake. Unlike `OPENROWSET`, which is ad-hoc, an external table makes it look and feel like a real SQL table. It requires three components: an `EXTERNAL DATA SOURCE` (the "where"), an `EXTERNAL FILE FORMAT` (the "what"), and the `EXTERNAL TABLE` (the "schema").

Step-by-Step Solution (T-SQL)

```
-- 1. Create the 'where' (the path to your data lake)
CREATE EXTERNAL DATA SOURCE MyDataLake
WITH (
    LOCATION = 'https://<storage_acct>.dfs.core.windows.net/<container>/'
);

-- 2. Create the 'what' (the file format)
CREATE EXTERNAL FILE FORMAT CsvFormat
WITH (
    FORMAT_TYPE = DELIMITEDTEXT,
    FORMAT_OPTIONS (
        FIELD_TERMINATOR = ',',
        FIRST_ROW = 2 -- Assumes header row
    )
);

-- 3. Create the 'schema' (the table itself)
CREATE EXTERNAL TABLE dbo.SalesData (
    TransactionID int,
    ProductID int,
    Amount float,
    TransactionDate date
)
WITH (
    LOCATION = 'path/to/sales_files/', -- Relative to the DATA_SOURCE
    DATA_SOURCE = MyDataLake,
    FILE_FORMAT = CsvFormat
);

-- 4. Query and Validate
SELECT TOP 10 * FROM dbo.SalesData;
EXEC sp_columns 'SalesData'; -- Validate schema
```

Listing 6: Synapse External Table Creation

Q27: ADF Data Flow - Aggregate + Derived Column

Lab Task: Create a Data Flow that performs an Aggregate, followed by a Derived Column transformation to compute new KPIs.

Conceptual Basis

This task demonstrates **chaining transformations**. You first reduce the data's granularity (e.g., from 1000 sales rows to 10 region rows) using **Aggregate**, and *then* you perform row-by-row calculations on that new, smaller dataset using **Derived Column**.

Step-by-Step Solution (Flow)

1. **Source:** Add your `SalesData` source.
2. **Aggregate:** Add an **Aggregate** transformation.
    - **Group by:** `Region`
    - **Aggregates:**
    - `TotalSales = sum(SalesAmount)`
    - `TotalCost = sum(Cost)`
3. **Derived Column:** Add a **Derived Column** transformation *after* the Aggregate.
    - **Column:** `ProfitMargin_KPI`
    - **Expression:** `( (TotalSales - TotalCost) / TotalSales ) * 100`

4. **Sink:** Add a **Sink**. The output is `Region`, `TotalSales`, `TotalCost`, `ProfitMargin_KPI`.

Q28: ADF Data Flow - Window + Conditional Split

Lab Task: Use Window and Conditional Split transformations together to rank sales by date and separate top-performing regions.

Conceptual Basis

The **Window** transformation is ADF's equivalent of SQL's Analytic Functions (e.g., `RANK() OVER (...)`). It lets you perform calculations across a "window" of rows (like `partitionBy`) without collapsing them (unlike `Aggregate`). This is perfect for ranking.

Step-by-Step Solution (Flow)

1. **Source:** Add your `SalesData` source.
2. **Window:** Add a **Window** transformation.
   - **Over (Partition):** `Region` (to rank sales *within* each region)
   - **Sort:** `SalesDate` (or `SalesAmount`) in `Descending` order.
   - **Columns:** `SalesRank = rank()`
3. **Conditional Split:** Add a **Conditional Split** transformation.
   - **Stream 1 (Top_Performers):** `SalesRank == 1`
   - **Default Stream (Others):** `default()`
4. **Sinks:** Add two **Sink** transformations, one for `Top_Performers` and one for `Others`.

Q29: ADF Data Flow - Filter + Join

Lab Task: Implement a pipeline with two connected transformations: Filter inactive records first, then Join with department data.

Conceptual Basis

This is a **performance optimization**. By **Filtering** *before* the **Join**, you reduce the number of rows being passed into the join operation. This "filter-early" strategy is a fundamental concept in data engineering to minimize data shuffling and processing load.

