SUMMARY

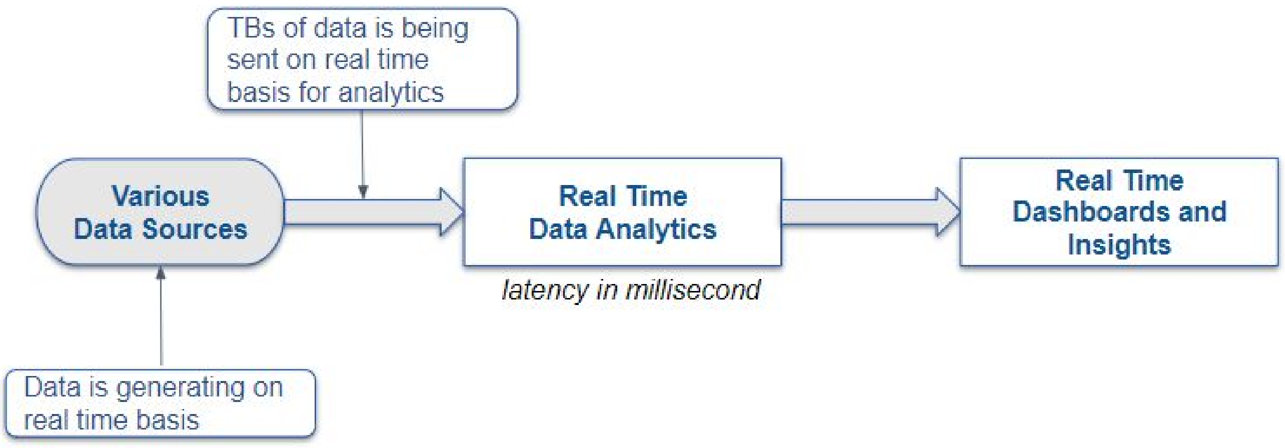
PySpark Implementation - Streaming and ALS

In this session, you implemented a case study on PySpark based on your learnings from the previous modules. You were also introduced to PySpark libraries that helped you in the implementation of the case study. Now, let's revise all these topics in this document.

Real-Time Analytics

In this segment, you explored the first case study. Let's quickly revise the concepts that you learnt and recall the implementation of this case study.

Streaming is a self-explanatory process. As the term suggests, it is the process of receiving the data by the end-user for continuous use. Real-time analytics (RTA) plays an important role in many day-to-day activities. For example, Google's self-driving cars continuously capture and process images and make decisions after analysing them. This process needs to occur in a significantly short duration of time, as the slightest delay might lead to accidents. All these decisions are based on real-time streaming of data. The diagram given below shows the process of real-time analytics.


TBs of data is being sent on real time basis for analytics
Various Data Sources
Real Time Data Analytics
latency in millisecond
Real Time Dashboards and Insights
Data is generating on real time basis

Some of the important terms related to real-time analytics are as follows:

1. **Data sources:** The system receives real-time data from various data sources in order to perform analytics on the data. For example, you can receive live Twitter data that includes users' tweets and perform analytics on it to identify trends in those tweets.

2. **Real-time analytics (RTA):** This refers to the process of analysing data that is received from various sources without any delay and delivering the results in real time.

3. **Real-time dashboards and insights:** Once the analysis is performed on the data, the insights from the data are displayed as visualisations on a real-time basis.

Now that you have revised real-time processing, let's recall the advantages of real-time processing over batch-time processing. Some of these advantages are as follows:

- **Self-driving cars:** You have already understood how RTA helps in processing real-time data from high-resolution cameras in self-driving cars to make any driving decisions.
- **Social media trends:** You can analyse the top trending topics using real-time data and posts from users across the world.
- **Recommendations:** After you watch a movie or video on Netflix or YouTube and rate it according to your liking, these applications use this rating data to recommend the most appropriate movies or videos to you on a real-time basis.
- **IoT Sensors:** Many IoT devices need to synchronise with one another in order to generate the desired results in real time. For example, electricity bills are generated using electric meters that record consumption in real time.

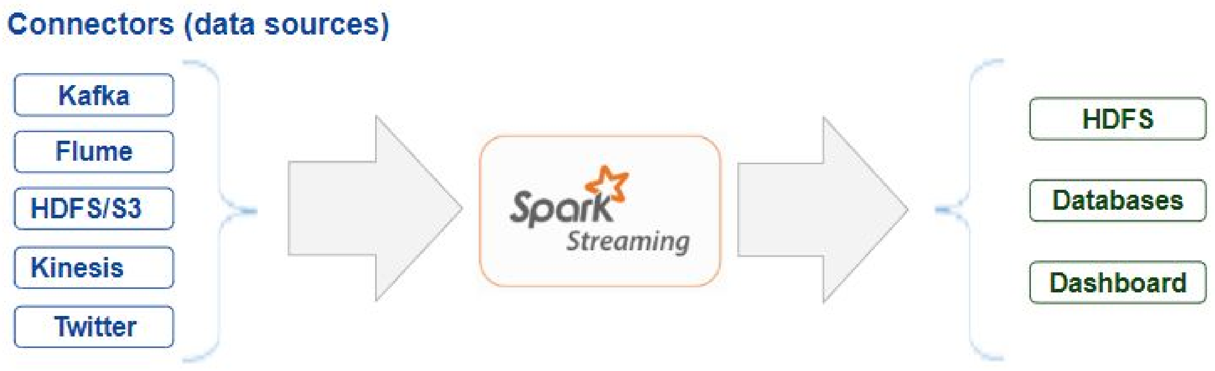
Introduction to Spark Streaming

So far, you have learnt about real-time analysis. Let's now recall the following points about Spark Streaming:

