Question 4: Collaborative filtering recommender system for recommending aisles

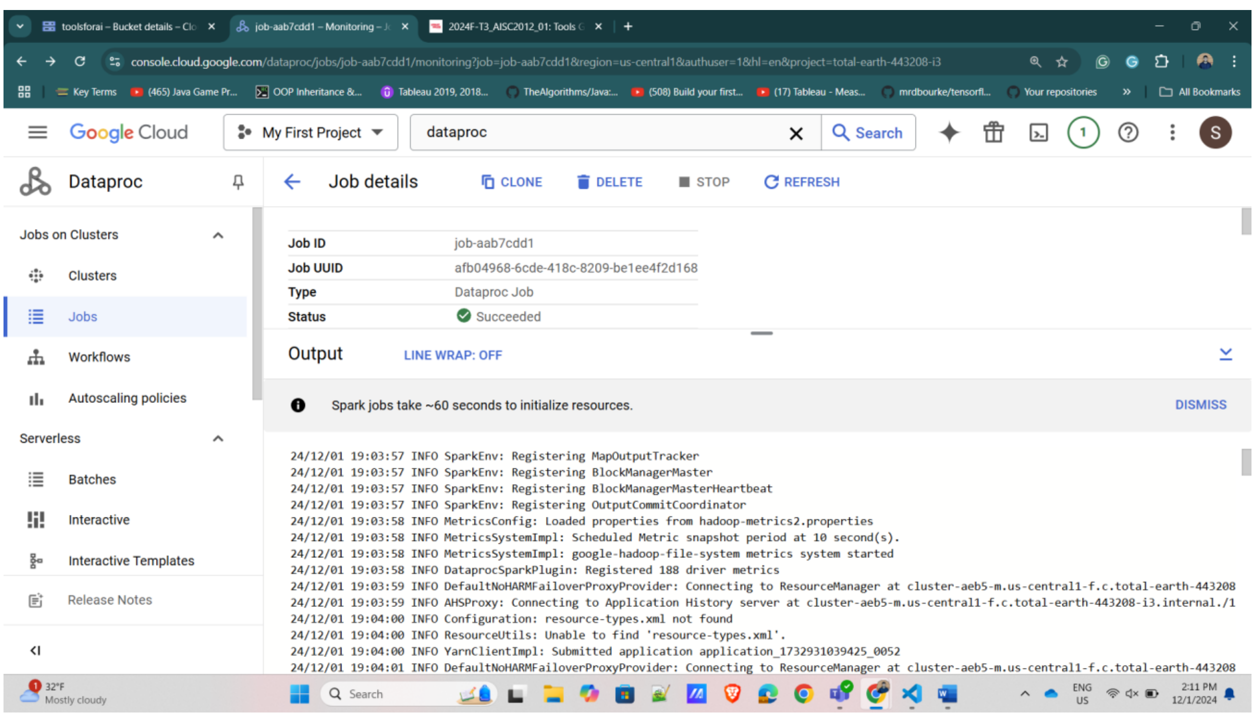
MECE:

Shubham Bhavsar: Model building, Evaluation and Aisle Recommendations for Top 5 Banana Buyers

Palash Lalani: Data Preprocessing, Rolling Cross Validation and Report Creation

Status Update Table:

Model	Task	Comment/Score
Model 4	Collaborative filtering recommender system for recommending aisles	Run-time = **41.80 seconds**
	Strong Generalization	MAP@5 = **0.35579837902969114** Recall: **0.07855490828189576**
	Weak Generalization	MAP@5 = **0.047336184386920645** Recall: **0.07075051890727846**
	Top 5 users who buy most bananas	Recommendations = [User 1: **189425**, User 2: **194931**, User 3: **178107**, User 4: **99707**, User 5: **69919**] See Aisle Recommendations section for more info.
	Non-Spark Libraries Used	Library: time Used for measuring training runtime



Initial Setup:

Imported required libraries

Set up a PySpark environment with adequate memory allocation for the driver and executor.

Loaded the datasets (products, orders, order_products, and aisles) from Google Cloud Storage.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, count, min, max
from pyspark.mllib.recommendation import ALS, Rating
from pyspark.mllib.evaluation import RankingMetrics
import time

# -----------------------------Initial Setup------------------------------
spark = SparkSession.builder \
    .appName("CollaborativeFiltering-RecommendAisles") \
    .config("spark.executor.memory", "8g") \
    .config("spark.driver.memory", "8g") \
    .getOrCreate()

