1. SparkML

Spark ML is the DataFrame-based API of Apache Spark's machine-learning stack. It standardizes ML workflows around *Pipelines*, where *Transformers* apply deterministic DataFrame-to-DataFrame transformations and *Estimators* learn parameters from data to produce Transformers (models). Some most used operators are *Evaluators*, *CrossValidator*, and *TrainValidationSplit*, users can handle feature preparation, model fitting, tuning, and scoring as a reproducible DAG that scales across a cluster. Since the API is native to DataFrames, it integrates tightly with Spark SQL and can serve both batch and streaming inference within the same application.

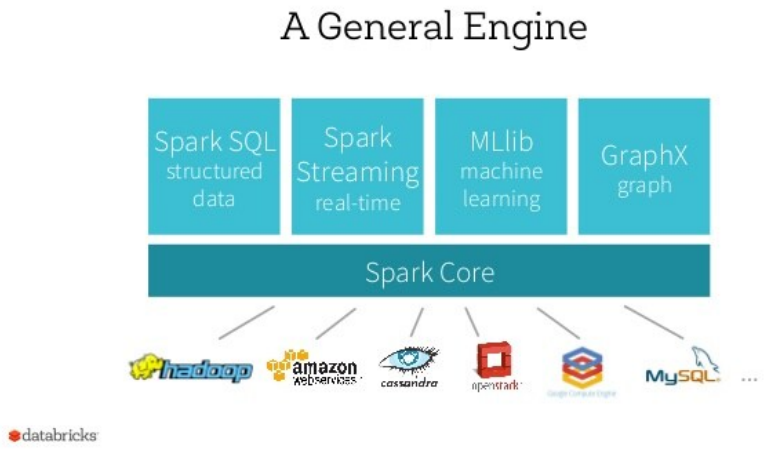

Figure 1: Components of Spark Core.

The library covers end-to-end needs: featurization (tokenization, hashing, string indexing, one-hot encoding, scaling, PCA/feature selection), a wide range of algorithms (classification, regression, clustering, and collaborative filtering with ALS), pipeline utilities (grid search, persistence), and model lifecycle (save/load of models/pipelines).

Practically, a typical flow assembles features with components such as `StringIndexer`, `OneHotEncoder`, `VectorAssembler`, and `StandardScaler`; trains with linear models or tree-based ensembles; evaluates with built-in metrics; and persists a `PipelineModel` for production reuse.

Historically, Spark also shipped an RDD-based MLlib API; today it is in *maintenance mode*, while the DataFrame-based API is the recommended path for new workloads. The ecosystem further extends Spark ML with popular learners such as XGBoost (via XGBoost4J-Spark), allowing teams to plug state-of-the-art algorithms into the same Pipeline abstraction and scale training/inference across executors. In short, Spark ML unifies data preparation, training, tuning, and deployment on a single distributed engine with consistent, SQL-friendly semantics.

2. SparkML – a set of plug-and-play executors

To train a model from a bunch of data, you must tell Spark how you manipulate your data.

- Getting the data (explained in previous labs).
- Pre-processing the data.
- Training and evaluate the model.

Spark will try to do so to the data iteratively $\longrightarrow$ follows a pipeline pattern:

- Get the data (spark.read.{csv, parquet, format("kafka"), etc.}, your choice).
- Pre-process the data (*i.e.*, pyspark.sql.functions.<your-chosen-function>), and prepare the data/extract features (*i.e.*, pyspark.ml.feature.<your-chosen-function> where <your-chosen-function> are mostly VectorAssembler or StandardScaler).
- Declare a model (*i.e.*, pyspark.ml.{classification, clustering, recommendation}.<your-chosen-algorithm>)

Then put them all into a pyspark.ml.Pipeline object and run.

3. TUTORIAL

Let's do some tutorial to get familiar with SparkML.