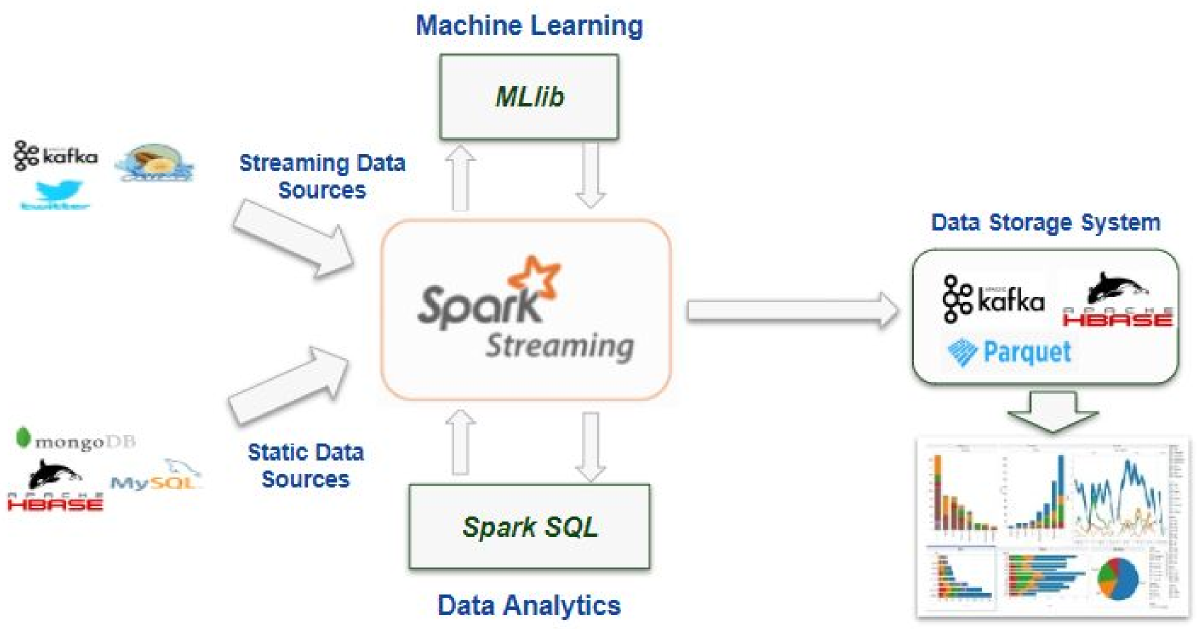
1. Spark Streaming is an extension of Spark Framework and is similar to other Spark APIs, such as MLlib and GraphX.
2. It provides streaming capabilities that help you fetch real-time data from various sources.
3. The coding pattern is similar to that of Spark-structured APIs and RDDs.
4. It is highly scalable, as you can fetch data in extremely high throughput and perform analytics on it.
5. Let's take the example of Twitter. Here, data related to thousands of Tweets are sent to the server on any given day. Now, suppose the number of Tweets increases to millions due to an activity or event in a certain part of the world, which further increases the size of the data. Using Spark Streaming, you can manage such a huge amount of data intake by increasing the size of the cluster.
6. As Spark provides streaming capabilities, it also ensures that the latency of processing the data is low.
7. It is highly fault-tolerant. For example, when a machine fails, Spark Streaming knows which task it should restart and assigns the remaining tasks to a new machine in order to resume the tasks.

Input/Output Connectors

In this segment, you will revise how Spark Streaming receives and performs analytics on data and then stores the result. As illustrated in the image given below, Spark Streaming fetches data from input connectors that are either static or real-time data sources.



The image given below shows how data is fetched and stored after it is analysed.



Essentially, data sources are of the following two types:

1. Streaming data sources
2. Static data sources

One of the differences between streaming data and static data is that the former contains real-time data.

Let's take the example of Twitter data to understand the process of Streaming.

- The real-time data that is generated from Twitter APIs is the streaming data, whereas data related to a user's name and location or user_id is considered static data that is stored in static data sources such as **HDFS**, **MongoDB** and **HBase**.

- The next step is to fetch real-time Tweets from the streaming data from Twitter and obtain user information of the Tweets from static data sources.

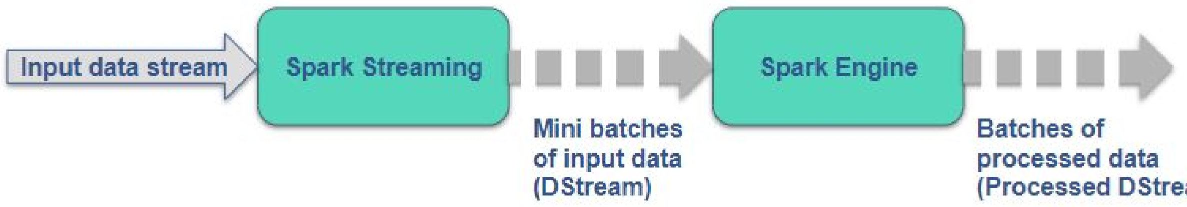
- Once you have collated the data, you can use various core Spark APIs, such as **SparkSQL** and **MLlib**, to perform data analytics and build machine learning models to derive insights from real-time data.

- After the analysis, you can store the output in any of the formats mentioned in the image above. You can also create interactive dashboards based on the real-time analysis of the data. This management of the output is done by output connectors, where you can store the data in a storage system or create real-time dashboards.

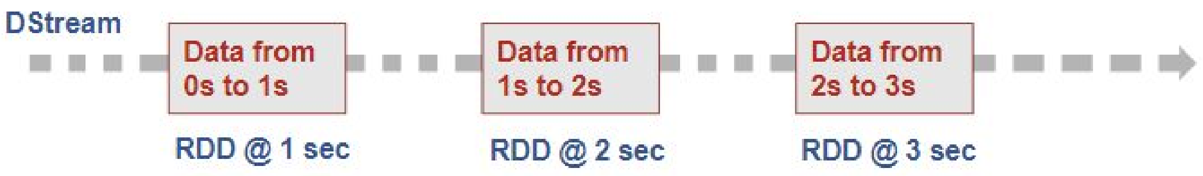
The Workings of Spark Streaming

Let's revise the important aspects of DStream and understand how data is divided into mini-batches. Spark Streaming divides real-time data streams into mini-batches of data called **DStreams (Discretized Streams).** This process is nothing but a **continuous sequence of RDDs of the same type** representing the continuous stream of data. Once the data is divided into DStreams, the Spark engine performs analytics on them and outputs the results in the same format as that of the DStreams.

The image given below shows how data is processed in the form of DStreams.


Input data stream
Spark Streaming
Mini batches of input data (DStream)
Spark Engine
Batches of processed data (Processed DStream)

As shown in the image provided above, data is received continuously from input connectors, and Spark Streaming divides this data internally into mini-batches called DStreams. Ultimately, Spark performs all the operations on these RDDs. As shown in the image given below, Spark Streaming collects data in the first second and creates a single RDD. After the first second, it creates another RDD in the next second, and so on. Ultimately, Spark Streaming creates a sequence of RDDs and performs analytics on them.

