Ramniranjan Jhunjhunwala College of Arts, Science and Commerce

Ghatkopar (W), Mumbai - 400 086

Department of Computer Science and Information Technology

M.Sc. DSAI SEM III 2022-23

RJSPDSAI3L6 BIG DATA ANALYTICS

Name: Ajaz Nafis Khan

Seat No.: 711

INDEX

Practical 1 - Using Spark Session and Spark Context

a)create Spark Session and Spark Context for prime or not prime number.

Spark session: SparkSession introduced in version 2.0 and is an entry point to underlying Spark functionality in order to programmatically create Spark RDD, DataFrame and DataSet. It's object *spark* is default available in spark-shell and it can be created programmatically using SparkSession builder pattern.

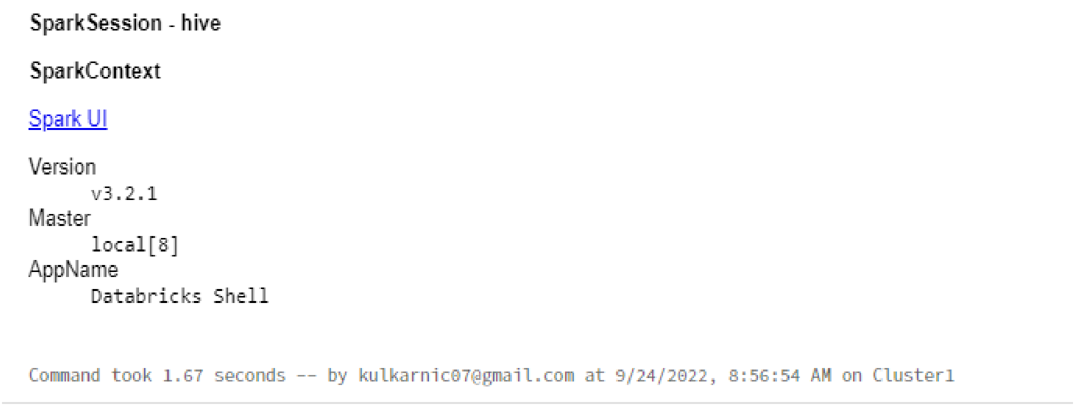
Spark Context: Spark SparkContext is an entry point to Spark and defined in org.apache.spark package since 1.x and used to programmatically create Spark RDD, accumulators and broadcast variables on the cluster. Since Spark 2.0 most of the functionalities (methods) available in SparkContext are also available in SparkSession. Its object *sc* is default available in spark-shell and it can be programmatically created using SparkContext class.

1.import libraries:

```
!pip install --upgrade pip
!pip install -q findspark
!wget -q https://archive.apache.org/dist/spark/spark-3.1.1/spark-3.1.1-bin-hadoop3.2.tgz

import os
os.environ['JAVA_HOME']='/usr/lib/jvm/java-8-openjdk-amd64'
os.environ['SPARK_HOME']='/content/spark-3.1.1-bin-hadoop3.2.tgz'

import findspark
findspark.init()

from pyspark import SparkConf
from pyspark.context import SparkContext
from pyspark.sql import SparkSession
```

2.Creating Session:

```
spark=SparkSession.builder.master("local[1]").getOrCreate()
spark.conf.set("spark.sql.repl.eagerEval.enabled",True)
spark
```



```
SparkSession - hive

SparkContext

Spark UI

Version
    v3.2.1
Master
    local[8]
AppName
    Databricks Shell

Command took 1.67 seconds -- by kulkarnic07@gmail.com at 9/24/2022, 8:56:54 AM on Cluster1
```

3.Creating spark context-Its like connecting to spark cluster:

```
from pyspark import SparkConf
from pyspark.context import SparkContext

sc = SparkContext.getOrCreate(SparkConf().setMaster("local[*]"))
```

4.Check whether number is prime or not

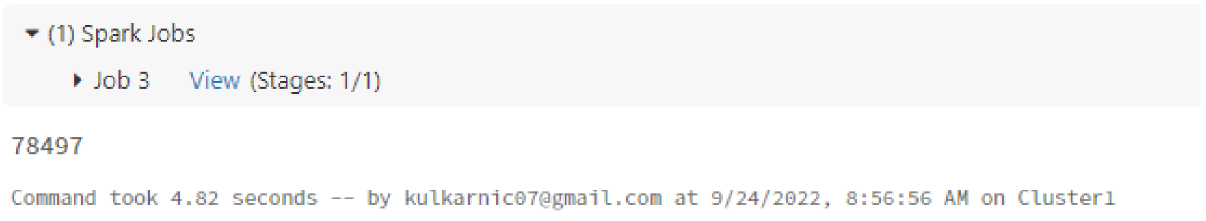
```
def isprime(n):
    n=abs(int(n))
    if n<2:
        return False
    elif(not n & 1):
        return False
    elif(n==2):
        return False
    for x in range(3,int(n**0.5)+1,2):
        if n%x==0:
            return False
    return True
```

5. Create an RDD of numbers from 0 to 1,000,000

```
nums = sc.parallelize(range(1000000))
```

7. Compute the number of primes in the RDD

```
print(nums.filter(isprime).count())
```



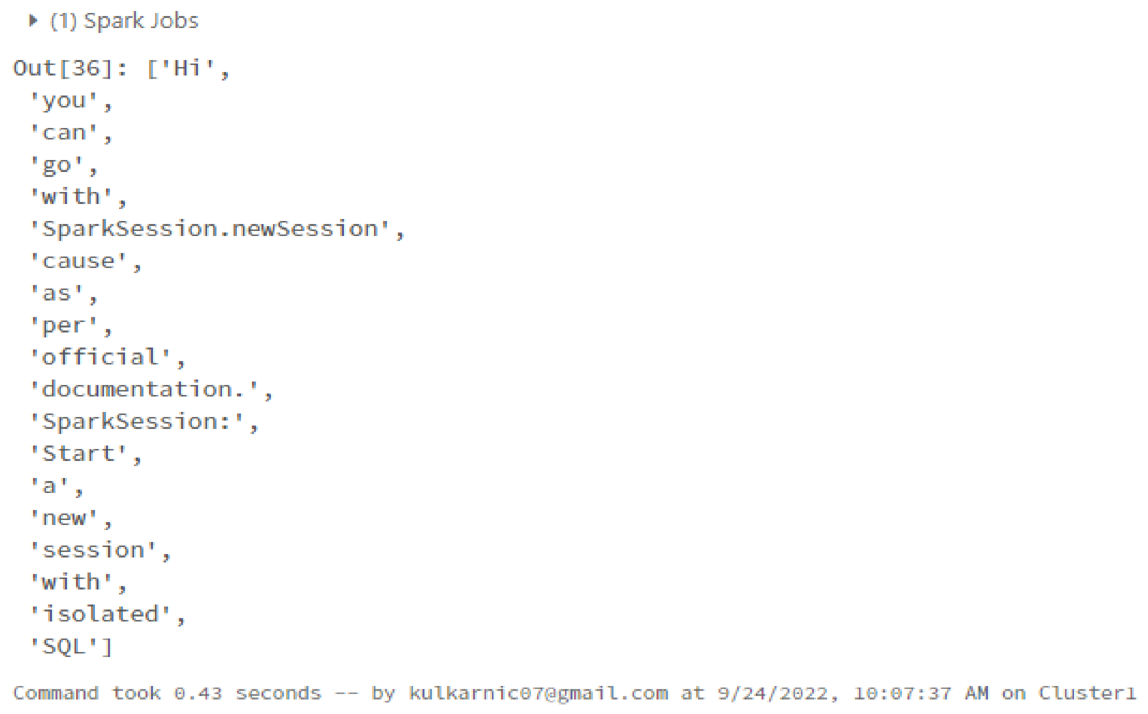
```
78497
Command took 4.82 seconds -- by kulkarnic07@gmail.com at 9/24/2022, 8:56:56 AM on Cluster1
```

b)Implement the word count program using pyspark.

(count the occurrences of unique words in a text line.)

1. Files loading and defining the functions in databricks.

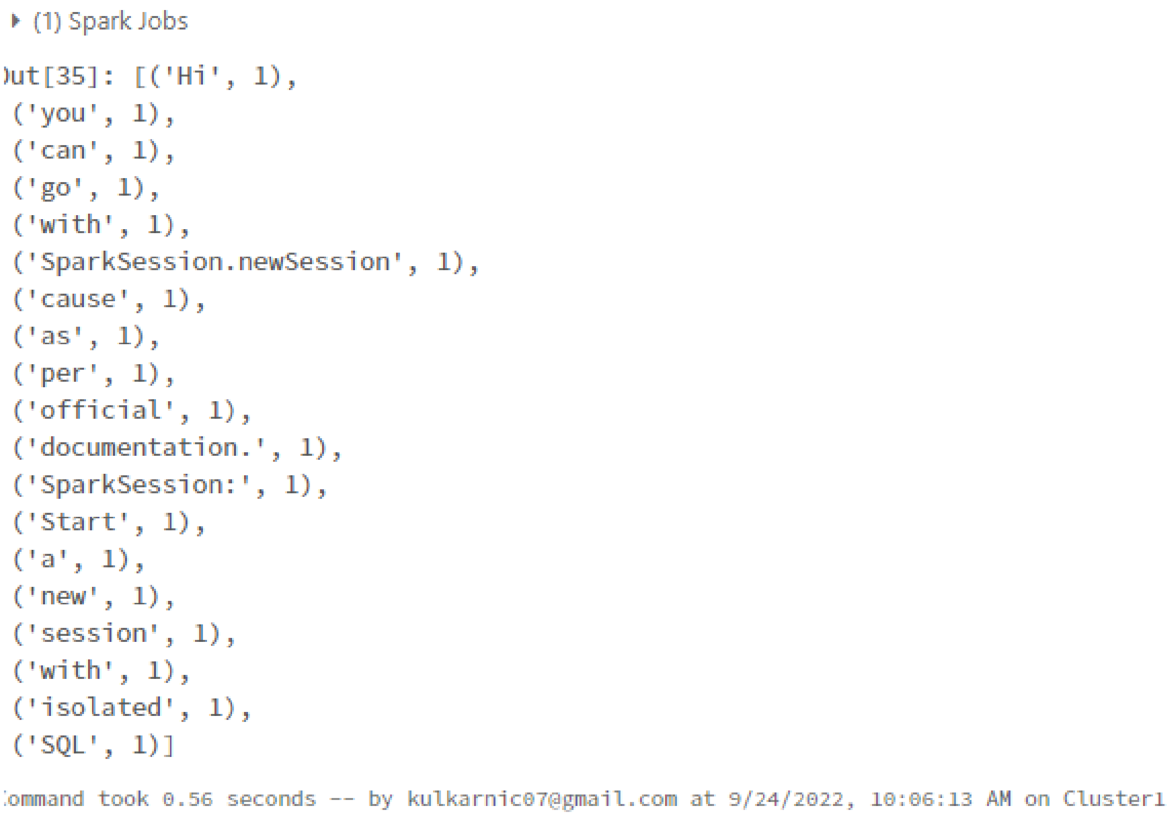
```
text=sc.textFile("/FileStore/tables/demo-1.txt")
text
from operator import add
def tokenize(text):
    return text.split()

words = text.flatMap(tokenize)
words.collect()
```



```
Out[36]: ['Hi',
 'you',
 'can',
 'go',
 'with',
 'SparkSession.newSession',
 'cause',
 'as',
 'per',
 'official',
 'documentation.',
 'SparkSession:',
 'Start',
 'a',
 'new',
 'session',
 'with',
 'isolated',
 'SQL']
Command took 0.43 seconds -- by kulkarnic07@gmail.com at 9/24/2022, 10:07:37 AM on Cluster1
```

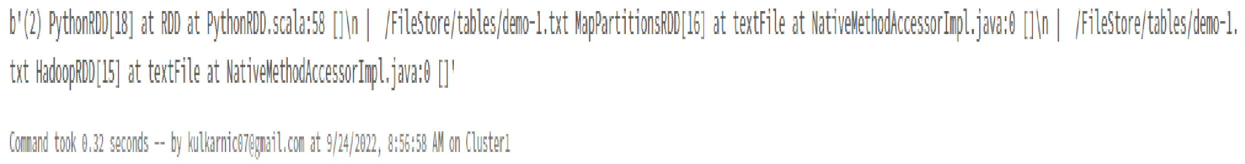
2. Mapping the data
wc = words.map(lambda x: (x,1))

wc.collect()



```
▸ (1) Spark Jobs

)ut[35]: [('Hi', 1),
 ('you', 1),
 ('can', 1),
 ('go', 1),
 ('with', 1),
 ('SparkSession.newSession', 1),
 ('cause', 1),
 ('as', 1),
 ('per', 1),
 ('official', 1),
 ('documentation.', 1),
 ('SparkSession:', 1),
 ('Start', 1),
 ('a', 1),
 ('new', 1),
 ('session', 1),
 ('with', 1),
 ('isolated', 1),
 ('SQL', 1)]

:ommand took 0.56 seconds -- by kulkarnic07@gmail.com at 9/24/2022, 10:06:13 AM on Cluster1
```

print(wc.toDebugString())



```
b'(2) PythonRDD[18] at RDD at PythonRDD.scala:58 []\n |  /FileStore/tables/demo-1.txt MapPartitionsRDD[16] at textFile at NativeMethodAccessorImpl.java:0 []\n |  /FileStore/tables/demo-1.
txt HadoopRDD[15] at textFile at NativeMethodAccessorImpl.java:0 []'

Command took 0.32 seconds -- by kulkarnic07@gmail.com at 9/24/2022, 8:56:58 AM on Cluster1
```

3.Reduce the value and it display counted words
counts = wc.reduceByKey(add)
counts.collect()

▸ (1) Spark Jobs

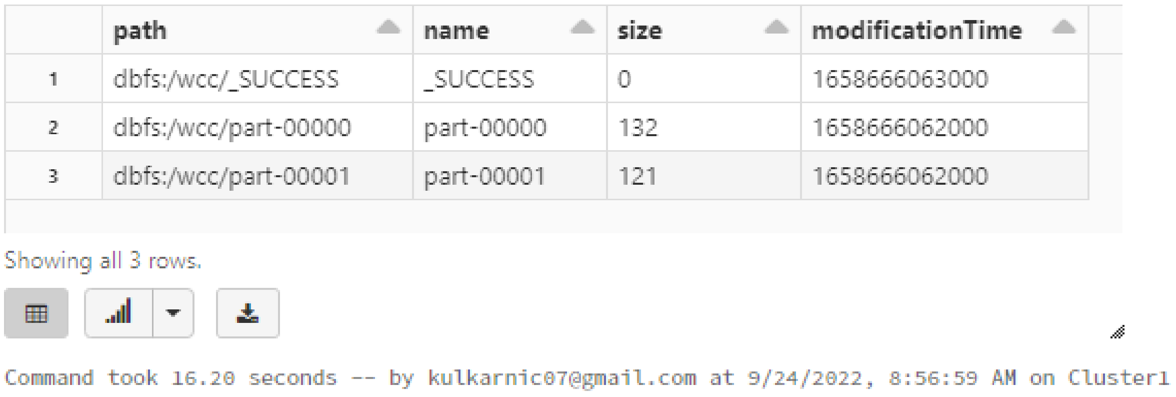
```
Out[34]: [('Hi', 1),
 ('go', 1),
 ('cause', 1),
 ('as', 1),
 ('official', 1),
 ('documentation.', 1),
 ('Start', 1),
 ('new', 1),
 ('isolated', 1),
 ('SQL', 1),
 ('you', 1),
 ('can', 1),
 ('with', 2),
 ('SparkSession.newSession', 1),
 ('per', 1),
 ('SparkSession:', 1),
 ('a', 1),
 ('session', 1)]
```

```
Command took 1.83 seconds -- by kulkarnic07@gmail.com at 9/24/2022, 10:04:46 AM on Cluster1
```

4.Save the file with name wcc

counts.saveAsTextFile("wcc")

%fs ls wcc/



	path	name	size	modificationTime
1	dbfs:/wcc/_SUCCESS	_SUCCESS	0	1658666063000
2	dbfs:/wcc/part-00000	part-00000	132	1658666062000
3	dbfs:/wcc/part-00001	part-00001	121	1658666062000

Showing all 3 rows.

```
Command took 16.20 seconds -- by kulkarnic07@gmail.com at 9/24/2022, 8:56:59 AM on Cluster1
```

%fs head wcc/part-00000

Practical 2: Using Spark Resilient Distributed Dataset

a) Transformations
b) Actions
c) Soccer Tweet Analysis

Solution

a) Transformations

RDD Transformations are lazy operations meaning none of the transformations get executed until you call an action on PySpark RDD. Since RDD's are immutable, any transformations on it result in a new RDD leaving the current one unchanged.