1. Importing necessary library.

Code 1: Necessary libraries.

```
from pyspark.sql import SparkSession
from pyspark.sql.types import *
from pyspark.ml import Pipeline
from pyspark.ml.feature import VectorAssembler, StandardScaler, StringIndexer
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.clustering import KMeans
from pyspark.ml.recommendation import ALS
from pyspark.ml.evaluation import BinaryClassificationEvaluator, ClusteringEvaluator, RegressionEvaluator
from pyspark.sql.functions import *
import pandas as pd
from datetime import datetime, timedelta
import numpy as np
```

2. Initializing Spark for standalone cluster.

Code 2: Spark Session Initialization.

```
spark = (SparkSession.builder
    .master("spark://10.1.11.5:7077")
    .appName("Lab4")
    .config("spark.sql.adaptive.enabled", "true")
    .config("spark.sql.adaptive.coalescePartitions.enabled", "true")
    .getOrCreate()
)
```

3. Generating enhanced dataset with realistic ML features and DataFrame for each dataset.

Code 3: Generate data for User.

```
# Generate data
users_data = [
    (1, 25, "low", "NYC", "2023-01-15", "evening", "mobile"),
    (2, 34, "medium", "LA", "2023-02-20", "morning", "desktop"),
    (3, 45, "high", "Chicago", "2023-01-10", "afternoon", "mobile"),
    (4, 28, "medium", "NYC", "2023-03-01", "evening", "tablet"),
    (5, 52, "high", "LA", "2023-01-05", "morning", "desktop"),
    (6, 31, "low", "Chicago", "2023-02-15", "afternoon", "mobile"),
    (7, 38, "medium", "NYC", "2023-01-20", "evening", "desktop"),
    (8, 29, "low", "LA", "2023-02-25", "morning", "mobile"),
    (9, 33, "high", "Chicago", "2023-01-12", "afternoon", "tablet"),
    (10, 27, "medium", "NYC", "2023-02-10", "morning", "mobile")
]
# Create dataframe
users_df = spark.createDataFrame(users_data,
    ["user_id", "age", "income_bracket", "location", "signup_date", "preferred_time", "device"])
```

Code 4: Generate data for Item.

```
# Generate data
items_data = [
    ("apple", "fruit", 1.50, 8.5, 9, 7, "west"),
    ("banana", "fruit", 0.80, 9.0, 8, 5, "south"),
    ("orange", "fruit", 1.20, 7.5, 9, 10, "west"),
    ("spinach", "vegetable", 2.00, 6.0, 10, 3, "north"),
    ("carrot", "vegetable", 1.10, 7.0, 8, 14, "north"),
    ("bread", "grain", 2.50, 3.0, 4, 3, "local"),
    ("milk", "dairy", 3.20, 4.0, 7, 7, "local"),
    ("cheese", "dairy", 4.50, 2.0, 6, 21, "local"),
    ("rice", "grain", 3.00, 1.0, 5, 365, "south"),
    ("chicken", "protein", 8.00, 5.0, 8, 3, "north")
]
# Create dataframe
items_df = spark.createDataFrame(items_data,
    ["item", "category", "price", "seasonality", "health_rating", "shelf_life", "supplier_region"])
```