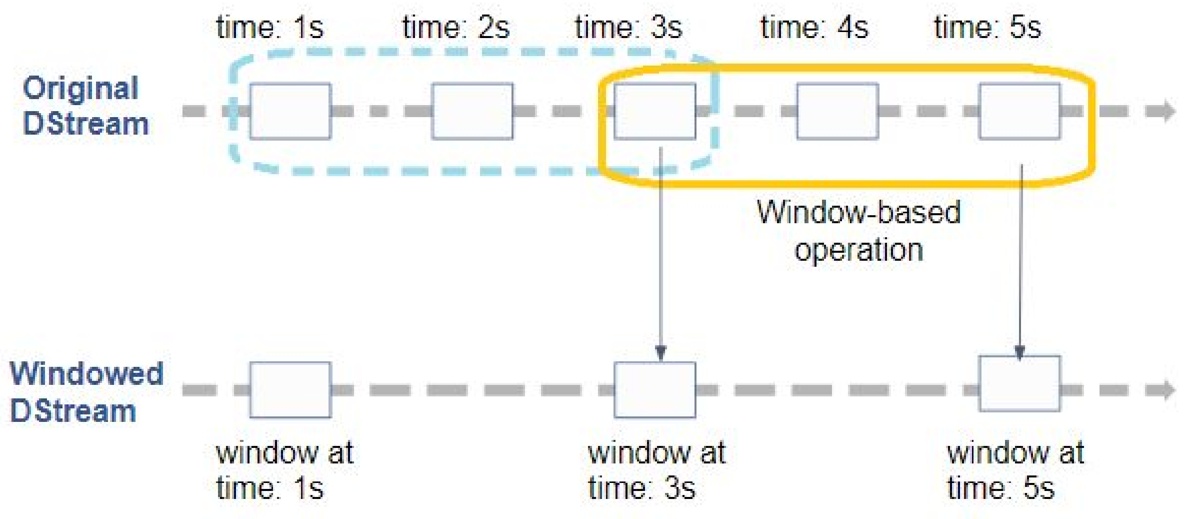


Spark supports both batch-processing and real-time processing. To implement these, it uses RDDs for batch-processing and DStreams for real-time processing. Essentially, DStream is a stream of RDDs that can be created using live data through Kafka or Twitter APIs. You can also transform an existing DStream in order to create a new DStream.

Window Operation and Spark Streaming Context

Let's recall what window operations are and how to set up the Spark Streaming Context.

A window operation is the process of applying a transformation over the sliding window of real-time data. When you perform a window operation, the source RDDs that fall within the window are combined and operated upon to produce the new RDDs of that windowed DStream. In other words, a window operation is the process of combining a series of RDDs into a single unit. You can recall this operation with the help of the diagram given below.



Let’s understand the image given above through the following points:

- Window length = 3 seconds; sliding interval = 2 seconds
- You need to group all the RDDs of the previous three seconds after every two seconds.
- If you are at ‘3 seconds’, then as per the window, you need to group the RDDs of ‘1 sec’, ‘2 sec’ and ‘3 sec’.
- Now, after two seconds (sliding interval), that is, at ‘5 seconds’, you need to group the RDDs of the last three seconds, i.e., group the RDDs of ‘3 sec’, ‘4 sec’ and ‘5 sec’.

Let's summarise the concepts of batch interval, window length and sliding interval:

1. **Batch interval**: This is the duration in seconds for which data is collected one at a time before it is processed. For example, if you set a batch interval of 10 seconds, then Spark Streaming will collect data for 10 seconds and perform the calculations on that data in the form of an RDD.

2. **Window size**: This is the interval of time in seconds for which historical data will be contained in the RDD before it is processed. For example, if you have a 10 seconds batch interval and a window size of 2, then, in this case, the calculation will be performed on 2 batches, the current batch and the previous batch (i.e. 20 seconds of data). E.g at time=3 unit, you will have data from the batch at time=2 unit and time=3 unit.

3. **Sliding interval**: It is the amount of time in seconds for how much the window will shift. In the previous example, the sliding interval is 1, hence the calculation is performed in each second, i.e., at time=1 unit, time=2 unit, time=3 unit. If you set the sliding interval=2, you will get a calculation at time=1 unit, time=3 unit, time=5 unit.

Now, you can create a StreamingContext in Spark as shown below. There are many PySpark classes as well as parameters such as master, appname, sparkHome, pyFiles etc. that SparkContext can assume. However, we are using only the following two parameters for now:

1. **master**: This is the URL of the distributed cluster to which SparkContext connects or the **local** machine to run locally with one thread. Here, in the example shown below, you can see that the master is set to the **local[2],** which means Spark will run locally with two worker threads as the logical cores on your machine.
2. **appName**: This indicates the name of your job. It shows the name of your application on the cluster UI.

```
#create local StreamingContext with two working thread and batch interval of 10s

from pyspark import SparkContext

from pyspark.streaming import StreamingContext

sc= SparkContext ("local[2]", "streamingDemo")

ssc= StreamingContext(sc, 300, 10)
```

upGrad

Summary

Twitter Hashtags Analysis

In this session, you implemented a case study on Spark Streaming. Let's recall your learnings from this segment.

Import Libraries and Tweet Formats

The process of performing real-time hashtags analysis can be divided into the following two parts:

1. Fetching real-time Twitter data
2. Performing Hashtag analysis on the real-time data

You began by using the '**Fetching_Twitter_Data**' notebook for fetching Tweets and used the following libraries to fetch real-time Twitter data:

- **tweepy:** This library is used to access the Twitter API. You can also use the tweepy library to extract Tweets from timelines, post or delete tweets, or follow or unfollow users on Twitter.
  Before importing this library, you need to install it in Python. The command to install the tweepy library in Python is as follows:

```
! pip3 install tweepy --user
```

- **OAuthHandler:** Before getting data from Twitter APIs, you need to register your client application with Twitter using the OAuthHandler library.

