

# University of Salford, MSc Data Science

**Module:** Big Data Tools and Techniques

**Date:** Trimester 2, 2024-2025

**Session:** Workshop Week 8

**Topic:** Recommender Systems Using Apache Spark

**Tools:** Databricks Community Edition

**Instructors:** Dr Kaveh Kiani, Dr Taha Mansouri, and Nathan Topping.

## **Objectives:**

After completing this workshop, you will be able to:

- Use Spark SQL and Spark DataFrames to carry out exploratory data analysis
- Use the ALS algorithm within the MLlib library to train a recommender system
- Use grid search to tune your hyperparameters when training a recommender system
- Use your recommender system to make recommendations for an individual user
- Track runs and experiments using MLflow and the Databricks Experiment UI

# Table of Contents

A Brief Introduction to Recommender Systems.....	3
MLlib .....	4
MLflow.....	5
Part 1: Fire up the Databricks workspace .....	6
Part 2: Creating a new notebook.....	7
Part 3: Exploratory Data Analysis & Data Pre-Processing .....	8
Challenge 1 .....	11
Challenge 2.....	12
Part 4: Training the Model.....	12
Part 5: Evaluating the Model .....	14
Part 6: Using the Model to Generate Movie Recommendations.....	16
Challenge 3.....	17
Part 8: Hyperparameter Tuning.....	19
Challenge 4.....	19

## A Brief Introduction to Recommender Systems

This week we are going to look recommender systems. Recommender systems are ubiquitous in the modern world – think about Amazon recommending what products to buy, Spotify recommending what songs to listen to and Netflix recommending what shows to watch. In industry, Apache Spark is widely used to power many companies' recommendation systems – including Netflix, OpenTable and MyFitnessPal. You can view a more complete list of companies known to be using Apache Spark here (not all of them for recommender systems.)

Apache Spark's scalability makes it well suited to recommender systems, as companies may be dealing with millions of customers and tens of thousands of products. For example, Netflix has over 230 million subscribers who watch over six billion hours per month from their catalogue of over 17,000 titles globally; from their subscriber data, they need to generate personalised recommendations for every user. One of Netflix's engineers spoke about how they utilise Apache Spark:

*"Netflix uses machine learning to inform nearly every aspect of the product, from the recommendations you get, to the boxart you see to the decisions made about which TV shows and movies are created. Given this scale, we utilized Apache Spark to be the engine of our recommendation pipeline. Apache Spark enables Netflix to use a single, unified framework/API – for ETL, feature generation, model training, and validation. With pipeline framework in Spark ML, each step within the Netflix recommendation pipeline (e.g., label generation, feature encoding, model training, model evaluation) is encapsulated as Transformers, Estimators and Evaluators – enabling modularity, composability, and testability. Thus, Netflix engineers can build our own feature engineering logics as Transformers, learning algorithms as Estimators, and customized metrics as Evaluators, and with these building blocks, we can more easily experiment with new pipelines and rapidly deploy them to production."*

<https://www.databricks.com/session/netflixs-recommendation-ml-pipeline-using-apache-spark>

In this workshop, we will be using an extract from the MovieLens dataset to train a collaborative filtering model which we can use to make personalised movie recommendations. Perhaps you can even use it to help pick the next movie to watch!

Broadly speaking, there are two main approaches to recommender systems. These are:

- **Content based filtering** uses item features to recommend other items similar to what a user likes. The disadvantages are that we have to engineer item features in some way so that we have a way of comparing two items to see how similar or dissimilar they are – and engineering these features requires domain knowledge. We also can't use this to expand on the existing interests of the user, since we are only recommending things similar to their existing likes.
- **Collaborative filtering** uses similarities between both users and items to generate recommendations. This means it can recommend an item to a user based on the fact that it was liked by a similar user. The advantages of this approach are that we don't need to

engineer features and the model can also help users discover new interests, based on those of other, similar users. This is the approach we will be exploring in today's workshop.

Feedback provided by a user on an item can either be explicit or implicit. Explicit feedback is where a user has provided feedback, for example, by rating a movie or a product. Implicit feedback is based on the actions taken by a user – for example, what a user has chosen to watch or to purchase provides us with an indication of what they are interested in.

The MovieLens data we are using for this workshop contains consists of three files:

- **movies.csv** contains data on the movie titles and genres, and a unique movieId for each movie.
- **ratings.csv** contains data on user ratings. Each row contains a userId, a movieId and the rating which that user gave to that movie. Note each movie and user can appear in this dataset many times, but each userId and movieId pair can only appear once (i.e., a user can only rate a specific movie once.) This data is explicit feedback as the users have given the movies a rating out of 5.
- **myratings.csv** contains an extract of 300 of the most highly rated movies. 10 of these have been rated. Optionally, you had the opportunity prior to the workshop to update this file with some of your own movie ratings so that you can use this to generate some recommendations specific to you.

## MLlib

MLlib is Spark's machine learning library. It provides implementations of many common machine learning algorithms, including classification, regression, clustering and collaborative filtering algorithms. It also provides tools for data pre-processing, such as feature extraction, transformation and selection.