Code 5: Generate data for Transaction.

```
# Generate data
transactions_data = [
    ("2025-09-01 10:30", 1, "apple", 2, "store_A", 0.0, "Monday", 10, "sunny", "sess_1"),
    ("2025-09-01 10:32", 1, "banana", 1, "store_A", 0.0, "Monday", 10, "sunny", "sess_1"),
    ("2025-09-01 14:15", 2, "orange", 3, "store_B", 0.1, "Monday", 14, "cloudy", "sess_2"),
    ("2025-09-01 19:20", 3, "spinach", 1, "store_A", 0.0, "Monday", 19, "rainy", "sess_3"),
    ("2025-09-02 09:45", 4, "apple", 1, "store_C", 0.05, "Tuesday", 9, "sunny", "sess_4"),
    ("2025-09-02 09:47", 4, "carrot", 2, "store_C", 0.05, "Tuesday", 9, "sunny", "sess_4"),
    ("2025-09-02 16:30", 5, "bread", 1, "store_B", 0.15, "Tuesday", 16, "cloudy", "sess_5"),
    ("2025-09-03 11:00", 6, "milk", 1, "store_A", 0.0, "Wednesday", 11, "sunny", "sess_6"),
    ("2025-09-03 18:45", 7, "banana", 2, "store_C", 0.0, "Wednesday", 18, "rainy", "sess_7"),
    ("2025-09-03 18:50", 7, "apple", 1, "store_C", 0.0, "Wednesday", 18, "rainy", "sess_7"),
    ("2025-09-04 08:30", 8, "orange", 1, "store_B", 0.2, "Thursday", 8, "sunny", "sess_8"),
    ("2025-09-04 15:20", 1, "spinach", 1, "store_A", 0.0, "Thursday", 15, "cloudy", "sess_9"),
    ("2025-09-05 12:15", 2, "carrot", 1, "store_C", 0.1, "Friday", 12, "rainy", "sess_10"),
    ("2025-09-05 13:20", 3, "cheese", 1, "store_B", 0.0, "Friday", 13, "sunny", "sess_11"),
    ("2025-09-06 10:10", 9, "rice", 2, "store_A", 0.05, "Saturday", 10, "cloudy", "sess_12"),
    ("2025-09-06 14:45", 10, "chicken", 1, "store_C", 0.1, "Saturday", 14, "rainy", "sess_13"),
    ("2025-09-07 09:30", 5, "apple", 3, "store_B", 0.0, "Sunday", 9, "sunny", "sess_14"),
    ("2025-09-07 16:20", 6, "banana", 1, "store_A", 0.0, "Sunday", 16, "cloudy", "sess_15"),
    ("2025-09-08 11:15", 8, "spinach", 2, "store_C", 0.15, "Monday", 11, "rainy", "sess_16"),
    ("2025-09-08 19:00", 9, "milk", 1, "store_B", 0.0, "Monday", 19, "sunny", "sess_17")
]
# Create dataframe
transactions_df = spark.createDataFrame(transactions_data,
    ["timestamp", "user_id", "item", "quantity", "store", "discount", "day_of_week", "hour", "weather", "session_id"])
```

Let's resolve the following problems.

Problem 1: Classification - Will User Purchase Next Week? Predict whether a user will make a purchase in the next 7 days based on their historical behavior, demographics, and recent activity patterns.

- Check pandas memory usage just for feature engineering.

Code 6: Do the traditional Spark's way.

```
users_pd = users_df.toPandas()
transactions_pd = transactions_df.toPandas()

user_features_pd = transactions_pd.groupby('user_id').agg({
    'quantity': ['sum', 'mean', 'count'],
    'hour': ['mean', 'std'],
    'session_id': 'nunique'
}).reset_index()
# Flatten column names.
\begin{lstlisting}[
language=Python,
caption={Generating data for Transaction.}
]
user_features_pd.columns = ['user_id', 'total_qty', 'avg_qty', 'purchase_count',
                            'avg_hour', 'hour_std', 'session_count']
```

Code 7: Check memory usage.

```
# Join with user data.
final_pd = users_pd.merge(user_features_pd, on='user_id', how='left').fillna(0)

# Memory usage check.
\begin{lstlisting}[
language=Python,
caption={Generate data for Transaction.}
]
memory_kb = final_pd.memory_usage(deep=True).sum() / 1024
print(f"Pandas memory usage: {memory_kb:.2f} KB")
print(f"With 10M users + 20M transactions (Uber: 36M daily transactions): {memory_kb * 10**6:.0f} KB = {memory_kb * 10**6 / 1024**2:.1f
  } GB")
```

- Feature engineering using Spark distributed processing.