- **Stream:** This library allows you to extract a stream of data (Tweets) from the Twitter API. You can use the library as follows:

```
def send_twitter_data(c_socket):
  auth = OAuthHandler(consumer_key, consumer_secret)
  auth.set_access_token(access_token, access_secret)
  twitter_stream_data = Stream(auth, TweetsListener(c_socket))
  twitter_stream_data.filter(track=['corona'])
```

- **StreamListener:** This is an object under the tweepy library. So, to read the JSON file, you need to create a class using a StreamListener object using the following code:

```
class TweetsListener(StreamListener):
  def __init__(self, csocket):
     self.client_socket = csocket
```

The on_data method receives all the messages and, hence, can call functions according to the type of the message. Here, it receives the Tweets and prints the needed information. You can also send the Tweet's text data to the client_socket. Refer to the following code:

```
def on_data(self, tweet_json):
    try:
       tweet_data = json.loads( tweet_json )
       print("tweet_text: ", tweet_data['text'].encode('utf-8') )
       print("created_at: ", tweet_data['created_at'])
       print("name: ", tweet_data['user']['name'])
       print("location: ", tweet_data['user']['location'])
       print("\n")
       self.client_socket.send( tweet_data['text'].encode('utf-8') )
       return True
    except BaseException as e:
       print("Error on_data: %s" % str(e))
    return True
```

The *on_error* method is used to deal with any errors, e.g, if the stream gets disconnected. It can be implemented as follows:

```
def on_error(self, status):
    print(status)
    return True
```

- **Import JSON:** All the tweets extracted from the Twitter API are in the JSON format. Hence, to read the JSON format, you need to import this library as follows:

```
tweet_data = json.loads( tweet_json )
```

- **Import socket:** A socket is used to connect two nodes on a network so that they can communicate with each other. A server needs to bind the socket to a specific IP and port so that it can listen to the incoming requests on that IP and port. Hence, you need to create a socket object using the socket library. So, a socket connection is useful to get all the Tweets on a local server from Twitter APIs.

Fetching Tweets

Let's revise the process of authentication and socket creation. The steps involved in this process are as follows:

- **twitter_stream_data:** Essentially, you are receiving the Tweets from the **socket** that you have defined in your local machine. The Tweets are received from the Twitter API using the **Stream** library. As you previously learnt, the Tweets are in the JSON format. So, to read the JSON file format, we are using the **TweetsListener** class object, which was already defined in the previous segment.

- In the last step, the Tweets are filtered to get those that contain the word 'corona'. This gives you the DStream of the tweets that contain the word 'corona' into the variable twitter_stream_data. These steps are implemented in the code given below.

```
def send_twitter_data(c_socket):
 auth = OAuthHandler(consumer_key, consumer_secret)
 auth.set_access_token(access_token, access_secret)
 twitter_stream_data = Stream(auth, TweetsListener(c_socket))
 twitter_stream_data.filter(track=['corona'])
```

- You can create and use a socket as shown below:

```
# Create a socket object
s = socket.socket()

# Get local machine name
host = "127.0.0.1"

# Reserve a port for your service.
port = 7773

# Bind to the port
s.bind((host, port))

# print
print("Listening on port: %s" % str(port))
```

By executing the send_twitter_data() function, you were able to see the structure in which the Tweets are printed. Remember that this structure was defined in the TweetListener() function.

Hashtag Analysis on Tweets

Let's revise the process of performing hashtag analysis on the Tweets extracted above. Before you begin with the analysis, you need run the following code in your Jupyter Notebook:

```
import os
import sys
# Here you need to have same Python version on your local machine adn on worker node i.e. EC2. here both should have python3.
os.environ["PYSPARK_PYTHON"] = "/bin/python3"
os.environ["JAVA_HOME"] = "/usr/java/jdk1.8.0_161/jre"
os.environ["SPARK_HOME"] = "/home/ec2-user/spark-2.4.4-bin-hadoop2.7"
os.environ["PYLIB"] = os.environ["SPARK_HOME"] + "/python/lib"
sys.path.insert(0, os.environ["PYLIB"] + "/py4j-0.10.7-src.zip")
sys.path.insert(0, os.environ["PYLIB"] + "/pyspark.zip")
```

```
os.environ["PYTHONIOENCODING"] = "utf8"
```

Note that while defining the Spark Streaming Context, you need to add the value of the sliding interval, which is 10 seconds in this case, as follows:

```
sc = SparkContext("local[2]","TwitterStreaming")
sc.setLogLevel('ERROR')
ssc = StreamingContext(sc, 10)
```

To obtain the desired results, you performed the following transformations on DStreams:

1. You already had an input DStream that received real-time Tweets in the variable twitter_stream_data.
2. Using the flatMap() command, you were splitting sentences into words and getting those sequences of RDDs into **word_data** using the following code:

   ```
   word_data=twitter_data.flatMap(lambda text: text.split(“ “))
   ```

3. Using a filter, you obtained all the words that started with a hashtag(#) and converted them into lower case, and then moved the words into the new DStream called **filtered_data** using the following code:

   ```
   filtered_data=word_data.filter(lambda word: word.lower().startswith(“#”))
   ```

4. To convert the words into lower case, you used the following code:

   ```
   hashtag_count = filtered_data.map(lambda word: (word.lower(), 1)).reduceByKey(lambda a, b:a+b)
   ```

5. To mapping each tag to (word, 1), you used the following code:

   ```
   hashtag_sorted = hashtag_count.transform(lambda foo: foo.sortBy(lambda x:x[1], ascending = False))
   ```

6. You reduced and counted the occurrences of each hashtag using the following code: (action: reduceByKey) hashtags

   ```
   hashtag_sorted.pprint()
   ```

7. Finally, you got the **hashtag_sorted** DStream, and using pprint, you printed the results one by one according to the window length and sliding interval.

So, you performed the following tasks in this process:

- You read the data from the socket into **stream_data** in the form of DStream using the following code:

  ```
  stream_data=ssc.socketTextStream(“127.0.0.1”,7773)
  ```

- Then, you took the sliding interval as 10 seconds to convert the aforementioned DStream into another DStream. You mentioned the window length as 20 seconds using the following code:

  ```
  twitter_data = stream_data.window(20)
  ```

- Finally, you applied the transformations on these internally generated DStreams and got the **hashtag_sorted** DStream that you wanted to print.

Getting the Trending Hashtags

Until now, we revised how to read data in a DStream and the various transformations on it. Now, we will recall how the output on this data is generated. The Spark Streaming that code we have written does not execute until we start 'ssc' due to lazy evaluation. As soon as we execute ssc.start(), Spark will use the lineage and DAG and start running the whole sequence of code. We can start 'ssc' using following command:

```
ssc.start()
```

The awaitTermination() command waits for the termination signal from the user. When it receives a stop signal from the user, its streaming context is stopped. To stop the Spark Streaming Context, you can use the following stop command:

```
ssc.awaitTermination()
```

Or

You can directly stop 'ssc' using the following command:

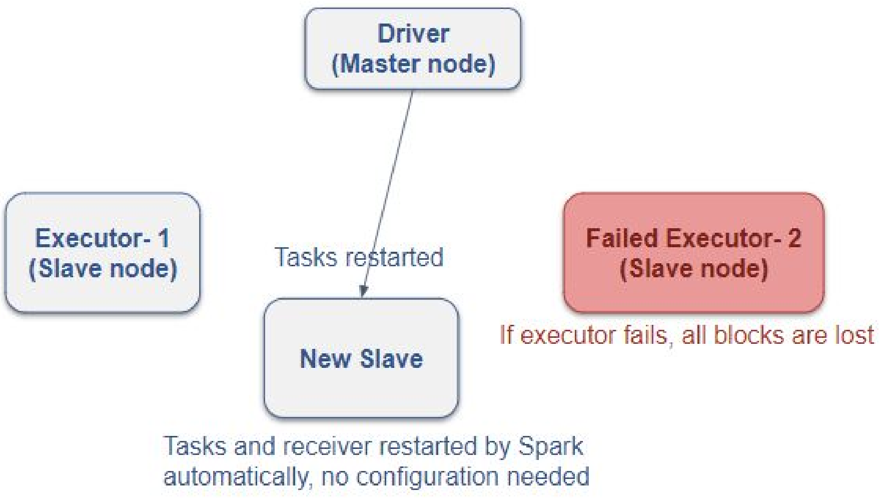
```
ssc.stop()
```

Fault Tolerance

We just revised that the output of transformations on the code does not generate until we start 'ssc' due to lazy evaluation. Let's now recall the concept of lazy evaluation. A faulty condition may result from either of the following failures:

- Driver (Master) failure
- Executor (Slave) failure

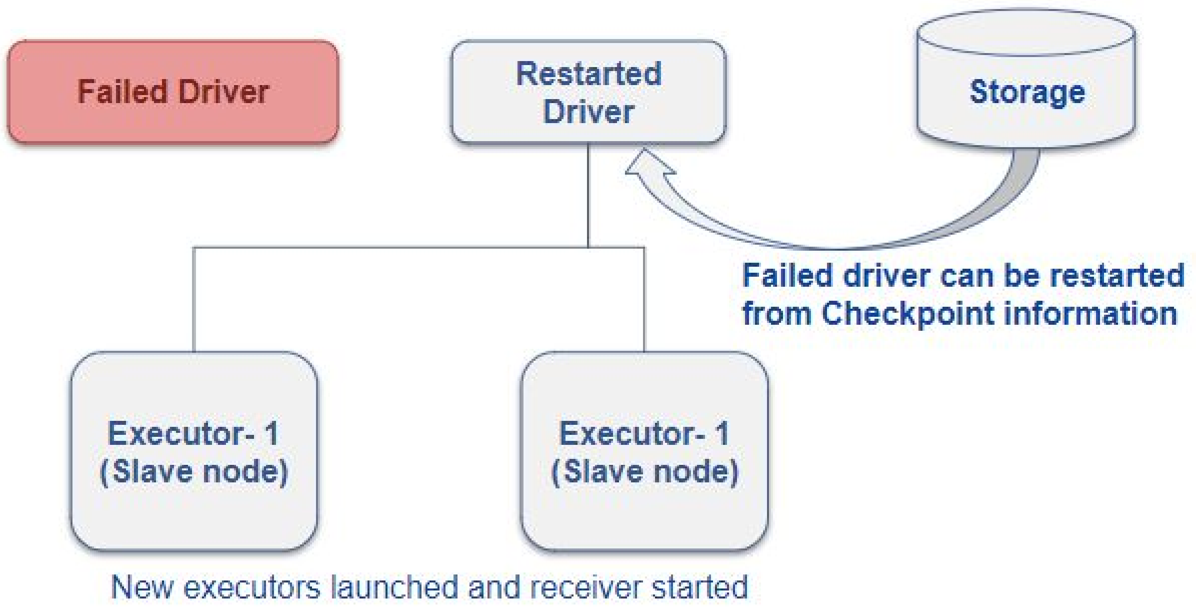
The diagram given below shows the first case, where the executor fails.


Driver
(Master node)
Executor- 1
(Slave node)
Tasks restarted
Failed Executor- 2
(Slave node)
If executor fails, all blocks are lost
New Slave
Tasks and receiver restarted by Spark
automatically, no configuration needed

In such cases, as you previously learnt, the master/driver node knows all the tasks that have been assigned to each node (slaves), owing to the lineage and DAG in Spark. Based on this knowledge, it will assign all the tasks to the newly created slaves. These slaves will start the tasks from that particular point where the fault occurred.

Checkpointing

You learnt that if an executor node fails, then the driver redirects all the tasks that were assigned to the executor node to a newly created executor. Since the master node contains all the metadata of all the slave nodes, i.e., the information of all the tasks that have been assigned to the slave nodes, the whole system will collapse if the driver fails. To prevent this, the concept of **checkpointing** was introduced. In checkpointing, all the metadata of the driver node is stored periodically in storage systems such as HDFS or S3. The image given below depicts the concept of checkpointing.


Failed Driver
Restarted Driver
Storage
Failed driver can be restarted from Checkpoint information
Executor- 1 (Slave node)
Executor- 1 (Slave node)
New executors launched and receiver started

To recover from a faulty condition caused by the failure of the driver, a new driver gets started, fetches all the metadata information from the storage, and resumes the task by assigning the tasks to the executors accordingly. This is called checkpointing.

The two types of checkpointing are as follows:

1. **Metadata checkpointing**: Here, only the metadata of the executors gets stored in the storage system, such as HDFS or S3. This is similar to a bookkeeping system. The metadata contains the information of the tasks that have been assigned to the executors. It does not store the real data in the form of RDDs.
2. **Data checkpointing**: Here, along with the metadata information of the executors, the intermediate RDDs are also stored periodically in the storage system.

Summary

Recommendation System

In this session, you were introduced to an end-to-end case study on the workings of a recommendation system, and you also learnt how to build a model for a recommendation system. Now, let's revise this concept.

Recommendation System

You must have observed that when you search for products such as Nike Hypervenom on Amazon or Myntra, the site recommends shoes that look similar, have similar costs and fulfil your purpose. The main objective of a recommendation system is to provide the most relevant and similar entity to users.

