

Program 5. Machine Learning with MLlib

i. Implement a linear regression model using MLlib in Scala

ii. Evaluate the model's performance.

iii. Build a classification model (e.g., Logistic Regression) using MLlib in Scala

iv. Evaluate the classification model.

Sol:

MLlib is Spark's library of machine learning functions. Designed to run in parallel on clusters, **MLlib contains a variety of learning algorithms** and is accessible from all of Spark's programming languages.

MLlib lets us invoke various algorithms on distributed datasets, representing all data as **RDDs**

MLlib introduces a few data types (e.g., labeled points and vectors), but at the end of the day, it is simply a set of functions to call on **RDDs**.

For example, to use MLlib for a text classification task (e.g., Identifying spammy emails), you might do the following:

1. Start with an RDD of strings representing your messages.
2. Run one of MLlib's **feature extraction** algorithms to convert text into numerical
Features (suitable for learning algorithms); this will give back an RDD of vectors.
3. Call a **classification** algorithm (e.g., logistic regression) on the RDD of vectors;
this will give back a model object that can be used to classify new points.
4. Evaluate the model on a test dataset using one of MLlib's evaluation functions.

```
import org.apache.spark.mllib.regression.LabeledPoint
import org.apache.spark.mllib.feature.HashingTF
import org.apache.spark.mllib.classification.LogisticRegressionWithSGD
val spam = sc.textFile("spam.txt")
val normal = sc.textFile("normal.txt")
// Create a HashingTF instance to map email text to vectors of 10,000 features.
val tf = new HashingTF(numFeatures = 10000)
// Each email is split into words, and each word is mapped to one feature.
val spamFeatures = spam.map(email => tf.transform(email.split(" ")))
val normalFeatures = normal.map(email => tf.transform(email.split(" ")))
// Create LabeledPoint datasets for positive (spam) and negative (normal) examples.
```

```

val positiveExamples = spamFeatures.map(features => LabeledPoint(1, features))
val negativeExamples = normalFeatures.map(features => LabeledPoint(0, features))
val trainingData = positiveExamples.union(negativeExamples)
trainingData.cache() // Cache since Logistic Regression is an iterative algorithm.
// Run Logistic Regression using the SGD algorithm.
val model = new LogisticRegressionWithSGD().run(trainingData)
// Test on a positive example (spam) and a negative one (normal).
val posTest = tf.transform(
  "O M G GET cheap stuff by sending money to ...".split(" "))
val negTest = tf.transform(
  "Hi Dad, I started studying Spark the other ...".split(" "))
println("Prediction for positive test example: " + model.predict(posTest))
println("Prediction for negative test example: " + model.predict(negTest))

```

Classification and **regression** are two common forms of supervised learning, where Algorithms attempt to **predict a variable from features of objects using labeled training data**

The difference between them is the type of variable predicted.

Regression is used to predict **continuous values**. Classification is used to predict which class a data point is part of (**discrete value**).

EXAMPLES:

Example on Regression: I have a house with W rooms, X bathrooms, Y square-footage and Z lot-size. Based on other houses in the area that have recently sold how much (amount) can I sell my house for? I would use regression for this kind of problem....here we are following the continuity based on some other similar model

Example on Classification: I have an unknown fruit that is yellow in color, 5.5 inches long, diameter of an inch, and density of X. What fruit is this? I would use classification for this kind of problem to classify it as a banana (as opposed to an apple or orange).

In classification, the **variable is discrete** (i.e., it takes on a finite set of values called classes)

Example: classes might be **spam** or **nospam** for emails, or the language in which the text is written

In regression, the **variable predicted** is **continuous** (e.g., the height of a person given her age and weight)

Both classification and regression use the **LabeledPoint** class in **MLlib**

NOTE: Regression is a technique for determining the statistical relationship between two or more variables where a change in a dependent variable is associated with, and depends on, a change in one or more independent variables.

Linear regression is a basic and commonly used type of predictive analysis. The overall idea of regression is to examine two things:

Classification

MLlib introduces a few data types (e.g., **labeled points** and **vectors**), but at the end of the day, it is simply a set of functions to call on **RDDs**.