There are two packages within MLlib. The first is `pyspark.mllib` – this leverages the RDD APIs provided by Spark. However, this is now in 'maintenance mode' and the primary API is `pyspark.ml` which is a DataFrame-based API. We will be using this one in today's workshop.

In MLDM, we introduce Scikit Learn and Tensorflow, which are libraries which are more widely used for machine learning. These libraries are generally the most appropriate choices unless you are working with big data, and it becomes difficult or inconvenient to use these libraries. When working with big data, MLlib makes ***distributed*** machine learning very simple. The dataset we use today, for example, is very small (around one thousand rows) so in practice we would be likely to use Scikit Learn for this. However, the aim of today is to introduce you to MLlib, because if you are working with big data, MLlib becomes a useful tool to complement the other libraries we cover on this course.

You can view the documentation for MLlib here:

<https://spark.apache.org/docs/latest/ml-guide.html>

## **MLflow**

MLflow is an open-source platform to manage the ML lifecycle, including experimentation, reproducibility, deployment, and a central model registry. MLflow currently offers four components:

- MLflow Tracking: Record and query experiments: code, data, config, and results
- MLflow Projects: Package data science code in a format to reproduce runs on any platform
- MLflow Models: Deploy machine learning models in diverse serving environments
- Model Registry: Store, annotate, discover, and manage models in a central repository

MLflow is included in Databricks Community Edition, meaning that you can utilize its Tracking and Model APIs easily within a notebook. It can be used with many machine learning libraries, including Scikit Learn and Tensorflow, as well as MLlib.

You can read more online here:

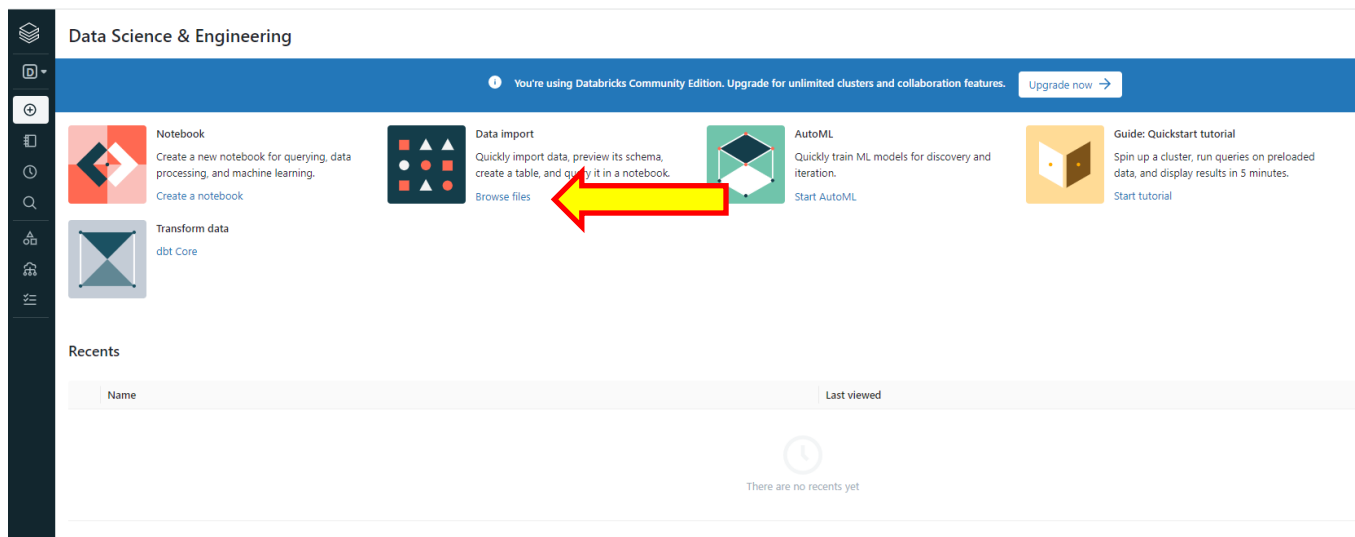
<https://mlflow.org/>

## Part 1: Fire up the Databricks workspace

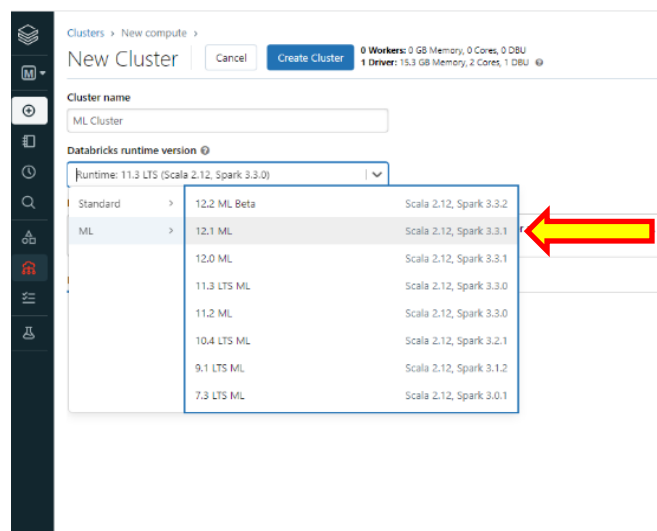
1. Log in to your Databricks Community Edition account. Here is the link:

<https://community.cloud.databricks.com/login.html?nextUrl=%2F>

2. You will need to download three csv files from Blackboard for use in this workshop. These are **ratings.csv** and **movies.csv**, both of which are saved in the Workshop folder. You will also need the **myratings.csv** file which was saved in the Pre-Class Activity folder. You had the option to update this file with some of your own ratings. Then on your Databricks home screen select 'Browse files' under the Data Import option on your screen, and then navigate to the directory location where you have saved the files. Upload all three to Databricks.