RDD Transformation Types: There are two types of transformations.

- **Narrow Transformation**

Narrow transformations are the de of map() and filter() functions and these compute data that live on a single partition meaning there will not be any data movement between partitions to execute narrow transformations.

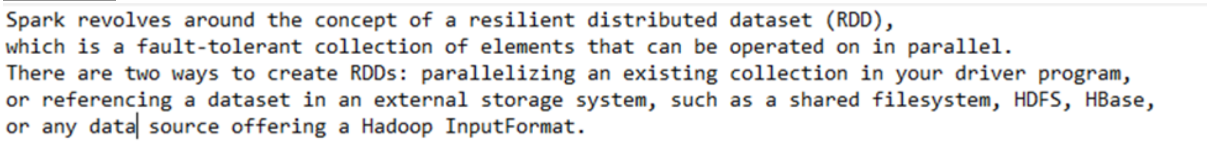
- **Wider Transformation**

Wider transformations are the result of *groupByKey()* and *reduceByKey()* functions and these compute data that live on many partitions meaning there will be data movements between partitions to execute wider transformations.

- **Steps:**

1) Create RDD by reading text file

Text file



```
Spark revolves around the concept of a resilient distributed dataset (RDD),
which is a fault-tolerant collection of elements that can be operated on in parallel.
There are two ways to create RDDs: parallelizing an existing collection in your driver program,
or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase,
or any data source offering a Hadoop InputFormat.
```

Code:

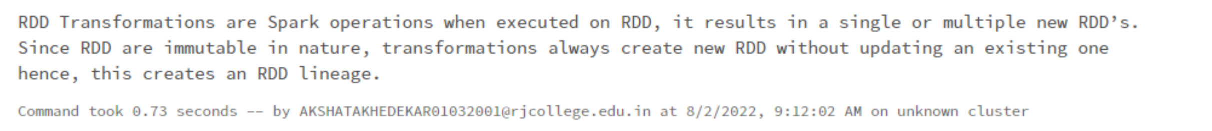
```
 from pyspark.sql import SparkSession
spark = SparkSession.builder.appName('Rdd_trans').getOrCreate()
rdd=spark.sparkContext.textFile('dbfs:/FileStore/shared_uploads/AKSHATAKHEDEKAR01032001@rjcollege.edu. in/text-3.txt')
```

2) View the elements from the text file

Code:

```
for element in rdd.collect():
        print(element)
```

Output:



```
RDD Transformations are Spark operations when executed on RDD, it results in a single or multiple new RDD's.
Since RDD are immutable in nature, transformations always create new RDD without updating an existing one
hence, this creates an RDD lineage.

Command took 0.73 seconds -- by AKSHATAKHEDEKAR01032001@rjcollege.edu.in at 8/2/2022, 9:12:02 AM on unknown cluster
```

By creating the RDD now we can apply the transformations as follows:

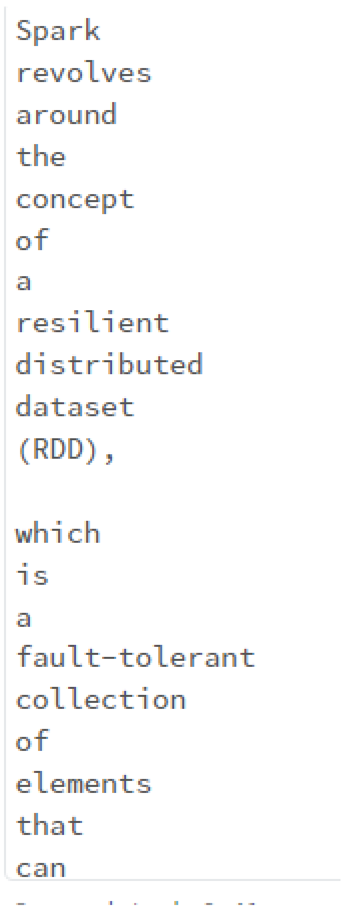
1) flatMap()

flatMap() transformation flattens the RDD after applying the function and returns a new RDD. First, it splits each record by space in an RDD and finally flattens it. Resulting
Z
RDD consists of a single word on each record.

Code:

```
rdd2 = rdd.flatMap(lambda x: x.split(" "))
for element in rdd2.collect():
   print(element)
```

Output:



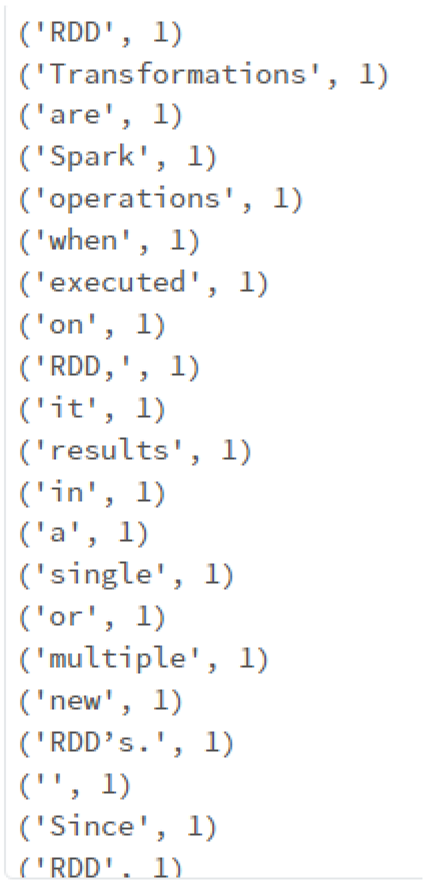
```
Spark
revolves
around
the
concept
of
a
resilient
distributed
dataset
(RDD),

which
is
a
fault-tolerant
collection
of
elements
that
can
```

2) map()

Code:

```
rdd3 = rdd2.map(lambda x: (x,1))
for element in rdd3.collect():
   print(element)
```

Output:



```
('RDD', 1)
('Transformations', 1)
('are', 1)
('Spark', 1)
('operations', 1)
('when', 1)
('executed', 1)
('on', 1)
('RDD,', 1)
('it', 1)
('results', 1)
('in', 1)
('a', 1)
('single', 1)
('or', 1)
('multiple', 1)
('new', 1)
('RDD’s.', 1)
('', 1)
('Since', 1)
('RDD', 1)
```

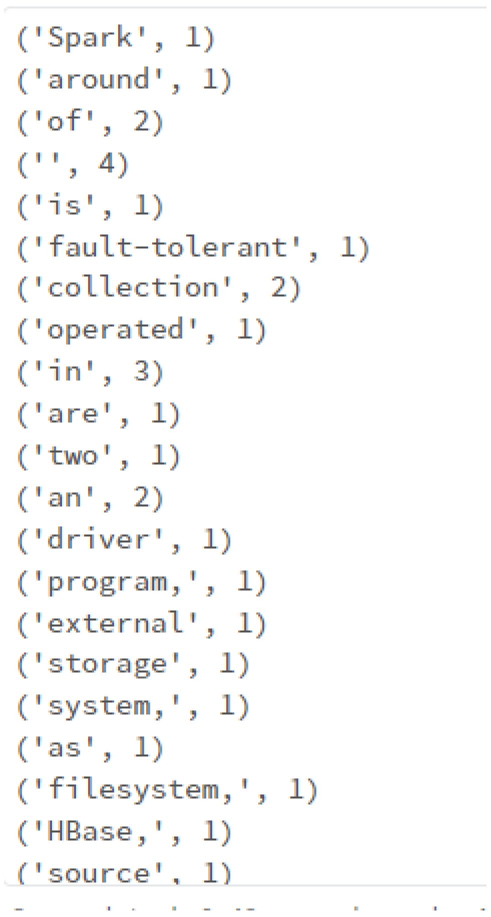
3) reduceByKey()

reduceByKey() merges the values for each key with the function specified. In our example, it reduces the word string by applying the sum function on value. The result of our RDD contains unique words and their count.

Code:

```
rdd4=rdd3.reduceByKey(lambda a,b: a+b)
for element in rdd4.collect():
   print(element)
```

Output:


```
('Spark', 1)
('around', 1)
('of', 2)
('', 4)
('is', 1)
('fault-tolerant', 1)
('collection', 2)
('operated', 1)
('in', 3)
('are', 1)
('two', 1)
('an', 2)
('driver', 1)
('program,', 1)
('external', 1)
('storage', 1)
('system,', 1)
('as', 1)
('filesystem,', 1)
('HBase,', 1)
('source', 1)
```

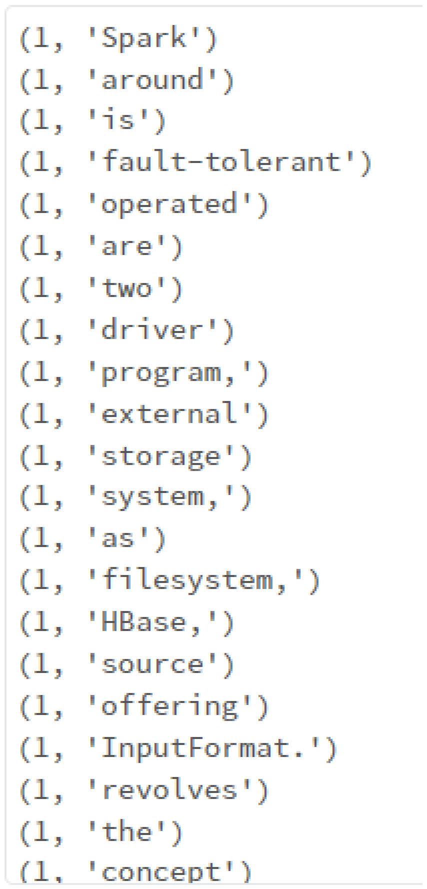
4) sortByKey()

sortByKey() transformation is used to sort RDD elements on key. In our example, first, we convert RDD[(String,Int]) to RDD[(Int,String]) using map transformation and later apply sortByKey which ideally does sort on an integer value.

Code:

```
rdd5 = rdd4.map(lambda x: (x[1],x[0])).sortByKey()
for element in rdd5.collect():
   print(element)
```

Output:

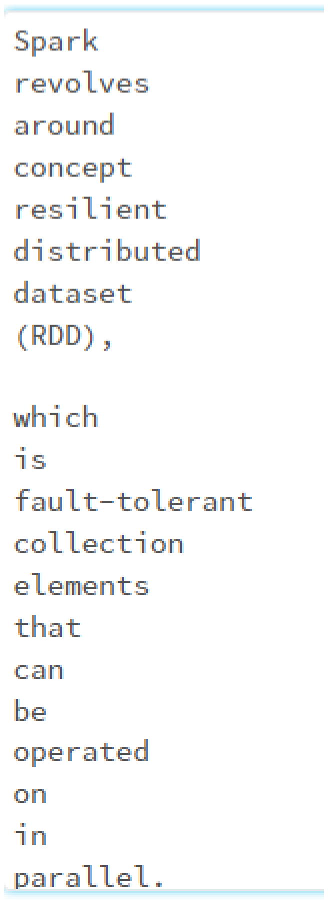

```
(1, 'Spark')
(1, 'around')
(1, 'is')
(1, 'fault-tolerant')
(1, 'operated')
(1, 'are')
(1, 'two')
(1, 'driver')
(1, 'program,')
(1, 'external')
(1, 'storage')
(1, 'system,')
(1, 'as')
(1, 'filesystem,')
(1, 'HBase,')
(1, 'source')
(1, 'offering')
(1, 'InputFormat.')
(1, 'revolves')
(1, 'the')
(1, 'concept')
```

5) filter()

Code:

```
stwords = ['a','an','the']
rdd3 = rdd2.filter(lambda x: x not in stwords)

for element in rdd3.collect():
   print(element)
```

Output:



```
Spark
revolves
around
concept
resilient
distributed
dataset
(RDD),

which
is
fault-tolerant
collection
elements
that
can
be
operated
on
in
parallel.
```

b) Actions

RDD actions are PySpark operations that return the values to the driver program. Any function on RDD that returns other than RDD is considered as an action in PySpark programming.