For example, to use MLlib for a text classification task (e.g, Identifying spammy emails), you might do the following:

1. Start with an RDD of strings representing your messages.
 2. Run one of MLlib's **feature extraction** algorithms to convert text into numerical
Features (suitable for learning algorithms); this will give back an RDD of vectors.
 3. Call a **classification** algorithm (e.g., logistic regression) on the RDD of vectors;
this will give back a model object that can be used to classify new points.
1. Evaluate the model on a test dataset using one of MLlib's evaluation functions

EXAMPLE 2: If we want to drive Sentimental Analytics on Demonetization

Some +ve words , some -ve words & neutral words we will formulate initially (**un-serviced**), **based** on the same ...we are doing to classify the different messages on that

Good, fantastic, superb, marvellous → +ve indicators

Poor, bad, not good, worst, waste, not happy → -ve indicators

Once we Classify the data , then only we can divide them into sectors called "Clustering" and we make "recommendations" which are the best once

In the entire process...one key aspect is "How to Model the Solution"?

DENSE VECTOR: every entry will be stored including ZERO Values

```
scala> import org.apache.spark.mllib.linalg.{Vector, Vectors}
```

```
import org.apache.spark.mllib.linalg.{Vector, Vectors}
```

```
scala> val dv: Vector = Vectors.dense(1.0, 0.0, 3.0)
```

```
dv: org.apache.spark.mllib.linalg.Vector = [1.0,0.0,3.0]
```

```
scala> dv.
```

```
apply argmax asML compressed copy equals foreachActive hashCode numActives  
numNonzeros size toArray toDense toJson toSparse
```

```
scala> dv.size
```

```
res0: Int = 3
```

SPARSE VECTOR: only NON-ZERO entries will be stored to save the space

```
scala> val ds = dv.toSparse
```

```
ds: org.apache.spark.mllib.linalg.SparseVector = (3,[0,2],[1.0,3.0])
```

```
scala> val sv2: Vector = Vectors.sparse(3, Seq((0, 1.0), (2, 3.0)))
```

```
sv2: org.apache.spark.mllib.linalg.Vector = (3,[0,2],[1.0,3.0])
```

```
scala> val dv: Vector = Vectors.dense(1.0, 0.0, 3.0,4.0,0.0,5.0)
```

```
dv: org.apache.spark.mllib.linalg.Vector = [1.0,0.0,3.0,4.0,0.0,5.0]
```

```
scala> val ds = dv.toSparse
```

```
ds: org.apache.spark.mllib.linalg.SparseVector = (6,[0,2,3,5],[1.0,3.0,4.0,5.0])
```

```
scala> ds.size
```

```
res1: Int = 6
```

```
scala>
```

Matrix : Collections of Such vectors

Program 6. Spark Streaming in Scala

- i. Set up a Spark Streaming application using Scala
- ii. Process and analyse real-time streaming data
- iii. Implement windowed operations on streaming data (e.g., windowed counts).

Sol:

Spark Streaming is a distributed data stream processing framework. It makes it easy to develop distributed applications for processing live data streams in near real time. It not only provides a simple programming model but also enables an application to process **high-velocity stream data**. It also allows the combining of data streams and historical data for processing.

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.

Data can be ingested from many sources like **Kafka, Flume, Twitter, ZeroMQ, Kinesis or TCP sockets** can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window.

Finally, processed data can be pushed out to file systems, databases, and live dashboards



Process Flow in Spark Streaming



Spark Streaming works in the below fashion:

- Spark Streaming receives live input data streams and divides the data into batches.
- Spark Engine will process the same data
- Once processing is done Spark engine will generate the final stream of results in batches.

StreamingContext

- **StreamingContext**, a class defined in the Spark Streaming library, is the **main entry point into the Spark Streaming library**.
- It allows a Spark Streaming application to connect to a Spark cluster.
- It also provides methods for creating an instance of the data stream abstraction provided by Spark Streaming.
- Every Spark Streaming application must create an instance of this class.


```
import org.apache.spark._
import org.apache.spark.streaming._

val config = new SparkConf().setMaster("spark://host:port")
                              .setAppName("big streaming app")
val batchInterval = 10
val ssc = new StreamingContext(config, Seconds(batchInterval))
```

NOTE: The batch size can be as small as 500 milliseconds. The upper bound for the batch size is determined by the latency requirements of your application and the available memory

Starting Stream Computation

The **start method** begins stream computation. Nothing really happens in a Spark Streaming application **until the start method is called on an instance of the StreamingContext class.**

A Spark Streaming application begins receiving data after it calls the start method.

```
ssc.start()
```