3. Click on "Create Compute" and type in a new name for the cluster, any name that you like.
4. From the dropdown, you **must** select an ML runtime and not a standard runtime. This will ensure that it has all the necessary libraries installed for this session. Select a 12.1 ML runtime.



Note that it will take a few minutes to create the cluster. After some time, the green circle next to the cluster name will gain a green tick meaning the cluster has successfully started up.

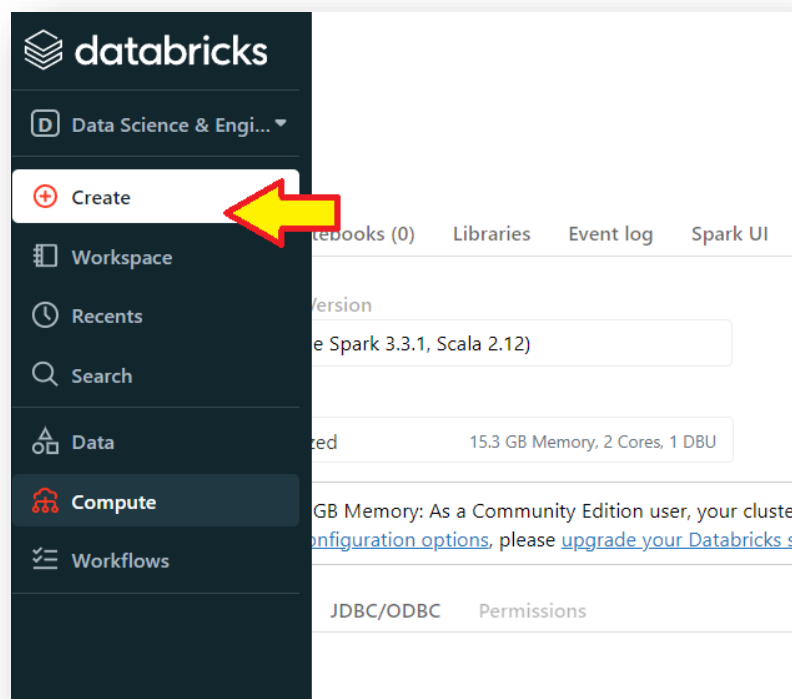
## Part 2: Creating a new notebook

Databricks programs are written in Notebooks. A notebook is a collection of runnable cells which contain your commands. Look at:

<https://docs.databricks.com/notebooks/notebooks-use.html>

for an overview of how to use a Databricks notebook.

1. Click on the button “Create” and select “Notebook”.



2. Name your notebook, e.g. “Week 8 Machine Learning”, leave the language as the default Python and the cluster should be the cluster you have just created. (NB – you can open the notebook and start working through the below while your cluster is starting, you will just be unable to run any of the code until your cluster has successfully started.)

## Part 3: Exploratory Data Analysis & Data Pre-Processing

Before we carry out any machine learning on a dataset, it is good practice to explore the dataset and get a better understanding of the data we are working with. This typically includes plotting graphs to help us visualise the data and/or computing summary statistics.

1. Before we start, we need to import the MLflow library and enable autologging.

```
Cmd 1

1  # import mlflow and autolog machine learning runs
2
3  import mlflow
4
5  mlflow.pyspark.ml.autolog()

Command took 2.07 seconds -- by n.j.topping@salford.ac.uk at 2/26/2023, 9:58:01 PM on ML Cluster V2
```

2. Now we can read the three datasets which we uploaded into a Spark DataFrame using `spark.read.csv()`. Because the first row of the file contains the header, we set `header` to `true`. We also want to infer the schema from the file so have set the option `inferSchema` to `true`. This will create three DataFrames from the csv files we have imported to Databricks.

```
1  ratings = spark.read.csv("/FileStore/tables/ratings.csv",
2                           header = "true",
3                           inferSchema="true")

▶ (2) Spark Jobs
▶ ratings: pyspark.sql.dataframe.DataFrame = [userId: integer, movielid: integer ... 2 more fields]

Command took 1.78 seconds -- by n.j.topping@salford.ac.uk at 3/2/2023, 7:59:26 PM on Recommender v1
```

```
Cmd 3

1  movies = spark.read.csv("/FileStore/tables/movies.csv",
2                           header = "true",
3                           inferSchema="true")

▶ (2) Spark Jobs
▶ movies: pyspark.sql.dataframe.DataFrame = [movielid: integer, title: string ... 1 more field]


Command took 0.99 seconds -- by n.j.topping@salford.ac.uk at 3/2/2023, 7:59:26 PM on Recommender v1
```