1) Create an RDD

Parallelize

PySpark parallelize() is a function in SparkContext and is used to create an RDD from a list collection

Code:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
data=[("Z", 1),("A", 20),("B", 30),("C", 40),("B", 30),("B", 60)]
inputRDD = spark.sparkContext.parallelize(data)
listRdd = spark.sparkContext.parallelize([1,2,3,4,5,3,2])
```

Output:ParallelCollectionRDD[1] at readRDDFromInputStream at PythonRDD.scala:413

2) Aggregate

Aggregate functions that could be very handy when it comes to segmenting out the data according to the requirements so that it would become a bit easier task to analyze the chunks of data separately based on the groups.

Code:
```
seqOp2 = (lambda x, y: (x[0] + y, x[1] + 1))
combOp2 = (lambda x, y: (x[0] + y[0], x[1] + y[1]))
agg2=listRdd.aggregate((0, 0), seqOp2, combOp2)
print(agg2)
```

Output: (20, 7)

Code:
```
seqOp2=(lambda x,y:(x[1] + 1))
seqOp2([1,2,3],[4,5,6])
```
Output:Out[3]: 3

3) Fold

Aggregate the elements of each partition, and then the results for all the partitions, using a given associative function and a neutral “zero value.”The function op(t1, t2) is allowed to modify t1 and return it as its result value to avoid object allocation; however, it should not modify t2.This behaves somewhat differently from fold operations implemented for non-distributed collections in functional languages like Scala. This fold operation may be applied to partitions individually, and then fold those results into the final result, rather than apply the fold to each element sequentially in some defined ordering. For functions that are not commutative, the result may differ from that of a fold applied to a non-distributed collection.

Code:
```
from operator import add
foldRes=listRdd.fold(0, add)
print(foldRes)
```

Output: 20

4) Reduce

Spark RDD reduce() aggregate action function is used to calculate min, max, and total of elements in a dataset.

Code:
```
redRes=listRdd.reduce(add)
print(redRes)
```

Output: 20

5) Collect

PySpark RDD/DataFrame collect() is an action operation that is used to retrieve all the elements of the dataset (from all nodes) to the driver node. We should use the collect() on smaller dataset usually after filter(), group() e.t.c. Retrieving larger datasets results in OutOfMemory error.

Code:
```
data = listRdd.collect()
print(data)
data = listRdd.count()
print(data)
```

Output :
```
[1, 2, 3, 4, 5, 3, 2]
  7
```

6) Count

a)countApprox

Approximate version of count() that returns a potentially incomplete result within a timeout, even if not all tasks have finished.

Code:
```
data =listRdd.countApprox(1200)
print(data)
```
Output: 7

b)countApproxDistinct

Return approximate number of distinct elements in the RDD.

Code:
```
data= listRdd.countApproxDistinct()
print(data)
```
Output: 5

c)countByValue

Return the count of each unique value in this RDD as a dictionary of (value, count) pairs.

Code:
```
print("countByValue :  "+str(listRdd.countByValue()))
```
Output : countByValue : defaultdict(<class 'int'>, {1: 1, 2: 2, 3: 2, 4: 1, 5: 1})

7)First

Return the first element in this RDD.

Code:Code:
```
df1 = listRdd.first()
print(df1)
df1 = inputRDD.first()
```

print(df1)
Output: 1
('Z', 1)

8)Top
Get the top N elements from an RDD.This method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver's memory.It returns the list sorted in descending order.
Code:
print("top : "+str(listRdd.top(2)))
print("top : "+str(inputRDD.top(2)))
Output: top : [5, 4]
top : [('Z', 1), ('C', 40)]

9)Min
Find the minimum item in this RDD.
Code:
print("min : "+str(listRdd.min()))
print("min : "+str(inputRDD.min()))
Output : min : 1
min : ('A', 20)

10)Max
Find the maximum item in this RDD.
Code:
print("max : "+str(listRdd.max()))
print("max : "+str(inputRDD.max()))
Output: max : 5
max : ('Z', 1)

11) a)take
Take the first num elements of the RDD.
It works by first scanning one partition, and use the results from that part
This method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver's memory.ition to estimate the number of additional partitions needed to satisfy the limit.
Code:print("take : "+str(listRdd.take(2)))
Output: take : [1, 2]

b)takeOrdered
Get the N elements from an RDD ordered in ascending order or as specified by the optional key function.

This method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver's memory.
Code:print("takeOrdered : "+ str(listRdd.takeOrdered(2)))
Output: takeOrdered : [1, 2]

c)takeSampleReturn a fixed-size sampled subset of this RDD.
This method should only be used if the resulting array is expected to be small, as all the data is loaded into the driver's memory.
Code:print("take : "+str(listRdd.takeSample(1,2)))
Output: take : [3, 2]

c) Soccer Tweet Analysis
Using following Datasets Perform the RDD Transformation.

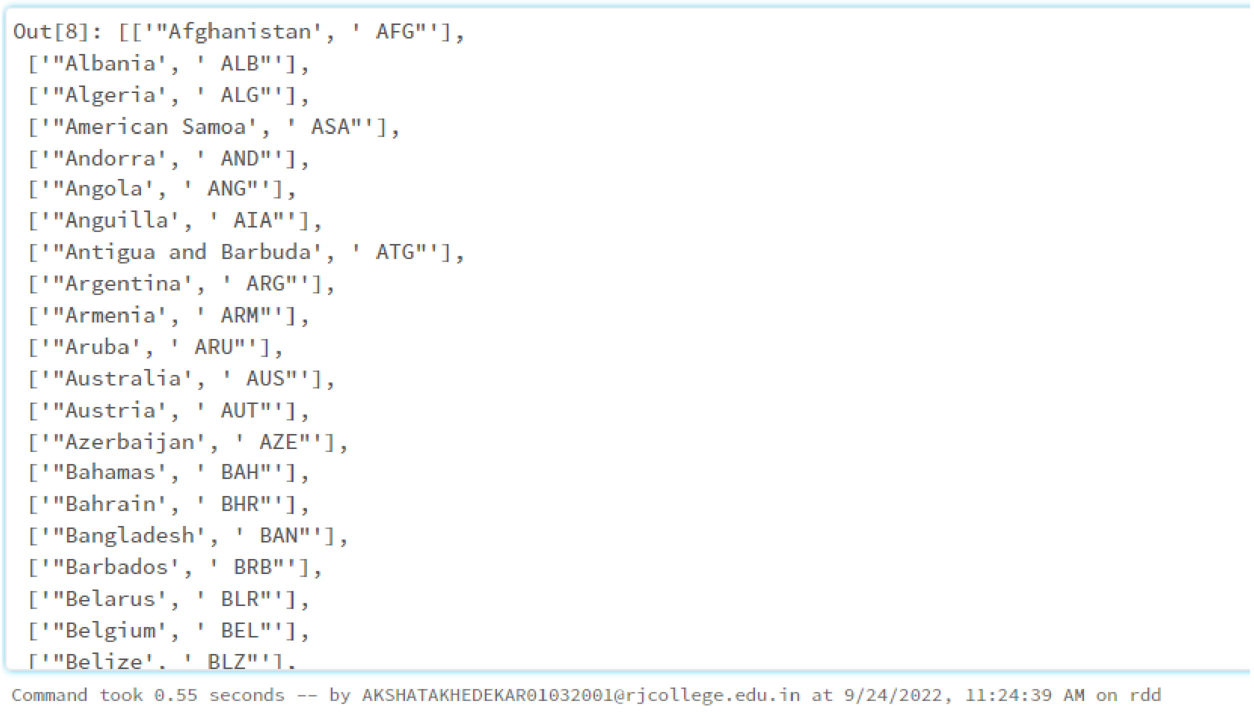
1. Import and create a new SQLContext

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
```

2. Read the country CSV file into an RDD.
```
country_lines =
sc.textFile('dbfs:/FileStore/shared_uploads/AKSHATAKHEDEKAR01032001@rjcollege.edu.in/country_list.csv')
```

3. Convert each line into a pair of words
```
country_lines.map(lambda a: a.split(",")).collect()
```

output:


```
Out[8]: [['"Afghanistan', ' AFG"'],
 ['"Albania', ' ALB"'],
 ['"Algeria', ' ALG"'],
 ['"American Samoa', ' ASA"'],
 ['"Andorra', ' AND"'],
 ['"Angola', ' ANG"'],
 ['"Anguilla', ' AIA"'],
 ['"Antigua and Barbuda', ' ATG"'],
 ['"Argentina', ' ARG"'],
 ['"Armenia', ' ARM"'],
 ['"Aruba', ' ARU"'],
 ['"Australia', ' AUS"'],
 ['"Austria', ' AUT"'],
 ['"Azerbaijan', ' AZE"'],
 ['"Bahamas', ' BAH"'],
 ['"Bahrain', ' BHR"'],
 ['"Bangladesh', ' BAN"'],
 ['"Barbados', ' BRB"'],
 ['"Belarus', ' BLR"'],
 ['"Belgium', ' BEL"'],
 ['"Belize', ' BLZ"'],
Command took 0.55 seconds -- by AKSHATAKHEDEKAR01032001@rjcollege.edu.in at 9/24/2022, 11:24:39 AM on rdd
```

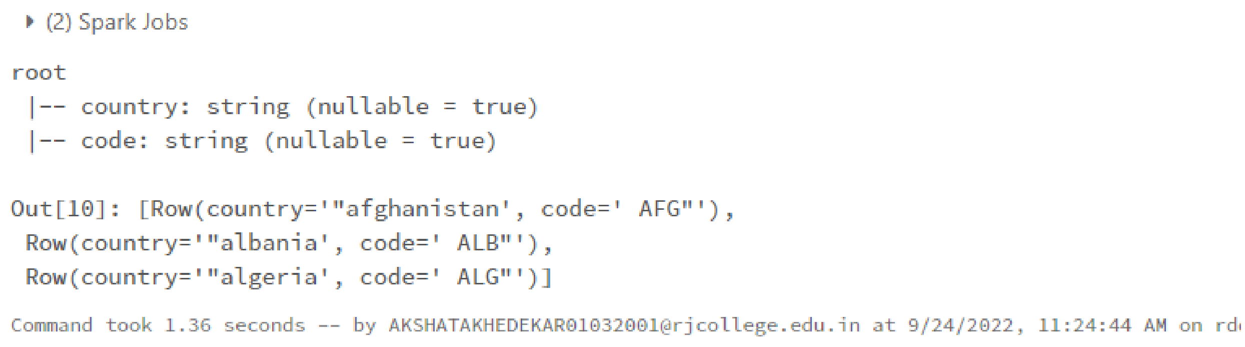
4. Convert each pair of words into a tuple

country_tuples = country_lines.map(lambda a: (a.split(",")[0].lower(), a.split(",")[1]))

5. Create the DataFrame, look at schema and contents

countryDF = sqlContext.createDataFrame(country_tuples, ["country", "code"])
countryDF.printSchema()
countryDF.take(3)

output:

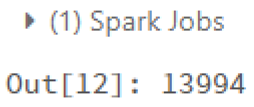


```
root
 |-- country: string (nullable = true)
 |-- code: string (nullable = true)

Out[10]: [Row(country='"afghanistan', code=' AFG"'),
 Row(country='"albania', code=' ALB"'),
 Row(country='"algeria', code=' ALG"')]
Command took 1.36 seconds -- by AKSHATAKHEDEKAR01032001@rjcollege.edu.in at 9/24/2022, 11:24:44 AM on rd
```

6. Read tweets CSV file into RDD of lines

```
tweets =
sc.textFile('dbfs:/FileStore/shared_uploads/AKSHATAKHEDEKAR01032001@rjcollege.edu.in
/tweets.csv')
tweets.count()
```

output:

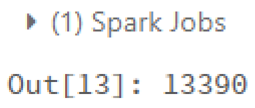


```
▸ (1) Spark Jobs

Out[12]: 13994
```

7.Clean the data: some tweets are empty. Remove the empty tweets using filter()

```
filtered_tweets = tweets.filter(lambda a: len(a) > 0)
filtered_tweets.count()
```

Output:



```
▸ (1) Spark Jobs

Out[13]: 13390
```

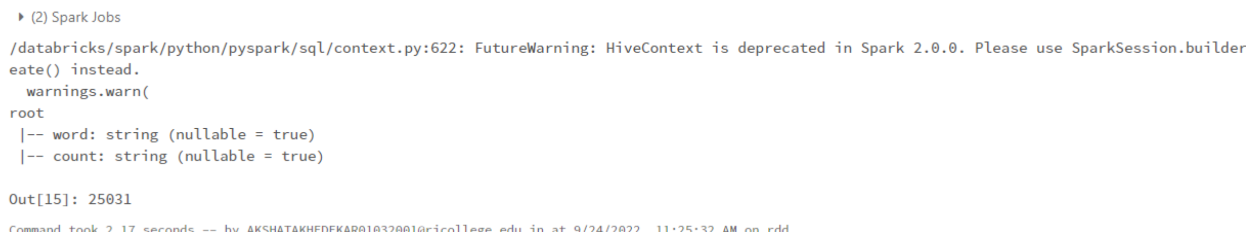
8. Perform WordCount on the cleaned tweet texts. (note: this is several lines.)

```
word_counts = filtered_tweets.flatMap(lambda a: a.split(" ")) \
    .map(lambda word: (word.lower(), 1)) \
    .reduceByKey(lambda a, b: a + b)

from pyspark.sql import HiveContext
from pyspark.sql.types import *

# sc is an existing SparkContext.
sqlContext = HiveContext(sc)

schemaString = "word count"

fields = [StructField(field_name, StringType(), True) for field_name in schemaString.split()]
schema = StructType(fields)
```

9. Create the DataFrame of tweet word counts

```
tweetsDF = sqlContext.createDataFrame(word_counts, schema)
tweetsDF.printSchema()
tweetsDF.count()
```

Output:

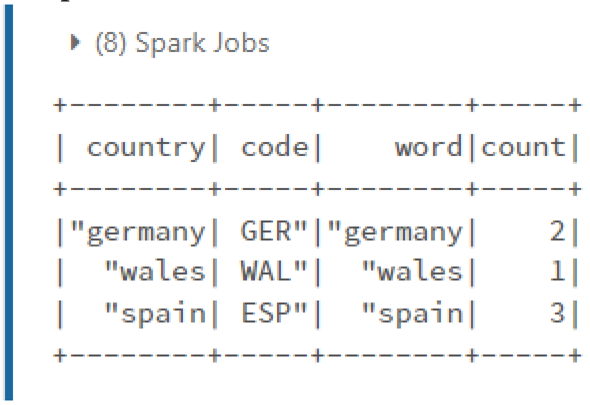


```
▸ (2) Spark Jobs
/databricks/spark/python/pyspark/sql/context.py:622: FutureWarning: HiveContext is deprecated in Spark 2.0.0. Please use SparkSession.builder
eate() instead.
  warnings.warn(
root
 |-- word: string (nullable = true)
 |-- count: string (nullable = true)

Out[15]: 25031

Command took 2.17 seconds -- by AKSHATAKHEDEKAR01032001@ricollege.edu in at 9/24/2022, 11:25:32 AM on rdd
```

10. Join the country and tweet DataFrames (on the appropriate column)

```
joined = countryDF.join(tweetsDF, countryDF.country == tweetsDF.word)
joined.take(5)
joined.show()
```

Output:

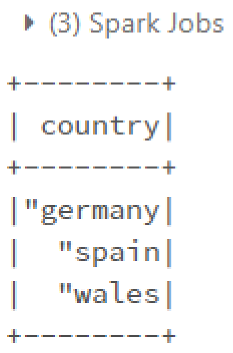


```
▸ (8) Spark Jobs

+--------+-----+--------+-----+
| country| code|    word|count|
+--------+-----+--------+-----+
|"germany| GER"|"germany|    2|
|  "wales| WAL"|  "wales|    1|
|  "spain| ESP"|  "spain|    3|
+--------+-----+--------+-----+
```

11. Question 1: number of distinct countries mentioned

```
distinct_countries = joined.select("country").distinct()
distinct_countries.show(100)
```

output:



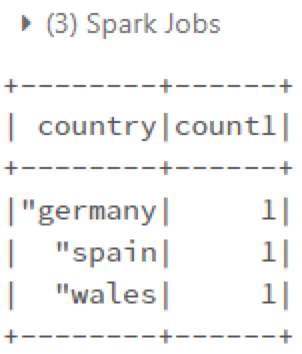
```
▸ (3) Spark Jobs

+--------+
| country|
+--------+
|"germany|
|  "spain|
|  "wales|
+--------+
```

12.Question 2: number of countries mentioned in tweets.

```
from pyspark.sql.functions import sum
from pyspark.sql import SparkSession
from pyspark.sql import Row
```

countries_count = joined.groupBy("country")
joined.createOrReplaceTempView("records")
spark.sql("SELECT country, count(*) count1 FROM records group by country order by count1 desc, country asc").show(100)

output:



```
▸ (3) Spark Jobs

+--------+------+
| country|count1|
+--------+------+
|"germany|     1|
|  "spain|     1|
|  "wales|     1|
+--------+------+
```

Practical 3 - Using Spark Dataset/Dataframe in PySpark/Scala

Solution:

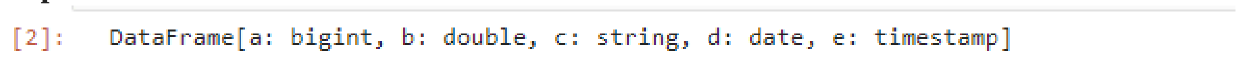
1. **We have to import pyspark**

   Code-
   ```
   ! pip install pyspark
   ```

2. **PySpark applications start with initializing SparkSession which is the entry point of PySpark as below.**

   Code-
   ```
   from pyspark.sql import SparkSession

   spark = SparkSession.builder.getOrCreate()
   ```
3. **DataFrame Creation**
   Firstly, you can create a PySpark DataFrame from a list of rows.