Waiting for Stream Computation to Finish

The **awaitTermination** method in the StreamingContext class makes an application thread **wait for stream computation to stop.**

It's syntax is:

```
ssc.awaitTermination()
```

```
scala> import org.apache.spark._
import org.apache.spark._

scala> import org.apache.spark.streaming._
import org.apache.spark.streaming._

scala> import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.streaming.StreamingContext._

scala> █
```

```
scala> import org.apache.spark._
import org.apache.spark._

scala> import org.apache.spark.streaming._
import org.apache.spark.streaming._

scala> import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.streaming.StreamingContext._

scala> val ssc = new StreamingContext(sc, Seconds(10) )
ssc: org.apache.spark.streaming.StreamingContext = org.apache.spark.streaming.StreamingContext@43e53e8f

scala> val data = ssc.socketTextStream("localhost", 9999)
data: org.apache.spark.streaming.dstream.ReceiverInputDStream[String] = org.apache.spark.streaming.dstream.SocketInputDStream@3c2efae8

scala> val lines = data.flatMap(x => x.split(" "))
lines: org.apache.spark.streaming.dstream.DStream[String] = org.apache.spark.streaming.dstream.FlatMappedDStream@39f6f419

scala> val words = lines.map(x => (x,1) )
words: org.apache.spark.streaming.dstream.DStream[(String, Int)] = org.apache.spark.streaming.dstream.MappedDStream@32a1ad3

scala> val result = words.reduceByKey(_+_)
```

Activate Windows
Go to Settings to activate Windows.

```
scala> import org.apache.spark.streaming._
import org.apache.spark.streaming._

scala> import org.apache.spark.streaming.StreamingContext._
import org.apache.spark.streaming.StreamingContext._

scala> val ssc = new StreamingContext(sc, Seconds(10) )
ssc: org.apache.spark.streaming.StreamingContext = org.apache.spark.streaming.StreamingContext@632fc85e

scala> val data = ssc.socketTextStream("localhost", 9999)
data: org.apache.spark.streaming.dstream.ReceiverInputDStream[String] = org.apache.spark.streaming.dstream.SocketInputDStream@660df45a

scala> val lines = data.flatMap(x => x.split(" "))
lines: org.apache.spark.streaming.dstream.DStream[String] = org.apache.spark.streaming.dstream.FlatMappedDStream@28d9d062

scala> val words = lines.map(x => (x,1) )
words: org.apache.spark.streaming.dstream.DStream[(String, Int)] = org.apache.spark.streaming.dstream.MappedDStream@384006ca

scala> val result = words.reduceByKey(_+_ )
result: org.apache.spark.streaming.dstream.DStream[(String, Int)] = org.apache.spark.streaming.dstream.ShuffledDStream@2d1ce4b

scala> result.print()
```