Cmd 4

```
1 myRatings = spark.read.csv("/FileStore/tables/myratings.csv",
2                             header = "true",
3                             inferSchema="true")
```

▸ (2) Spark Jobs

▸  myRatings: pyspark.sql.dataframe.DataFrame = [userId: integer, movieId: integer ... 2 more fields]


Command took 1.17 seconds -- by n.j.topping@salford.ac.uk at 3/2/2023, 8:09:26 PM on Recommender v1

3. We can use `show()` to view the first few rows of each DataFrame, and gain an understanding of their contents. However, first, we can use `dropna()` to drop the rows from the `myRatings` DataFrame which do not have any movie ratings in them. If we then view the `myRatings` DataFrame you should see the ratings which you provided prior to the workshop (or the default ratings that were provided for you if you opted not to complete this pre-class activity.)

Cmd 5

```
1 myRatings = myRatings.dropna()
2
3 myRatings.show()
```

▸ (1) Spark Jobs

▸  myRatings: pyspark.sql.dataframe.DataFrame = [userId: integer, movieId: integer ... 2 more fields]

```
+-----+-----+-----+-----+
|userId|movieId|          title|rating|
+-----+-----+-----+-----+
|    0|    356|Forrest Gump (1994)|    3|
|    0|    318|Shawshank Redempt...|    4|
|    0|    296|Pulp Fiction (1994)|    1|
|    0|    593|Silence of the La...|    0|
|    0|   2571|Matrix, The (1999)|    5|
|    0|    260|Star Wars: Episod...|    4|
|    0|    480|Jurassic Park (1993)|    4|
|    0|    110|Braveheart (1995)|    3|
|    0|    589|Terminator 2: Jud...|    4|
|    0|    527|Schindler's List ...|    5|
+-----+-----+-----+-----+
```

Command took 1.03 seconds -- by n.j.topping@salford.ac.uk at 3/5/2023, 5:36:50 PM on My Cluster

4. If we take a look at the ratings DataFrame we can see each row is formed of a `userId`, `movieId`, a rating and a timestamp (NB – you should note from the schema that the timestamp was not inferred as being a timestamp data type and therefore is not correctly formatted. If we were using this column, we would want to change this, but we will be dropping this column instead)

Cmd 6

```
1 ratings.show(truncate=False)
```

► (1) Spark Jobs

```
+-----+-----+-----+-----+
|userId|movieId|rating|timestamp|
+-----+-----+-----+-----+
|1      |1       |4.0   |964982703|
|1      |3       |4.0   |964981247|
|1      |6       |4.0   |964982224|
|1      |47      |5.0   |964983815|
|1      |50      |5.0   |964982931|
|1      |70      |3.0   |964982400|
|1      |101     |5.0   |964980868|
|1      |110     |4.0   |964982176|
|1      |151     |5.0   |964984041|
|1      |157     |5.0   |964984100|
|1      |163     |5.0   |964983650|
|1      |216     |5.0   |964981208|
|1      |223     |3.0   |964980985|
|1      |231     |5.0   |964981179|
|1      |235     |4.0   |964980908|
|1      |260     |5.0   |964981680|
|1      |296     |3.0   |964982967|
|1      |316     |3.0   |964982310|
```

Command took 0.61 seconds -- by n.j.topping@salford.ac.uk at 3/5/2023, 5:37:53 PM on My Cluster

Cmd 7

```
1 movies.show(truncate=False)
```

► (1) Spark Jobs

```
+-----+-----+-----+
|movieId|title                                     |genres
+-----+-----+-----+
|1      |Toy Story (1995)                         |Adventure|Animation|Children|Comedy|Fantasy|
|2      |Jumanji (1995)                           |Adventure|Children|Fantasy
|3      |Grumpier Old Men (1995)                   |Comedy|Romance
|4      |Waiting to Exhale (1995)                 |Comedy|Drama|Romance
|5      |Father of the Bride Part II (1995)       |Comedy
|6      |Heat (1995)                              |Action|Crime|Thriller
|7      |Sabrina (1995)                           |Comedy|Romance
|8      |Tom and Huck (1995)                      |Adventure|Children
|9      |Sudden Death (1995)                      |Action
|10     |GoldenEye (1995)                         |Action|Adventure|Thriller
|11     |American President, The (1995)            |Comedy|Drama|Romance
|12     |Dracula: Dead and Loving It (1995)        |Comedy|Horror
|13     |Balto (1995)                             |Adventure|Animation|Children
|14     |Nixon (1995)                             |Drama
|15     |Cutthroat Island (1995)                   |Action|Adventure|Romance
|16     |Casino (1995)                             |Crime|Drama
|17     |Sense and Sensibility (1995)              |Drama|Romance
|18     |Four Rooms (1995)                       |Comedy
```

Command took 0.59 seconds -- by n.j.topping@salford.ac.uk at 3/5/2023, 5:39:14 PM on My Cluster

- Now take a quick look at the movies DataFrame. This contains more information on each of the movies – including the title and genre tags.