Code 8: Function for creating user features.

```
def create_user_features_spark():
    # User purchase behavior features - distributed across partitions
    user_stats = transactions_df.groupBy("user_id").agg(           # For each user
        count("*").alias("total_purchases"),                       # Get total number of purchases (transaction)
        sum("quantity").alias("total_quantity"),                   # Total quantity purchased
        count_distinct("item").alias("unique_items"),              # Unique items purchased, one-hot encode later
        avg("hour").alias("avg_purchase_hour"),                    # Average hour of purchase
        stddev("hour").alias("hour_variance"),                     # Variance in purchase hour
        count_distinct("session_id").alias("total_sessions"),      # Unique sessions
        avg("discount").alias("avg_discount")                      # Average discount used
    )

    # Recent activity features (distributed window operations)
    recent_cutoff = "2025-09-05"
    recent_activity = (transactions_df.filter(col("timestamp") >= recent_cutoff)    # In the recent week
        .groupBy("user_id")                                                         # For each user
        .agg(count("*").alias("recent_purchases"),                                  # Get number of purchases
             sum("quantity").alias("recent_quantity")))                             # Total item quantity

    # Category preference features
    category_prefs = (transactions_df.join(items_df, "item")    # Join to get categories
        .groupBy("user_id", "category")                         # For each user-category pair
        .agg(sum("quantity").alias("cat_quantity"))             # Get total quantity per category
        .groupBy("user_id")                                     # For each user
        .pivot("category")                                      # Pivot on category
        .sum("cat_quantity")                                    # Sum quantities per category
        .fillna(0))

    # Join all features together - distributed joins
    user_features = (users_df.join(user_stats, "user_id", "left")   # Left-join user demographics
```

```
            .join(recent_activity, "user_id", "left")                    # Left-join recent activity
            .join(category_prefs, "user_id", "left")                     # Left-join category preferences
            .fillna(0))

        return user_features
```

- Creating target variable based on patterns in data.

Code 9: Generate data features and function for creating labels.

```
def create_target_labels(user_features):
    # Target: will purchase next week (based on recent activity and patterns). Assumption: users with recent purchases or high overall
  purchases are likely to purchase next week
    return user_features.withColumn("will_purchase",            # Create target column
        when((col("recent_purchases") > 1) |                    # If recent purchases > 1
             (col("total_purchases") > 3) |                     # And total purchases > 3
             (col("fruit") > 2), 1).otherwise(0))               # And fruit purchases > 2, then 1 else 0

# Execute feature engineering
user_features = create_user_features_spark()
labeled_data = create_target_labels(user_features)
```

- Showing feature engineering results.

Code 10: Create user features and labels.

```
# Execute feature engineering
user_features = create_user_features_spark()
labeled_data = create_target_labels(user_features)

# Show feature engineering results
print("Feature engineering completed - distributed processing")
labeled_data.select("user_id", "total_purchases", "recent_purchases", "fruit", "will_purchase").show()
```

```
Feature engineering completed - distributed processing
+-------+---------------+----------------+-----+-------------+
|user_id|total_purchases|recent_purchases|fruit|will_purchase|
+-------+---------------+----------------+-----+-------------+
|      5|              2|               1|    3|            1|
|     10|              1|               1|    0|            0|
|      3|              2|               1|    0|            0|
|      8|              2|               1|    1|            0|
|      4|              2|               0|    1|            0|
|      2|              2|               1|    3|            1|
|      1|              3|               0|    3|            1|
|      7|              2|               0|    3|            1|
|      9|              2|               2|    0|            1|
|      6|              2|               1|    1|            0|
+-------+---------------+----------------+-----+-------------+
```

Figure 2: Components of Spark Core.

- Building classification pipeline.

Code 11: Create a pipeline consisting of 3 indexers, 1 assembler, 1 scaler, and 1 classifier.

```
# StringIndexer: Convert string labels (categorical) to numbers
location_indexer = StringIndexer(inputCol="location", outputCol="location_idx", handleInvalid="keep")
income_indexer = StringIndexer(inputCol="income_bracket", outputCol="income_idx", handleInvalid="keep")
device_indexer = StringIndexer(inputCol="device", outputCol="device_idx", handleInvalid="keep")

# Select numerical features for the model
feature_cols = ["age", "location_idx", "income_idx", "device_idx",
                "total_purchases", "total_quantity", "unique_items",
                "recent_purchases", "fruit", "vegetable", "dairy"]
"""VectorAssembler
feat1 | feat2 | feat3
0       1       true
becomes
feat1 | feat2 | feat3 | feat_vect
0       1       true    [0,1,true]
Spark models require a single column of features as a vector (weird huh?)
"""
assembler = VectorAssembler(inputCols=feature_cols, outputCol="features_raw", handleInvalid="skip")
scaler = StandardScaler(inputCol="features_raw", outputCol="features")                              # Standardize features around their
   mean, scaled to unit variance
classifier = LogisticRegression(featuresCol="features", labelCol="will_purchase", maxIter=10)      # A SparkML model requires col of
  features (made with VectorAssembler) + col of labels

# Create pipeline, data flows through each stage iteratively
classification_pipeline = Pipeline(stages=[
    location_indexer, income_indexer, device_indexer,
    assembler, scaler, classifier
])
```

