

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

The screenshot displays the Google Cloud Dataproc console interface. The top navigation bar shows the Google Cloud logo, the project name 'My First Project', and the 'dataproc' service. The left sidebar contains a navigation menu with options like 'Jobs on Clusters', 'Clusters', 'Jobs', 'Workflows', 'Autoscaling policies', 'Serverless', 'Batches', 'Interactive', 'Interactive Templates', and 'Release Notes'. The main content area is titled 'Job details' and shows the following information:

- Job ID:** job-aab7cdd1
- Job UUID:** afb04968-6cde-418c-8209-be1ee4f2d168
- Type:** Dataproc Job
- Status:** Succeeded

Below the job details, there is an 'Output' section with a 'LINE WRAP: OFF' toggle. The output shows a series of log messages from the Spark environment, including:

- 24/12/01 19:03:57 INFO SparkEnv: Registering MapOutputTracker
- 24/12/01 19:03:57 INFO SparkEnv: Registering BlockManagerMaster
- 24/12/01 19:03:57 INFO SparkEnv: Registering BlockManagerMasterHeartbeat
- 24/12/01 19:03:57 INFO SparkEnv: Registering OutputCommitCoordinator
- 24/12/01 19:03:58 INFO MetricsConfig: Loaded properties from hadoop-metrics2.properties
- 24/12/01 19:03:58 INFO MetricsSystemImpl: Scheduled Metric snapshot period at 10 second(s).
- 24/12/01 19:03:58 INFO MetricsSystemImpl: google-hadoop-file-system metrics system started
- 24/12/01 19:03:58 INFO DataprocSparkPlugin: Registered 188 driver metrics
- 24/12/01 19:03:59 INFO DefaultNoHARMPFailoverProxyProvider: Connecting to ResourceManager at cluster-aeb5-m.us-central1-f.c.total-earth-443208
- 24/12/01 19:03:59 INFO AHSProxy: Connecting to Application History server at cluster-aeb5-m.us-central1-f.c.total-earth-443208-i3.internal./1
- 24/12/01 19:04:00 INFO Configuration: resource-types.xml not found
- 24/12/01 19:04:00 INFO ResourceUtils: Unable to find 'resource-types.xml'.
- 24/12/01 19:04:00 INFO YarnClientImpl: Submitted application application_1732931039425_0052
- 24/12/01 19:04:01 INFO DefaultNoHARMPFailoverProxyProvider: Connecting to ResourceManager at cluster-aeb5-m.us-central1-f.c.total-earth-443208

The bottom of the screenshot shows a Windows taskbar with various application icons and a system tray indicating the time as 2:11 PM on 12/1/2024.

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:

```

20 # -----Data Preprocessing-----
21 # Limiting Data for Banana Buyers Only
22 banana_product_id = products.filter(col("product_name") == "Banana").select("product_id").first()["product_id"]
23 banana_buyers = order_products.join(orders, "order_id") \
24     .filter(col("product_id") == banana_product_id) \
25     .select("user_id").distinct()
26
27 filtered_orders = orders.join(banana_buyers, "user_id")
28 filtered_order_products = order_products.join(filtered_orders, "order_id")
29
30 # Aggregate User-Aisle Interactions
31 filtered_order_products = filtered_order_products.join(products, "product_id").join(aisles, "aisle_id")
32 user_aisle_interactions = filtered_order_products.groupBy("user_id", "aisle_id") \
33     .agg(count("*").alias("interaction_strength"))
34
35 # Normalize Interaction Strengths
36 interaction_stats = user_aisle_interactions.agg(
37     min("interaction_strength").alias("min_interaction"),
38     max("interaction_strength").alias("max_interaction")
39 ).collect()
40
41 min_interaction = interaction_stats[0]["min_interaction"]
42 max_interaction = interaction_stats[0]["max_interaction"]
43
44 normalized_user_aisle_interactions = user_aisle_interactions.withColumn(
45     "normalized_strength",
46     (col("interaction_strength") - min_interaction) / (max_interaction - min_interaction)
47 )
48

```

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

toolsforai - Bucket details - Cl... job-aab7cdd1 - Monitoring - 2024f-T3_AISC2012.01: Tools

console.cloud.google.com/dataproc/jobs/job-aab7cdd1/monitoring?job=job-aab7cdd1®ion=us-central1&authuser=1&hl=en&project=total-earth-443208-i3

Google Cloud My First Project dataproc

Dataproc Job details CLONE DELETE STOP REFRESH

Jobs on Clusters Clusters Jobs Workflows Autoscaling policies Serverless Batches Interactive Interactive Templates Release Notes