- We can drop the columns we don't need from the myRatings DataFrame and the ratings DataFrame. This means that both DataFrames will therefore have the same columns which means we will be able to use the unionAll() transformation to merge the two DataFrames

before we train the recommender system. This means we will be including our own ratings as part of the training data and can subsequently generate recommendations for ourselves.

Cmd 8

```
1 myRatings = myRatings.drop("title")
2 ratings = ratings.drop("timestamp")
```

▶ myRatings: pyspark.sql.dataframe.DataFrame = [userId: integer, movielid: integer ... 1 more field]  
▶ ratings: pyspark.sql.dataframe.DataFrame = [userId: integer, movielid: integer ... 1 more field]

Command took 0.15 seconds -- by n.j.topping@salford.ac.uk at 3/5/2023, 5:40:01 PM on My Cluster

Before we train our recommendation system, we should do some initial exploration of the dataset.

7. We can use the below code to count the number of reviews from each user and display the result.

Cmd 9

```
1 ratings.groupBy("userId").count().display()
```

▶ (2) Spark Jobs

Table ▾ +

	userId	count
1	148	48
2	463	33
3	471	28
4	496	29
5	243	36
6	392	25
7	540	42

610 rows | 4.67 seconds runtime

Command took 4.67 seconds -- by n.j.topping@salford.ac.uk at 3/5/2023, 5:44:00 PM on My Cluster

## Challenge 1

- Using the above data, create a **histogram** using the inbuilt Databricks visualisations so we can see the distribution of the count data – i.e., the distribution of the number of reviews per person (you may want to increase the number of bins in the histogram).
  - Use Spark SQL to conduct the same query and satisfy yourself that both approaches produce the same result.
8. If we want to see the films which have the highest number of ratings in the data, we can use the below:

Cmd 12

```
1 ratings.groupBy("movieId").count().join(movies, ["movieId"]).orderBy("count", ascending=False).limit(10).display()
```

▶ (3) Spark Jobs

Table ▾ +

	movieId	count	title	genres
1	356	329	Forrest Gump (1994)	Comedy Drama Romance War
2	318	317	Shawshank Redemption, The (1994)	Crime Drama
3	296	307	Pulp Fiction (1994)	Comedy Crime Drama Thriller
4	593	279	Silence of the Lambs, The (1991)	Crime Horror Thriller
5	2571	278	Matrix, The (1999)	Action Sci-Fi Thriller
6	260	251	Star Wars: Episode IV - A New Hope (1977)	Action Adventure Sci-Fi
7	480	238	Jurassic Park (1993)	Action Adventure Sci-Fi Thriller

[↓](#) 10 rows | 1.27 seconds runtime

Command took 1.27 seconds -- by n.j.topping@salford.ac.uk at 3/5/2023, 6:02:51 PM on My Cluster

## Challenge 2

- Using the above data, create a **horizontal bar plot** using the inbuilt Databricks visualisations so we can see the number of reviews for each of the top 10 reviewed films. (Under the Y Axis tab of the Databricks visualisation tool, you should un-select 'Sort Values' so that the values remain sorted by the number of reviews and not the movie title.)
  - Use Spark SQL to conduct the same query and satisfy yourself that both approaches produce the same result.
9. The next step before we train our model is to split the data into a training and a test dataset using `randomSplit()` as we did last week. We will also use `unionAll()` to merge our own ratings data with the training data, so that we can generate some recommendations.

Cmd 15

```
1 (training, test) = ratings.randomSplit([0.8, 0.2], seed=100)
2
3 training = training.unionAll(myRatings)
```

▶ test: pyspark.sql.dataframe.DataFrame = [userId: integer, movieId: integer ... 1 more field]

▶ training: pyspark.sql.dataframe.DataFrame = [userId: integer, movieId: integer ... 1 more field]

Command took 0.19 seconds -- by n.j.topping@salford.ac.uk at 3/5/2023, 6:21:47 PM on My Cluster

## Part 4: Training the Model

As discussed in the lecture, we can use our user feedback to create a **user-item matrix** which summarises the ratings each user has given to each item (note this will be a **sparse** matrix because most users won't have rated very many products)

					
Alice		5		2	
Bob			4		
Carole		5			1
Toni			2		3

Matrix factorization involves finding two lower dimensional matrices which when multiplied given you a good approximation of the original matrix. If we have  $m$  users and  $n$  products, then we will have a matrix of size  $m \times n$ . Matrix factorization means finding two matrices of size  $m \times d$  and  $d \times n$  so that when we multiply then we get a matrix which is a good approximation of the user-item matrix we started with. Here,  $d$  is the number of *latent factors* (or dimensions) in our embedding, and it is a parameter we can choose.

To carry this out, we use the Alternating Least Squares (ALS) algorithm, which is already implemented for us in the MLlib library.

7. For this workshop, we are going to use alternating least squares (ALS) algorithm to learn the *latent factors*. To implement this we will use the ALS estimator in the MLlib library.

To implement this using MLlib, we first instantiate an instance of the estimator and specify the column in the Data Frame which is the userCol and the column which is the itemCol. We can then use the fit() method to fit the model to our data – i.e., to train it. We will be discussing the hyperparameters for this model in more depth later in this workshop, but for now we set the maxIter parameter to 5 and the regParam to 0.01. As last week, you should note that running this cell leads to a MLflow run being logged to an experiment in MLflow.