Activate Windows
Go to Settings to activate Windows.

```
gopalkrishna@ubuntu:~$ nc -lk 9999
HI HI HI HI
```

```
scala> val words = lines.map(x => (x,1) )
words: org.apache.spark.streaming.dstream.DStream[(String, Int)] = org.apache.spark.streaming.dstream.MappedDStream@384006ca

scala> val result = words.reduceByKey(_+_ )
result: org.apache.spark.streaming.dstream.DStream[(String, Int)] = org.apache.spark.streaming.dstream.ShuffledDStream@2d1ce4b
```



```
gopalkrishna@ubuntu:~$ nc -lk 9999
HI HI HI HI

scala> val words = lines.map(x => (x, 1))
words: org.apache.spark.streaming.dstream.DStream[(String, Int)] = org.apache.spark.streaming.dstream.MappedDStream[...]

scala> val result = words.reduceByKey(_+_ )
result: org.apache.spark.streaming.dstream.DStream[(String, Int)] = org.apache.spark.streaming.dstream.ShuffledDStream[...]

scala> result.print()

scala> ssc.start()
```

```
gopalkrishna@ubuntu:~$ nc -lk 9999
HI HI HI HI
hi hi hi hi hi hi
for for for today today today me me me to to to an an an

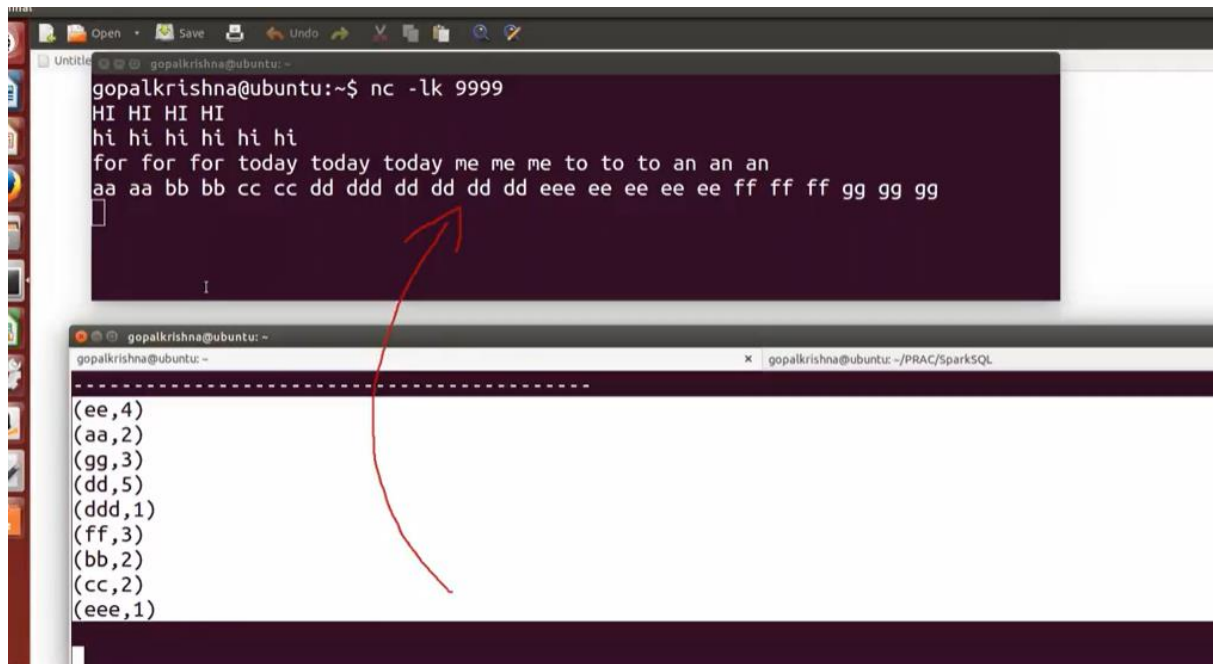
Time: 1723512170000 ms
-----
(hi,6)
-----
Time: 1723512180000 ms
-----

24/08/12 18:23:03 WARN RandomBlockReplicationPolicy: Expecting 1 replicas with only 0 peer/s.
24/08/12 18:23:03 WARN BlockManager: Block input-0-1723512183600 replicated to only 0 peer(s) instead of 1 peers
```

```
gopalkrishna@ubuntu:~$ nc -lk 9999
HI HI HI HI
hi hi hi hi hi hi
for for for today today today me me me to to to an an an
aa aa bb bb cc cc dd dd dd dd dd

(me,3)
(today,3)
(for,3)
(an,3)
(to,3)

Time: 1723512200000 ms
```



```
gopalkrishna@ubuntu:~$ nc -lk 9999
HI HI HI HI
hi hi hi hi hi hi
for for for today today me me me to to to an an an
aa aa bb bb cc cc dd ddd dd dd dd dd eee ee ee ee ff ff ff gg gg gg

(gg,3)
(dd,5)
(ddd,1)
(ff,3)
(bb,2)
(cc,2)
(eee,1)
```

2)Example:

```
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._

val ssc = new StreamingContext(sc, Seconds(10) )
val data = ssc.socketTextStream("localhost", 9999)
val lines = data.flatMap(x => x.split(" "))
val words = lines.map(x => (x,1) )
val result = words.reduceByKey(_+_ )
result.saveAsTextFiles("hdfs://localhost:8020/kafka-spark-out")
result.print()

// To Start Receives the data and compute the data
ssc.start()
```

```
NetworkWordCount.scala x HdfsWordCount.scala x StatefulNetworkWordCount.scala x
gopalkrishna@ubuntu: ~
gopalkrishna@ubuntu:~$ nc -lk 9999
HI HI HI HI HI
HI HI HI HI HI HI HI HI HI HI HI HI HI HI HI
hello hello hello for for for an an an to to to my my my job job today today today today
for for for for for for my my my

gopalkrishna@ubuntu:~$ $SPARK_HOME/bin/run-example streaming.StatefulNetworkWordCount localhost 9999
```