   Code-
   ```
   from datetime import datetime, date
   import pandas as pd
   from pyspark.sql import Row

   df = spark.createDataFrame([
       Row(a=1, b=2., c='string1', d=date(2000, 1, 1), e=datetime(2000, 1, 1, 12, 0)),
       Row(a=2, b=3., c='string2', d=date(2000, 2, 1), e=datetime(2000, 1, 2, 12, 0)),
       Row(a=4, b=5., c='string3', d=date(2000, 3, 1), e=datetime(2000, 1, 3, 12, 0))
   ])
   df
   ```

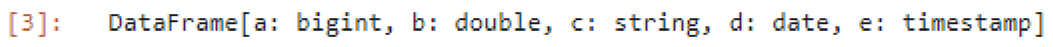
   Output-
   
   ```
   [2]:   DataFrame[a: bigint, b: double, c: string, d: date, e: timestamp]
   ```

4. **Create a PySpark DataFrame with an explicit schema.**

   Code-
   ```
   df = spark.createDataFrame([
   ```

```
    (1, 2., 'string1', date(2000, 1, 1), datetime(2000, 1, 1, 12, 0)),
    (2, 3., 'string2', date(2000, 2, 1), datetime(2000, 1, 2, 12, 0)),
    (3, 4., 'string3', date(2000, 3, 1), datetime(2000, 1, 3, 12, 0))
], schema='a long, b double, c string, d date, e timestamp')
df
```

Output-



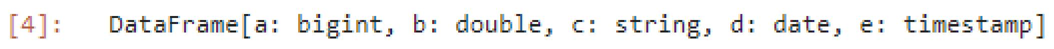
```
[3]:   DataFrame[a: bigint, b: double, c: string, d: date, e: timestamp]
```

5. **Create a PySpark DataFrame from a pandas DataFrame.**

Code-

```
pandas_df = pd.DataFrame({
    'a': [1, 2, 3],
    'b': [2., 3., 4.],
    'c': ['string1', 'string2', 'string3'],
    'd': [date(2000, 1, 1), date(2000, 2, 1), date(2000, 3, 1)],
    'e': [datetime(2000, 1, 1, 12, 0), datetime(2000, 1, 2, 12, 0), datetime(2000, 1, 3, 12,
0)]
})
df = spark.createDataFrame(pandas_df)
df
```

Output-



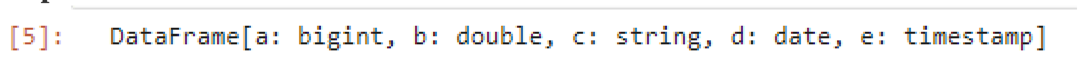
```
[4]:   DataFrame[a: bigint, b: double, c: string, d: date, e: timestamp]
```

6. **Create a PySpark DataFrame from an RDD consisting of a list of tuples.**

Code-

```
rdd = spark.sparkContext.parallelize([
    (1, 2., 'string1', date(2000, 1, 1), datetime(2000, 1, 1, 12, 0)),
    (2, 3., 'string2', date(2000, 2, 1), datetime(2000, 1, 2, 12, 0)),
    (3, 4., 'string3', date(2000, 3, 1), datetime(2000, 1, 3, 12, 0))
])
df = spark.createDataFrame(rdd, schema=['a', 'b', 'c', 'd', 'e'])
df
```

Output-



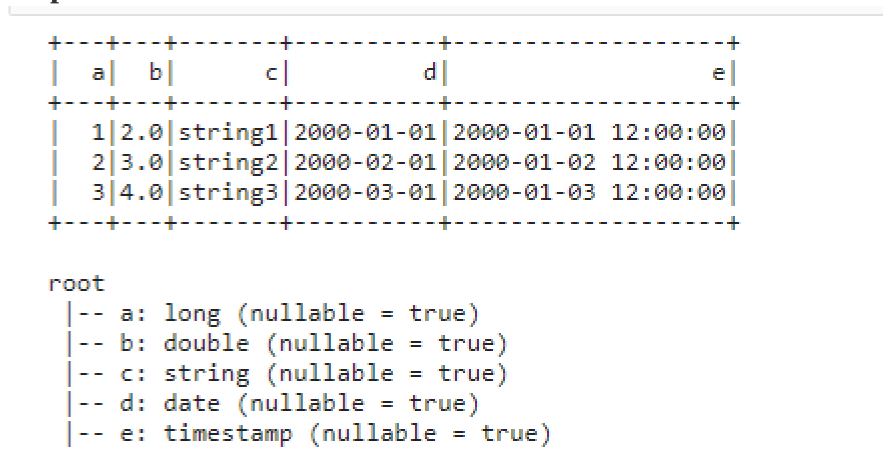
```
[5]:   DataFrame[a: bigint, b: double, c: string, d: date, e: timestamp]
```

7. **The DataFrames created above all have the same results and schema.**

Code-

# All DataFrames above result the same.
df.show()
df.printSchema()

Output-

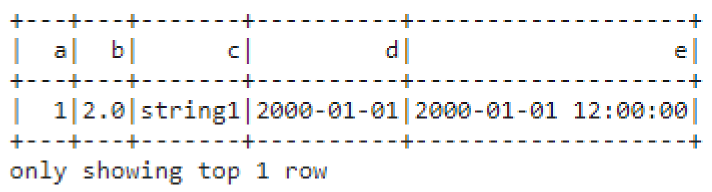


```
+---+---+-------+----------+-------------------+
|  a|  b|      c|         d|                  e|
+---+---+-------+----------+-------------------+
|  1|2.0|string1|2000-01-01|2000-01-01 12:00:00|
|  2|3.0|string2|2000-02-01|2000-01-02 12:00:00|
|  3|4.0|string3|2000-03-01|2000-01-03 12:00:00|
+---+---+-------+----------+-------------------+

root
 |-- a: long (nullable = true)
 |-- b: double (nullable = true)
 |-- c: string (nullable = true)
 |-- d: date (nullable = true)
 |-- e: timestamp (nullable = true)
```

8. Viewing Data
The top rows of a DataFrame can be displayed using DataFrame.show().

Code-
df.show(1)

Output-

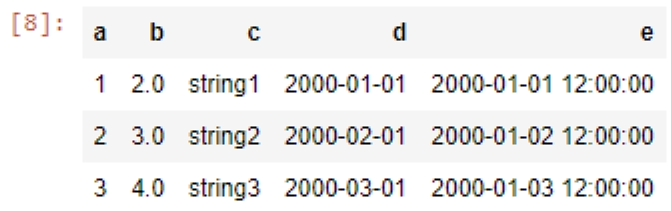


```
+---+---+-------+----------+-------------------+
|  a|  b|      c|         d|                  e|
+---+---+-------+----------+-------------------+
|  1|2.0|string1|2000-01-01|2000-01-01 12:00:00|
+---+---+-------+----------+-------------------+
only showing top 1 row
```

9. Alternatively, you can enable spark.sql.repl.eagerEval.enabled configuration for the eager evaluation of PySpark DataFrame in notebooks such as Jupyter. The number of rows to show can be controlled via spark.sql.repl.eagerEval.maxNumRows configuration.

Code-
spark.conf.set('spark.sql.repl.eagerEval.enabled', True)
df

Output-



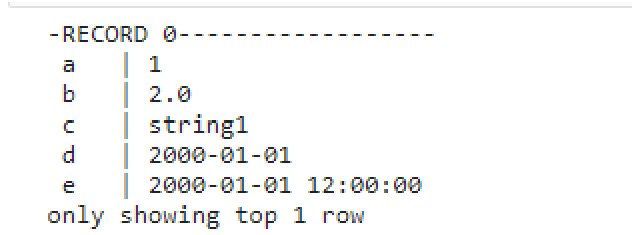
[8]:

a	b	c	d	e
1	2.0	string1	2000-01-01	2000-01-01 12:00:00
2	3.0	string2	2000-02-01	2000-01-02 12:00:00
3	4.0	string3	2000-03-01	2000-01-03 12:00:00

10. **The rows can also be shown vertically. This is useful when rows are too long to show horizontally.**

Code-
df.show(1, vertical=True)

Output-

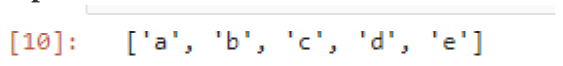


```
-RECORD 0------------------
 a   | 1
 b   | 2.0
 c   | string1
 d   | 2000-01-01
 e   | 2000-01-01 12:00:00
only showing top 1 row
```

`11. You can see the DataFrame's schema and column names as follows:

Code-
df.columns

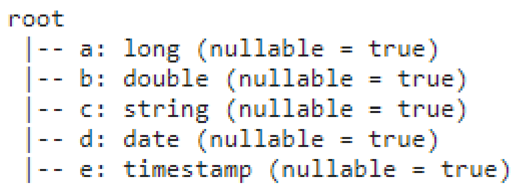
Output-



```
[10]:   ['a', 'b', 'c', 'd', 'e']
```

Code-
df.printSchema()

Output-

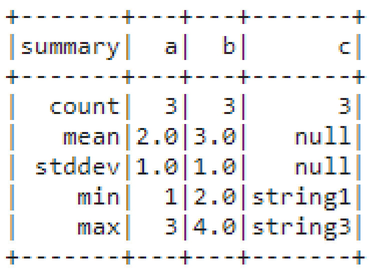


```
root
 |-- a: long (nullable = true)
 |-- b: double (nullable = true)
 |-- c: string (nullable = true)
 |-- d: date (nullable = true)
 |-- e: timestamp (nullable = true)
```

12. Show the summary of the DataFrame.

Code-
df.select("a", "b", "c").describe().show()

Output-

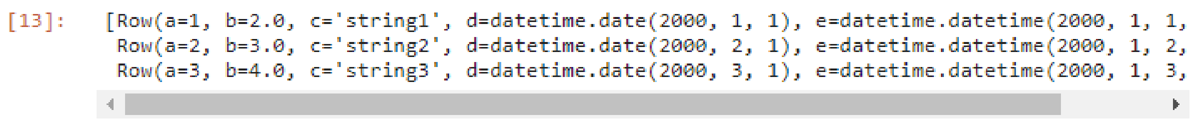


```
+-------+---+---+-------+
|summary|  a|  b|      c|
+-------+---+---+-------+
|  count|  3|  3|      3|
|   mean|2.0|3.0|   null|
| stddev|1.0|1.0|   null|
|    min|  1|2.0|string1|
|    max|  3|4.0|string3|
+-------+---+---+-------+
```

13. DataFrame.collect() collects the distributed data to the driver side as the local data in Python. Note that this can throw an out-of-memory error when the dataset is too large to fit in the driver side because it collects all the data from executors to the driver side.

Code-
df.collect()

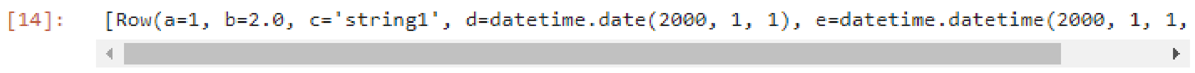
Output-



```
[13]:  [Row(a=1, b=2.0, c='string1', d=datetime.date(2000, 1, 1), e=datetime.datetime(2000, 1, 1,
        Row(a=2, b=3.0, c='string2', d=datetime.date(2000, 2, 1), e=datetime.datetime(2000, 1, 2,
        Row(a=3, b=4.0, c='string3', d=datetime.date(2000, 3, 1), e=datetime.datetime(2000, 1, 3,
```

14. In order to avoid throwing an out-of-memory exception, use DataFrame.take() or DataFrame.tail().

Code-
df.take(1)
Output-

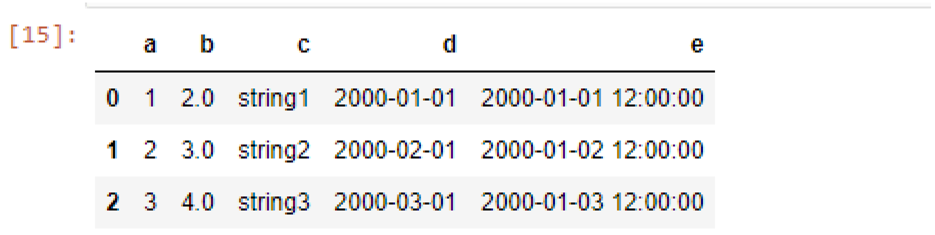


```
[14]:  [Row(a=1, b=2.0, c='string1', d=datetime.date(2000, 1, 1), e=datetime.datetime(2000, 1, 1,
```

15. PySpark DataFrame also provides the conversion back to a pandas DataFrame to leverage pandas API. Note that toPandas also collects all data into the driver side that can easily cause an out-of-memory-error when the data is too large to fit into the driver side.

Code-
df.toPandas()

Output-



[15]:

	a	b	c	d	e
0	1	2.0	string1	2000-01-01	2000-01-01 12:00:00
1	2	3.0	string2	2000-02-01	2000-01-02 12:00:00
2	3	4.0	string3	2000-03-01	2000-01-03 12:00:00

16. Selecting and Accessing Data
PySpark DataFrame is lazily evaluated and simply selecting a column does not trigger the computation but it returns a Column instance.

Code-
```
df.a
```

Output-
```
[16]:   Column<b'a'>
```

17. In fact, most column-wise operations return Columns.

Code-
```
from pyspark.sql import Column
 from pyspark.sql.functions import upper

 type(df.c) == type(upper(df.c)) == type(df.c.isNull())
```
Output-
```
[17]:   True
```

18. These Columns can be used to select the columns from a DataFrame. For example, DataFrame.select() takes the Column instances that return another DataFrame.

Code-
```
df.select(df.c).show()
```

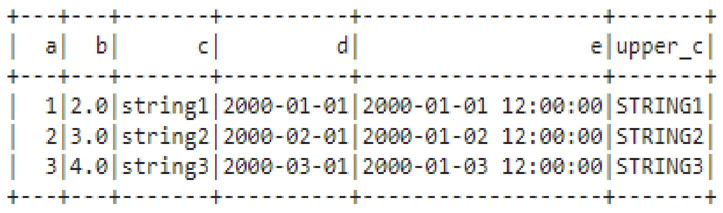
Output-
```
+-------+
|      c|
+-------+
|string1|
|string2|
|string3|
+-------+
```

19. Assign a new Column instance.

Code-

```
df.withColumn('upper_c', upper(df.c)).show()
```

Output-



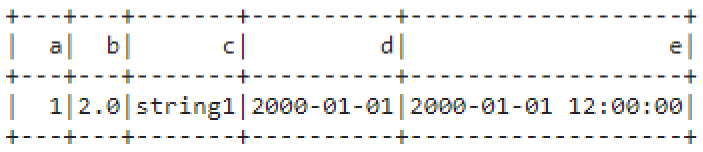
```
+---+---+-------+----------+-------------------+-------+
|  a|  b|      c|         d|                  e|upper_c|
+---+---+-------+----------+-------------------+-------+
|  1|2.0|string1|2000-01-01|2000-01-01 12:00:00|STRING1|
|  2|3.0|string2|2000-02-01|2000-01-02 12:00:00|STRING2|
|  3|4.0|string3|2000-03-01|2000-01-03 12:00:00|STRING3|
+---+---+-------+----------+-------------------+-------+
```

20.To select a subset of rows, use DataFrame.filter().

