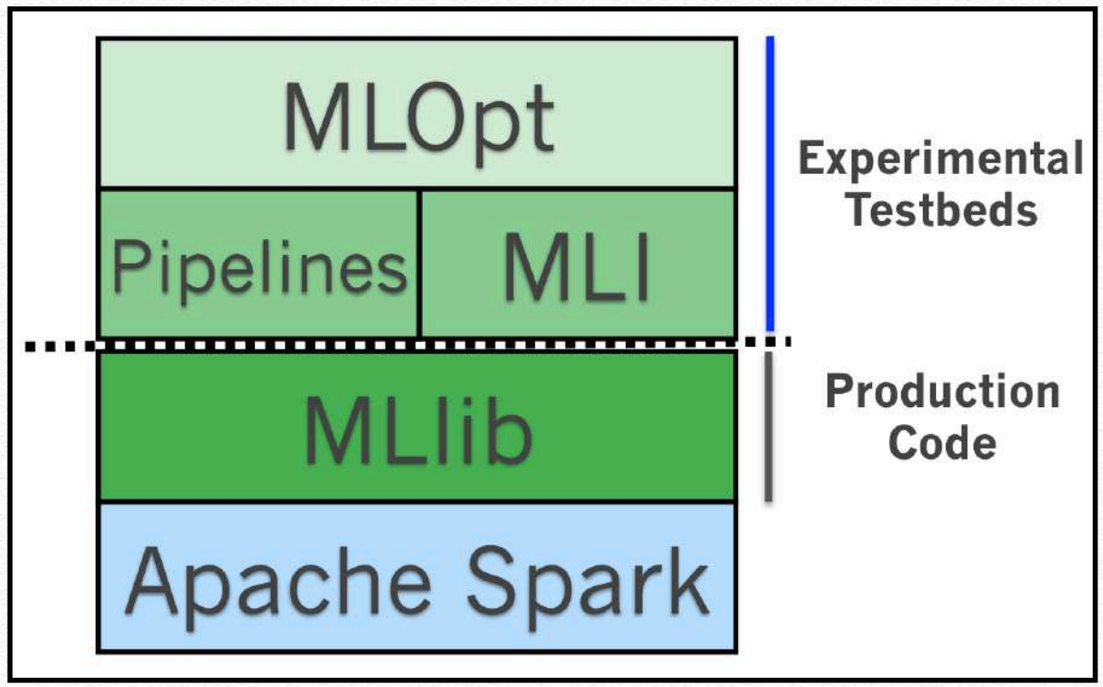
What is Spark MLlib

MLlib



- MLbase aims to simplify development and deployment of scalable ML pipelines
- MLlib: Spark's core ML library
- MLI, Pipelines: APIs to simplify ML development
  - Tables, Matrices, Optimization, ML Pipelines
- MLOpt: Declarative layer to automate hyperparameter tuning

What's in Spark MLlib

MLlib

- Alternating Least Squares
- Lasso
- Ridge Regression
- Logistic Regression
- Decision Trees
- Naïve Bayes
- Support Vector Machines
- K-Means
- Gradient descent
- L-BFGS
- Random data generation
- Linear algebra
- Feature transformations
- Statistics: testing, correlation
- Evaluation metrics