Step-by-Step Solution (Flow)

1. **Sources:** Add `Source_Employees` and `Source_Departments`.
2. **Filter:** Add a **Filter** transformation immediately after `Source_Employees`.
   - **Filter on:** `Status == 'Active'`
3. **Join:** Add a **Join** transformation, using the *output* of the Filter* as the left stream.
   - **Left stream:** `Filter_ActiveEmployees`
   - **Right stream:** `Source_Departments`
   - **Join conditions:** `DeptID == DeptID`
4. **Sink:** Add a **Sink**. The output contains only Active employees enriched with their department names.

Q30: SAS Token - Read + Write + List

Lab Task: Generate a SAS token with read + write + list permissions, valid for 1 hour. Upload a file using this SAS link through Storage Explorer.

Conceptual Basis

This task combines concepts from Q1 and Q2. The key is to create a token with a specific set of permissions (**Read, Write, List**) that allows a client to perform a full CRUD-like operation on a container.

- **List:** To see the container's contents.
- **Read:** To download files.
- **Write:** To upload new files.

Step-by-Step Solution

1. **Generate SAS:** In Azure Storage Explorer, right-click your target container (e.g., `lab-data`) and select "Get Shared Access Signature...".
2. **Configure SAS:**
   - **Permissions:** Check **Read**, **Write**, and **List**.
   - **Expiry time:** Set the "End time" to be 1 hour from the current time.
3. **Create and Copy:** Click "Create" and copy the generated **URI**.
4. **Test Connection:** Detach any existing connections. Click the "Connect" icon.
5. Select "Blob container" → "Shared access signature (SAS)" → Paste the SAS URI.
6. **Test Permissions:**
   - You will see the container contents (verifies **List**).
   - Click "Upload" and upload a new file. The operation will **succeed** (verifies **Write**).
   - Right-click an existing file and download it (verifies **Read**).

Q31: ADF Pipeline - Excel to CSV

Lab Task: Design a pipeline in ADF to copy data from an Excel file stored in Azure Blob Storage to another Blob container as CSV.

Conceptual Basis

ADF's **Copy data** activity is dataset-driven. The "smarts" for handling file formats are in the **Dataset** definition, not the pipeline itself. By defining a source dataset as "Excel" and a sink as "DelimitedText" (CSV), the Copy activity will handle the format conversion automatically.

Step-by-Step Solution

1. **Source Dataset:**
   - Create a new Dataset → **Azure Blob Storage** → **Excel**.
   - Point it to the source `.xlsx` file in your blob container.
   - Select the **Sheet name** (e.g., "Sheet1") to be read.

2. **Sink Dataset:**
    - Create a new Dataset → **Azure Blob Storage** → **DelimitedText**.
    - Point it to the output *container* and give it a file name (e.g., `output.csv`).
3. **Pipeline:** Create a new pipeline.
4. **Copy Activity:** Drag a **Copy data** activity onto the canvas.
    - **Source:** Select your `Excel` dataset.
    - **Sink:** Select your `DelimitedText` dataset.
5. **Run:** Debug the pipeline. It will read the specified Excel sheet and write its contents into a new CSV file.

Q32: ADF Trigger - Scheduled

Lab Task: Use Scheduled Trigger to run a pipeline automatically every day at 9:00 AM IST and verify the execution in the Monitor tab.

Conceptual Basis

A **Schedule Trigger** is a "wall clock" trigger. It runs at a fixed, recurring time (like a cron job) regardless of data or other events. The key is to correctly set the time, time zone, and recurrence.

Step-by-Step Solution

1. **Create Pipeline:** Have a pipeline ready to be triggered.
2. **Create Trigger:** In the ADF "Author" pane, go to "Triggers" (under "Manage") and click "New".
3. **Configure Trigger:**
    - **Type:** Select **Schedule**.
    - **Start Date:** Set to today's date.
    - **Time Zone: Crucial step.** Select **(UTC+05:30) Chennai, Kolkata, Mumbai, New Delhi**.
    - **Time:** `09:00:00`
    - **Recurrence:** `Every 1 Day`
    - **End Date:** (Optional) Leave blank for it to run indefinitely.
4. **Associate & Publish:** Associate this trigger with your pipeline and **Publish** all changes.
5. **Verify:** After 9:00 AM IST passes, go to the **Monitor** tab. Under "Trigger Runs", you will see your trigger execution and the corresponding "Pipeline Run" it initiated.