products = spark.read.csv('gs://toolsforai/products.csv', header=True, inferSchema=True)
orders = spark.read.csv('gs://toolsforai/orders.csv', header=True, inferSchema=True)
order_products = spark.read.csv('gs://toolsforai/order_products.csv', header=True, inferSchema=True)
aisles = spark.read.csv('gs://toolsforai/aisles.csv', header=True, inferSchema=True)
```

Data Preprocessing

What we did?

- Filtered the dataset for users who purchased bananas by identifying the product_id corresponding to bananas.
- Extracted user-aisle interaction data by aggregating the count of purchases per user per aisle.
- Normalized the interaction strengths to scale the values between 0 and 1.

Why we did?

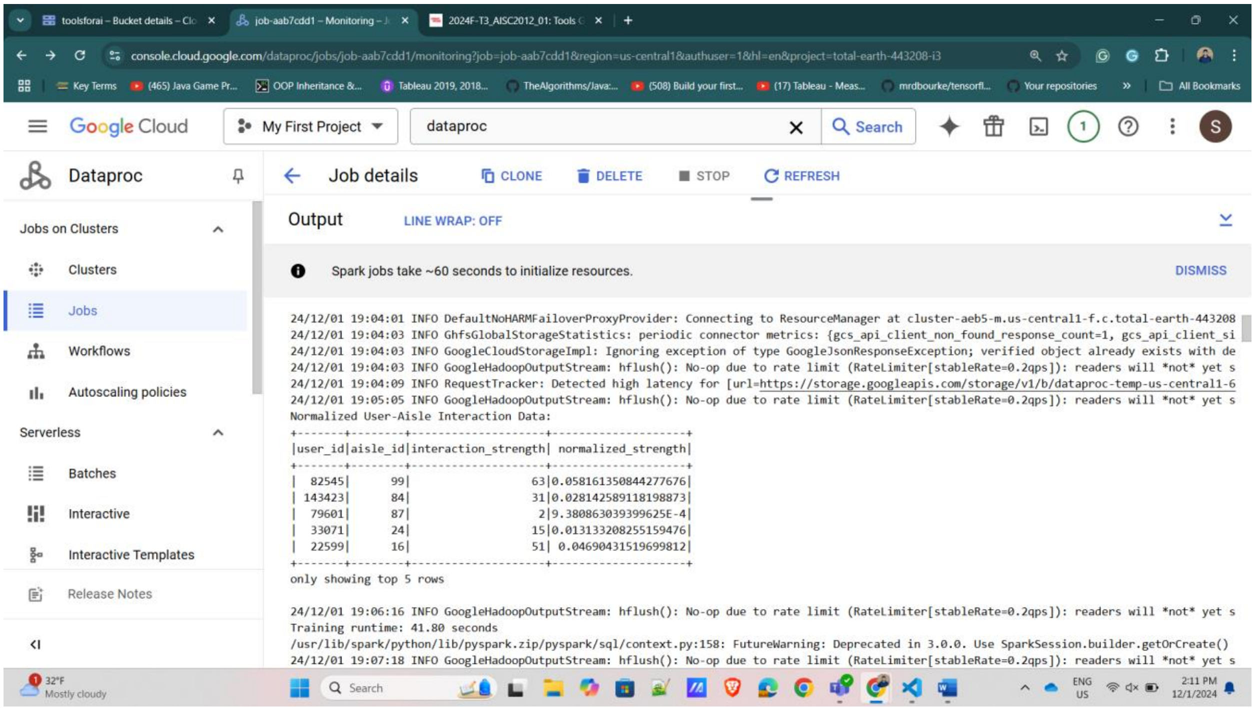
- To focus the recommendation system on banana buyers and create a meaningful representation of user-aisle interactions that can be used as input for the ALS model.

Code:

```python
# ------------------------------Data Preprocessing------------------------
# Limiting Data for Banana Buyers Only
banana_product_id = products.filter(col("product_name") == "Banana").select("product_id").first()["product_id"]
banana_buyers = order_products.join(orders, "order_id") \
    .filter(col("product_id") == banana_product_id) \
    .select("user_id").distinct()

filtered_orders = orders.join(banana_buyers, "user_id")
filtered_order_products = order_products.join(filtered_orders, "order_id")

# Aggregate User-Aisle Interactions
filtered_order_products = filtered_order_products.join(products, "product_id").join(aisles, "aisle_id")
user_aisle_interactions = filtered_order_products.groupBy("user_id", "aisle_id") \
    .agg(count("*").alias("interaction_strength"))

# Normalize Interaction Strengths
interaction_stats = user_aisle_interactions.agg(
    min("interaction_strength").alias("min_interaction"),
    max("interaction_strength").alias("max_interaction")
).collect()

min_interaction = interaction_stats[0]["min_interaction"]
max_interaction = interaction_stats[0]["max_interaction"]

normalized_user_aisle_interactions = user_aisle_interactions.withColumn(
    "normalized_strength",
    (col("interaction_strength") - min_interaction) / (max_interaction - min_interaction)
)
```

Results:

```
Normalized User-Aisle Interaction Data:
+-------+--------+--------------------+--------------------+
|user_id|aisle_id|interaction_strength| normalized_strength|
+-------+--------+--------------------+--------------------+
|  82545|      99|                  63|0.058161350844277676|
| 143423|      84|                  31|0.028142589118198873|
|  79601|      87|                   2|9.380863039399625E-4|
|  33071|      24|                  15|0.013133208255159476|
|  22599|      16|                  51| 0.04690431519699812|
+-------+--------+--------------------+--------------------+
only showing top 5 rows
```



Model Training

Trained an ALS model using the interaction data.

```python
# Prepare data in RDD format for ALS modeling
als_data_raw = user_aisle_interactions.rdd.map(
    lambda row: Rating(row["user_id"], row["aisle_id"], row["interaction_strength"])
)

# Data Splitting
train_data_raw, test_data_raw = als_data_raw.randomSplit([0.8, 0.2], seed=42)

# --------------------------------Model Training--------------------------

start_time = time.time()

als_model = ALS.train(
    train_data_raw,
    rank=20,
    iterations=20,
    lambda_=0.01
)

end_time = time.time()

# Calculate runtime
training_runtime = end_time - start_time
print(f"Training runtime: {training_runtime:.2f} seconds")
```

Training time:

```
24/12/01 19:06:16 INFO GoogleHadoopOutputStream: hflush(): No-op due to rate limit
Training runtime: 41.80 seconds
```

Model Evaluation:

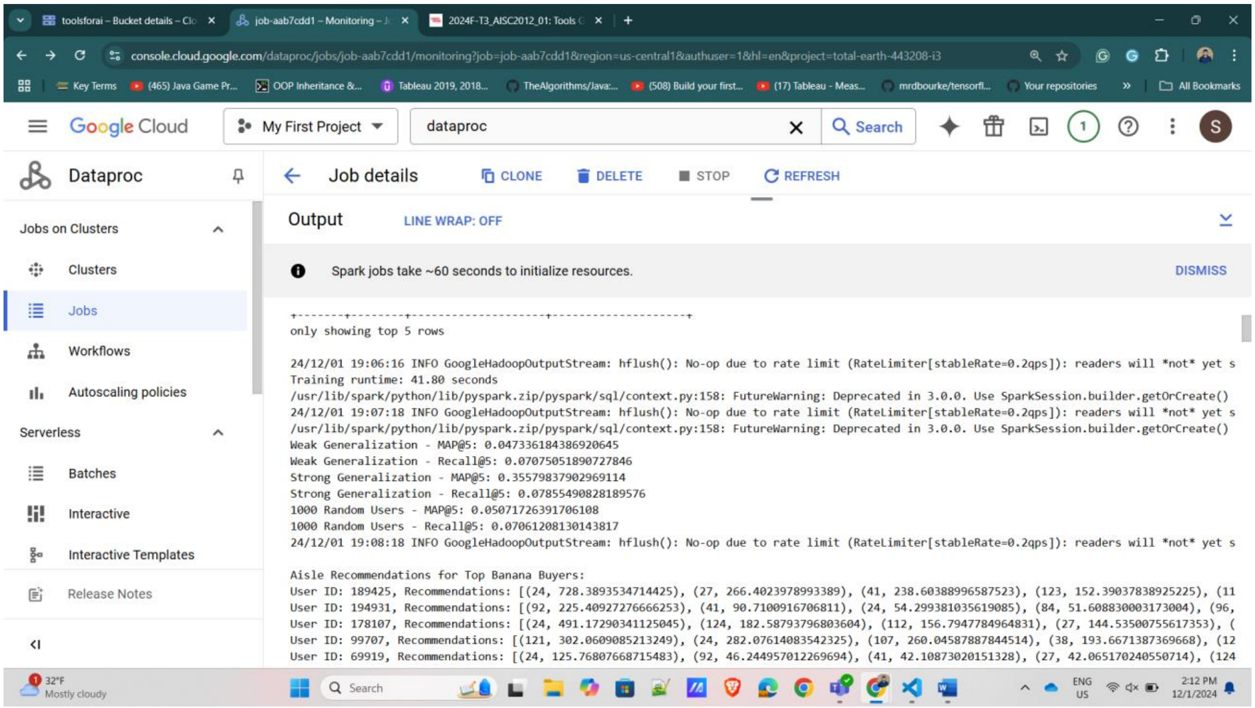
- Model Evaluation on Strong and Weak Generalization
- Evaluation on 1000 Random Users

```python
# ----------Model Evaluation Strong and Weak Generalization------------------------

unique_users = user_aisle_interactions.select("user_id").distinct()
_, test_users = unique_users.randomSplit([0.8, 0.2], seed=42)
test_users_df = test_users
test_data_strong = user_aisle_interactions.join(test_users_df, "user_id")

recommendations = als_model.recommendProductsForUsers(5)

# Weak Generalization
test_data_weak = test_data_raw
predicted_ranking_weak = recommendations.mapValues(lambda recs: [rec.product for rec in recs])
actual_ranking_weak = test_data_weak.map(lambda x: (x.user, int(x.product))).groupByKey().mapValues(list)
formatted_ranking_weak = predicted_ranking_weak.join(actual_ranking_weak).map(lambda x: (x[1][0], x[1][1]))

metrics_weak = RankingMetrics(formatted_ranking_weak)
map_at_5_weak = metrics_weak.meanAveragePrecisionAt(5)
recall_at_5_weak = metrics_weak.recallAt(5)

# Strong Generalization
predicted_ranking_strong = recommendations.mapValues(lambda recs: [rec.product for rec in recs])
actual_ranking_strong = test_data_strong.rdd.map(lambda x: (x["user_id"], int(x["aisle_id"]))).groupByKey().mapValues(list)
formatted_ranking_strong = predicted_ranking_strong.join(actual_ranking_strong).map(lambda x: (x[1][0], x[1][1]))

metrics_strong = RankingMetrics(formatted_ranking_strong)
map_at_5_strong = metrics_strong.meanAveragePrecisionAt(5)
recall_at_5_strong = metrics_strong.recallAt(5)

# Print Generalization Results
```

```python
# Print Generalization Results
print("Weak Generalization - MAP@5:", map_at_5_weak)
print("Weak Generalization - Recall@5:", recall_at_5_weak)
print("Strong Generalization - MAP@5:", map_at_5_strong)
print("Strong Generalization - Recall@5:", recall_at_5_strong)

# ----------Model Evaluation for 1000 Random Users------------------------

random_users_sample = test_data_raw.map(lambda x: x.user).distinct().takeSample(False, 1000, seed=42)
random_users_sample_broadcast = spark.sparkContext.broadcast(set(random_users_sample))

# Filter for Random Users
filtered_recommendations_sample = recommendations.filter(lambda x: x[0] in random_users_sample_broadcast.value)
filtered_test_data_sample = test_data_raw.filter(lambda x: x.user in random_users_sample_broadcast.value)

predicted_ranking_sample = filtered_recommendations_sample.mapValues(lambda recs: [rec.product for rec in recs])
actual_ranking_sample = filtered_test_data_sample.map(lambda x: (x.user, int(x.product))).groupByKey().mapValues(list)
formatted_ranking_sample = predicted_ranking_sample.join(actual_ranking_sample).map(lambda x: (x[1][0], x[1][1]))

metrics_sample = RankingMetrics(formatted_ranking_sample)
map_at_5_sample = metrics_sample.meanAveragePrecisionAt(5)
recall_at_5_sample = metrics_sample.recallAt(5)

# Print Results
print("1000 Random Users - MAP@5:", map_at_5_sample)
print("1000 Random Users - Recall@5:", recall_at_5_sample)
```

Output:

```
Weak Generalization - MAP@5: 0.047336184386920645
Weak Generalization - Recall@5: 0.07075051890727846
Strong Generalization - MAP@5: 0.35579837902969114
Strong Generalization - Recall@5: 0.07855490828189576
1000 Random Users - MAP@5: 0.05071726391706108
1000 Random Users - Recall@5: 0.07061208130143817
```



Aisle Recommendations for Top Banana Buyers

- Identified the top 5 users who purchased the most bananas.
- Generated aisle recommendations for these users using the trained ALS model.

Code:

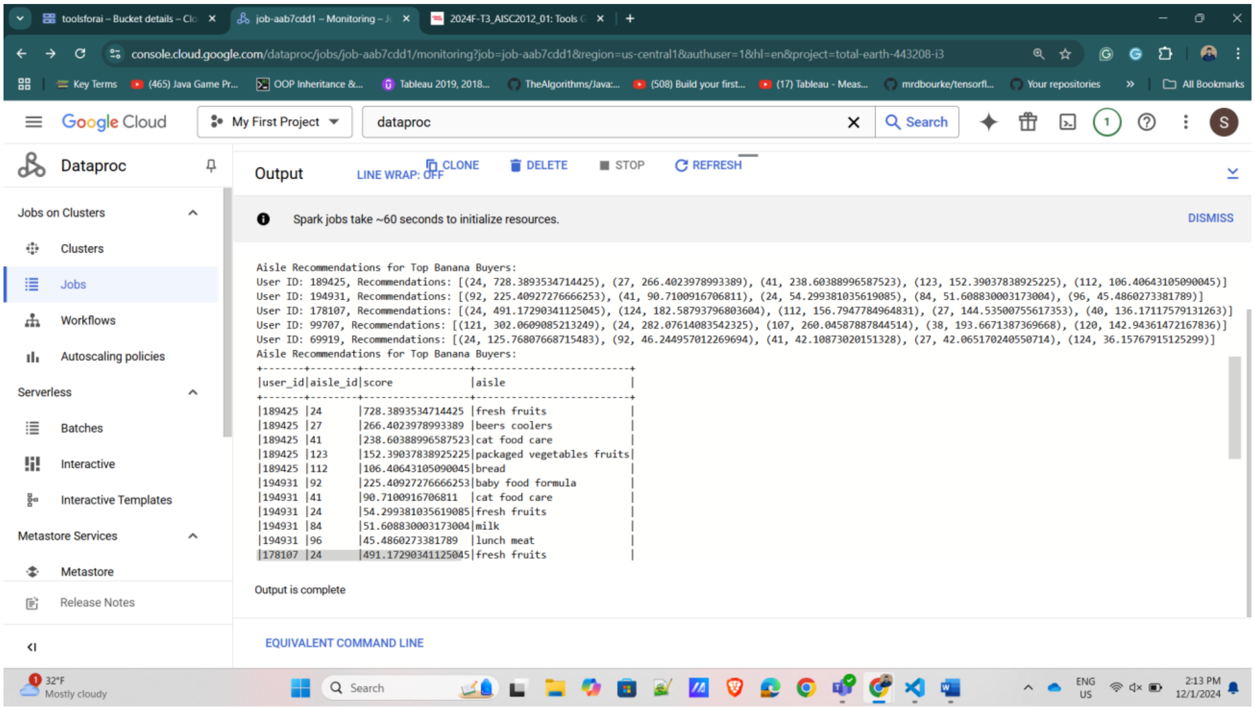
```python
# ----------------------Aisle Recommendations for Top Bannana Buyers------------------

top_banana_buyers = filtered_order_products.filter(col("product_id") == banana_product_id) \
    .groupBy("user_id") \
    .count() \
    .orderBy(col("count").desc()) \
    .limit(5)

top_banana_buyers_ids = [row["user_id"] for row in top_banana_buyers.collect()]
top_banana_buyers_broadcast = spark.sparkContext.broadcast(top_banana_buyers_ids)

recommendations_top_users = recommendations.filter(lambda x: x[0] in top_banana_buyers_broadcast.value).collect()

print("\nAisle Recommendations for Top Banana Buyers:")
if not recommendations_top_users:
    print("No Recommendations Found for Top Banana Buyers")
else:
    for user_id, recs in recommendations_top_users:
        recommendations_list = [(r.product, r.rating) for r in recs]
        print(f"User ID: {user_id}, Recommendations: {recommendations_list}")

recommendations_top_users_df = spark.createDataFrame([
    (user_id, r.product, r.rating)
    for user_id, recs in recommendations_top_users
    for r in recs
], schema=["user_id", "aisle_id", "score"])

# Join recommendations with aisle names

recommendations_with_names_df = recommendations_top_users_df.join(
    aisles.withColumnRenamed("aisle_id", "aisle_key"),
    recommendations_top_users_df["aisle_id"] == col("aisle_key"),
    "left"
).select(
    "user_id",
    "aisle_id",
    "score",
    "aisle"
)

print("Aisle Recommendations for Top Banana Buyers:")
recommendations_with_names_df.show(25, truncate=False)

recommendations_with_names_df = recommendations_with_names_df.limit(10000)  # Limit to 10,000 rows

output_path = "gs://toolsforai//recommendations_with_aisle_names.csv"

# Save the recommendations
recommendations_with_names_df.write.csv(output_path, header=True, mode="overwrite")
print(f"Results saved to {output_path}")

#---------------------------------------------------------------------------
```

Output:

```
Aisle Recommendations for Top Banana Buyers:
User ID: 189425, Recommendations: [(24, 728.3893534714425), (27, 266.4023978993389), (41, 238.60388996587523), (123, 152.39037838925225), (112, 106.40643105090045)]
User ID: 194931, Recommendations: [(92, 225.40927276666253), (41, 90.7100916706811), (24, 54.299381035619085), (84, 51.608830003173004), (96, 45.4860273381789)]
User ID: 178107, Recommendations: [(24, 491.17290341125045), (124, 182.58793796803604), (112, 156.7947784964831), (27, 144.53500755617353), (40, 136.17117579131263)]
User ID: 99707, Recommendations: [(121, 302.0609085213249), (24, 282.07614083542325), (107, 260.04587887844514), (38, 193.6671387369668), (120, 142.94361472167836)]
User ID: 69919, Recommendations: [(24, 125.76807668715483), (92, 46.244957012269694), (41, 42.10873020151328), (27, 42.065170240550714), (124, 36.15767915125299)]
```

Aisle Recommendations for Top Banana Buyers:

user_id	aisle_id	score	aisle
189425	24	728.3893534714425	fresh fruits
189425	27	266.4023978993389	beers coolers
189425	41	238.60388996587523	cat food care
189425	123	152.39037838925225	packaged vegetables fruits
189425	112	106.40643105090045	bread
194931	92	225.40927276666253	baby food formula
194931	41	90.7100916706811	cat food care
194931	24	54.299381035619085	fresh fruits
194931	84	51.608830003173004	milk
194931	96	45.4860273381789	lunch meat
178107	24	491.17290341125045	fresh fruits
178107	124	182.58793796803604	spirits
178107	112	156.7947784964831	bread
178107	27	144.53500755617353	beers coolers
178107	40	136.17117579131263	dog food care
99707	121	302.0609085213249	cereal
99707	24	282.07614083542325	fresh fruits
99707	107	260.04587887844514	chips pretzels
99707	38	193.6671387369668	frozen meals
99707	120	142.94361472167836	yogurt
69919	24	125.76807668715483	fresh fruits
69919	92	46.244957012269694	baby food formula
69919	41	42.10873020151328	cat food care
69919	27	42.065170240550714	beers coolers
69919	124	36.15767915125299	spirits



Entire Code:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, count, min, max
from pyspark.mllib.recommendation import ALS, Rating
from pyspark.mllib.evaluation import RankingMetrics
import time

# ------------------Initial Setup-------------------------------
spark = SparkSession.builder \
    .appName("CollaborativeFiltering-RecommendAisles") \
    .config("spark.executor.memory", "8g") \
    .config("spark.driver.memory", "8g") \
    .getOrCreate()

products = spark.read.csv('gs://toolsforai/products.csv', header=True,
inferSchema=True)
orders = spark.read.csv('gs://toolsforai/orders.csv', header=True,
inferSchema=True)
```

```python
order_products = spark.read.csv('gs://toolsforai/order_products.csv',
header=True, inferSchema=True)
aisles = spark.read.csv('gs://toolsforai/aisles.csv', header=True,
inferSchema=True)

# --------------Data Preprocessing------------------------
# Limiting Data for Banana Buyers Only
banana_product_id = products.filter(col("product_name") ==
"Banana").select("product_id").first()["product_id"]
banana_buyers = order_products.join(orders, "order_id") \
    .filter(col("product_id") == banana_product_id) \
    .select("user_id").distinct()

filtered_orders = orders.join(banana_buyers, "user_id")
filtered_order_products = order_products.join(filtered_orders,
"order_id")

# Aggregate User-Aisle Interactions
filtered_order_products = filtered_order_products.join(products,
"product_id").join(aisles, "aisle_id")
user_aisle_interactions = filtered_order_products.groupBy("user_id",
"aisle_id") \
    .agg(count("*").alias("interaction_strength"))

# Normalize Interaction Strengths
interaction_stats = user_aisle_interactions.agg(
    min("interaction_strength").alias("min_interaction"),
    max("interaction_strength").alias("max_interaction")
).collect()

min_interaction = interaction_stats[0]["min_interaction"]
max_interaction = interaction_stats[0]["max_interaction"]

normalized_user_aisle_interactions =
user_aisle_interactions.withColumn(
    "normalized_strength",
```

```
    (col("interaction_strength") - min_interaction) / (max_interaction
- min_interaction)
)

# Preview Normalized Interactions
print("Normalized User-Aisle Interaction Data:")
normalized_user_aisle_interactions.show(5)

# Prepare data in RDD format for ALS modeling
als_data_raw = user_aisle_interactions.rdd.map(
    lambda row: Rating(row["user_id"], row["aisle_id"],
row["interaction_strength"])
)

# Data Splitting
train_data_raw, test_data_raw = als_data_raw.randomSplit([0.8, 0.2],
seed=42)

# --------------------Model Training--------------------------
start_time = time.time()

als_model = ALS.train(
    train_data_raw,
    rank=20,
    iterations=20,
    lambda_=0.01
)

end_time = time.time()

# Calculate runtime
training_runtime = end_time - start_time
print(f"Training runtime: {training_runtime:.2f} seconds")

# ----------Model Evaluation Strong and Weak Generalization----------
unique_users = user_aisle_interactions.select("user_id").distinct()
```

```
_, test_users = unique_users.randomSplit([0.8, 0.2], seed=42)
test_users_df = test_users
test_data_strong = user_aisle_interactions.join(test_users_df,
"user_id")

recommendations = als_model.recommendProductsForUsers(5)

# Weak Generalization
test_data_weak = test_data_raw
predicted_ranking_weak = recommendations.mapValues(lambda recs:
[rec.product for rec in recs])
actual_ranking_weak = test_data_weak.map(lambda x: (x.user,
int(x.product))).groupByKey().mapValues(list)
formatted_ranking_weak =
predicted_ranking_weak.join(actual_ranking_weak).map(lambda x:
(x[1][0], x[1][1]))

metrics_weak = RankingMetrics(formatted_ranking_weak)
map_at_5_weak = metrics_weak.meanAveragePrecisionAt(5)
recall_at_5_weak = metrics_weak.recallAt(5)

# Strong Generalization
predicted_ranking_strong = recommendations.mapValues(lambda recs:
[rec.product for rec in recs])
actual_ranking_strong = test_data_strong.rdd.map(lambda x:
(x["user_id"], int(x["aisle_id"]))).groupByKey().mapValues(list)
formatted_ranking_strong =
predicted_ranking_strong.join(actual_ranking_strong).map(lambda x:
(x[1][0], x[1][1]))

metrics_strong = RankingMetrics(formatted_ranking_strong)
map_at_5_strong = metrics_strong.meanAveragePrecisionAt(5)
recall_at_5_strong = metrics_strong.recallAt(5)

# Print Generalization Results
print("Weak Generalization - MAP@5:", map_at_5_weak)
```

```python
print("Weak Generalization - Recall@5:", recall_at_5_weak)
print("Strong Generalization - MAP@5:", map_at_5_strong)
print("Strong Generalization - Recall@5:", recall_at_5_strong)

# ----------Model Evaluation for 1000 Random Users--------------------
random_users_sample = test_data_raw.map(lambda x:
x.user).distinct().takeSample(False, 1000, seed=42)
random_users_sample_broadcast =
spark.sparkContext.broadcast(set(random_users_sample))

# Filter for Random Users
filtered_recommendations_sample = recommendations.filter(lambda x:
x[0] in random_users_sample_broadcast.value)
filtered_test_data_sample = test_data_raw.filter(lambda x: x.user in
random_users_sample_broadcast.value)

predicted_ranking_sample =
filtered_recommendations_sample.mapValues(lambda recs: [rec.product
for rec in recs])
actual_ranking_sample = filtered_test_data_sample.map(lambda x:
(x.user, int(x.product))).groupByKey().mapValues(list)
formatted_ranking_sample =
predicted_ranking_sample.join(actual_ranking_sample).map(lambda x:
(x[1][0], x[1][1]))

metrics_sample = RankingMetrics(formatted_ranking_sample)
map_at_5_sample = metrics_sample.meanAveragePrecisionAt(5)
recall_at_5_sample = metrics_sample.recallAt(5)

# Print Results
print("1000 Random Users - MAP@5:", map_at_5_sample)
print("1000 Random Users - Recall@5:", recall_at_5_sample)

# ---------------Aisle Recommendations for Top Bannana Buyers---------
top_banana_buyers = filtered_order_products.filter(col("product_id")
== banana_product_id) \
```

```python
    .groupBy("user_id") \
    .count() \
    .orderBy(col("count").desc()) \
    .limit(5)

top_banana_buyers_ids = [row["user_id"] for row in
top_banana_buyers.collect()]
top_banana_buyers_broadcast =
spark.sparkContext.broadcast(top_banana_buyers_ids)

recommendations_top_users = recommendations.filter(lambda x: x[0] in
top_banana_buyers_broadcast.value).collect()

print("\nAisle Recommendations for Top Banana Buyers:")
if not recommendations_top_users:
    print("No Recommendations Found for Top Banana Buyers")
else:
    for user_id, recs in recommendations_top_users:
        recommendations_list = [(r.product, r.rating) for r in recs]
        print(f"User ID: {user_id}, Recommendations:
{recommendations_list}")

recommendations_top_users_df = spark.createDataFrame([
    (user_id, r.product, r.rating)
    for user_id, recs in recommendations_top_users
    for r in recs
], schema=["user_id", "aisle_id", "score"])

# Join recommendations with aisle names
recommendations_with_names_df = recommendations_top_users_df.join(
    aisles.withColumnRenamed("aisle_id", "aisle_key"),
    recommendations_top_users_df["aisle_id"] == col("aisle_key"),
    "left"
).select(
    "user_id",
    "aisle_id",
```

```
    "score",
    "aisle"
)

print("Aisle Recommendations for Top Banana Buyers:")
recommendations_with_names_df.show(25, truncate=False)

recommendations_with_names_df =
recommendations_with_names_df.limit(10000)  # Limit to 10,000 rows

output_path = "gs://toolsforai//recommendations_with_aisle_names.csv"

# Save the recommendations
recommendations_with_names_df.write.csv(output_path, header=True,
mode="overwrite")
print(f"Results saved to {output_path}")

#---------------------------------------------------------------
```