Collaborative Filtering for Recommendation

Prediction

Clustering

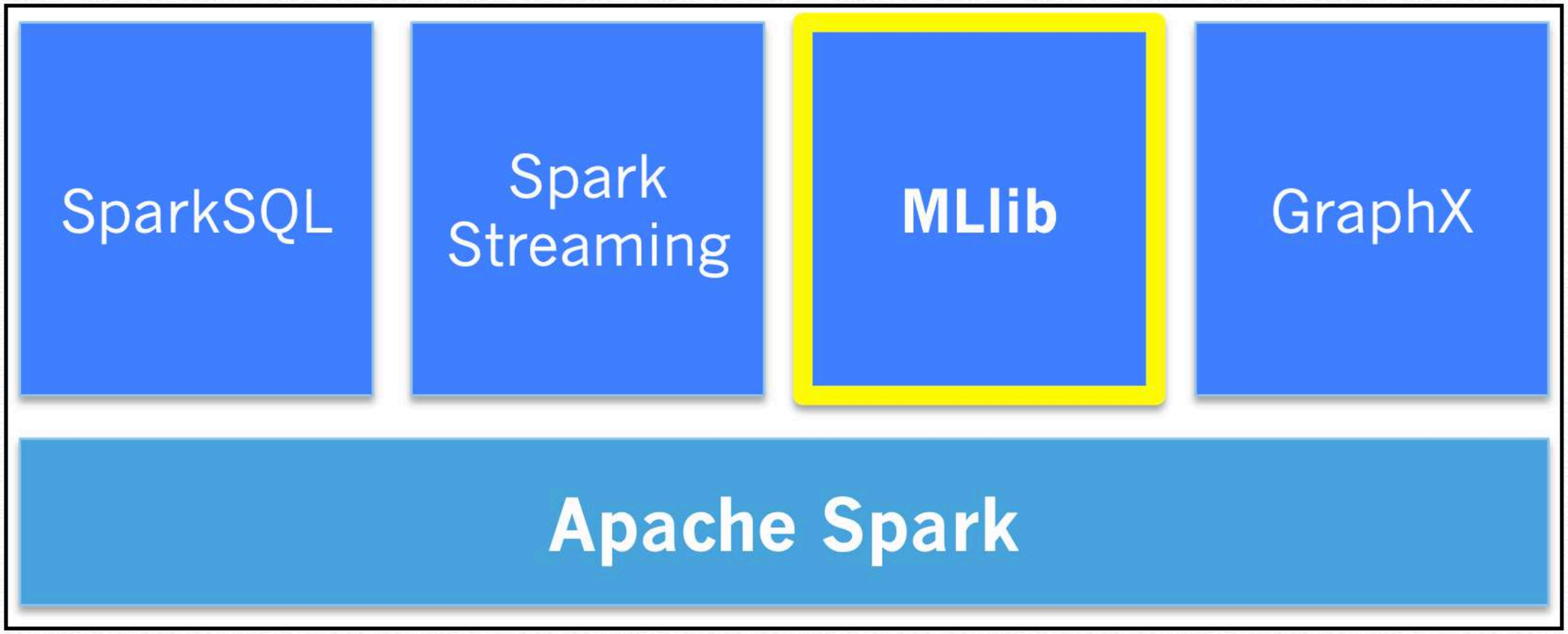
Optimization

Many Utilities

Spark

MLlib

Benefits of Spark MLlib

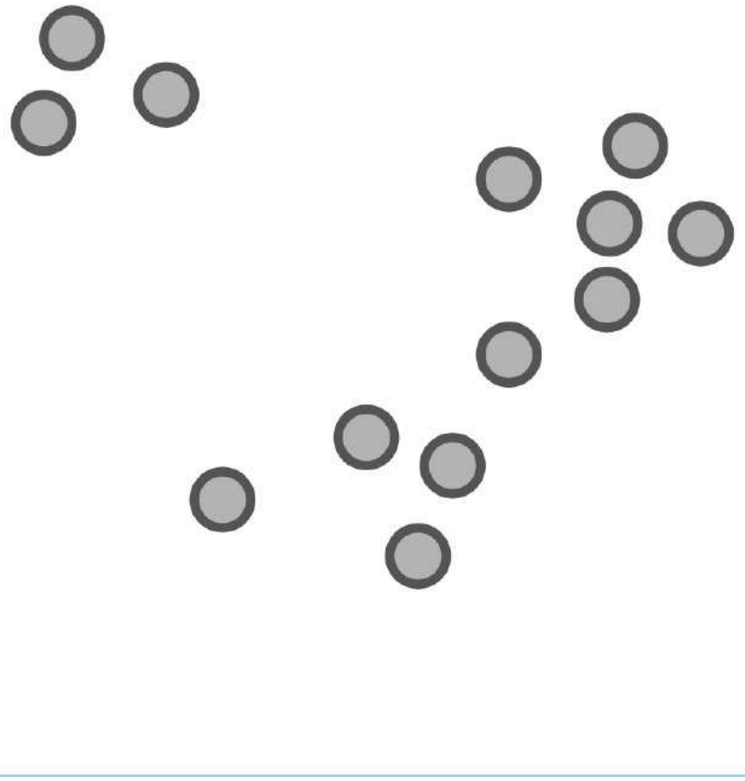


Spark

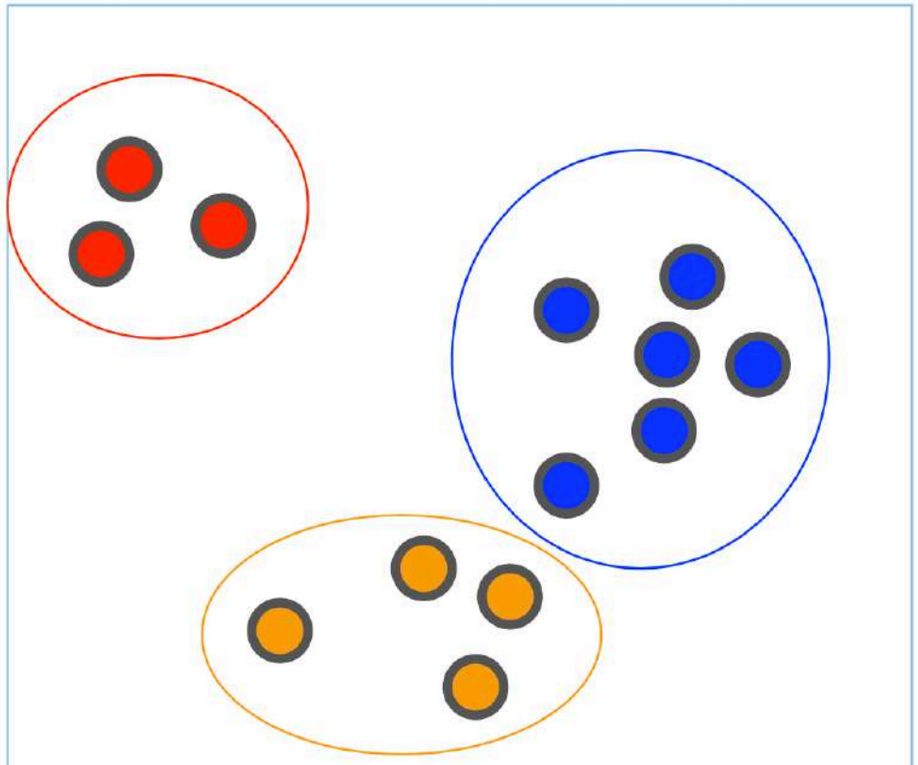
MLlib

Clustering with K-Means

Given data points



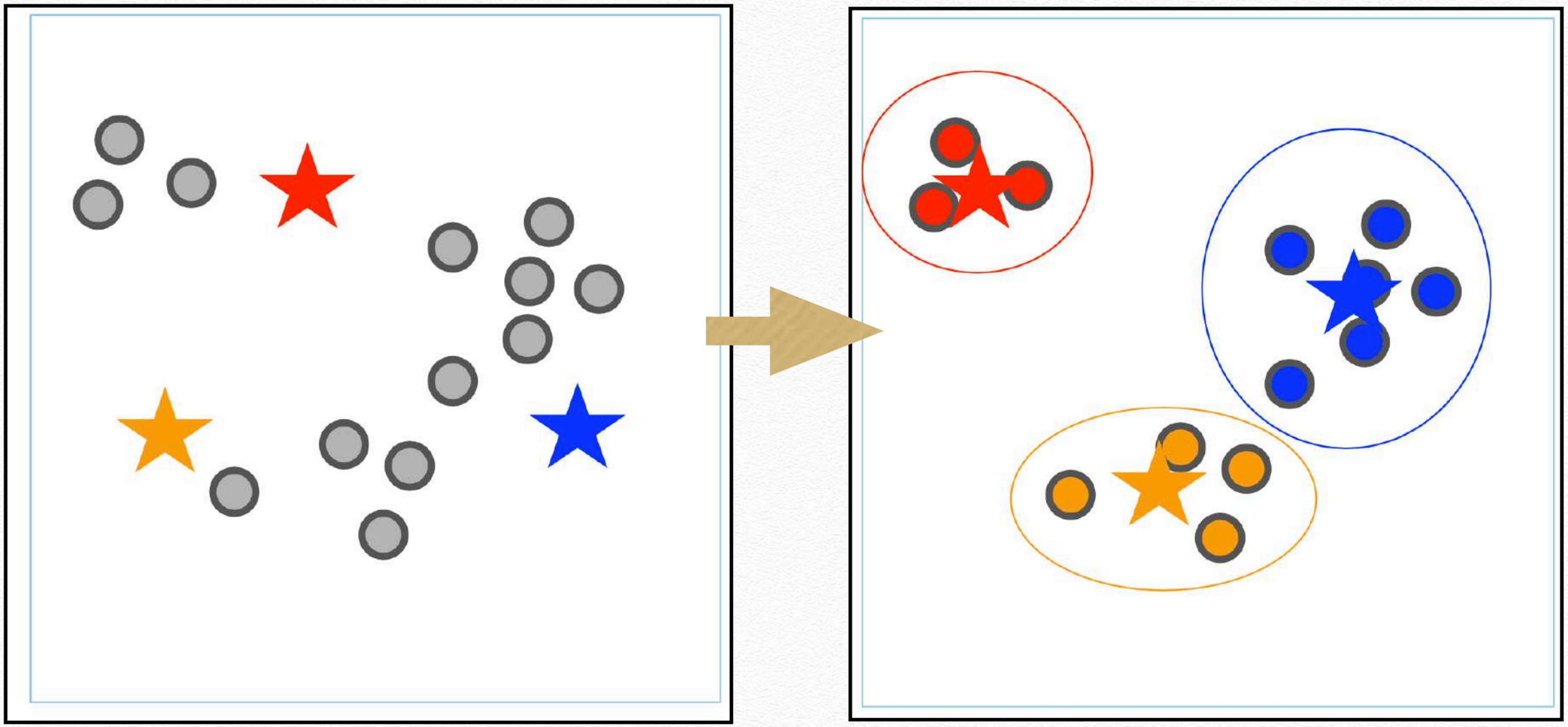
Find meaningful clusters



Spark

MLlib

Clustering with K-Means (centers)



Spark MLlib

K-means (JAVA)

kmeans-data.txt

```
0.0 0.0 0.0
0.1 0.1 0.1
0.2 0.2 0.2
9.0 9.0 9.0
9.1 9.1 9.1
9.2 9.2 9.2
```

```java
public class JavaKMeansExample {

  public static void main(String[] args) {

    SparkConf conf = new SparkConf().setAppName("JavaKMeansExample");
    JavaSparkContext jsc = new JavaSparkContext(conf);

    // Load and parse data
    String path = "data/kmeans_data.txt";
    JavaRDD<String> data = jsc.textFile(path);
    JavaRDD<Vector> parsedData = data.map(s -> {
      String[] sarray = s.split(" ");
      double[] values = new double[sarray.length];
      for (int i = 0; i < sarray.length; i++) {
        values[i] = Double.parseDouble(sarray[i]);
      }
      return Vectors.dense(values);
    });
    parsedData.cache();
```