3) Window function in Spark SQL

window functions or **windowed aggregates** are functions that perform a calculation over a group of records called **window** that are in *some* relation to the current record

Window Functions helps us to **compare current row** with other rows in the same data frame, calculating **running totals**, **sequencing** of events and **sessionization** of transactions etc.

```
scala> import org.apache.spark.sql.SparkSession
```

```
import org.apache.spark.sql.SparkSession
```

```
scala> val sparkSession = SparkSession.builder.master("local").appName("Window Function").getOrCreate()
```

```
19/10/16 04:32:58 WARN SparkSession$Builder: Using an existing SparkSession; some configuration may not take effect.
```

```
sparkSession: org.apache.spark.sql.SparkSession =
org.apache.spark.sql.SparkSession@5b0bf26b
```

```
scala> import sparkSession.implicits._
```

```
import sparkSession.implicits._
```

```
scala> val empDF = sparkSession.createDataFrame(Seq(
  (7369, "SMITH", "CLERK", 7902, "17-Dec-80", 800, 20, 10),
  (7499, "ALLEN", "SALESMAN", 7698, "20-Feb-81", 1600, 300, 30),
  (7521, "WARD", "SALESMAN", 7698, "22-Feb-81", 1250, 500, 30),
  (7566, "JONES", "MANAGER", 7839, "2-Apr-81", 2975, 0, 20),
  (7654, "MARTIN", "SALESMAN", 7698, "28-Sep-81", 1250, 1400, 30),
  (7698, "BLAKE", "MANAGER", 7839, "1-May-81", 2850, 0, 30),
  (7782, "CLARK", "MANAGER", 7839, "9-Jun-81", 2450, 0, 10),
  (7788, "SCOTT", "ANALYST", 7566, "19-Apr-87", 3000, 0, 20),
  (7839, "KING", "PRESIDENT", 0, "17-Nov-81", 5000, 0, 10),
```

```

(7844, "TURNER", "SALESMAN", 7698, "8-Sep-81", 1500, 0, 30),
(7889, "SYED", "ANALYST", 5698, "14-Oct-81", 1150, 1300, 30),
(7876, "ADAMS", "CLERK", 7788, "23-May-87", 1100, 0, 20)
)).toDF("empno", "ename", "job", "mgr", "hiredate", "sal", "comm", "deptno")
empDF: org.apache.spark.sql.DataFrame = [empno: int, ename: string ... 6 more fields]

```

```
scala> empDF.show
```

```
scala> val partitionWindow = Window.partitionBy($"deptno").orderBy($"sal".desc)
```

```
<console>:29: error: not found: value Window
```

```
    val partitionWindow = Window.partitionBy($"deptno").orderBy($"sal".desc)
```

```
scala> import org.apache.spark.sql.expressions.Window
```

```
import org.apache.spark.sql.expressions.Window
```

```
scala> import org.apache.spark.sql.functions._
```

```
import org.apache.spark.sql.functions._
```

```
scala> val partitionWindow = Window.partitionBy($"deptno").orderBy($"sal".desc)
```

```
partitionWindow: org.apache.spark.sql.expressions.WindowSpec =
org.apache.spark.sql.expressions.WindowSpec@2ab6e3de
```

```
RANK
```

```
scala> val rankTest = rank().over(partitionWindow)
```

```
rankTest: org.apache.spark.sql.Column = RANK() OVER (PARTITION BY deptno ORDER BY
sal DESC NULLS LAST UnspecifiedFrame)
```

```
scala> empDF.select($"*", rankTest as "rank").show
```

```

+-----+-----+-----+----+-----+----+----+-----+----+
|empno| ename|   job| mgr| hiredate| sal|comm|deptno|rank|
+-----+-----+-----+----+-----+----+----+-----+----+
| 7788| SCOTT| ANALYST|7566|19-Apr-87|3000| 0| 20| 1|
| 7566| JONES| MANAGER|7839| 2-Apr-81|2975| 0| 20| 2|
| 7876| ADAMS| CLERK|7788|23-May-87|1100| 0| 20| 3|
| 7839| KING| PRESIDENT| 0|17-Nov-81|5000| 0| 10| 1|
| 7782| CLARK| MANAGER|7839| 9-Jun-81|2450| 0| 10| 2|
| 7369| SMITH| CLERK|7902|17-Dec-80| 800| 20| 10| 3|
| 7698| BLAKE| MANAGER|7839| 1-May-81|2850| 0| 30| 1|
| 7499| ALLEN| SALESMAN|7698|20-Feb-81|1600| 300| 30| 2|
| 7844| TURNER| SALESMAN|7698| 8-Sep-81|1500| 0| 30| 3|
| 7521| WARD| SALESMAN|7698|22-Feb-81|1250| 500| 30| 4|
| 7654| MARTIN| SALESMAN|7698|28-Sep-81|1250|1400| 30| 4|
+-----+-----+-----+----+-----+----+----+-----+----+