Cross Validation Code:

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, row_number
from pyspark.sql.window import Window
from pyspark.ml.recommendation import ALS
from pyspark.ml.evaluation import RegressionEvaluator

# Initialize Spark Session
spark =
SparkSession.builder.appName("RecommenderSystem").getOrCreate()

# Load the datasets
aisles = spark.read.csv("gs://dataproc-staging-us-central1-
116615940763-tcdeaetb/retail/aisles.csv", header=True,
inferSchema=True)
```

```
departments = spark.read.csv("gs://dataproc-staging-us-central1-
116615940763-tcdeaetb/retail/departments.csv", header=True,
inferSchema=True)
orders = spark.read.csv("gs://dataproc-staging-us-central1-
116615940763-tcdeaetb/retail/orders.csv", header=True,
inferSchema=True)
products = spark.read.csv("gs://dataproc-staging-us-central1-
116615940763-tcdeaetb/retail/products.csv", header=True,
inferSchema=True)
order_products = spark.read.csv("gs://dataproc-staging-us-central1-
116615940763-tcdeaetb/retail/order_products.csv", header=True,
inferSchema=True)

# Join datasets to create a unified DataFrame for analysis
data = order_products.join(orders, on="order_id", how="inner") \
                     .join(products, on="product_id", how="inner") \
                     .join(aisles, on="aisle_id", how="inner") \
                     .join(departments, on="department_id",
how="inner")

# Select relevant columns
data = data.select("order_id", "product_id", "user_id",
"add_to_cart_order", "reordered")

# Assign row numbers to simulate timestamps for rolling cross-
validation
window_spec = Window.orderBy("order_id")
data = data.withColumn("row_num", row_number().over(window_spec))

# Define rolling splits
def create_rolling_splits(data, num_splits=3):
    total_rows = data.count()
    split_size = total_rows // (num_splits + 1)

    splits = []
    for i in range(num_splits):
```

```
        train = data.filter(col("row_num") <= split_size * (i + 1))
        test = data.filter((col("row_num") > split_size * (i + 1)) &
                           (col("row_num") <= split_size * (i + 2)))
        splits.append((train, test))
    return splits

rolling_splits = create_rolling_splits(data)