Spark MLlib

K-means (JAVA)

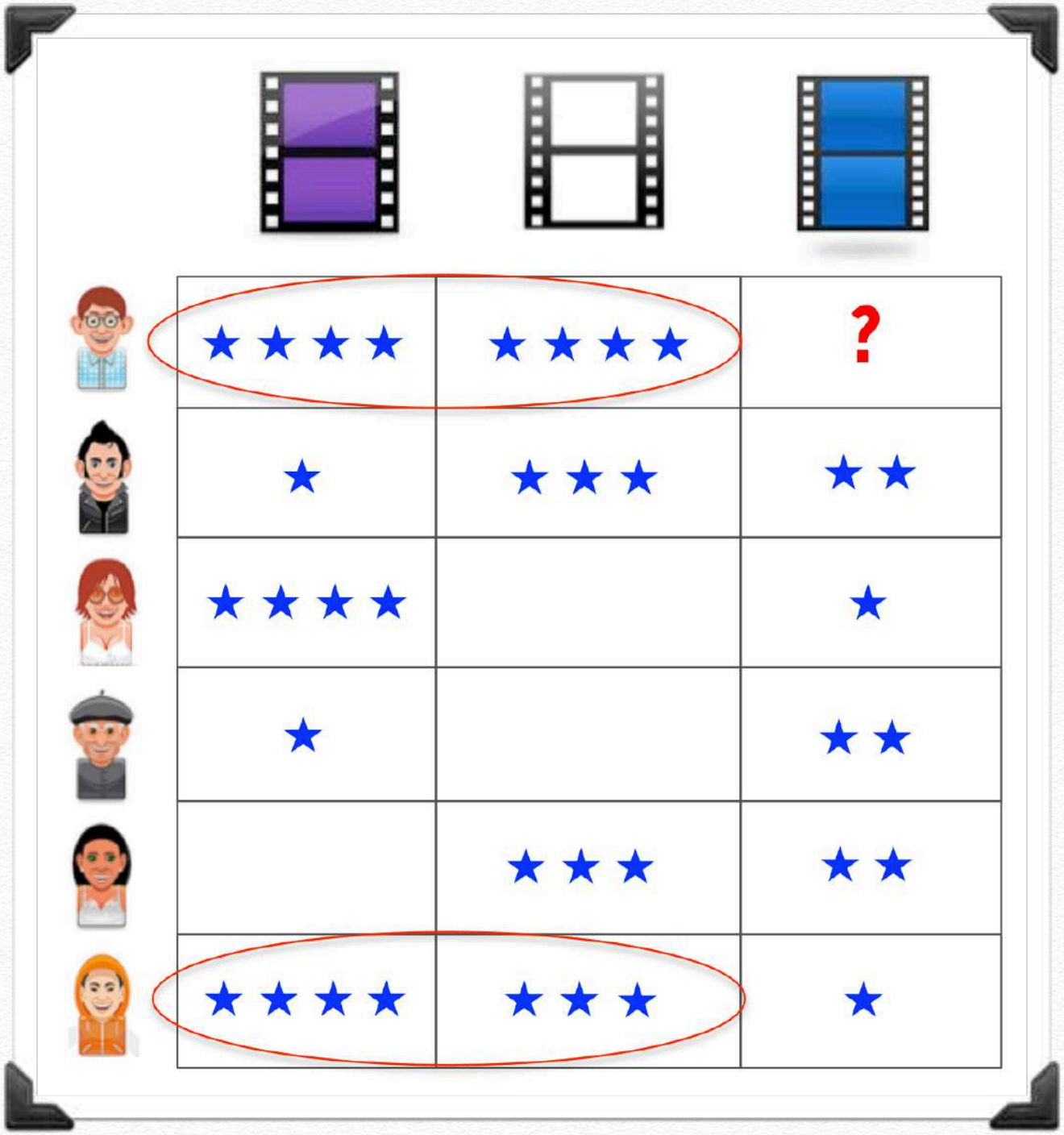
kmeans-data.txt

```
0.0 0.0 0.0
0.1 0.1 0.1
0.2 0.2 0.2
9.0 9.0 9.0
9.1 9.1 9.1
9.2 9.2 9.2
```

```
    // Cluster the data into two classes using KMeans
    int numClusters = 2;
    int numIterations = 20;
    KMeansModel clusters = KMeans.train(parsedData.rdd(), numClusters, numIterations);

    System.out.println("Cluster centers:");
    for (Vector center: clusters.clusterCenters()) {
      System.out.println(" " + center);
    }
    double cost = clusters.computeCost(parsedData.rdd());
    System.out.println("Cost: " + cost);

    // Evaluate clustering by computing Within Set Sum of Squared Errors
    double WSSSE = clusters.computeCost(parsedData.rdd());
    System.out.println("Within Set Sum of Squared Errors = " + WSSSE);

    // Save and load model
    clusters.save(jsc.sc(), "target/JavaKMeansExample/KMeansModel");
    KMeansModel sameModel = KMeansModel.load(jsc.sc(),
      "target/JavaKMeansExample/KMeansModel");
    // $example off$

    jsc.stop();
  }
}
```