- Training model.

Code 12: Define train data and test data ratio.

```
# Train-test split
train_data, test_data = labeled_data.randomSplit([0.7, 0.3], seed=42)

# Train model
classification_model = classification_pipeline.fit(train_data)
```

- Making predictions.

Code 13: Train the model.

```
predictions = classification_model.transform(test_data)
```

```
+-------+-------------+----------+--------------------+
|user_id|will_purchase|prediction|         probability|
+-------+-------------+----------+--------------------+
|      3|            0|       0.0|[0.99999201117049...|
|      7|            1|       1.0|[0.02143435369988...|
|      9|            1|       0.0|[0.99482614470013...|
|     10|            0|       0.0|[0.99999999287099...|
+-------+-------------+----------+--------------------+
```

Figure 3: Components of Spark Core.

- Evaluation.

Code 14: Use Binary Classification Evaluator and calculate AUC.

```
evaluator = BinaryClassificationEvaluator(labelCol="will_purchase")
auc = evaluator.evaluate(predictions)

print(f"Classification AUC: {auc:.3f}")
predictions.select("user_id", "will_purchase", "prediction", "probability").show()
```

Problem 2: Customer Segmentation Clustering – Creating Behavior Vectors for Clustering

Firstly, we create comprehensive user behavior profiles for clustering with the following demands.

- Purchase patterns: item preferences (pivot table of items).
- Temporal behavior: avg purchase hour, variance, active days, avg discount used, avg quantity per purchase, total purchases.
- Store preferences: pivot table of stores.
- Join with user demographics.
- Fill missing values with 0.
- Return final user behavior vectors DataFrame.

Code 15: Create a function that generates a user feature vectors.

```
def create_user_behavior_vectors():
    # Purchase pattern features - item preferences as pivot
    user_item_matrix = (transactions_df.groupBy("user_id", "item")  # For each user-item pair
        .agg(sum("quantity").alias("total_qty"))                    # Get total quantity per item
        .groupBy("user_id")                                         # For each user
        .pivot("item")                                              # Pivot on item
        .sum("total_qty")                                           # Sum quantities per item
        .fillna(0))

    # Temporal behavior patterns
    temporal_features = transactions_df.groupBy("user_id").agg(     # For each user
        avg("hour").alias("avg_hour"),                              # Get average hour of purchase
        stddev("hour").alias("hour_variance"),                      # Variance in purchase hour
        count_distinct(col("day_of_week")).alias("active_days"),    # Number of unique active days
        avg("discount").alias("avg_discount_used"),                 # Average discount used
        avg("quantity").alias("avg_quantity_per_purchase"),         # Average quantity per purchase
        count("*").alias("total_purchases")                         # Total number of purchases
    ).fillna(0)

    # Store preference features
    store_preferences = (transactions_df.groupBy("user_id")         # For each user
        .pivot("store")                                             # Pivot on store
        .count()                                                    # Count purchases per store
        .fillna(0)
    )
    # Join demographics with behavioral features
    user_vectors = (users_df.join(user_item_matrix, "user_id")      # Join user demographics
        .join(temporal_features, "user_id")                         # Join temporal features
        .join(store_preferences, "user_id")                         # Join store preferences
        .fillna(0))

    return user_vectors
```

Secondly, we execute behavior vector creation.

Code 16: Generating data for Transaction.

```
user_behavior_data = create_user_behavior_vectors() # Feature extraction
user_behavior_data.show()
```

Next, we prepare features for clustering (exclude non-numeric columns).