Cmd 18

```

1  from pyspark.ml.recommendation import ALS
2
3  als = ALS(maxIter=5, regParam=0.01, userCol="userId", itemCol="movieId", ratingCol="rating", seed=100)
4
5  model = als.fit(training)

```

► (5) Spark Jobs

▼ (1) MLflow run

Logged 1 run to an [experiment](#) in MLflow. [Learn more](#)

## Part 5: Evaluating the Model

8. To evaluate the model, we need to use our trained model to make predictions on the test data, which we did not use when training the model. We generate predictions by using the `transform()` method on the test data. The model is also unable to make predictions for users who did not have any reviews in the training dataset (this is known as the **cold start problem**) and so for these users the predictions will be NaN, so we use `dropna()` to remove these rows from the predictions DataFrame.

```
Cmd 19
1 predictions = model.transform(test).dropna()
2
3 predictions.show()
```

▶ (7) Spark Jobs

▶ predictions: pyspark.sql.dataframe.DataFrame = [userId: integer, movieId: integer ... 2 more fields]

userId	movieId	rating	prediction
1	47	5.0	4.506145
1	163	5.0	4.2417793
1	235	4.0	3.8748116
1	356	4.0	4.709498
1	457	5.0	4.6837845
1	543	4.0	4.2651477
1	593	4.0	4.9132996
1	596	5.0	4.3999677
1	648	3.0	4.3810983
1	661	5.0	3.5649142
1	733	4.0	3.8588338
1	736	3.0	3.3673422
1	1025	5.0	4.9205837
1	1049	5.0	2.9955406
1	1060	4.0	5.18416
1	1097	5.0	4.9711194
1	1208	4.0	4.130768
1	1222	5.0	4.978036

Command took 3.23 seconds -- by n.j.topping@salford.ac.uk at 3/5/2023, 9:57:22 PM on My Cluster

We can see from the output that the predictions Data Frame includes the original columns from the data, but with one new column appended – *prediction*. The prediction column gives a ‘prediction’ for the rating (note, when we use collaborative filtering, the predictions generated are continuous and can therefore be outside the range of 0-5 which the original reviews had to be within. It should therefore not be treated as a prediction *exactly*, but as a score intended to reflect how likely the user is to enjoy the film. The higher the score, the more likely they are to enjoy it.)

9. We then need to instantiate an evaluator which will use an evaluation metric to provide a measure of how effective the model was. In this case we are using **Root Mean Square Error** as our evaluation metric. This is a metric which is commonly used in regression and measures the average difference between the values predicted by a model and the actual values in the data. We use the `evaluate()` method to generate the metric in question and then use `print()` to return this value in the output.

The root mean square error (or RMSE) is given by the below:

$$RMSE = \sqrt{\sum_{i=1}^n \frac{(\hat{y}_i - y_i)^2}{n}}$$

We can interpret the RMSE as the average difference between the values we predict for the ratings and the true values for these predictions – i.e., the lower this is, the more accurate the predictions from our model are.

Cmd 20

```
1  from pyspark.ml.evaluation import RegressionEvaluator
2
3  evaluator = RegressionEvaluator(metricName="rmse", labelCol="rating", predictionCol="prediction")
4
5  rmse = evaluator.evaluate(predictions)
6
7  print('Root Mean Square Error is %g' %rmse)
```

► (7) Spark Jobs

Root Mean Square Error is 1.08036

Command took 2.91 seconds -- by n.j.topping@salford.ac.uk at 3/5/2023, 9:57:22 PM on My Cluster


The accuracy is 1.08036, which means that the average difference between the actual rating and the predicted rating is just over 1. (You may see very slightly different results here if you have used your own ratings for this exercise.)

We can also generate predictions for our own ratings specifically and look at the RMSE for our own ratings. (**NB** – the RMSE and the output you see here will depend on what ratings data you used for this exercise)

Cmd 19

```
1  mySampledMovies = model.transform(myRatings)
2
3  myRmse = evaluator.evaluate(mySampledMovies)
4
5  print('Root Mean Square Error is %g' %myRmse)
6  mySampledMovies.show()
```

► (11) Spark Jobs

►  mySampledMovies: pyspark.sql.dataframe.DataFrame = [userId: integer, movieId: integer ... 2 more fields]

Root Mean Square Error is 0.304837

+-----+-----+-----+-----+  userId movieId rating prediction  +-----+-----+-----+-----+			
	0	356	3  3.239606
	0	318	4  3.5869942
	0	296	1  0.9295597
	0	593	0 0.75178903
	0	2571	5  4.8786364
	0	260	4  4.0124216
	0	480	4  3.9582357
	0	110	3  2.6647182
	0	589	4  3.954423
	0	527	5  4.9986477
+-----+-----+-----+-----+			

## Part 6: Using the Model to Generate Movie Recommendations

We have generated predictions for the movies we **have** rated. But if we are going to use this as a recommender system, we need to generate predictions for the movies we **haven't** rated. We can then sort the result to see which ones have the highest scores and these are the recommended movies for us.