```

ROW NUMBER

```
scala> val rankTest = row_number().over(partitionWindow)
```

```
rankTest: org.apache.spark.sql.Column = row_number() OVER (PARTITION BY deptno  
ORDER BY sal DESC NULLS LAST UnspecifiedFrame)
```

```
scala> empDF.select($"*", rankTest as "Row Number").show
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
|empno| ename|   job| mgr| hiredate| sal|comm|deptno|Row Number|
+-----+-----+-----+-----+-----+-----+-----+-----+
| 7788| SCOTT| ANALYST|7566|19-Apr-87|3000| 0| 20|      1|
| 7566| JONES|  MANAGER|7839| 2-Apr-81|2975| 0| 20|      2|
| 7876| ADAMS|  CLERK|7788|23-May-87|1100| 0| 20|      3|
| 7839| KING|PRESIDENT| 0|17-Nov-81|5000| 0| 10|      1|
| 7782| CLARK|  MANAGER|7839| 9-Jun-81|2450| 0| 10|      2|
| 7369| SMITH|  CLERK|7902|17-Dec-80| 800| 20| 10|      3|
| 7698| BLAKE|  MANAGER|7839| 1-May-81|2850| 0| 30|      1|
| 7499| ALLEN| SALESMAN|7698|20-Feb-81|1600| 300| 30|      2|
| 7844| TURNER| SALESMAN|7698| 8-Sep-81|1500| 0| 30|      3|
| 7521| WARD| SALESMAN|7698|22-Feb-81|1250| 500| 30|      4|
| 7654| MARTIN| SALESMAN|7698|28-Sep-81|1250|1400| 30|      5|
+-----+-----+-----+-----+-----+-----+-----+-----+
```

DENSE RANK

```
scala> val rankTest = dense_rank().over(partitionWindow)
```

```
rankTest: org.apache.spark.sql.Column = DENSE_RANK() OVER (PARTITION BY deptno  
ORDER BY sal DESC NULLS LAST UnspecifiedFrame)
```

```
scala> empDF.select($"*", rankTest as "dense rank").show
```

```
+-----+-----+-----+-----+-----+-----+-----+-----+
|empno| ename|   job| mgr| hiredate| sal|comm|deptno|dense rank|
+-----+-----+-----+-----+-----+-----+-----+-----+
| 7788| SCOTT| ANALYST|7566|19-Apr-87|3000| 0| 20|      1|
| 7566| JONES|  MANAGER|7839| 2-Apr-81|2975| 0| 20|      2|
| 7876| ADAMS|  CLERK|7788|23-May-87|1100| 0| 20|      3|
| 7839| KING|PRESIDENT| 0|17-Nov-81|5000| 0| 10|      1|
| 7782| CLARK|  MANAGER|7839| 9-Jun-81|2450| 0| 10|      2|
```


[illegible]

Program 7.Advanced Spark and Scala Applications

- i. Explore the use of broadcast variables and accumulators in Spark.
- ii. Create and analyse a graph using Spark's GraphX library.

Sol;

GraphX :

- ✓ **GraphX** is a **distributed graph analytics framework**.
- ✓ It is a **Spark library** that extends Spark for **large-scale graph processing**.
- ✓ It provides a higher-level abstraction for graphs analytics than that provided by the Spark core API.

GraphX provides both fundamental graph operators and advanced operators implementing graph algorithms such as **PageRank, strongly connected components, and triangle count**.

It also provides an implementation of Google's Pregel API. These operators simplify graph analytics tasks.

GraphX allows the same data to be operated on as a distributed graph or distributed collections.

It provides collection operators similar to those provided by the RDD API and graph operators similar to those provided by specialized graph analytics libraries.