Code-

```
df.filter(df.a == 1).show()
```

Output-



```
+---+---+-------+----------+-------------------+
|  a|  b|      c|         d|                  e|
+---+---+-------+----------+-------------------+
|  1|2.0|string1|2000-01-01|2000-01-01 12:00:00|
+---+---+-------+----------+-------------------+
```

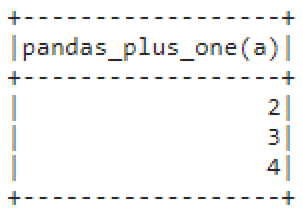
21. Applying a Function

PySpark supports various UDFs and APIs to allow users to execute Python native functions. See also the latest Pandas UDFs and Pandas Function APIs. For instance, the example below allows users to directly use the APIs in a pandas Series within Python native function.

Code-

```
import pandas as pd
from pyspark.sql.functions import pandas_udf

@pandas_udf('long')
def pandas_plus_one(series: pd.Series) -> pd.Series:
    # Simply plus one by using the pandas Series.
    return series + 1

df.select(pandas_plus_one(df.a)).show()
```

Output-



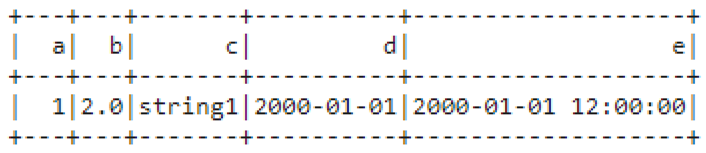
```
+------------------+
|pandas_plus_one(a)|
+------------------+
|                 2|
|                 3|
|                 4|
+------------------+
```

22. Another example is DataFrame.mapInPandas which allows users directly use the APIs in a pandas DataFrame without any restrictions such as the result length.

Code-

```
def pandas_filter_func(iterator):
    for pandas_df in iterator:
        yield pandas_df[pandas_df.a == 1]

df.mapInPandas(pandas_filter_func, schema=df.schema).show()
```

Output-



```
+---+---+-------+----------+-------------------+
|  a|  b|      c|         d|                  e|
+---+---+-------+----------+-------------------+
|  1|2.0|string1|2000-01-01|2000-01-01 12:00:00|
+---+---+-------+----------+-------------------+
```

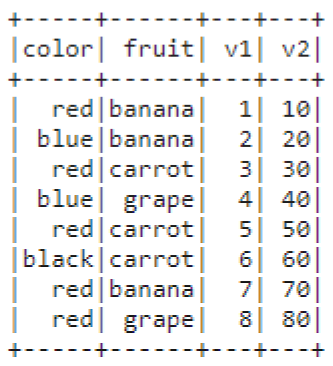
23. Grouping Data.

PySpark DataFrame also provides a way of handling grouped data by using the common approach, split-apply-combine strategy. It groups the data by a certain condition, applies a function to each group and then combines them back to the DataFrame.

Code-

```
df = spark.createDataFrame([['red', 'banana', 1, 10], ['blue', 'banana', 2, 20], ['red',
'carrot', 3, 30],['blue', 'grape', 4, 40], ['red', 'carrot', 5, 50], ['black', 'carrot', 6, 60],['red',
'banana', 7, 70], ['red', 'grape', 8, 80]], schema=['color', 'fruit', 'v1', 'v2'])
df.show()
```

Output-



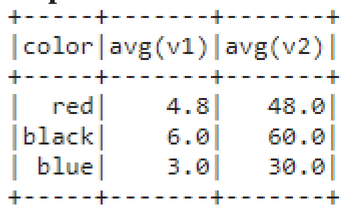
```
+-----+------+---+---+
|color| fruit| v1| v2|
+-----+------+---+---+
|  red|banana|  1| 10|
| blue|banana|  2| 20|
|  red|carrot|  3| 30|
| blue| grape|  4| 40|
|  red|carrot|  5| 50|
|black|carrot|  6| 60|
|  red|banana|  7| 70|
|  red| grape|  8| 80|
+-----+------+---+---+
```

24. Grouping and then applying the avg() function to the resulting groups.

Code-

```
df.groupby('color').avg().show()
```

Output-



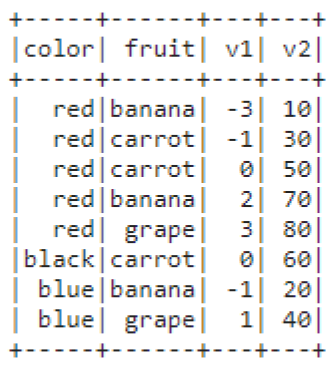
```
+-----+-------+-------+
|color|avg(v1)|avg(v2)|
+-----+-------+-------+
|  red|    4.8|   48.0|
|black|    6.0|   60.0|
| blue|    3.0|   30.0|
+-----+-------+-------+
```

25. You can also apply a Python native function against each group by using pandas API.

Code-

```
def plus_mean(pandas_df):
    return pandas_df.assign(v1=pandas_df.v1 - pandas_df.v1.mean())

df.groupby('color').applyInPandas(plus_mean, schema=df.schema).show()
```

Output-



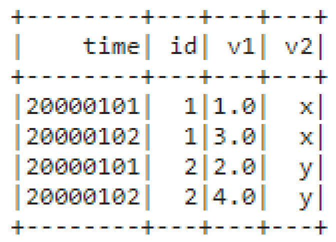
```
+-----+------+---+---+
|color| fruit| v1| v2|
+-----+------+---+---+
|  red|banana| -3| 10|
|  red|carrot| -1| 30|
|  red|carrot|  0| 50|
|  red|banana|  2| 70|
|  red| grape|  3| 80|
|black|carrot|  0| 60|
| blue|banana| -1| 20|
| blue| grape|  1| 40|
+-----+------+---+---+
```

26. Co-grouping and applying a function.

Code-

```
df1 = spark.createDataFrame(
   [(20000101, 1, 1.0), (20000101, 2, 2.0), (20000102, 1, 3.0), (20000102, 2, 4.0)],
   ('time', 'id', 'v1'))

df2 = spark.createDataFrame(
   [(20000101, 1, 'x'), (20000101, 2, 'y')],
   ('time', 'id', 'v2'))

def asof_join(l, r):
   return pd.merge_asof(l, r, on='time', by='id')

df1.groupby('id').cogroup(df2.groupby('id')).applyInPandas(
   asof_join, schema='time int, id int, v1 double, v2 string').show()
```

Output-



```
+--------+---+---+---+
|    time| id| v1| v2|
+--------+---+---+---+
|20000101|  1|1.0|  x|
|20000102|  1|3.0|  x|
|20000101|  2|2.0|  y|
|20000102|  2|4.0|  y|
+--------+---+---+---+
```

27.Getting Data in/out

CSV is straightforward and easy to use. Parquet and ORC are efficient and compact file formats to read and write faster.

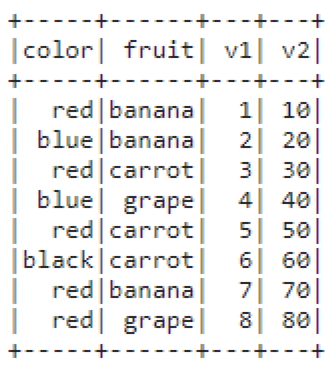
There are many other data sources available in PySpark such as JDBC, text, binaryFile, Avro, etc. See also the latest Spark SQL, DataFrames and Datasets Guide in Apache Spark documentation.

CSV

Code-

```
df.write.csv('foo.csv', header=True)
spark.read.csv('foo.csv', header=True).show()
```

Output-



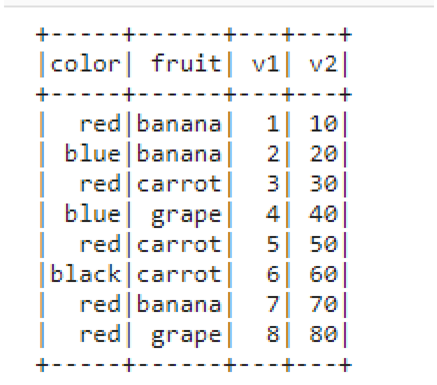
```
+-----+------+---+---+
|color| fruit| v1| v2|
+-----+------+---+---+
|  red|banana|  1| 10|
| blue|banana|  2| 20|
|  red|carrot|  3| 30|
| blue| grape|  4| 40|
|  red|carrot|  5| 50|
|black|carrot|  6| 60|
|  red|banana|  7| 70|
|  red| grape|  8| 80|
+-----+------+---+---+
```

28. Parquet

Code-

```
df.write.parquet('bar.parquet')
spark.read.parquet('bar.parquet').show()
```

Output-



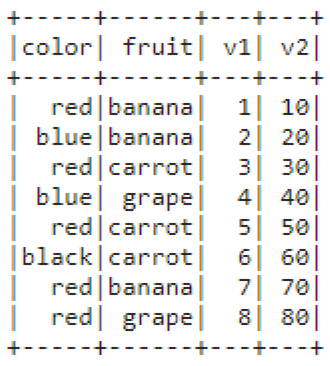
```
+-----+------+---+---+
|color| fruit| v1| v2|
+-----+------+---+---+
|  red|banana|  1| 10|
| blue|banana|  2| 20|
|  red|carrot|  3| 30|
| blue| grape|  4| 40|
|  red|carrot|  5| 50|
|black|carrot|  6| 60|
|  red|banana|  7| 70|
|  red| grape|  8| 80|
+-----+------+---+---+
```

29. ORC

Code-

```
df.write.orc('zoo.orc')
spark.read.orc('zoo.orc').show()
```

Output-



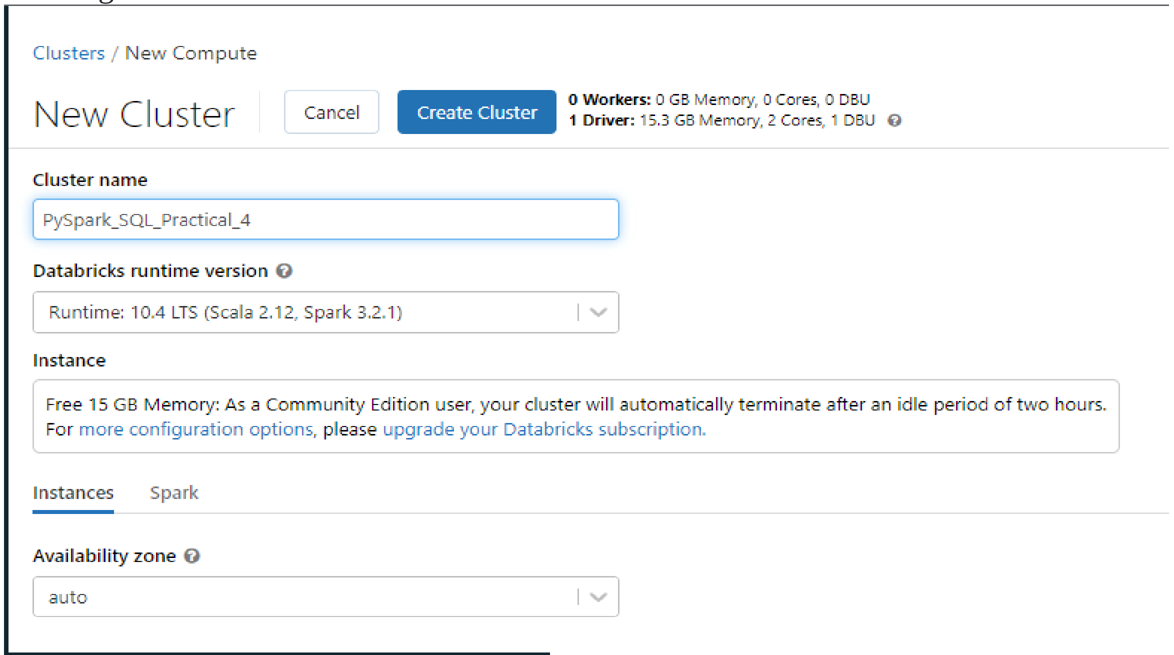
```
+-----+------+---+---+
|color| fruit| v1| v2|
+-----+------+---+---+
|  red|banana|  1| 10|
| blue|banana|  2| 20|
|  red|carrot|  3| 30|
| blue| grape|  4| 40|
|  red|carrot|  5| 50|
|black|carrot|  6| 60|
|  red|banana|  7| 70|
|  red| grape|  8| 80|
+-----+------+---+---+
```

Practical 4 - PySpark SQL

a) SparkSession and SparkContext
b) Creating DataFrames
c) DataFrame Operations
d) Running SQL Queries
e) Global Temporary View
f) Aggregations