10. To generate predictions for us, we create a new DataFrame which contains all the movieIds and a column populated with '0' (this is because we gave ourselves the userId 0, which you can see from the myRatings DataFrame on page 9). We can then use the transform method on this data, which will generate a prediction for the user with userId 0 for every film in the dataset. We can then use orderBy to sort on the movies with the highest values for the ratings – these are the movies which we are predicted to like the most.

Cmd 21

```
1 from pyspark.sql import functions
2
3 myGeneratedPredictions = movies.withColumn("userId", functions.expr("int('0')"))
4
5 myGeneratedPredictions = model.transform(myGeneratedPredictions)
6
7 myGeneratedPredictions = myGeneratedPredictions.dropna()
8
9 myGeneratedPredictions.orderBy("prediction",ascending=False).show(truncate=False)
```

Your output should look similar to the below, although the highest rated movies will be different depending on your own ratings.



movieId	title	genres	userId	prediction
1916	Buffalo '66 (a.k.a. Buffalo 66) (1998)	Drama Romance	0	19.567131
3844	Steel Magnolias (1989)	Drama	0	16.320473
2531	Battle for the Planet of the Apes (1973)	Action Sci-Fi	0	16.094702
2007	Polish Wedding (1998)	Comedy	0	15.985681
5135	Monsoon Wedding (2001)	Comedy Romance	0	15.597923
218	Boys on the Side (1995)	Comedy Drama	0	14.579158
27611	Battlestar Galactica (2003)	Drama Sci-Fi War	0	14.239786
4678	UHF (1989)	Comedy	0	14.075563
1390	My Fellow Americans (1996)	Comedy	0	14.007243
6880	Texas Chainsaw Massacre, The (2003)	Horror	0	13.815862
2950	Blue Lagoon, The (1980)	Adventure Drama Romance	0	13.428917
6322	Confidence (2003)	Crime Thriller	0	13.403403
1957	Chariots of Fire (1981)	Drama	0	13.131377
118696	The Hobbit: The Battle of the Five Armies (2014)	Adventure Fantasy	0	13.114573
37830	Final Fantasy VII: Advent Children (2004)	Action Adventure Animation Fantasy Sci-Fi	0	13.048952
5504	Spy Kids 2: The Island of Lost Dreams (2002)	Adventure Children	0	12.979152
26614	Bourne Identity, The (1988)	Action Adventure Drama Mystery Thriller	0	12.96915
4846	Iron Monkey (Siu nin Wong Fei-hung ii: Tit Ma Lau) (1993)	Action Comedy	0	12.956755

What do you think of the recommendations for you? (That is, if you used your own ratings!)

### Challenge 3

- Write a Spark SQL query to return the predicted top 10 predictions for you
- Adapt this Spark SQL query to filter the results on a genre of your choice, using the data in the genres column (e.g., this could be Romance, Animation etc)
- Try to also to the above steps using DataFrames.

We have been using the ratings we provided in the **myratings.csv** file to play around with recommendations specific to you (if you completed the Pre-Class Activity). This should have helped to illustrate how we can use the model to generate some recommendations, although, depending on the number of films you reviewed, the recommendations might not seem overly accurate for you (you can also experiment further if you want to by rating more films and re-running this workshop to see how this affects the recommendations for you.)

11. However, in the real-world once we have our model, we want to use it to generate recommendations for all users. We can generate these using the `recommendForAllUsers` method:

```
Cmd 31

1 userRecs = model.recommendForAllUsers(10)

userRecs: pyspark.sql.dataframe.DataFrame
  userId: integer
  recommendations: array
    element: struct
      movieId: integer
      rating: float

Command took 1.12 seconds -- by n.j.topping@salford.ac.uk at 3/5/2023, 7:56:25 PM on My Cluster
```

You can see from the above that this generates a DataFrame with a nested structure; the recommendations column is an array which contains the movieId and rating for the top 10 movies. You can also inspect the DataFrame directly to see this:

```
1 userRecs.show(truncate=False)
```