Thus, it unifies collections and graphs as first-class composable objects.

A key benefit of using GraphX is that it provides an integrated platform for complete graph analytics workflow or pipeline.

A graph analytics pipeline generally consists of the following steps:

- a) Read raw data.**
- b) Preprocess data (e.g., cleanse data).**
- c) Extract vertices and edges to create a property graph.**
- d) Slice a subgraph.**
- e) Run graph algorithms.**
- f) Analyze the results.**
- g) Repeat steps e and f with another slice of the graph**

GraphX API

- ✓ The GraphX API provides data types for representing graph-oriented data and operators for graph analytics.
- ✓ It allows you to work with either a graphs or a collections view of the same data.

Data Abstractions

The key data types provided by GraphX for working with property graphs include

- ✓ **VertexRDD**
- ✓ **Edge**
- ✓ **EdgeRDD**
- ✓ **EdgeTriplet**
- ✓ **Graph**

VertexRDD

- ✓ VertexRDD represents a distributed collection of vertices in a property graph.
- ✓ It provides a collection view of the vertices in a property graph.
- ✓ VertexRDD stores only one entry for each vertex. In addition, it indexes the entries for fast joins.
- ✓ VertexRDD is a generic or parameterized class; it requires a type parameter.
- ✓ It is defined as VertexRDD[VD], where the type parameter VD specifies the data type of the attribute or property associated with each vertex in a graph.
- ✓ For example, VD can be Int, Long, Double, String or a user defined type. Thus, vertices in a graph can be represented by VertexRDD[String], VertexRDD[Int], or VertexRDD of some user defined type.

Edge

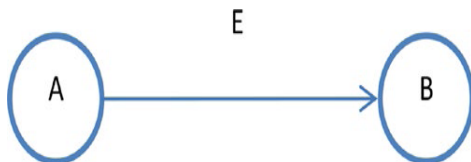
- ✓ The Edge class abstracts a directed edge in a property graph.
- ✓ An instance of the Edge class contains source vertex id, destination vertex id and edge attributes.
- ✓ Edge is also a generic class requiring a type parameter. The type parameter specifies the data type of the edge attributes.
- ✓ For example, an edge can have attribute of type Int, Long, Double, String or a user defined type

EdgeRDD

- ✓ EdgeRDD represents a distributed collection of the edges in a property graph.
- ✓ The EdgeRDD class is parameterized over the data type of the edge attributes.

EdgeTriplet

- ✓ An instance of the EdgeTriplet class represents a combination of an edge and the two vertices that it connects.
- ✓ It stores the attributes of an edge and the two vertices that it connects.
- ✓ It also contains the unique identifiers for the source and destination vertices of an edge.



Graph

- ✓ Graph is **GraphX's abstraction** for representing **property graphs**; an instance of the Graph class represents a property graph.
- ✓ Similar to RDD, it is immutable, distributed, and fault-tolerant.
- ✓ GraphX partitions and distributes a graph across a cluster using vertex partitioning heuristics.
- ✓ It recreates a partition on a different machine if a machine fails

Example: Once you are inside the Spark-shell, import the GraphX library.

```
import org.apache.spark.graphx._
```

Next, create a case class to represent a user. Each user will have two attributes or properties: **name** and **age**.

```
case class User(name: String, age: Int)
```

Now let's create an RDD of id and user pairs.

```
val users = List((1L, User("Alex", 26)), (2L, User("Bill", 42)),  
  (3L, User("Carol", 18)),  
  (4L, User("Dave", 16)), (5L, User("Eve", 45)), (6L,  
  User("Farell", 30)),
```

```
(7L, User("Garry", 32)), (8L, User("Harry", 36)), (9L,
User("Ivan", 28)),
(10L, User("Jill", 48))
)
```

```
val usersRDD = sc.parallelize(users)
```

The preceding code creates an RDD of key value pairs, where the key is a unique identifier and value is a user attributes. The key represents a vertex id.

Next, let's create an RDD of connections between users. A connection can be represented by an instance of the Edge case class.