Output LINE WRAP: OFF

Spark jobs take ~60 seconds to initialize resources. DISMISS

```
24/12/01 19:04:01 INFO DefaultNoHARMFaloverProxyProvider: Connecting to ResourceManager at cluster-aab5-m.us-central1-f.c.total-earth-443208
24/12/01 19:04:03 INFO GhfsGlobalStorageStatistics: periodic connector metrics: {gcs_api_client_non_found_response_count=1, gcs_api_client_si
24/12/01 19:04:03 INFO GoogleCloudStorageImpl: Ignoring exception of type GoogleJsonResponseException; verified object already exists with de
24/12/01 19:04:03 INFO GoogleHadoopOutputStream: hflush(): No-op due to rate limit (RateLimiter[stableRate=0.2qps]): readers will *not* yet s
24/12/01 19:04:09 INFO RequestTracker: Detected high latency for [url=https://storage.googleapis.com/storage/v1/b/dataproc-temp-us-central1-6
24/12/01 19:05:05 INFO GoogleHadoopOutputStream: hflush(): No-op due to rate limit (RateLimiter[stableRate=0.2qps]): readers will *not* yet s
```

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

```
24/12/01 19:06:16 INFO GoogleHadoopOutputStream: hflush(): No-op due to rate limit (RateLimiter[stableRate=0.2qps]): readers will *not* yet s
Training runtime: 41.80 seconds
/usr/lib/spark/python/lib/pyspark.zip/pyspark/sql/context.py:158: FutureWarning: Deprecated in 3.0.0. Use SparkSession.builder.getOrCreate()
24/12/01 19:07:18 INFO GoogleHadoopOutputStream: hflush(): No-op due to rate limit (RateLimiter[stableRate=0.2qps]): readers will *not* yet s
```

32°F Mostly cloudy 2:11 PM 12/1/2024

Model Training

Trained an ALS model using the interaction data.

```

53 # Prepare data in RDD format for ALS modeling
54 als_data_raw = user_aisle_interactions.rdd.map(
55     lambda row: Rating(row["user_id"], row["aisle_id"], row["interaction_strength"])
56 )
57
58 # Data Splitting
59 train_data_raw, test_data_raw = als_data_raw.randomSplit([0.8, 0.2], seed=42)
60
61 # -----Model Training-----
62
63 start_time = time.time()
64
65 als_model = ALS.train(
66     train_data_raw,
67     rank=20,
68     iterations=20,
69     lambda_=0.01
70 )
71
72 end_time = time.time()
73
74 # Calculate runtime
75 training_runtime = end_time - start_time
76 print(f"Training runtime: {training_runtime:.2f} seconds")
77