► (2) Spark Jobs

```
+-----+
|userId|recommendations
+-----+
|0      |[[{1916, 19.567131}, {3844, 16.320473}, {2531, 16.094702}, {2007, 15.985681}, {5135, 15.597923},
|1      |[[{47629, 7.089125}, {166534, 6.5254145}, {1411, 6.3896356}, {3388, 6.296742}, {2732, 6.286619},
|2      |[[{2843, 8.401294}, {6650, 7.546653}, {112290, 7.424156}, {3476, 7.374549}, {8973, 7.284415}, {
|3      |[[{74754, 6.9189706}, {42418, 5.9798703}, {2340, 5.7462544}, {26865, 5.6173134}, {93563, 5.5503
|4      |[[{71520, 7.6689963}, {8957, 6.8813376}, {55363, 6.8511453}, {2318, 6.7889676}, {2867, 6.417162
|5      |[[{106100, 9.06338}, {2459, 8.730718}, {67695, 8.573079}, {3784, 8.053167}, {94015, 7.9077263},
|6      |[[{103228, 6.434021}, {71264, 6.313224}, {91976, 6.2418904}, {6942, 5.996287}, {7842, 5.917795}
|7      |[[{1241, 7.8019238}, {74754, 7.7769284}, {6993, 7.134972}, {1916, 7.0035996}, {28, 6.613567}, {
|8      |[[{955, 7.484451}, {2349, 7.3731976}, {177593, 6.501405}, {3503, 6.479802}, {1272, 6.456595}, {
|9      |[[{215, 7.6763163}, {3814, 6.950236}, {501, 6.9111977}, {968, 6.8424053}, {112623, 6.774157}, {
|10     |[[{86911, 7.039669}, {59258, 6.9749646}, {7842, 6.910271}, {87529, 6.3486876}, {2459, 6.338048},
|11     |[[{1218, 7.7179837}, {27611, 6.9998403}, {2202, 6.9703884}, {3022, 6.932374}, {148881, 6.899632
|12     |[[{1255, 7.4046845}, {25771, 7.1023526}, {3266, 7.0181875}, {119141, 7.0031247}, {148881, 7.001
|13     |[[{86347, 7.119792}, {417, 6.9651504}, {86345, 6.724663}, {52885, 6.4428496}, {194, 6.346309},
|14     |[[{3910, 11.405136}, {89753, 11.2978}, {535, 11.252523}, {1212, 10.694298}, {3358, 10.009666},
|15     |[[{3099, 8.493177}, {1218, 8.19281}, {3503, 7.6743593}, {69406, 7.639838}, {80693, 7.5364633},
|16     |[[{7842, 5.193494}, {95441, 5.0837846}, {3266, 5.079641}, {119141, 5.002669}, {8477, 4.8916326}
|17     |[[{7842, 6.414626}, {8477, 5.9649644}, {3030, 5.8375034}, {3200, 5.7261043}, {4102, 5.707037},
Command took 9.88 seconds -- by n.j.topping@salford.ac.uk at 3/5/2023, 10:01:26 PM on My Cluster
```

12. We can return the recommendations for a specific user using the below (in this case, we have selected the user with userId of 1, but we could do the same for any user.)

```
1 from pyspark.sql.functions import explode
2
3 userRecs.where(userRecs.userId == 1).select("recommendations")\
4 .withColumn("recommendations", explode("recommendations"))\
5 .select("recommendations.movieId", "recommendations.rating")\
6 .join(movies, ["movieId"])\
7 .show(truncate=False)
```

► (3) Spark Jobs

```
+-----+-----+-----+-----+
|movieId|rating  |title                                     |genres      |
+-----+-----+-----+-----+
|47629   |7.089125|The Queen (2006)                         |Drama       | | |
|166534  |6.5254145|Split (2017)                             |Drama|Horror|Thriller|
|1411    |6.3896356|Hamlet (1996)                            |Crime|Drama|Romance |
|3388    |6.296742 |Harry and the Hendersons (1987)          |Children|Comedy    |
|2732    |6.286619 |Jules and Jim (Jules et Jim) (1961)       |Drama|Romance     |
|5015    |6.2266955|Monster's Ball (2001)                     |Drama|Romance     |
|1243    |6.172601 |Rosenkrantz and Guildenstern Are Dead (1990)|Comedy|Drama      |
|67695   |6.1405964|Observe and Report (2009)                 |Action|Comedy     |
|161582  |6.1061063|Hell or High Water (2016)                 |Crime|Drama       |
|106766  |6.0577126|Inside Llewyn Davis (2013)                |Drama         |
+-----+-----+-----+-----+
```

## Part 8: Hyperparameter Tuning

There are a number of hyperparameters we can set when using ALS in Spark. Some main parameters are:

- rank: the number of latent factors in the model (defaults to 10).
- maxIter: the maximum number of iterations to run (defaults to 10).
- regParam: specifies the regularization parameter in ALS (defaults to 1.0). Regularization is used to help avoid the model overfitting the training data
- implicitPrefs: specifies whether to use the explicit feedback ALS variant or one adapted for implicit feedback data (defaults to false which means using explicit feedback).

### Challenge 4

- For the fourth challenge, you should follow the same steps as last week to tune the hyperparameters and explore whether you can improve the model performance. You should try the following settings for the parameters:
  - Try values of 5, 10 and 15 for the rank
  - Try values of 0.001, 0.005, 0.01, 0.05, 0.1 for the regParam
- Generate some new predictions for yourself based on the new model you have built

You can use the below code to print the hyperparameters selected through grid search. As we did last week, you can also use the Databricks Experiment UI to compare performance of each of the models and verify that the model with the below hyperparameters did indeed perform best.

Cmd 38

```
1 # Select best model and identify the parameters
2
3 bestModel = gridsearchModel.bestModel
4
5 print("Parameters for the best model:")
6 print("Rank Parameter: %g" %bestModel.rank)
7 print("RegParam Parameter: %g" %bestModel._java_obj.parent().getRegParam())
```

Parameters for the best model:

Rank Parameter: 5

RegParam Parameter: 0.1

Command took 0.08 seconds -- by n.j.topping@salford.ac.uk at 3/5/2023, 9:51:29 PM on My Cluster