An edge can have any number of attributes; however, to keep things simple, assign a single attribute of type Int to each edge.

```
val follows = List(Edge(1L, 2L, 1), Edge(2L, 3L, 1), Edge(3L,
1L, 1), Edge(3L, 4L, 1),
Edge(3L, 5L, 1), Edge(4L, 5L, 1), Edge(6L, 5L, 1), Edge(7L, 6L,
1),
Edge(6L, 8L, 1), Edge(7L, 8L, 1), Edge(7L, 9L, 1), Edge(9L, 8L,
1),
Edge(8L, 10L, 1), Edge(10L, 9L, 1), Edge(1L, 11L, 1)
)
```

```
val followsRDD = sc.parallelize(follows)
```

In the preceding code snippet, the first argument to Edge is the source vertex id.

The second argument is the destination vertex id and the last argument is the edge attribute.

Note that there is an edge connecting vertex with id 1 to vertex with id 11. However, the vertex with id 11 does not have any property.

GraphX allows you to handle such cases by creating a default set of properties.

It will assign the default properties to the vertices that have not been explicitly assigned any properties.

```
val defaultUser = User("NA", 0)
```


Now you have all the components required to construct a property graph.

```
val socialGraph = Graph(usersRDD, followsRDD, defaultUser)
```

```
val numEdges = socialGraph.numEdges
```

```
val numVertices = socialGraph.numVertices
```

```
val inDegrees = socialGraph.inDegrees
```

```
inDegrees.collect
```

Example:2

```
scala> import org.apache.spark.graphx._  
import org.apache.spark.graphx._
```

```
scala> import org.apache.spark.rdd.RDD  
import org.apache.spark.rdd.RDD
```

```
scala> val vertexArray = Array(  
(1L,("Ram",31)),(2L,("Ramya",18)),(3L,("Raja",34)),(4L,("Mohan",28  
)),(5L,("Sonu",30)),(6L,("Yamini",36)) )  
vertexArray: Array[(Long, (String, Int))] = Array((1,(Ram,31)),  
(2,(Ramya,18)), (3,(Raja,34)), (4,(Mohan,28)), (5,(Sonu,30)),  
(6,(Yamini,36)))
```

```
scala> val edgeArray = Array(Edge(2L, 1L, 7),Edge(2L, 4L, 2),  
Edge(3L, 2L, 4),Edge(3L, 6L, 3),Edge(4L, 1L, 1),Edge(5L, 2L,  
2),Edge(5L, 3L, 8), Edge(5L, 6L, 3) )  
edgeArray: Array[org.apache.spark.graphx.Edge[Int]] =  
Array(Edge(2,1,7), Edge(2,4,2), Edge(3,2,4), Edge(3,6,3), Edge(4,1,1),  
Edge(5,2,2), Edge(5,3,8), Edge(5,6,3))
```

```
scala> val vRDD= sc.parallelize(vertexArray)  
vRDD: org.apache.spark.rdd.RDD[(Long, (String, Int))] =  
ParallelCollectionRDD[0] at parallelize at <console>:34
```

```
scala> val eRDD= sc.parallelize(edgeArray)  
eRDD: org.apache.spark.rdd.RDD[org.apache.spark.graphx.Edge[Int]]  
= ParallelCollectionRDD[1] at parallelize at <console>:34
```

```
scala> val graph: Graph[(String, Int), Int] = Graph(vRDD, eRDD)  
graph: org.apache.spark.graphx.Graph[(String, Int),Int] =  
org.apache.spark.graphx.impl.GraphImpl@635968ce
```

```
scala> graph.edges.foreach(println)  
[Stage 0:> (0 + 4) / 4]Edge(5,3,8)  
Edge(3,2,4)  
Edge(3,6,3)
```

Edge(4,1,1)

Edge(5,2,2)

Edge(2,1,7)

Edge(2,4,2)

Edge(5,6,3)

**18/08/10 02:46:24 WARN Executor: 1 block locks were not released
by TID = 1:**

[rdd_12_1]

**18/08/10 02:46:24 WARN Executor: 1 block locks were not released
by TID = 3:**

[rdd_12_3]

**18/08/10 02:46:24 WARN Executor: 1 block locks were not released
by TID = 0:**

[rdd_12_0]

**18/08/10 02:46:24 WARN Executor: 1 block locks were not released
by TID = 2:**

[rdd_12_2]

scala> graph.vertices.foreach(println)

[Stage 2:>

(0 + 4) / 4]

(4,(Mohan,28))

(6,(Yamini,36))

(2,(Ramya,18))

(1,(Ram,31))

8. Capstone Project

i. Develop a data science project showcasing data processing, analysis, and machine learning using Spark