Solution

Creating Cluster and Notebook



Clusters / New Compute

New Cluster | Cancel | Create Cluster

0 Workers: 0 GB Memory, 0 Cores, 0 DBU
1 Driver: 15.3 GB Memory, 2 Cores, 1 DBU

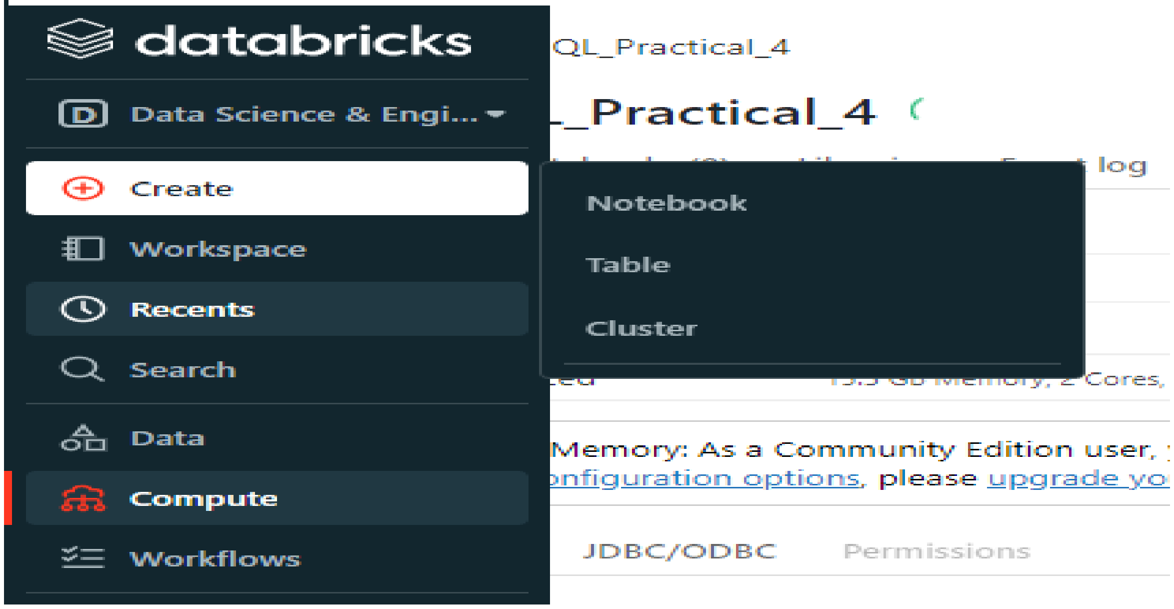
Cluster name
PySpark_SQL_Practical_4

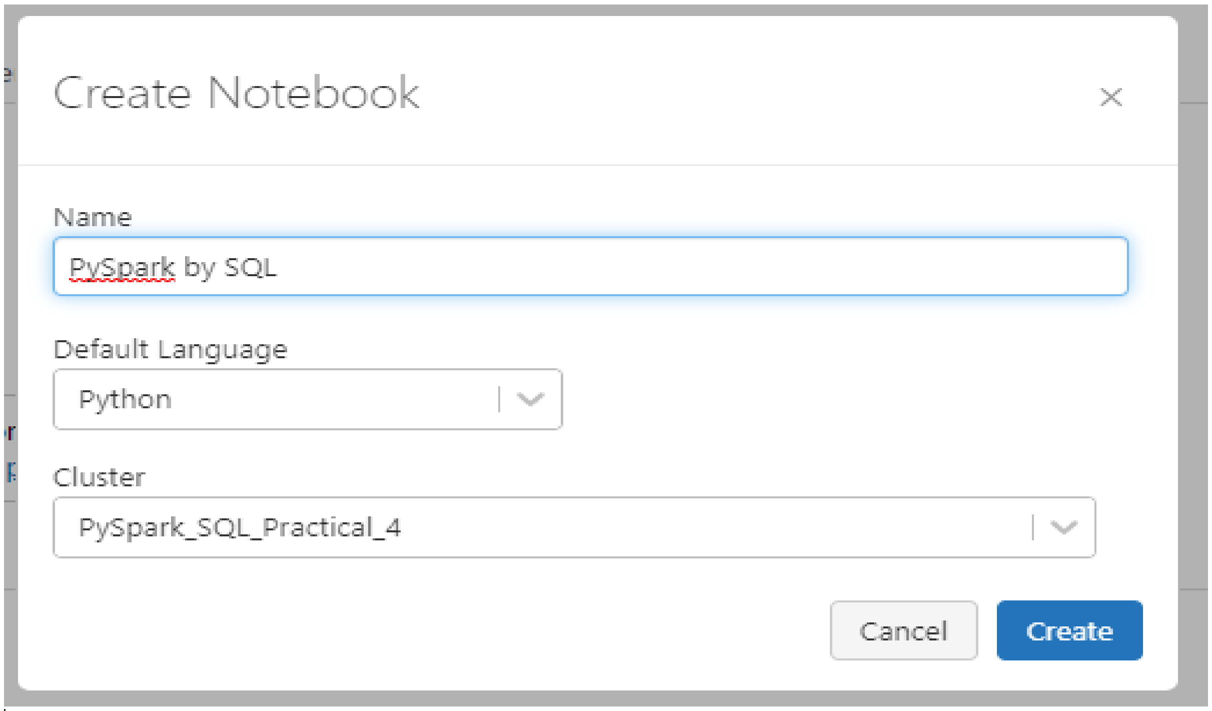
Databricks runtime version
Runtime: 10.4 LTS (Scala 2.12, Spark 3.2.1)

Instance
Free 15 GB Memory: As a Community Edition user, your cluster will automatically terminate after an idle period of two hours.
For more configuration options, please upgrade your Databricks subscription.

Instances | Spark

Availability zone
auto

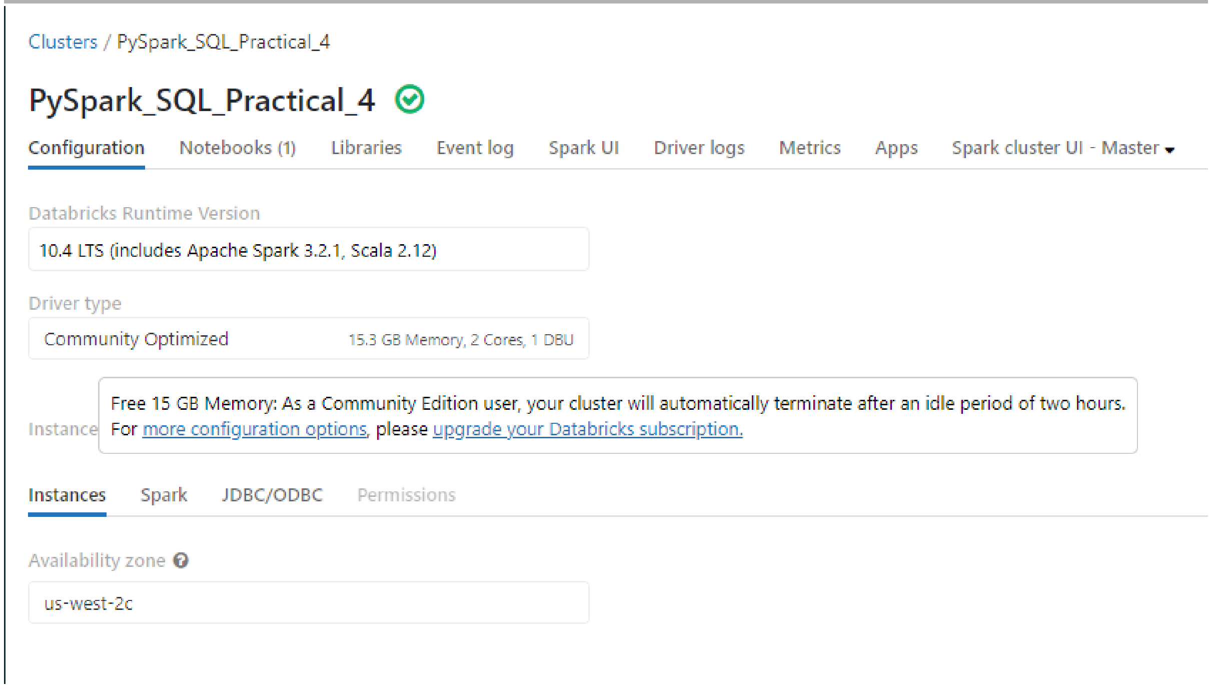




PySpark by SQL

Python

PySpark_SQL_Practical_4



Databricks Runtime Version: 10.4 LTS (includes Apache Spark 3.2.1, Scala 2.12)

Driver type: Community Optimized 15.3 GB Memory, 2 Cores, 1 DBU

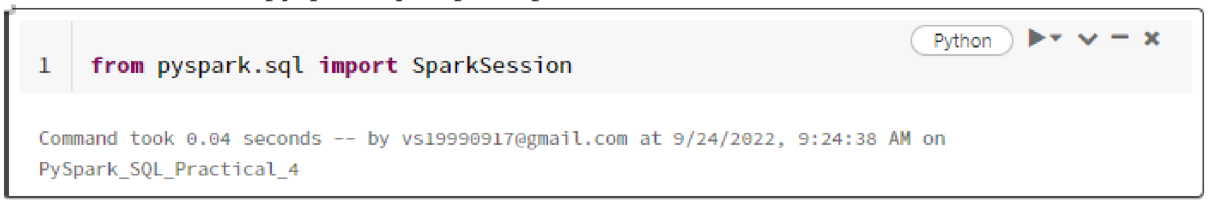
Instance: Free 15 GB Memory: As a Community Edition user, your cluster will automatically terminate after an idle period of two hours. For more configuration options, please upgrade your Databricks subscription.

Availability zone: us-west-2c

a) SparkSession and SparkContext

Creating SparkSession and SparkContext

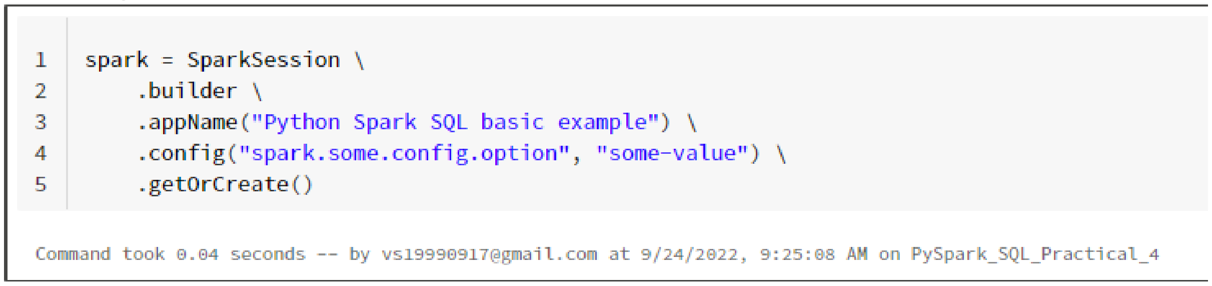
### The entry point into all functionality in Spark is the SparkSession class. To create a basic SparkSession, just use SparkSession.builder:
Code:- from pyspark.sql import SparkSession



```
from pyspark.sql import SparkSession
```

```
Command took 0.04 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:24:38 AM on
PySpark_SQL_Practical_4
```

Code:- spark = SparkSession \
.builder \
.appName("Python Spark SQL basic example") \
.config("spark.some.config.option", "some-value") \
.getOrCreate()



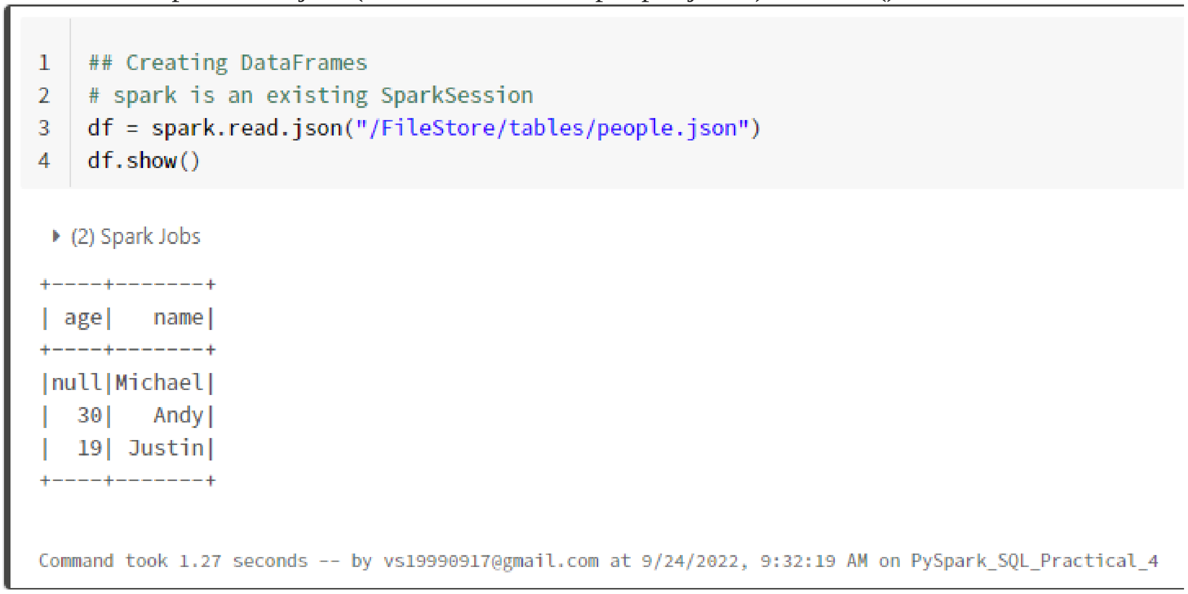
```
spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

```
Command took 0.04 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:25:08 AM on PySpark_SQL_Practical_4
```

b) Creating DataFrames

Creating DataFrames
Code :- ## Creating DataFrames
# spark is an existing SparkSession

df = spark.read.json("/FileStore/tables/people.json") df.show()



```
## Creating DataFrames
# spark is an existing SparkSession
df = spark.read.json("/FileStore/tables/people.json")
df.show()
```

(2) Spark Jobs

```
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

```
Command took 1.27 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:32:19 AM on PySpark_SQL_Practical_4
```

Note:- If the dataset is not previously in the examples then we need to upload or create a data set named as people.json when the column names are name and age

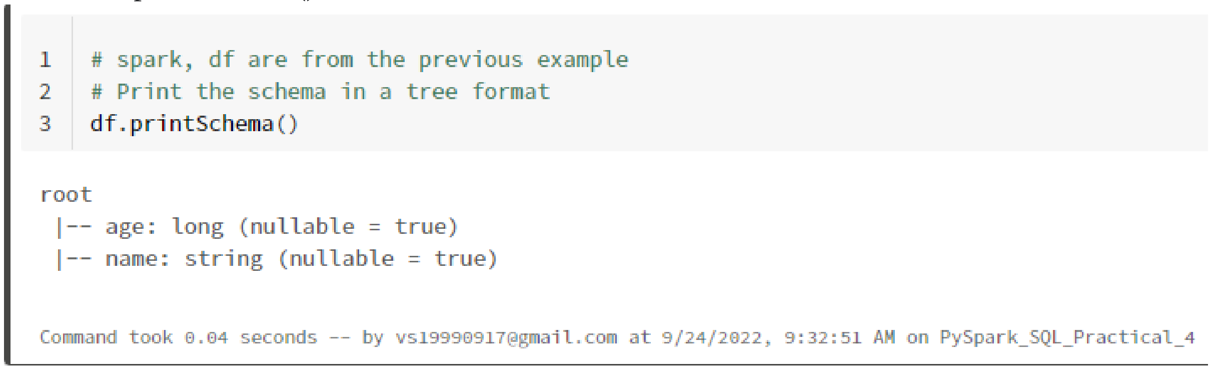
Sample Data set:- {"name":"Michael"}
{"name":"Andy", "age":30}
{"name":"Justin", "age":19}

C)DataFrame Operation

Creating DataFrame Operations

## In Python it's possible to access a DataFrame's columns either by attribute (df.age) or by indexing (df['age']). While the former is convenient for interactive data exploration, users are highly encouraged to use the latter form, which is future proof and won't break with column names that are also attributes on the DataFrame class.