Code 17: Same as above.

```
clustering_feature_cols = [col for col in user_behavior_data.columns
                           if col not in ["user_id", "location", "income_bracket",
                                          "signup_date", "preferred_time", "device"]]

clustering_assembler = VectorAssembler(inputCols=clustering_feature_cols,
                                       outputCol="features_raw", handleInvalid="skip")
clustering_scaler = StandardScaler(inputCol="features_raw", outputCol="features")
```

Initializing K-means clustering operator.

Code 18: Call the K-Means classifier operator.

```
kmeans = KMeans(featuresCol="features", predictionCol="cluster", k=3, seed=42)
```

Analyzing cluster characteristics

Code 19: Get insights from cluster.

```
cluster_analysis = clustered_users.groupBy("cluster").agg(          # For each cluster
    count("*").alias("cluster_size"),                               # Get cluster size
    avg("age").alias("avg_age"),                                    # Average age
    avg("apple").alias("avg_apple_purchases"),                      # Average apple purchases
    avg("avg_hour").alias("avg_purchase_hour"),                     # Average purchase hour
    avg("total_purchases").alias("avg_total_purchases")             # And total purchases
)

print("Cluster Analysis Results:")
cluster_analysis.show()
```

Finally, evaluating clustering quality.

Code 20: Evaluate the clustering model.

```
clustering_evaluator = ClusteringEvaluator(featuresCol="features", predictionCol="cluster")
silhouette_score = clustering_evaluator.evaluate(clustered_users)
print(f"Clustering Silhouette Score: {silhouette_score:.3f}")       # This score is between -1 and 1, higher is better. Depicts how well-separated
      the clusters are.
```

```
Cluster Analysis Results:
+-------+------------+-------+------------------+------------------+------------------+
|cluster|cluster_size|avg_age|avg_apple_purchases| avg_purchase_hour|avg_total_purchases|
+-------+------------+-------+------------------+------------------+------------------+
|      1|           1|   52.0|               3.0|              12.5|               2.0|
|      2|           4|   33.5|               0.5|13.916666666666666|              2.25|
|      0|           5|   31.2|               0.4|              12.7|               1.8|
+-------+------------+-------+------------------+------------------+------------------+

Clustering Silhouette Score: 0.225
```

Figure 4: Expected results.

Problem 3: ALS Recommendation System – Prepare Recommendation Data.

Firstly, we create user-item ratings from transaction data.

Code 21: Create a function to prepare the dataset.

```
def prepare_recommendation_data():
    # Aggregate purchase data to create implicit ratings
    user_item_ratings = (transactions_df.groupBy("user_id", "item") # For each user-item pair
        .agg(                                                       # Get
            sum("quantity").alias("total_quantity"),                # Total quantity purchased
            count("*").alias("purchase_frequency"),                 # Number of purchases
            avg("discount").alias("avg_discount")                   # Average discount applied
        )
        .withColumn("rating",                                                                   # Also get a rating based on this formula (I made it
       up lol)
            col("total_quantity") * col("purchase_frequency") * (1 - col("avg_discount"))       # (quantity * frequency * (1 - avg_discount)
        )
        .select("user_id", "item", "rating"))

    # Convert item names to numeric IDs for ALS
    item_indexer = StringIndexer(inputCol="item", outputCol="item_id", handleInvalid="keep")
    indexed_ratings = item_indexer.fit(user_item_ratings).transform(user_item_ratings)

    return indexed_ratings.select("user_id", "item_id", "rating", "item")
```

Secondly, we execute recommendation data preparation.

Code 22: Generate the dataset.

```
recommendation_data = prepare_recommendation_data()
recommendation_data.show()
```

Initializing and training ALS model.