```

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

```

78 # -----Model Evaluation Strong and Weak Generalization-----
79
80 unique_users = user_aisle_interactions.select("user_id").distinct()
81 _, test_users = unique_users.randomSplit([0.8, 0.2], seed=42)
82 test_users_df = test_users
83 test_data_strong = user_aisle_interactions.join(test_users_df, "user_id")
84
85 recommendations = als_model.recommendProductsForUsers(5)
86
87 # Weak Generalization
88 test_data_weak = test_data_raw
89 predicted_ranking_weak = recommendations.mapValues(lambda recs: [rec.product for rec in recs])
90 actual_ranking_weak = test_data_weak.map(lambda x: (x.user, int(x.product))).groupByKey().mapValues(list)
91 formatted_ranking_weak = predicted_ranking_weak.join(actual_ranking_weak).map(lambda x: (x[1][0], x[1][1]))
92
93 metrics_weak = RankingMetrics(formatted_ranking_weak)
94 map_at_5_weak = metrics_weak.meanAveragePrecisionAt(5)
95 recall_at_5_weak = metrics_weak.recallAt(5)
96
97 # Strong Generalization
98 predicted_ranking_strong = recommendations.mapValues(lambda recs: [rec.product for rec in recs])
99 actual_ranking_strong = test_data_strong.rdd.map(lambda x: (x["user_id"], int(x["aisle_id"]))).groupByKey().mapValues(list)
100 formatted_ranking_strong = predicted_ranking_strong.join(actual_ranking_strong).map(lambda x: (x[1][0], x[1][1]))
101
102 metrics_strong = RankingMetrics(formatted_ranking_strong)
103 map_at_5_strong = metrics_strong.meanAveragePrecisionAt(5)
104 recall_at_5_strong = metrics_strong.recallAt(5)
105
106 # Print Generalization Results
107
108 # Print Generalization Results
109 print("Weak Generalization - MAP@5:", map_at_5_weak)
110 print("Weak Generalization - Recall@5:", recall_at_5_weak)
111 print("Strong Generalization - MAP@5:", map_at_5_strong)
112 print("Strong Generalization - Recall@5:", recall_at_5_strong)
113
114 # -----Model Evaluation for 1000 Random Users-----
115
116 random_users_sample = test_data_raw.map(lambda x: x.user).distinct().takeSample(False, 1000, seed=42)
117 random_users_sample_broadcast = spark.sparkContext.broadcast(set(random_users_sample))
118
119 # Filter for Random Users
120 filtered_recommendations_sample = recommendations.filter(lambda x: x[0] in random_users_sample_broadcast.value)
121 filtered_test_data_sample = test_data_raw.filter(lambda x: x.user in random_users_sample_broadcast.value)
122
123 predicted_ranking_sample = filtered_recommendations_sample.mapValues(lambda recs: [rec.product for rec in recs])
124 actual_ranking_sample = filtered_test_data_sample.map(lambda x: (x.user, int(x.product))).groupByKey().mapValues(list)
125 formatted_ranking_sample = predicted_ranking_sample.join(actual_ranking_sample).map(lambda x: (x[1][0], x[1][1]))
126
127 metrics_sample = RankingMetrics(formatted_ranking_sample)
128 map_at_5_sample = metrics_sample.meanAveragePrecisionAt(5)
129 recall_at_5_sample = metrics_sample.recallAt(5)
130
131 # Print Results
132 print("1000 Random Users - MAP@5:", map_at_5_sample)
133 print("1000 Random Users - Recall@5:", recall_at_5_sample)

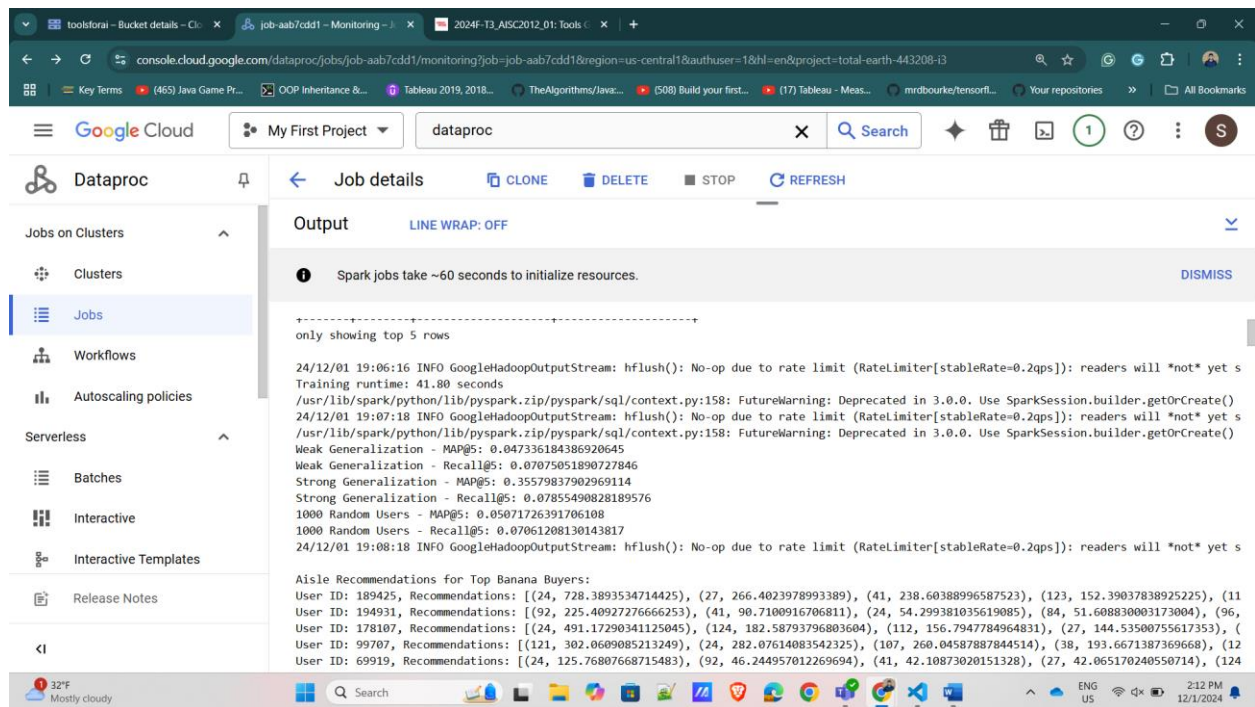
```