Code:- df.printSchema()

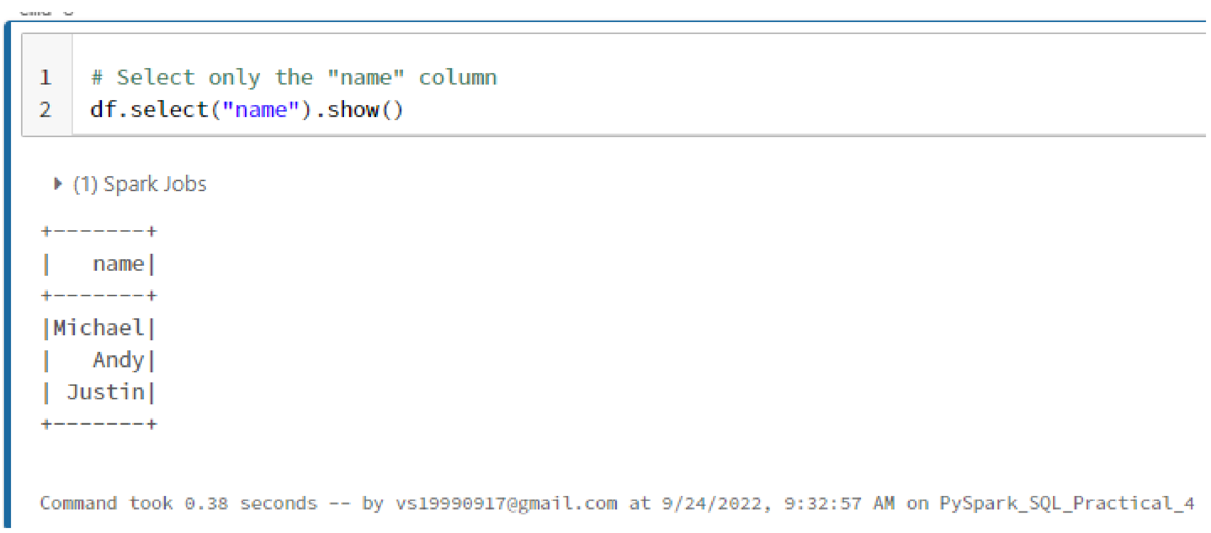


```
# spark, df are from the previous example
# Print the schema in a tree format
df.printSchema()
```

```
root
 |-- age: long (nullable = true)
 |-- name: string (nullable = true)
```

```
Command took 0.04 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:32:51 AM on PySpark_SQL_Practical_4
```

Code:- # Select only the "name" column
df.select("name").show()



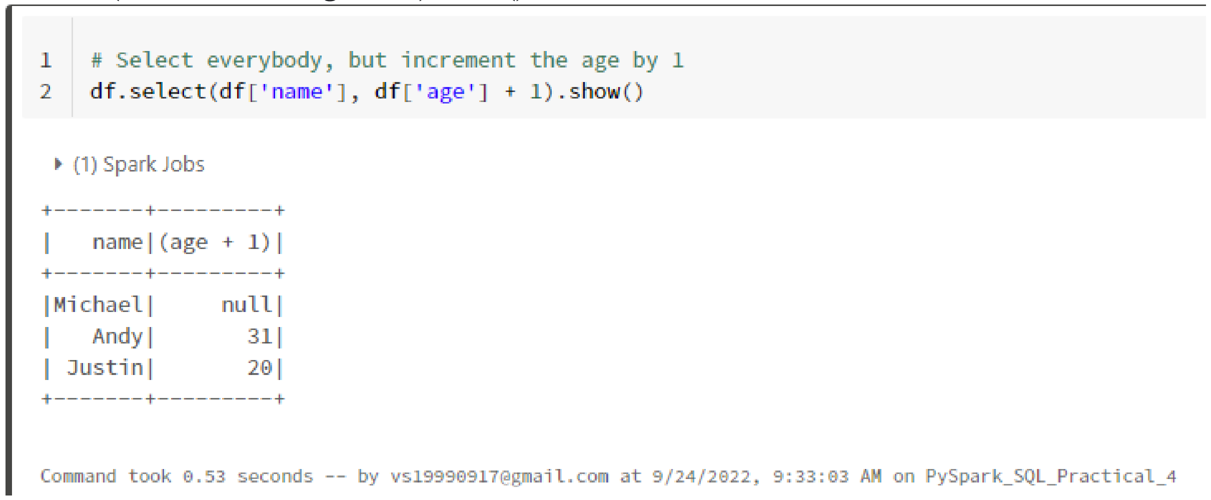
```
# Select only the "name" column
df.select("name").show()
```

▸ (1) Spark Jobs

```
+-------+
|   name|
+-------+
|Michael|
|   Andy|
| Justin|
+-------+
```

Command took 0.38 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:32:57 AM on PySpark_SQL_Practical_4

Code:- # Select everybody, but increment the age by 1
df.select(df['name'], df['age'] + 1).show()



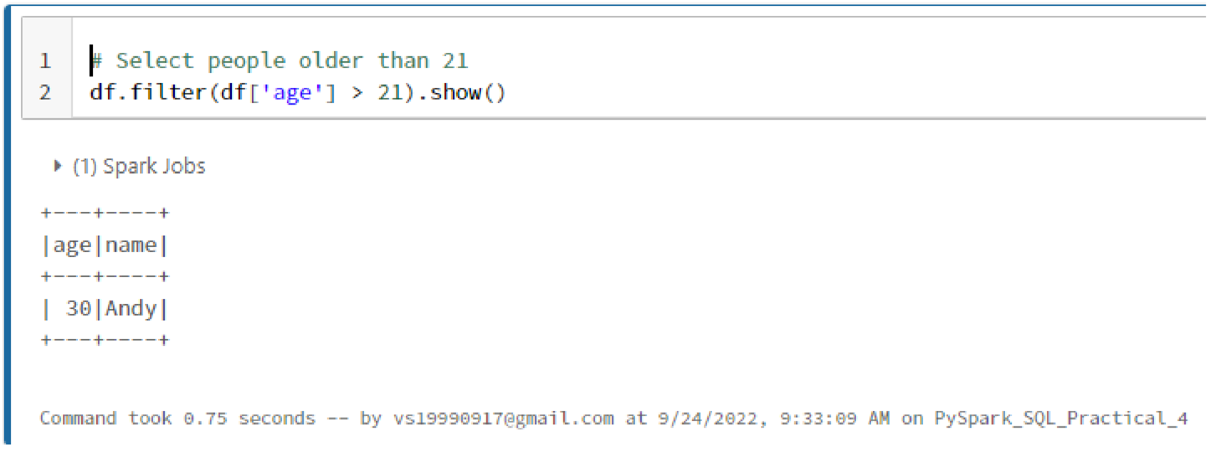
```
# Select everybody, but increment the age by 1
df.select(df['name'], df['age'] + 1).show()
```

▸ (1) Spark Jobs

```
+-------+---------+
|   name|(age + 1)|
+-------+---------+
|Michael|     null|
|   Andy|       31|
| Justin|       20|
+-------+---------+
```

Command took 0.53 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:33:03 AM on PySpark_SQL_Practical_4

Code:- # Select people older than 21
df.filter(df['age'] > 21).show()



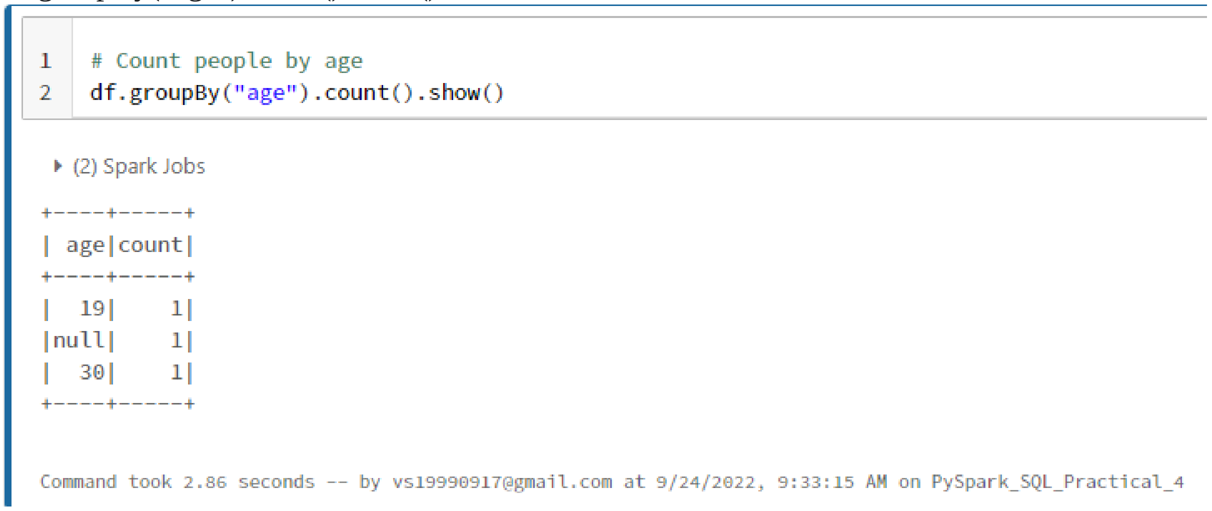
```
# Select people older than 21
df.filter(df['age'] > 21).show()
```

▸ (1) Spark Jobs

```
+---+----+
|age|name|
+---+----+
| 30|Andy|
+---+----+
```

Command took 0.75 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:33:09 AM on PySpark_SQL_Practical_4

Code:- # Count people by age
df.groupBy("age").count().show()



```
# Count people by age
df.groupBy("age").count().show()
```

(2) Spark Jobs

```
+----+-----+
| age|count|
+----+-----+
|  19|    1|
|null|    1|
|  30|    1|
+----+-----+
```

Command took 2.86 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:33:15 AM on PySpark_SQL_Practical_4

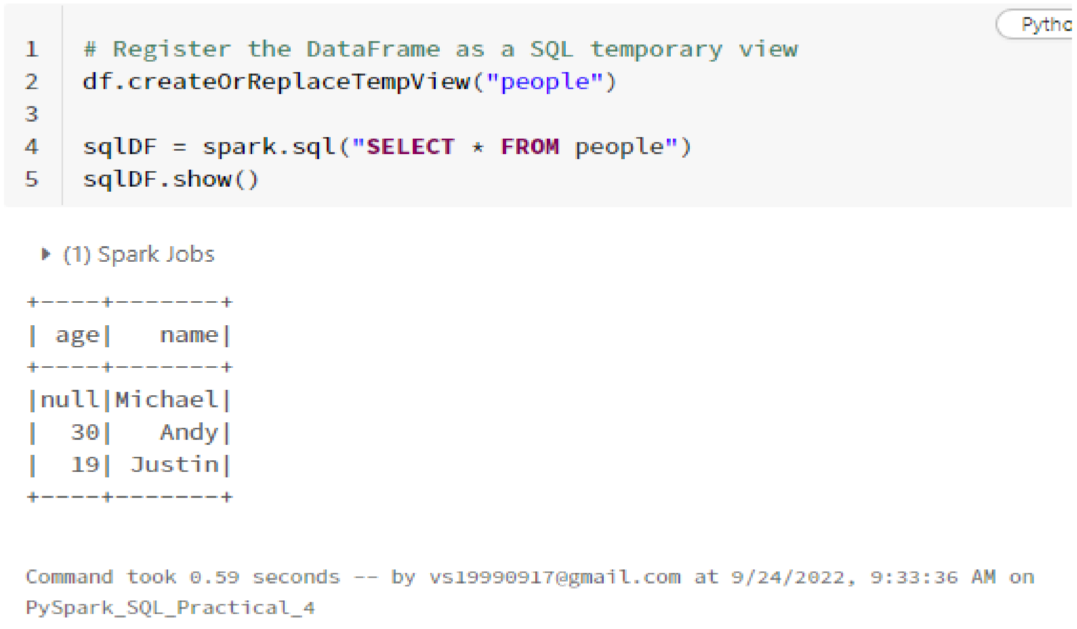
d)Running SQL Queries

Run SQL Queries

The sql function on a SparkSession enables applications to run SQL queries programmatically and returns the result as a DataFrame.

Code:-
df.createOrReplaceTempView("people")

sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()



```
# Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
```

(1) Spark Jobs

```
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

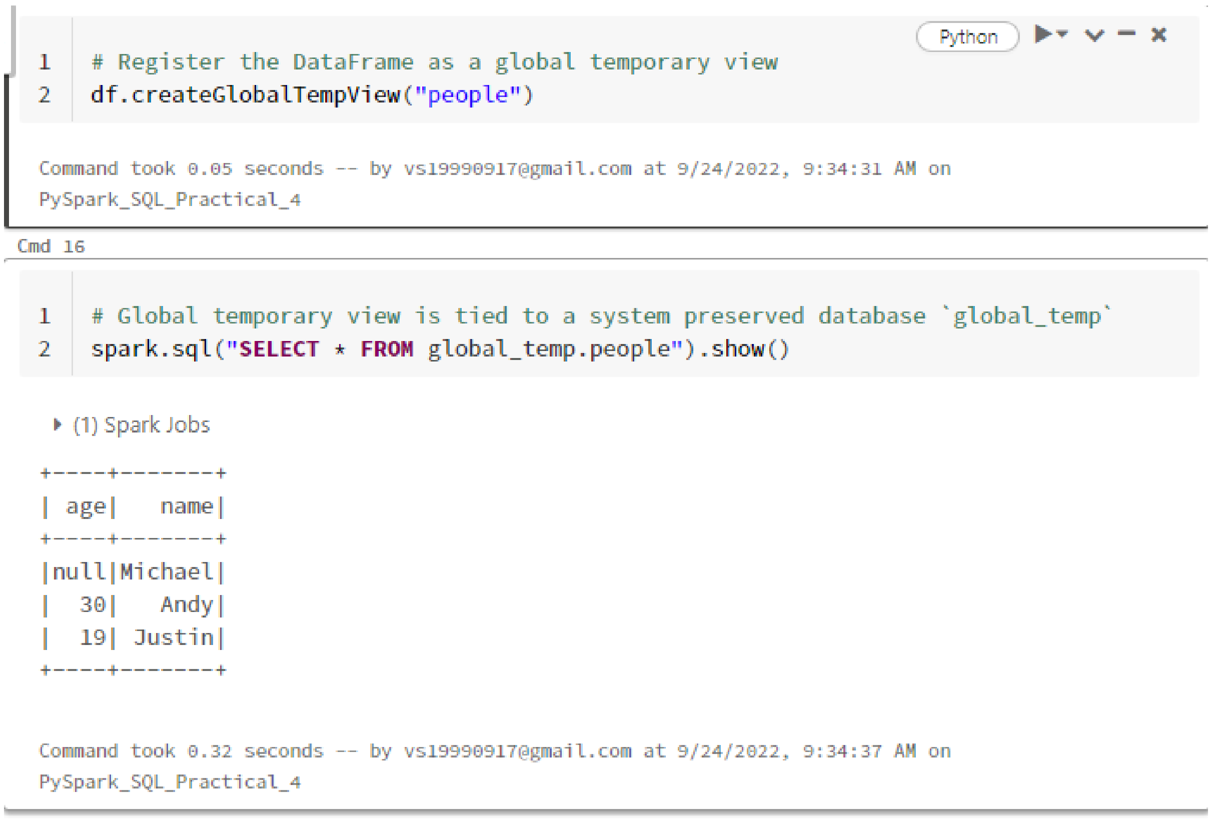
Command took 0.59 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:33:36 AM on PySpark_SQL_Practical_4

e)Global Temporary View
Creating Temporary View

Temporary views in Spark SQL are session-scoped and will disappear if the session that creates it terminates. If you want to have a temporary view that is shared among all sessions and keep alive until the Spark application terminates, you can create a global temporary view. Global temporary view is tied to a system preserved database global_temp, and we must use the qualified name to refer it, e.g. SELECT * FROM global_temp.view1.

Code:- *# Register the DataFrame as a global temporary view*
df.createGlobalTempView("people")

# Global temporary view is tied to a system preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show()



```
# Register the DataFrame as a global temporary view
df.createGlobalTempView("people")
```