Code 23: Define the train and test data, and define the Alternating Least Square classifier.

```
# Split data for training and evaluation
train_rec, test_rec = recommendation_data.randomSplit([0.8, 0.2], seed=42)
# Configure ALS model for implicit feedback
als = ALS(
    userCol="user_id",
    itemCol="item_id",
    ratingCol="rating",
    nonnegative=True,
    implicitPrefs=True,  # Using implicit feedback from purchases
    rank=5,  # Latent factors (reduced for small dataset)
    maxIter=10,
    regParam=0.1,
    seed=42
)

# Train ALS model
als_model = als.fit(train_rec)
```

Generating recommendations.

Code 24: Get top bought items/top buying users.

```
user_recs = als_model.recommendForAllUsers(3)  # Top 3 items per user
item_recs = als_model.recommendForAllItems(2)  # Top 2 users per item

print("User Recommendations (Top 3 items per user):")
user_recs.show(truncate=False)

print("Item Recommendations (Top 2 users per item):")
item_recs.show(truncate=False)
```

Evaluating recommendation quality.

Code 25: Get recommendations.

```
predictions = als_model.transform(test_rec)
rec_evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating", predictionCol="prediction")
rmse = rec_evaluator.evaluate(predictions.filter(col("prediction").isNotNull()))

print(f"Recommendation RMSE: {rmse:.3f}")
```

Finally, showing specific user recommendations with item names.

Code 26: Get Item Recommendations for Users.

```
user_recs_with_names = user_recs.select("user_id", explode("recommendations").alias("rec")) \
    .select("user_id", "rec.item_id", "rec.rating") \
    .join(recommendation_data.select("item_id", "item").distinct(), "item_id") \
    .groupBy("user_id") \
    .agg(collect_list(struct("item", "rating")).alias("recommendations"))

print("User Recommendations with Item Names:")
user_recs_with_names.show(truncate=False)
```

4. EXERCISES

This exercises is leveraged to help students practice SparkML and create a Pipeline to transform data from scratch to a complete model. The requirements for this exercise are shown below.

Exercise 0: Prepare movie data. This is mandatory to do any of the exercises.

- Reading data from 3 topics, including:
  - Movies (with “Lab1_movies" as topic name (*i.e.*, subscribe))
  - Ratings (with “Lab1_ratings" as topic name (*i.e.*, subscribe))
  - Tags similar to ratings (with “Lab1_tags" as topic name (*i.e.*, subscribe))

  Kafka addresses are 10.1.1.10:30090, 10.1.1.27:30091, 10.1.1.203:30092.
- Defining schema for movies, ratings and tags. Subsequently, converting data type of the DataFrame to its corresponding schema.

Exercise 1: Create a binary classifier and answer the question *"Will a user rate a movie* $\geqslant 4$*?"*

- Build a Pipeline that predicts 1 if rating $\geqslant 4$ else 0, using:
  - Text: user-movie tags and movie title.
  - Categorical multi-label: genres_tokens.
  - Numerics: user/movies aggregates.
- Output:
  - AUC (primary), F1, and a 2×2 confusion matrix.
  - Top-10 positive & top-10 negative feature signals (words/genres)

Exercise 2: Create a clustering model to group movies by genres.

- Build a movie clustering model using tags (TF-IDF) + genres.
- Output:
  - Silhouette score for candidate K with K = 6, 8, 10, 12 (report all tried values).
  - For each K, print out the top 10 terms describing each cluster + sample top 10 movies per cluster.

Exercise 3: Create a recommendataion system using Alternating Least Square and recommend 10 films for 3 random users.

- Build an ALS recommender trained on ratings.
- Output:
  - RMSE on test dataset.
  - Precision at 10 recommendations (averaged across users with $\geqslant 1$ relevant item).
  - Print out the top 10 recommendations for 3 random users.

Submission: The works must be done in a group of at most 3. Due to the difficulty of these exercise, **students are required to only do one of the three exercises**. Additional work beyond the first one will be treated as a bonus to other Lab exercises. **All exercises will be graded by effort** rather than result.

- Compress all necessary files (sources and a report) as **CO3137_BigDataLab_Lab04_<Student ID-1>_<Student ID-2>_<Student ID-3>.zip**.
- Students must submit and demonstrate their results before the end of the class.
- Early submitters (within the lab class) get bonus points. The rest can still get a 10 if they deserve it comprehensively.