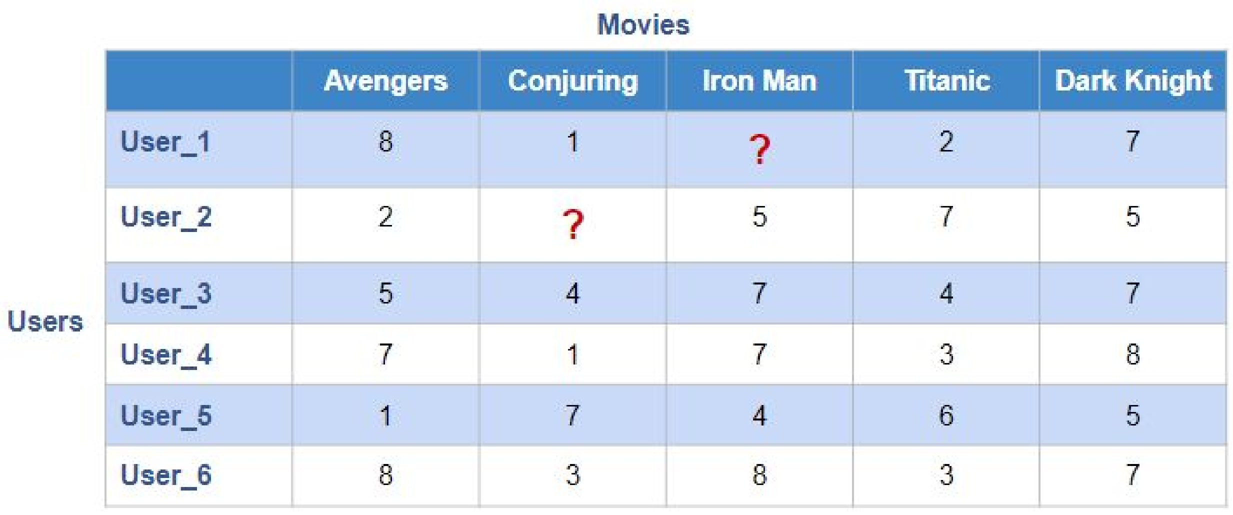
A recommendation system algorithm learns users' interests based on their browsing behaviour and then suggests products that have a high probability of being bought or liked by users. Such recommendation systems are used by almost all businesses to improve their customer experience and maximise revenue.

Collaborative Filtering

You know that recommendation systems play a significant role in improving customer experience and maximising revenue for certain businesses. Now, let's revise the methods of recommendation.

The two types of collaborative filtering are as follows:

1. **User-based collaborative filtering:** This method identifies users who are similar to other already existing users who have rated or queried in a similar way.
2. **Item-based collaborative filtering:** This method identifies items that are similar to other items based on user ratings. In the example shown in the image given below, each movie is an item.



Movies

Users

	Avengers	Conjuring	Iron Man	Titanic	Dark Knight
User_1	8	1	?	2	7
User_2	2	?	5	7	5
User_3	5	4	7	4	7
User_4	7	1	7	3	8
User_5	1	7	4	6	5
User_6	8	3	8	3	7

Introduction to Alternating Least Squares (ALS)

You previously learnt how to find similar users and items based on the ratings given by different users and accordingly recommend these items to the users. Now, let's revise the alternating least squares (ALS) algorithm, which is a state-of-the-art recommendation algorithm. Some of the features of the ALS algorithm are as follows:

1. Alternating least squares (ALS) is a matrix factorisation algorithm that separates the user rating matrix (which you learnt about in the previous segment) into two matrices and performs multiplication of these two separated matrices to deliver predicted ratings given by individual users for each item in the list.

2. The matrix multiplication of these two matrices occurs parallelly, not sequentially. You will learn more about this concept in the segment on parallelism.

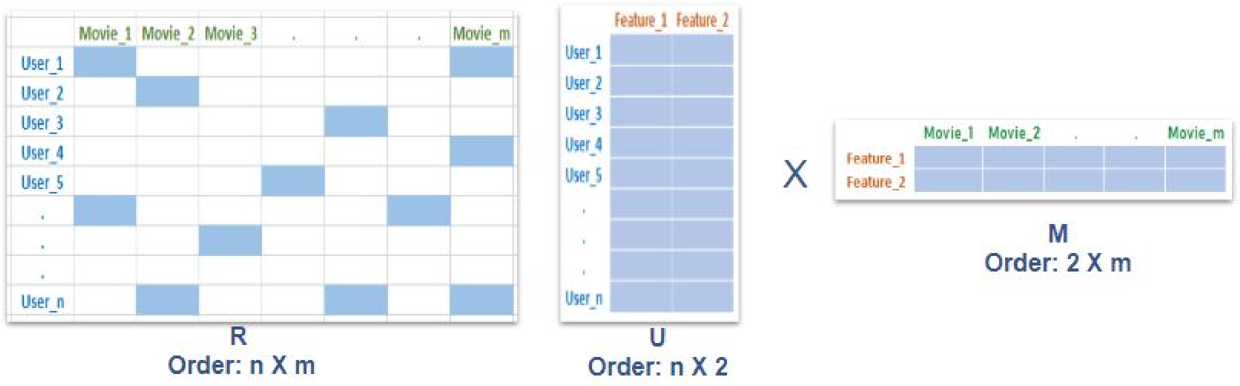
3. This algorithm is implemented in the Apache Spark ML library.

4. It is built for implementing large-scale collaborative filtering on huge data sets.

5. The ALS algorithm can also work with sparse matrices unlike other recommendation algorithms.

Matrix Factorisation

In order to revise the concept of matrix factorisation, let's assume that 'R' is an enormous matrix of the ratings of Netflix movies, as shown in the image given below:



In this matrix, a certain amount of ratings is available, but many of them are missing; hence, the entry of matrix will remain blank. So, you can imagine how sparse this matrix is going to be. Techniques such as collaborative filtering tend to fail when the matrix is this sparse. In such cases, the solution is provided by methods such as **ALS** that factorise the original matrix 'R' into smaller matrices. Here, the user matrix is divided into two smaller matrices: matrix 'U' (users) and matrix 'M' (movies).

Before beginning with the example, let's understand the following features based on which one would essentially differentiate between movies:

- Genres: Action, superhero, horror, romantic or comedy

- Actors
- Type of industry: Hollywood or Bollywood
- Time of release: An old or a new movie
- Directors

Now, let’s consider the feature ‘genres’ of movies to understand how ALS functions. Suppose you like action and superhero movies, dislike horror movies and moderately like comedy and romantic movies.
So, considering the five genres, action, superhero, horror, romance and comedy, you can create your own vector, with the range (0: lowest, 1: highest), as follows:

	Action	Superhero	Horror	Romantic	Comedy
User_1	0.4	0.4	0.0	0.1	0.1
User_2	0.1	0.1	0.4	0.4	0.0
User_3	0.4	0.05	0.4	0.15	0.0

Based on this single-user matrix, it is clear that User_1 is interested in action and superhero movies, not at all interested in horror movies and moderately interested in romantic and comedy movies. Similarly, User_2 likes horror and romantic movies but is the least interested in comedy and action movies.