Spark MLlib

Recommendation

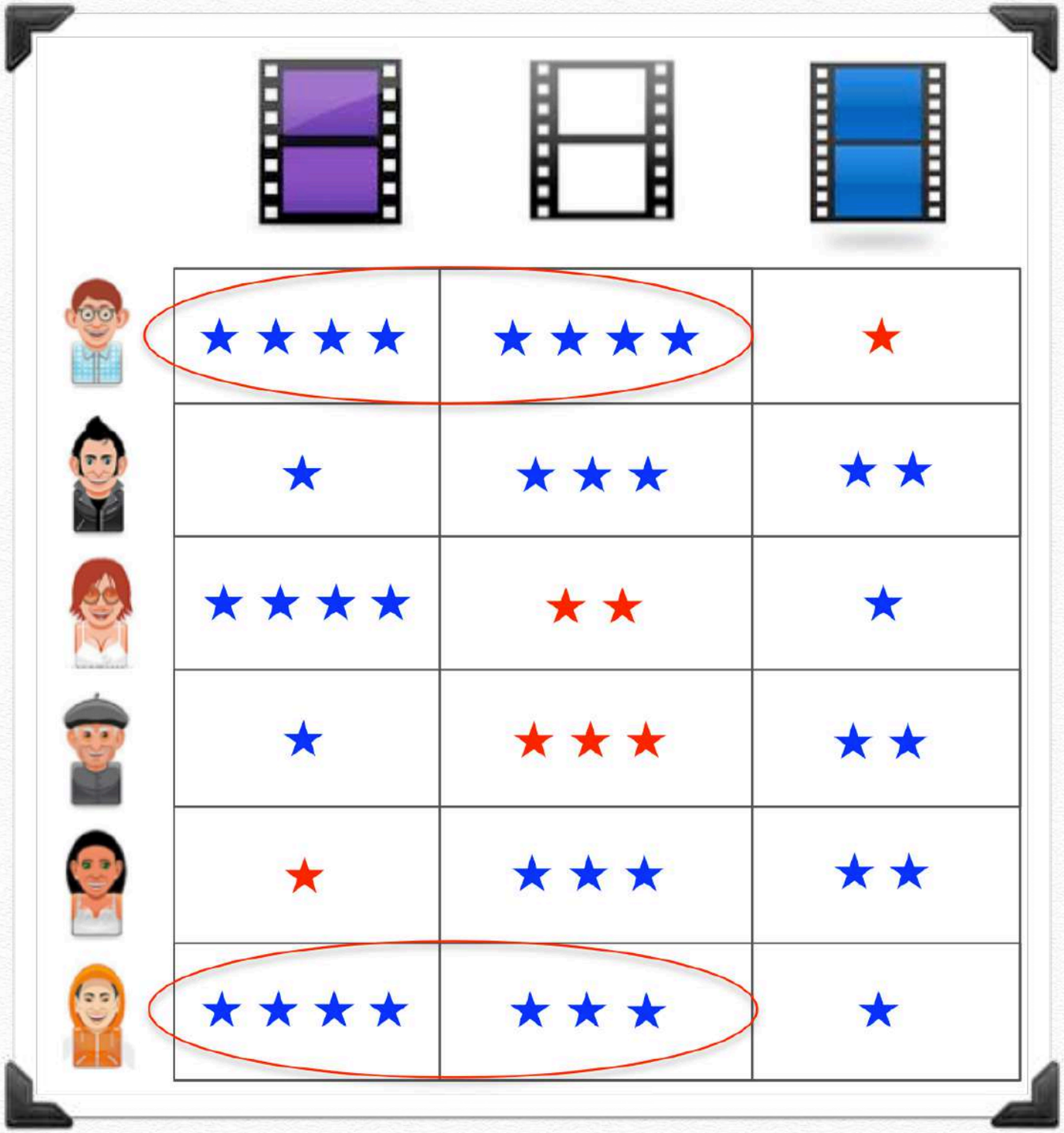


Goal:

Recommend movies to users

Spark
MLlib

Recommendation



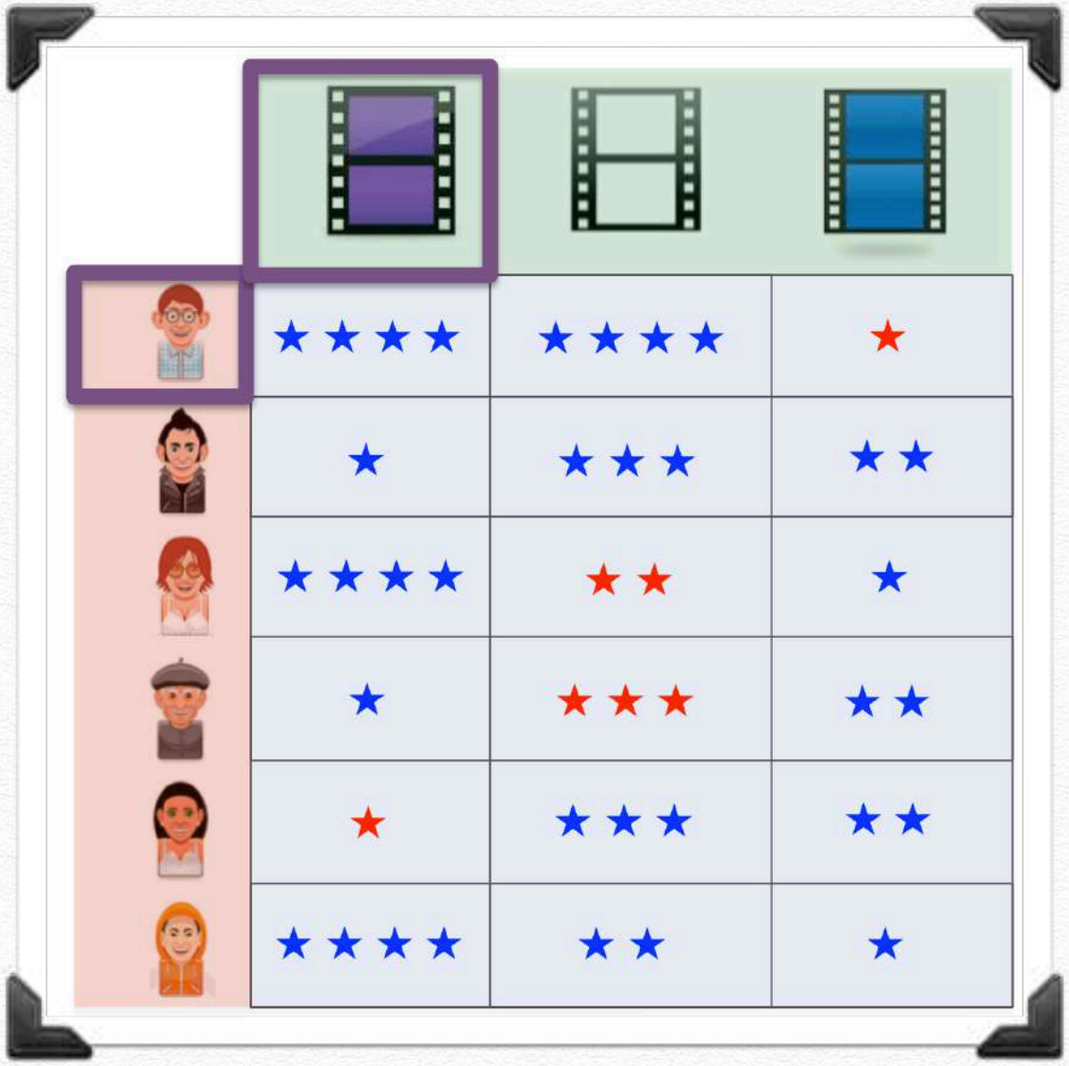
Collaborative Filtering:

- making
  automatic predictions (*filtering*)
  about the interests of a user
- by collecting preferences
  or taste information from many
  users (*collaborating*)

Spark

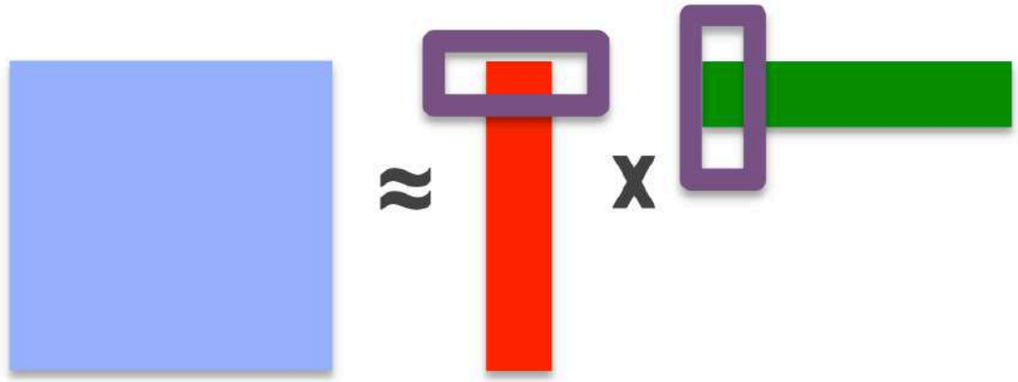
MLlib

Recommendation



	★★★★	★★★★	★
	★	★★★	★★
	★★★★	★★	★
	★	★★★	★★
	★	★★★	★★
	★★★★	★★	★

Solution: Assume ratings are determined by a small number of factors.