Q33: ADF Trigger - Event-Based

Lab Task: Use Event-Based Trigger to run a pipeline when a new .csv file is uploaded to a specified Blob Storage path.

Conceptual Basis

An **Event-Based Trigger** (or Storage Event Trigger) is a reactive trigger. It subscribes to **Azure Event Grid** topics. When a file is "Blob created" in your storage, Event Grid fires an event, and the trigger catches it to start your pipeline. This is far more efficient than a schedule trigger that runs every 5 minutes "just in case" a file arrived.

Step-by-Step Solution

1. **Create Trigger:** In ADF ("Author" → "Manage" → "Triggers"), click "New".
2. **Configure Trigger:**
   - **Type:** Select **Storage events**.
   - **Storage Account:** Select your storage account (ADLS Gen2 or Blob).
   - **Container name:** `/input` (for example).
   - **Blob path begins with:** (Optional) `uploads/` (if you only want to watch a subfolder).
   - **Blob path ends with:** `.csv`
   - **Event:** Check only **Blob created**.
3. **Associate & Publish:** Associate this trigger with your pipeline and **Publish**.
4. **Test:** Upload a new `.csv` file to the `/input` container. Within a minute, you will see a new pipeline run in the **Monitor** tab. Upload a `.txt` file; it will be ignored.

Q34: ADF Trigger - Scheduled with Date Range

Lab Task: Use Scheduled Trigger with Date Range to run a pipeline only between two dates (e.g., from Nov 15 to Nov 20) and confirm it does not run outside that window.

Conceptual Basis

This is a standard feature of the Schedule Trigger. By setting both a "Start Date" and an "End Date", you define a finite execution window for the trigger. This is commonly used for short-term data migration tasks or holiday-specific jobs.

Step-by-Step Solution

1. **Create Trigger:** Create a new **Schedule** Trigger (as in Q32).
2. **Configure Date Range:**
   - **Start Date:** Set to `2025-11-15T09:00:00Z` (or your local time).
   - **Recurrence:** `Every 1 Day`.
   - **Specify an end date: Check this box.**
   - **End Date:** Set to `2025-11-20T09:00:00Z`.
3. **Associate & Publish:** Attach to a pipeline and **Publish**.
4. **Confirm:** The trigger will run on Nov 15, 16, 17, 18, and 19. It will *not* run on or after Nov 20. The "End Date" is the "stop-before" timestamp.

Q35: ADF Trigger - Using Trigger Parameters

Lab Task: Use Trigger Parameters to pass dynamic values (like fileName) into a pipeline and log the received parameter.

Conceptual Basis

This is how you make event-driven pipelines dynamic. The **Event Trigger** (Q33) captures metadata about the event (e.g., the file name and path). You then **pass** this metadata from the trigger into your **Pipeline Parameters**, making the pipeline "aware" of which file triggered it.

Step-by-Step Solution

1. **Pipeline Parameter:**
   - In your pipeline, go to the "Parameters" tab.
   - Click "New" and create a parameter named `p_FileName`.
2. **Create Event Trigger:** Create a **Storage Event** Trigger (as in Q33).
3. **Map Parameters:**
   - When you attach the trigger to the pipeline, a "Parameters" pane will appear.
   - The pipeline parameter `p_FileName` will be listed.
   - In the "Value" box for it, enter the dynamic content expression: `@triggerBody().fileName`
4. **Log the Parameter:**
   - Add a **Lookup** activity to your pipeline (this is a simple way to "use" a parameter).
   - **Settings:** Use a dummy dataset.
   - In an arbitrary field (like a query), use the parameter: `@pipeline().p_FileName`. This isn't a real log, but it proves the parameter was passed.
   - (A better way is to use a `Stored Procedure` activity to write `@pipeline().p_FileName` to a log table).
5. **Run Verify:** Publish and upload a file named `report-2025.csv`. In the "Monitor" tab, check the "Input" of the pipeline run. You will see `p_FileName: "report-2025.csv"`.