Now, let’s consider a specific movie, The Avengers. This movie is categorised in both genres, action and superhero. Also, the movie Titanic is categorised as a romantic and partly comedy movie. So, if you create a similar matrix based on this information, then you will derive the following data:

	Action	Superhero	Horror	Romantic	Comedy
The Avengers	0.5	0.4	0.0	0.0	0.1
Titanic	0.0	0.0	0.0	0.8	0.2
Conjuring	0.0	0.0	0.9	0.05	0.05

As you can see in this table, the genres of movies act as features for the user and movie matrices. In the case of ALS, these features are a combination of different features that you can think of, which are decided by the model and, hence, are called latent features. Thus, features play a significant role in factoring the rating matrix, R, into two different matrices.

Matrix Factorisation and Cost Function

Let’s recall the cost function of the ALS algorithm and understand how the algorithm minimises the cost function, which, in turn, reduces errors in the predicted ratings.

You learnt that ALS segregates the main user matrix into two different matrices. Now, based on the features, the ALS algorithm fills the scores for each feature in the feature matrices and, finally, reduces the errors in the predicted results using the cost function.

Any machine learning model trains by minimising a cost function or a loss function using the following formula:

$$min \sum (R_{ij} - U_i X M_j^T)^2 + Regularization$$

Where,

- R_{ij} is the rating for the ith user and the jth movie
- U_i is a vector for the ith user
- $M_j$is a vector for the jth movie

This formula is the cost function that is reduced by the ALS to derive the most appropriate values corresponding to the matrix. If you calculate all the errors in this manner and the sum of the squares of all the errors, then it will result in the following:

$$\sum (R_{ij} - U_i X M_j^T)^2$$

The second term in the formula given above is the regularisation term, which is used to avoid overfitting of the model. Overfitting is a situation in which the model is also able to predict correctly for noise data, which is not favourable for a model. Do not worry about all these concepts discussed here because the upcoming modules will cover these.

ALS Algorithm and Parallelisation

In this segment, we will revise the ALS algorithm and the concept of parallelisation.

The singular value decomposition (SVD) theorem can be used to fill the entries in these matrices, but this algorithm will not work owing to the following reasons:

1. The matrix may not fit into the Spark memory.
2. It is a sparse matrix, which means that millions of entries are unavailable in it.

When multiple iterations occur in the ALS algorithm, they ultimately lead to the convergence of the matrix values. Let's revise this process in the following steps:

1. The algorithm starts with random value initialisation in matrix 'U' and matrix 'M'.
2. Next, it performs matrix multiplication to derive the values of the predicted ratings and then calculates the error by subtracting the resulting values from the actual values.
3. Then, it updates matrix 'U' using the values of the old matrix 'U' and matrix 'M' as well as the error terms, as shown in the algorithm given above.
4. Similarly, it updates matrix 'M' using the values of the old matrix 'M' and matrix 'U'.

5. This process continues until the algorithm error converges, which means that the user and movie matrices do not change with the iterations. When you implement ALS, you can fix the number of iterations to a particular value, for example, 3 or 5, and stop the process. In the next session on the implementation of ALS, you will get a clearer understanding of iteration and error convergence.

To summarise, the algorithm fixes matrix ‘M’ while updating matrix ‘U’, and vice versa. Simply put, you are alternating between the matrices to calculate the value of each matrix. Hence, this algorithm is called alternating least squares.

```
Initialize matrix U and M with random values
repeat
    for random $R_i^j$ in R do
        error = $R_{ij} - U_i X M_j^T$
        $U_i^{new}$ = $U_i^{old}$ - alpha (error * $M_j^T$ + Regularization)
        $M_j^{new}$ = $M_j^{old}$ - alpha (error* $U_i^T$ + Regularization)
    end for
until convergence
```

All the vector multiplications of the user and movie matrices occur parallelly, not sequentially. To understand this concept better, let’s consider an example. Suppose you have two users and two movies with three features each. The user and movie matrices are as follows:

User Matrix:

	Feature_1	Feature_2	Feature_3
User_1	2	3	4
User_2	1	4	5

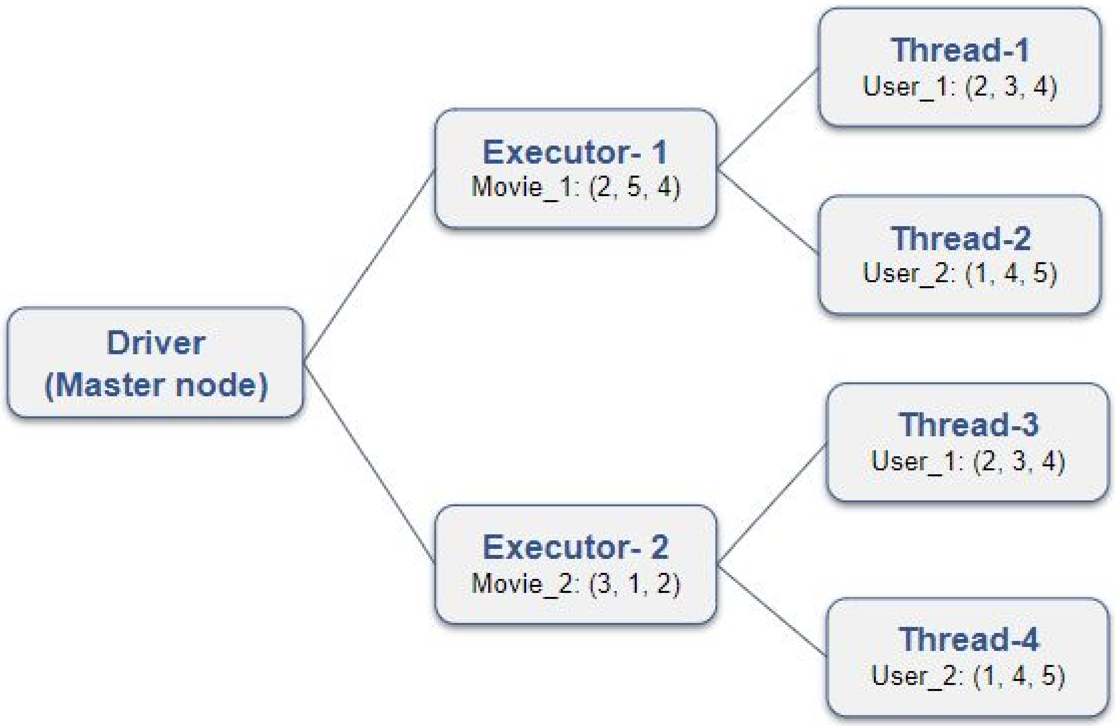
Movie Matrix:

	Feature_1	Feature_2	Feature_3
Movie_1	2	5	4
Movie_2	3	1	2

So basically, you have four tasks that have to occur in order to find the predictions, which are as follows:
Task A: User_1 X Movie_1: (2, 3, 4) X (2, 5, 4)
Task B: User_2 X Movie_1: (1, 4, 5) X (2, 5, 4)
Task C: User_1 X Movie_2: (2, 3, 4) X (3, 1, 2)

Task D: User_2 X Movie_2: (1, 4, 5) X (3, 1, 2)

The process is depicted in the diagram given below:



So, in the parallelisation process, all four tasks occur parallely on different executors in the cluster.

upGrad

Summary

ALS Implementation

Let’s recall how you implemented the ALS algorithm in PySpark.