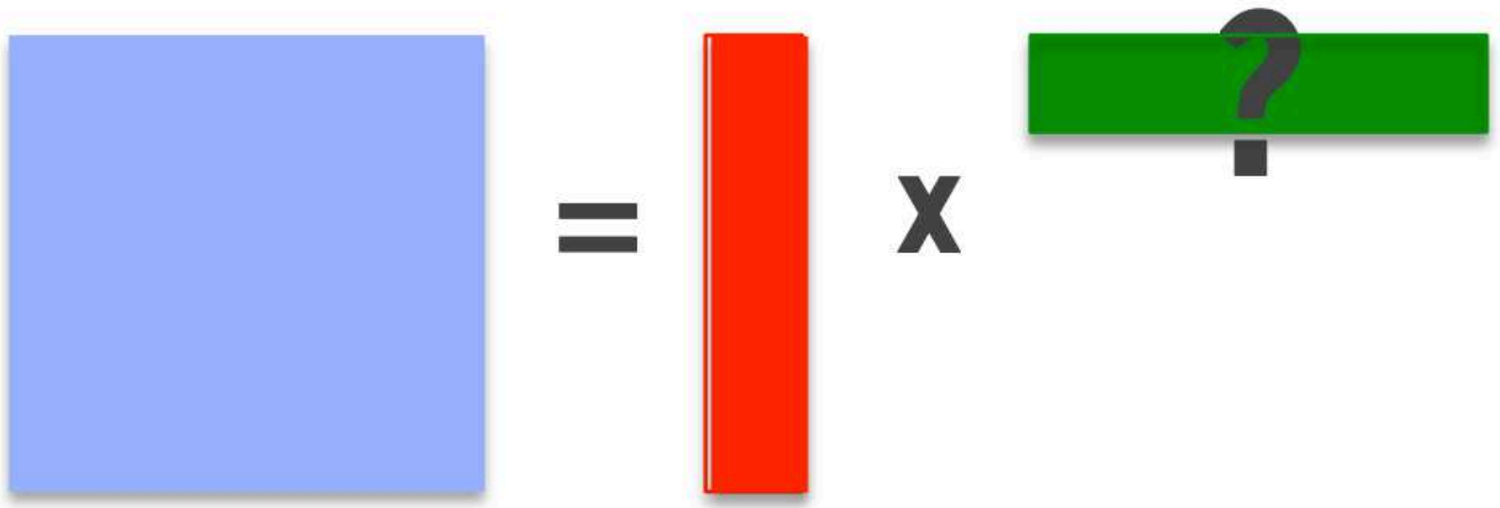


Spark
MLlib

Recommendation with Alternating Least Squares (ALS)

Algorithm

Alternating update of user/movie factors



Collaborative Filtering (JAVA)

MLlib

test.data

```
1,1,5.0
1,2,1.0
1,3,5.0
1,4,1.0
2,1,5.0
2,2,1.0
2,3,5.0
2,4,1.0
```

```
public class JavaRecommendationExample {
  public static void main(String[] args) {
    // $example on$
    SparkConf conf = new SparkConf()
            .setAppName("Java Collaborative Filtering Example");
    JavaSparkContext jsc = new JavaSparkContext(conf);

    // Load and parse the data
    String path = "data/test.data";
    JavaRDD<String> data = jsc.textFile(path);
    JavaRDD<Rating> ratings = data.map(s -> {
      String[] sarray = s.split(",");
      return new Rating(Integer.parseInt(sarray[0]),
        Integer.parseInt(sarray[1]),
        Double.parseDouble(sarray[2]));
    });
```

Collaborative Filtering (JAVA)

Spark MLlib

test.data

```
1,1,5.0
1,2,1.0
1,3,5.0
1,4,1.0
2,1,5.0
2,2,1.0
2,3,5.0
2,4,1.0
```

```
// Build the recommendation model using ALS
    int rank = 10;
    int numIterations = 10;
    MatrixFactorizationModel model = ALS
              .train(JavaRDD.toRDD(ratings), rank, numIterations, 0.01);
```

Spark MLlib

Collaborative Filtering (JAVA)

test.data

```
1,1,5.0
1,2,1.0
1,3,5.0
1,4,1.0
2,1,5.0
2,2,1.0
2,3,5.0
2,4,1.0
```

```
// Evaluate the model on rating data
    JavaRDD<Tuple2<Object, Object>> userProducts =
      ratings.map(r -> new Tuple2<>(r.user(), r.product()));

    JavaPairRDD<Tuple2<Integer, Integer>, Double> predictions =
                JavaPairRDD.fromJavaRDD(
      model.predict(JavaRDD.toRDD(userProducts)).toJavaRDD()
          .map(r -> new Tuple2<>(new Tuple2<>(r.user(), r.product()), r.rating()))
        );

    JavaRDD<Tuple2<Double, Double>> ratesAndPreds = JavaPairRDD.fromJavaRDD(
      ratings.map(r -> new Tuple2<>(new Tuple2<>(r.user(), r.product()), r.rating())))
      .join(predictions).values();
```

Collaborative Filtering (JAVA)