# Inspect each train-test split
for i, (train, test) in enumerate(rolling_splits):
    print(f"Fold {i+1}:")

    # Print train and test split counts
    print(f"Train count: {train.count()}, Test count: {test.count()}")

    # Show first few rows of train set
    print("Train Split:")
    train.show(5)

    # Show first few rows of test set
    print("Test Split:")
    test.show(5)

    # Check overlap between train and test
    train_users = train.select("user_id").distinct()
    test_users = test.select("user_id").distinct()
    common_users = train_users.intersect(test_users).count()

    train_products = train.select("product_id").distinct()
    test_products = test.select("product_id").distinct()
    common_products = train_products.intersect(test_products).count()

    print(f"Common users: {common_users}, Common products:
{common_products}")
    print("-" * 50)
```

```
    # Configure ALS
als = ALS(userCol="user_id", itemCol="product_id",
ratingCol="reordered",
          rank=10, maxIter=10, regParam=0.1, coldStartStrategy="drop")

# Initialize RMSE evaluator
evaluator = RegressionEvaluator(metricName="rmse",
labelCol="reordered", predictionCol="prediction")

# Train and evaluate on each split
rmse_scores = []
for i, (train, test) in enumerate(rolling_splits):
    print(f"Fold {i+1}:")

    # Train the ALS model
    model = als.fit(train)

    # Make predictions on the test set
    predictions = model.transform(test)

    # Check if predictions are not empty
    if predictions.count() > 0:
        # Evaluate RMSE
        rmse = evaluator.evaluate(predictions)
        rmse_scores.append(rmse)
        print(f"Fold {i+1} RMSE: {rmse}")
    else:
        print(f"Fold {i+1}: No predictions generated. Skipping RMSE
evaluation.")

    print("-" * 50)

# Calculate average RMSE
if len(rmse_scores) > 0:
    average_rmse = sum(rmse_scores) / len(rmse_scores)
    print(f"Average RMSE across all folds: {average_rmse}")
```

```python
else:
    print("No valid predictions generated for any fold.")
```

Results:

For Fold-1

Train count: 8108621, Test count: 8108621

Train Split:

```
+--------+----------+-------+-----------------+---------+-------+
|order_id|product_id|user_id|add_to_cart_order|reordered|row_num|
+--------+----------+-------+-----------------+---------+-------+
|       2|     33120| 202279|                1|        1|      1|
|       2|     28985| 202279|                2|        1|      2|
|       2|      9327| 202279|                3|        0|      3|
|       2|     45918| 202279|                4|        1|      4|
|       2|     30035| 202279|                5|        0|      5|
+--------+----------+-------+-----------------+---------+-------+
only showing top 5 rows
```

Test Split:

```
+--------+----------+-------+-----------------+---------+-------+
|order_id|product_id|user_id|add_to_cart_order|reordered|row_num|
+--------+----------+-------+-----------------+---------+-------+
|  855943|     13631| 174676|                8|        1|8108622|
|  855943|     22169| 174676|                9|        1|8108623|
|  855943|     14462| 174676|               10|        0|8108624|
|  855943|      6849| 174676|               11|        0|8108625|
|  855943|     45445| 174676|               12|        1|8108626|
+--------+----------+-------+-----------------+---------+-------+
only showing top 5 rows
```

Common users: 156604, Common products: 46506

Fold 1 RMSE: 0.46764134162378995

For Fold-2

Train count: 16217242, Test count: 8108621

Train Split:

```
+--------+----------+-------+----------------+---------+-------+
|order_id|product_id|user_id|add_to_cart_order|reordered|row_num|
+--------+----------+-------+----------------+---------+-------+
|       2|     33120| 202279|               1|        1|      1|
|       2|     28985| 202279|               2|        1|      2|
|       2|      9327| 202279|               3|        0|      3|
|       2|     45918| 202279|               4|        1|      4|
|       2|     30035| 202279|               5|        0|      5|
+--------+----------+-------+----------------+---------+-------+
only showing top 5 rows
```

Test Split:

```
+--------+----------+-------+----------------+---------+--------+
|order_id|product_id|user_id|add_to_cart_order|reordered| row_num|
+--------+----------+-------+----------------+---------+--------+
| 1711047|     44683| 159337|              12|        0|16217243|
| 1711048|     39812|  36017|               1|        1|16217244|
| 1711048|     24964|  36017|               2|        1|16217245|
| 1711048|      2966|  36017|               3|        1|16217246|
| 1711048|     45007|  36017|               4|        1|16217247|
+--------+----------+-------+----------------+---------+--------+
only showing top 5 rows
```

Common users: 174350, Common products: 47662

Fold 2 RMSE: 0.46176252481905483

For Fold-3

Train count: 24325863, Test count: 8108621

Train Split:

```
+--------+----------+-------+----------------+---------+-------+
|order_id|product_id|user_id|add_to_cart_order|reordered|row_num|
+--------+----------+-------+----------------+---------+-------+
|       2|     33120| 202279|               1|        1|      1|
|       2|     28985| 202279|               2|        1|      2|
|       2|      9327| 202279|               3|        0|      3|
|       2|     45918| 202279|               4|        1|      4|
|       2|     30035| 202279|               5|        0|      5|
+--------+----------+-------+----------------+---------+-------+
only showing top 5 rows
```

Test Split:

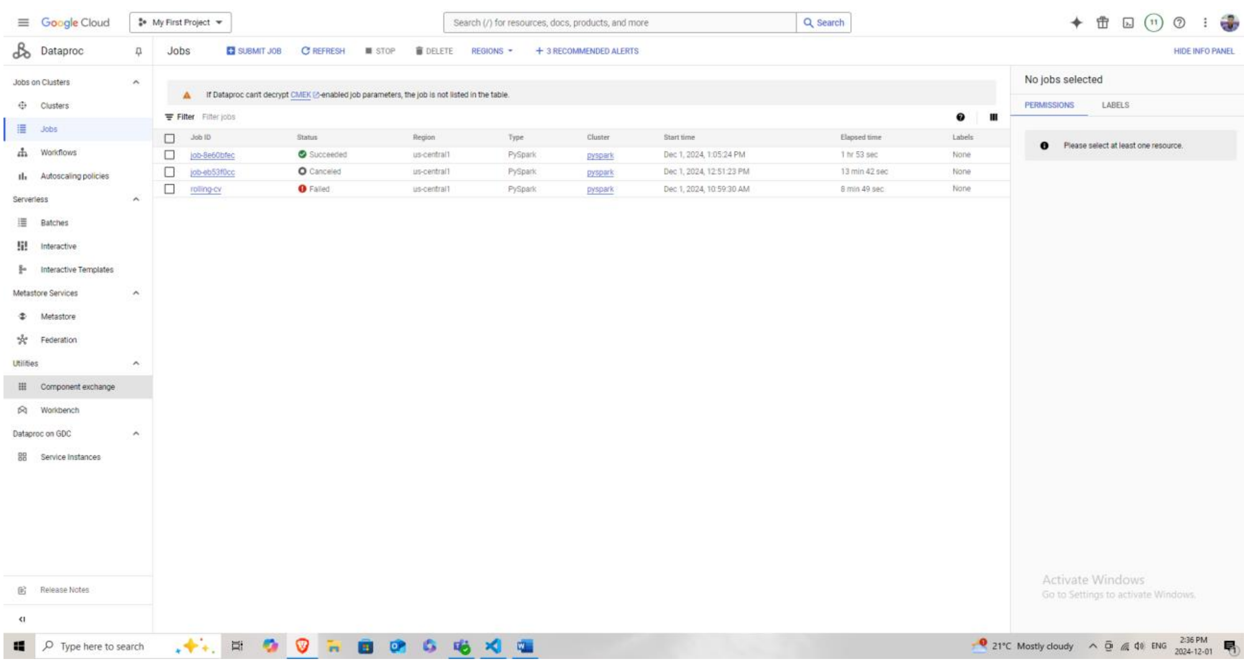
```
+--------+----------+-------+-----------------+---------+--------+
|order_id|product_id|user_id|add_to_cart_order|reordered| row_num|
+--------+----------+-------+-----------------+---------+--------+
| 2565514|     44098|  85474|                7|        1|24325864|
| 2565514|     12144|  85474|                8|        1|24325865|
| 2565514|      3849|  85474|                9|        1|24325866|
| 2565514|     18370|  85474|               10|        1|24325867|
| 2565514|     24852|  85474|               11|        1|24325868|
+--------+----------+-------+-----------------+---------+--------+
only showing top 5 rows
```

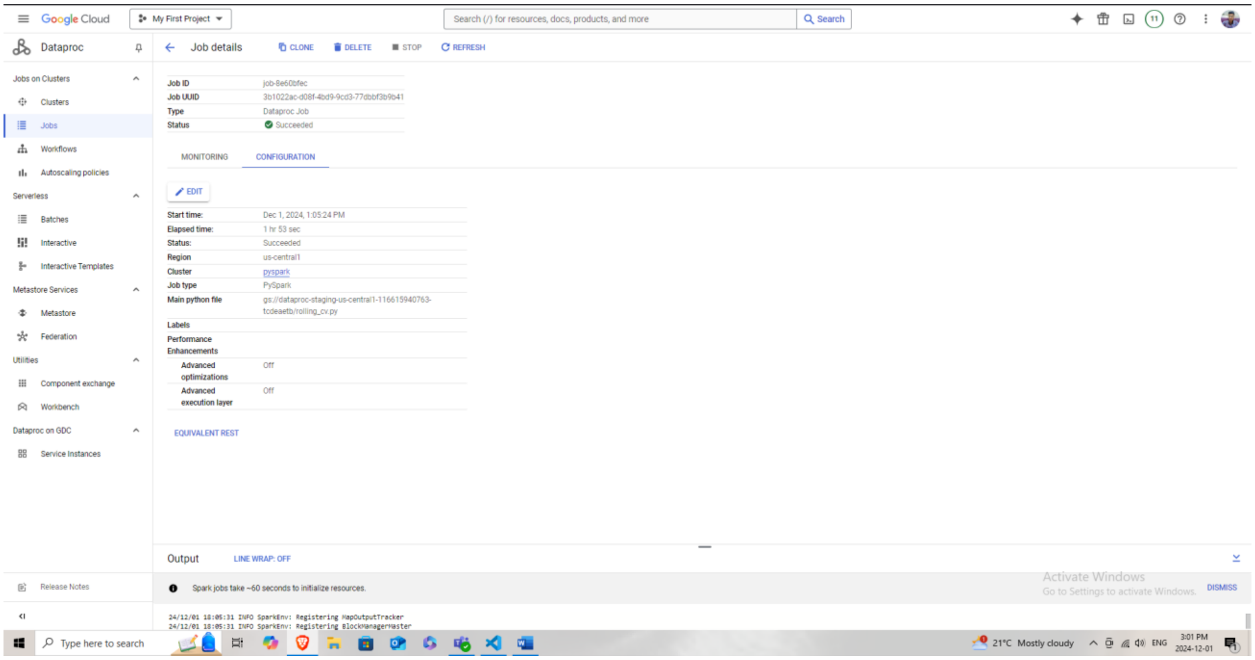
Common users: 178624, Common products: 47848

Fold 3 RMSE: 0.45927021393916434

Average RMSE across all folds: 0.46289136012733634

Jobs Screenshot:






```
-----------------------------------------------------
Average RMSE across all folds: 0.46289136012733634
```