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:


```

133 # -----Aisle Recommendations for Top Bannana Buyers-----
134
135 top_banana_buyers = filtered_order_products.filter(col("product_id") == banana_product_id) \
136     .groupBy("user_id") \
137     .count() \
138     .orderBy(col("count").desc()) \
139     .limit(5)
140
141 top_banana_buyers_ids = [row["user_id"] for row in top_banana_buyers.collect()]
142 top_banana_buyers_broadcast = spark.sparkContext.broadcast(top_banana_buyers_ids)
143
144 recommendations_top_users = recommendations.filter(lambda x: x[0] in top_banana_buyers_broadcast.value).collect()
145
146 print("\nAisle Recommendations for Top Banana Buyers:")
147 if not recommendations_top_users:
148     print("No Recommendations Found for Top Banana Buyers")
149 else:
150     for user_id, recs in recommendations_top_users:
151         recommendations_list = [(r.product, r.rating) for r in recs]
152         print(f"User ID: {user_id}, Recommendations: {recommendations_list}")
153
154 recommendations_top_users_df = spark.createDataFrame([
155     (user_id, r.product, r.rating)
156     for user_id, recs in recommendations_top_users
157     for r in recs
158 ], schema=["user_id", "aisle_id", "score"])
159
160 # Join recommendations with aisle names
161
162 recommendations_with_names_df = recommendations_top_users_df.join(
163     aisles.withColumnRenamed("aisle_id", "aisle_key"),
164     recommendations_top_users_df["aisle_id"] == col("aisle_key"),
165     "left"
166 ).select(
167     "user_id",
168     "aisle_id",
169     "score",
170     "aisle"
171 )
172
173 print("Aisle Recommendations for Top Banana Buyers:")
174 recommendations_with_names_df.show(25, truncate=False)
175
176 recommendations_with_names_df = recommendations_with_names_df.limit(10000) # Limit to 10,000 rows
177
178 output_path = "gs://toolsforai//recommendations_with_aisle_names.csv"
179
180 # Save the recommendations
181 recommendations_with_names_df.write.csv(output_path, header=True, mode="overwrite")
182 print(f"Results saved to {output_path}")
183
184 #-----

```

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, 42.06517024050714), (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

Output [CLONE](#) [DELETE](#) [STOP](#) [REFRESH](#)

Spark jobs take ~60 seconds to initialize resources. [DISMISS](#)

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, 382.069085213249), (24, 282.07614083542325), (107, 260.04587807844514), (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

Output is complete

[EQUIVALENT COMMAND LINE](#)

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)
```

```
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)

```

```

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 Banana 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}")

#-----

```

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}")

```

```
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:

Common users: 178624, Common products: 47848

Average RMSE across all folds: 0.46289136012733634

The screenshot displays the Google Cloud Dataproc console interface. The top navigation bar includes the Google Cloud logo, "My First Project", a search bar, and utility icons. The left sidebar shows a menu with options like Clusters, Jobs, Workflows, Autoscaling policies, Services, Batches, Interactive, Interactive Templates, Metastore Services, Metastore, Federation, Utilities, Component exchange, Workbench, Dataproc on GDC, Release notes, and AI.

The main content area is titled "Jobs" and contains several action buttons: SUBMIT JOB, REFRESH, STOP, DELETE, REGIONS, and +3 RECOMMENDED ALERTS. A message states: "If Dataproc can't decrypt CMXK ID-enabled job parameters, the job is not listed in the table." Below this is a filter section labeled "Filter jobs".

<input type="checkbox"/>	Job ID	Status	Region	Type	Cluster	Start time	Elapsed time	Labels
<input type="checkbox"/>	job-bedotdec	Succeeded	us-central1	Pyspark	pyspark	Dec 1, 2024, 1:05:24 PM	1 hr 53 sec	None
<input type="checkbox"/>	job-elec3fmc	Canceled	us-central1	Pyspark	pyspark	Dec 1, 2024, 12:51:23 PM	13 min 42 sec	None
<input type="checkbox"/>	rolling-cv	Failed	us-central1	Pyspark	pyspark	Dec 1, 2024, 10:59:30 AM	8 min 49 sec	None

The right sidebar shows "No jobs selected" with tabs for PERMISSIONS and LABELS. A message below reads: "Please select at least one resource." At the bottom right, there is an "Activate Windows" watermark.

Google Cloud

My First Project

Search (/) for resources, docs, products, and more

Search

Dataproc

Jobs on Clusters

Clusters

Jobs

Workflows

Autoscaling policies

Serverless

Batches

Interactive

Interactive Templates

Metastore Services

Metastore

Federation

Utilities

Component exchange

Workbench

Dataproc on GDC

Service instances

Job details

CLONE

DELETE

STOP

REFRESH

Job ID

Job UUID

Type

Status

MONITORING

CONFIGURATION

EDIT

Start time

Elapsed time

Status

Region

Cluster

Job type

Main python file

Labels

Performance Enhancements

Advanced optimizations

Advanced execution layer

Output

LINE WRAP OFF

Release Notes

Spark jobs take ~60 seconds to initialize resources.

24/12/01 18:05:11 INFO SearchEnv: registering mapoutputtracker

24/12/01 18:05:11 INFO SearchEnv: registering elasticsearchmaster

Activate Windows

Go to Settings to activate Windows.

DISMISS

Type here to search

21°C Mostly cloudy

3:01 PM 2024-12-01

Average RMSE across all folds: 0.46289136012733634