MLlib

test.data

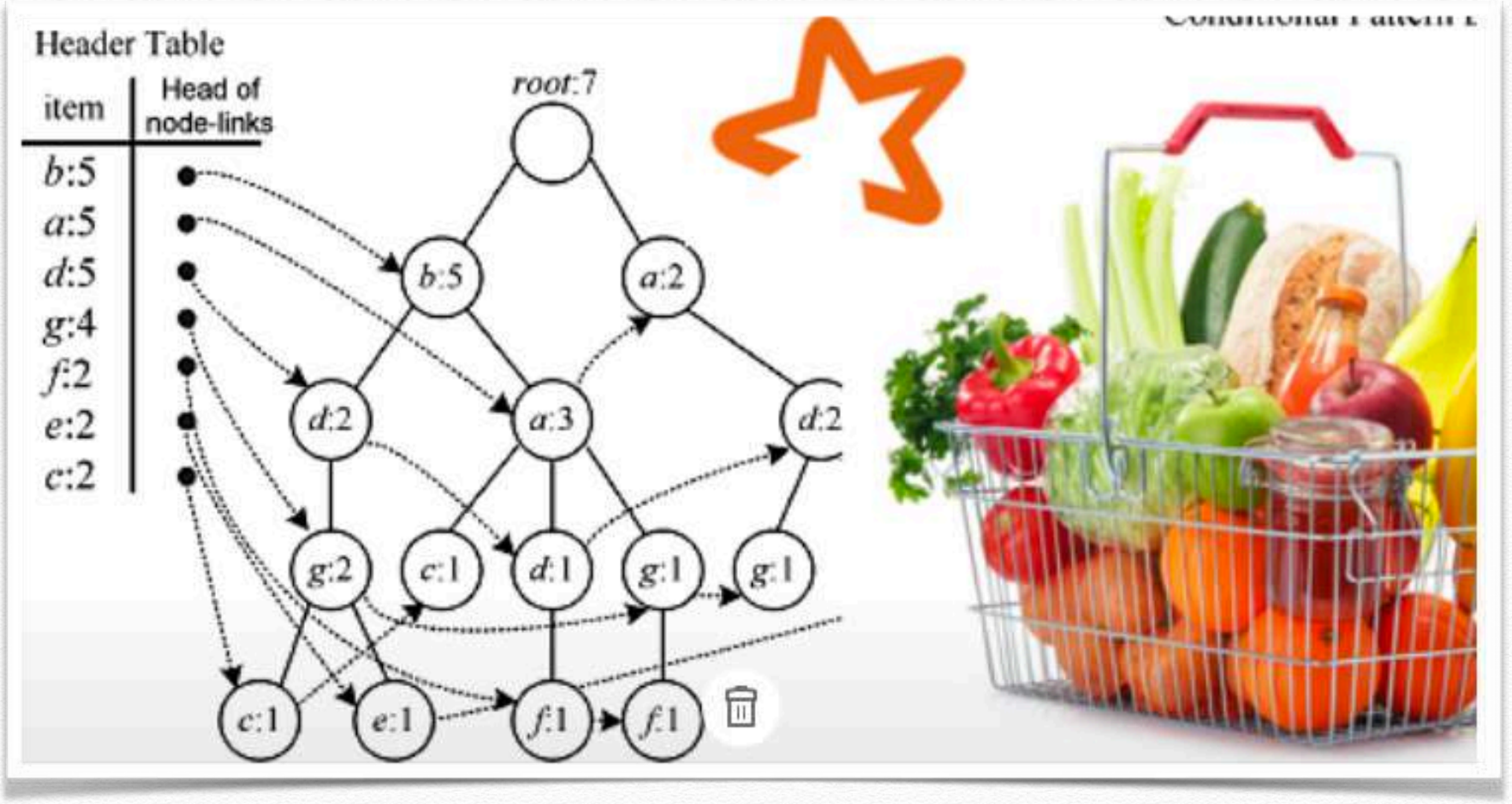
```
1,1,5.0
1,2,1.0
1,3,5.0
1,4,1.0
2,1,5.0
2,2,1.0
2,3,5.0
2,4,1.0
```

```
double MSE = ratesAndPreds.mapToDouble(pair -> {
   double err = pair._1() - pair._2();
   return err * err;
 }).mean();

    System.out.println("Mean Squared Error = " + MSE);

// Save and load model
     model.save(jsc.sc(), "target/myCollaborativeFilter");
    MatrixFactorizationModel sameModel =
           MatrixFactorizationModel.load(jsc.sc(),
                                         “target/myCollaborativeFilter");

    jsc.stop();
  }
}
```

Spark

MLlib

Association Rule



Spark

MLlib

Association Rule (JAVA)

```
public class JavaAssociationRulesExample {

  public static void main(String[] args) {

    SparkConf sparkConf = new SparkConf().setAppName("JavaAssociationRulesExample");
    JavaSparkContext sc = new JavaSparkContext(sparkConf);

    JavaRDD<FPGrowth.FreqItemset<String>> freqItemsets = sc.parallelize(Arrays.asList(
      new FreqItemset<>(new String[] {"a"}, 15L),
      new FreqItemset<>(new String[] {"b"}, 35L),
      new FreqItemset<>(new String[] {"a", "b"}, 12L)
    ));

    AssociationRules arules = new AssociationRules()
      .setMinConfidence(0.8);
    JavaRDD<AssociationRules.Rule<String>> results = arules.run(freqItemsets);

    for (AssociationRules.Rule<String> rule : results.collect()) {
      System.out.println(
       rule.javaAntecedent() + " => " + rule.javaConsequent() + ", " + rule.confidence());
    }

    sc.stop();
  }
}
```