Data Loading

Let’s recall how to load and visualise data in a matrix form using Spark.

Basically, Python operations are used only to build an understanding of some of the concepts; they do not find use in building the ALS model, which is why we are not focusing on gaining hands-on experience using Python'.

If you run the code files on the EC2 machine, then you can run both Python and PySpark commands; however, in the EMR cluster, you can run only one command at a time. Also, you cannot install the Python libraries in the EMR cluster using commands such as follows:

```
! pip3 install numpy
! pip3 install matplotlib
! pip3 install pandas
```

You will notice that the recommendation model building will only be performed in PySpark. Also, to load the data from the S3 bucket into data frames, you need to specify the path of the files as follows:

```
Rating Data: “s3a://sparkdemonstration-mlc/ratings.csv’
Movies Data: “s3a://sparkdemonstration-mlc/movies.csv”
```

You can see that the number of rows is approximately 25 million. Even with such a high number of rows, you will notice that the time taken by Spark is significantly less compared with that taken by Python. So, once you have loaded the rating data into the data frame, you can visualise the ratings in a matrix form using Spark SQL.

Building the ALS Model

Let’s recall how to use the loaded data in order to build an ALS model step by step.

1. First, you need to split the data set into training, validation and test data sets in a 60-20-20 ratio. The validation data set is used to analyse the variation of the root mean square error (RMSE) with respect to the number of iterations in the model building.

2. You have to import the ALS library, which is built in the ml.recommendation package. So, to build the ALS model, you need to specify the columns such as userCol, movieCol and ratingCol as well as the regularisation parameter.

3. Next, you need to specify the number of iterations for which the model will run. Once all the parameters are set, you can build the model and fit it on the training data.

4. Once you fit the model on the training data, you need to test it using the test data set and calculate the RMSE to determine the average deviation of the model. This is known as the model evaluation phase of the model building process, in which you judge the performance of the model on the basis of the value of the cost function. To perform this operation, you can follow the steps given below:

    a. First, you need to predict the ratings based on the model that you built. To obtain these ratings, you can apply .transform() on the test data and then store it in a new data frame, 'predictions'.

    b. Using these two values (the predicted and actual values), you can calculate the RMSE value as follows:

```
rmse = evaluator.evaluate(predictions)
```

So, in this way, you built the ALS model with an iteration value of 5.

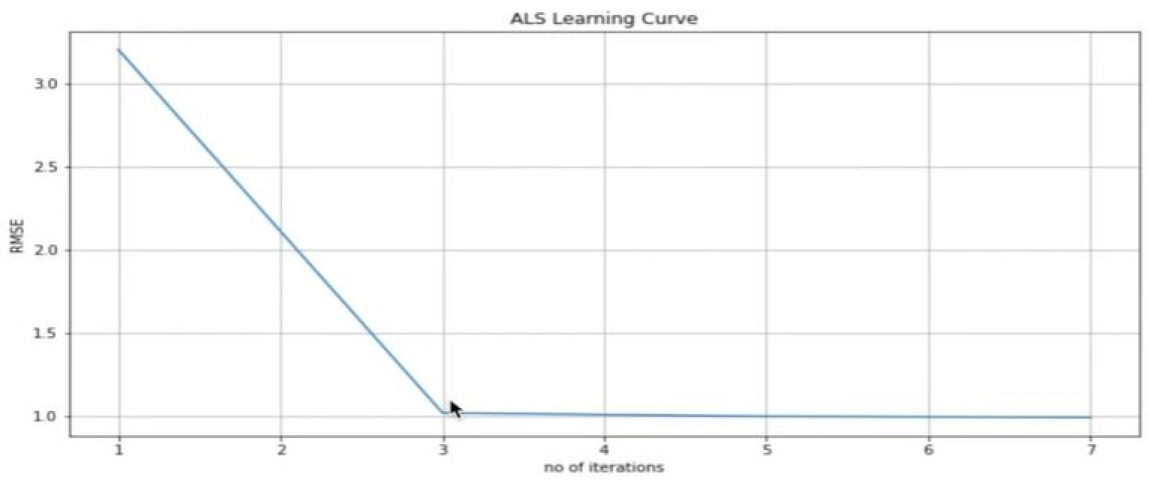
RMSE Value Versus Number of Iterations

The RMSE values vary with respect to the number of iterations. Now, let's recall this concept.

Recall a special type of parameter that you learnt about, hyperparameters. These remain constant during the training of the model. You can tune these hyperparameters to find the optimal values of the parameters of the model such that the cost function is minimised. Hyperparameter tuning is one of the key steps in the model evaluation phase. Let's learn how you can achieve this in ALS:

- To tune a hyperparameter, you need to observe the variation of the RMSE value in relation to the hyperparameter, which, here, is the number of iterations. For this, you can take an array of different values of maxIter, which represent the number of iterations and can run the model for the different values considered: iter_array = [1,3,5,7].

- When you plot the RMSE value against the different values of maxIter and the number of iterations, you will obtain the plot given below:



In this plot, you can notice a slight variation in the RMSE value when the number of iterations increases from 3 to 7. However, there is a drastic difference in the RMSE values when the number of iterations increases from 1 to 3. Therefore, instead of taking an iteration value of 5, you could take a value of 3 because this will reduce the amount of resources and the time required for training with almost the same RMSE values.

Recommendation Using the ALS Model

You created the ALS Model, and now, you can use it to recommend movies to users based on the predicted ratings as follows:

- First, you can generate a list of the top 10 user recommendations for a particular movie using the inbuilt function *.recommendForAllItems(10)*.
- Then, you can recommend this list of top 10 movies to each user.
- Similarly, you can repeat this for 20 movies or more.

An important point to note here is that you can fine-tune the model even further by selecting appropriate hyperparameters in the model and obtaining more appropriate values of the user-predicted ratings.