```
Command took 0.05 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:34:31 AM on
PySpark_SQL_Practical_4
```

Cmd 16

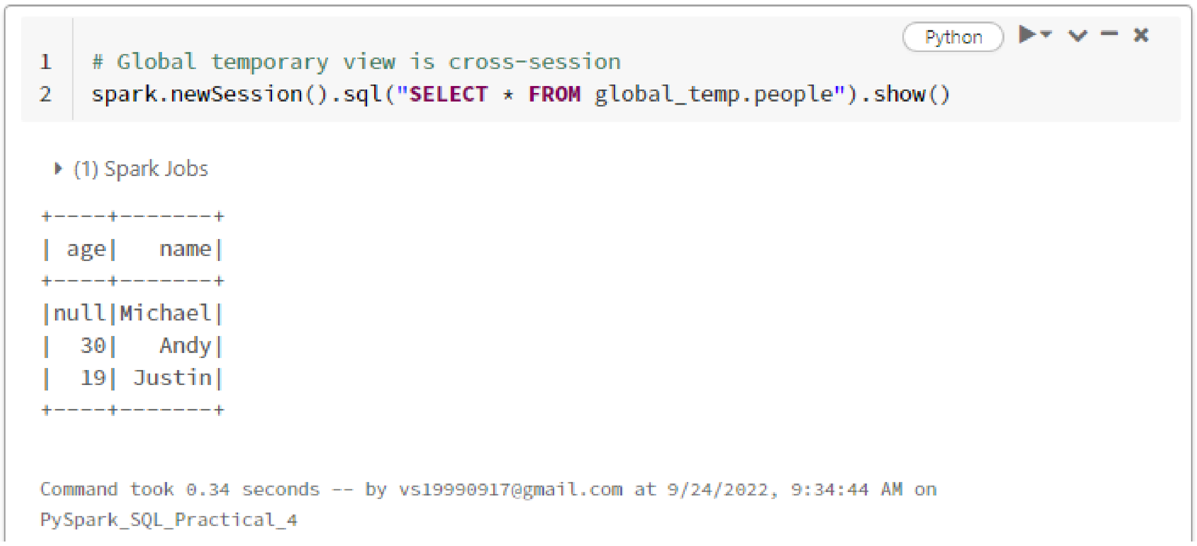
```
# Global temporary view is tied to a system preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show()
```

▸ (1) Spark Jobs

```
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

```
Command took 0.32 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:34:37 AM on
PySpark_SQL_Practical_4
```

Code:- *# Global temporary view is cross-session*
spark.newSession().sql("SELECT * FROM global_temp.people").show()



```
# Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show()
```

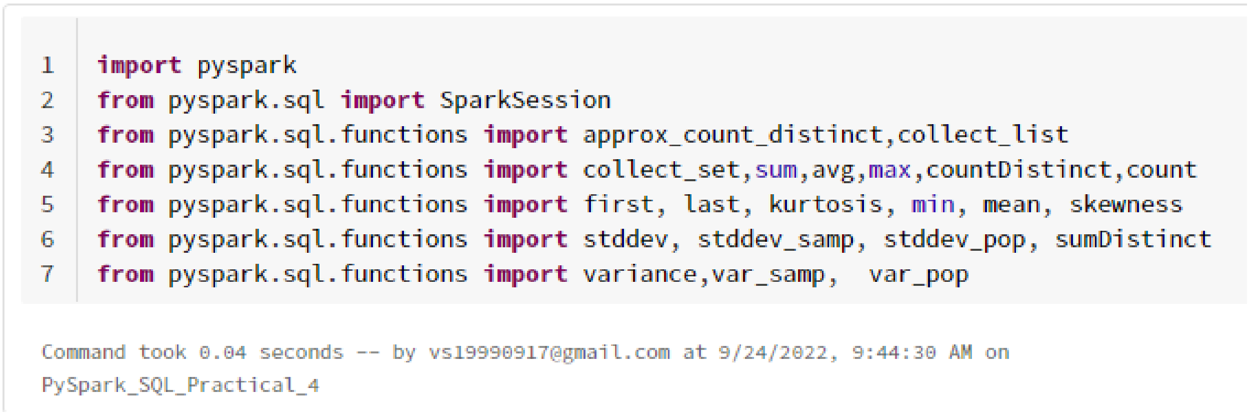
(1) Spark Jobs

```
+----+-------+
| age|   name|
+----+-------+
|null|Michael|
|  30|   Andy|
|  19| Justin|
+----+-------+
```

Command took 0.34 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:34:44 AM on PySpark_SQL_Practical_4

f)Aggregations

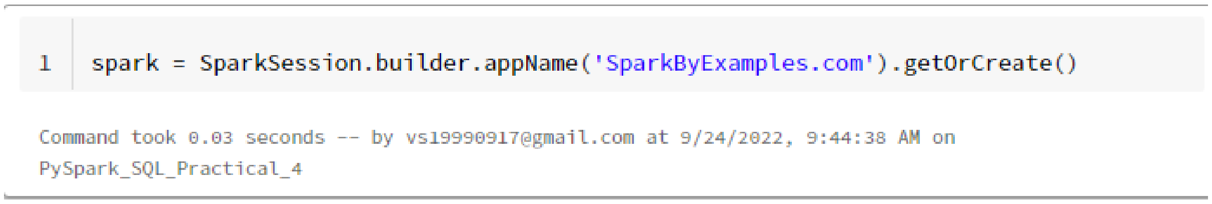
PySpark Aggregate Functions
PySpark SQL Aggregate functions are grouped as "agg_funcs" in Pyspark. Below is a list of functions defined under this group.
# Importing Libraries
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import approx_count_distinct,collect_list
from pyspark.sql.functions import collect_set,sum,avg,max,countDistinct,count
from pyspark.sql.functions import first, last, kurtosis, min, mean, skewness
from pyspark.sql.functions import stddev, stddev_samp, stddev_pop, sumDistinct
from pyspark.sql.functions import variance,var_samp, var_pop



```
import pyspark
from pyspark.sql import SparkSession
from pyspark.sql.functions import approx_count_distinct,collect_list
from pyspark.sql.functions import collect_set,sum,avg,max,countDistinct,count
from pyspark.sql.functions import first, last, kurtosis, min, mean, skewness
from pyspark.sql.functions import stddev, stddev_samp, stddev_pop, sumDistinct
from pyspark.sql.functions import variance,var_samp,  var_pop
```

Command took 0.04 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:44:30 AM on PySpark_SQL_Practical_4

#Creating SparkSession
 Creating a SparkSession
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()



```
spark = SparkSession.builder.appName('SparkByExamples.com').getOrCreate()
```

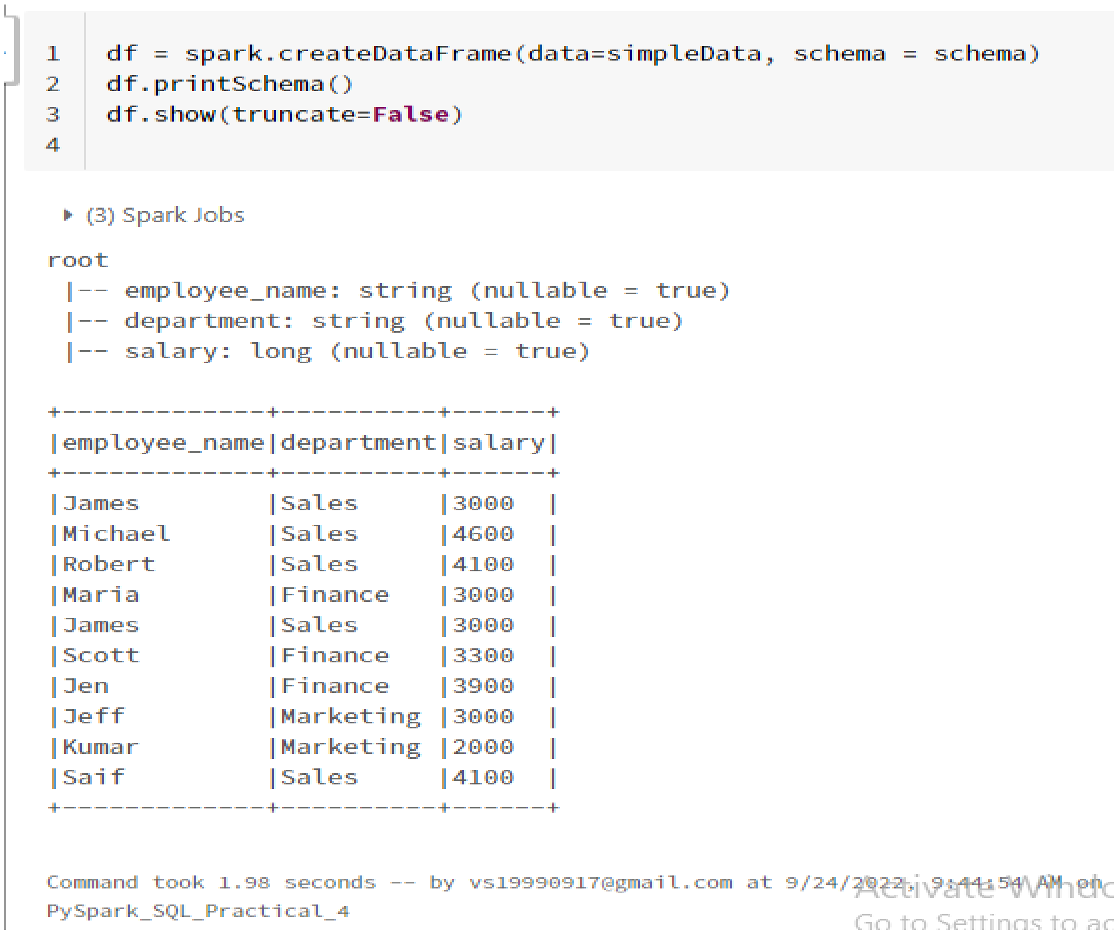
```
Command took 0.03 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:44:38 AM on
PySpark_SQL_Practical_4
```

#Sample Dataset
 Create sample Dataset
simpleData = [("James", "Sales", 3000),
 ("Michael", "Sales", 4600),
 ("Robert", "Sales", 4100),
 ("Maria", "Finance", 3000),
 ("James", "Sales", 3000),
 ("Scott", "Finance", 3300),
 ("Jen", "Finance", 3900),
 ("Jeff", "Marketing", 3000),
 ("Kumar", "Marketing", 2000),
 ("Saif", "Sales", 4100)
 ]
schema = ["employee_name", "department", "salary"]

```
simpleData = [("James", "Sales", 3000),
    ("Michael", "Sales", 4600),
    ("Robert", "Sales", 4100),
    ("Maria", "Finance", 3000),
    ("James", "Sales", 3000),
    ("Scott", "Finance", 3300),
    ("Jen", "Finance", 3900),
    ("Jeff", "Marketing", 3000),
    ("Kumar", "Marketing", 2000),
    ("Saif", "Sales", 4100)
  ]
schema = ["employee_name", "department", "salary"]

```

```
Command took 0.05 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:44:47 AM on
PySpark_SQL_Practical_4
```

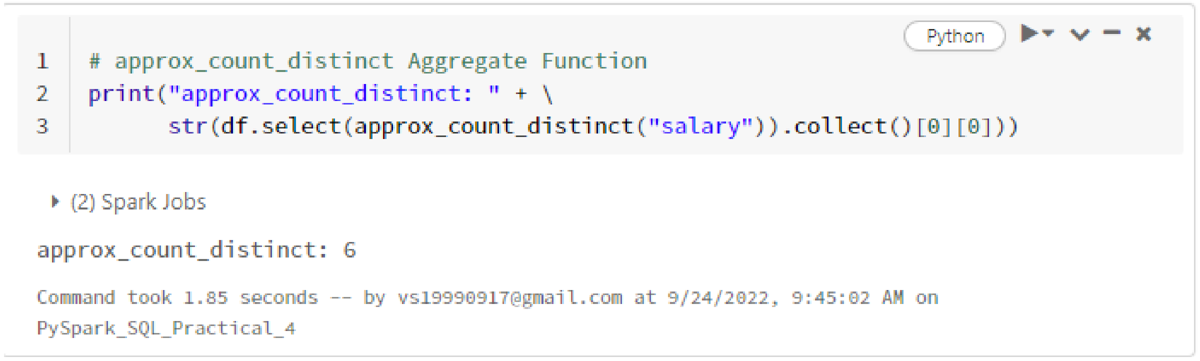
Code:-df = spark.createDataFrame(data=simpleData, schema = schema)
df.printSchema()
df.show(truncate=False)



```
df = spark.createDataFrame(data=simpleData, schema = schema)
df.printSchema()
df.show(truncate=False)
```

```
root
 |-- employee_name: string (nullable = true)
 |-- department: string (nullable = true)
 |-- salary: long (nullable = true)

+-------------+----------+------+
|employee_name|department|salary|
+-------------+----------+------+
|James        |Sales     |3000  |
|Michael      |Sales     |4600  |
|Robert       |Sales     |4100  |
|Maria        |Finance   |3000  |
|James        |Sales     |3000  |
|Scott        |Finance   |3300  |
|Jen          |Finance   |3900  |
|Jeff         |Marketing |3000  |
|Kumar        |Marketing |2000  |
|Saif         |Sales     |4100  |
+-------------+----------+------+


Command took 1.98 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:44:54 AM on
PySpark_SQL_Practical_4
```

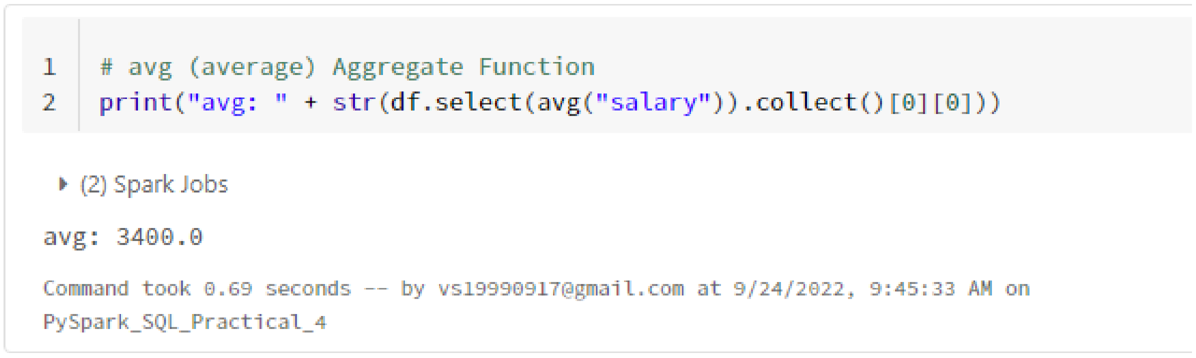
# Calculate approx_count_distinct
# In PySpark approx_count_distinct() function returns the count of distinct items in a group.
Code:-print("approx_count_distinct: " + \
    str(df.select(approx_count_distinct("salary")).collect()[0][0]))



```
# approx_count_distinct Aggregate Function
print("approx_count_distinct: " + \
      str(df.select(approx_count_distinct("salary")).collect()[0][0]))
```

```
approx_count_distinct: 6

Command took 1.85 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:45:02 AM on
PySpark_SQL_Practical_4
```

# Calculate average
# avg() function returns the average of values in the input column.
Code:-print("avg: " + str(df.select(avg("salary")).collect()[0][0]))

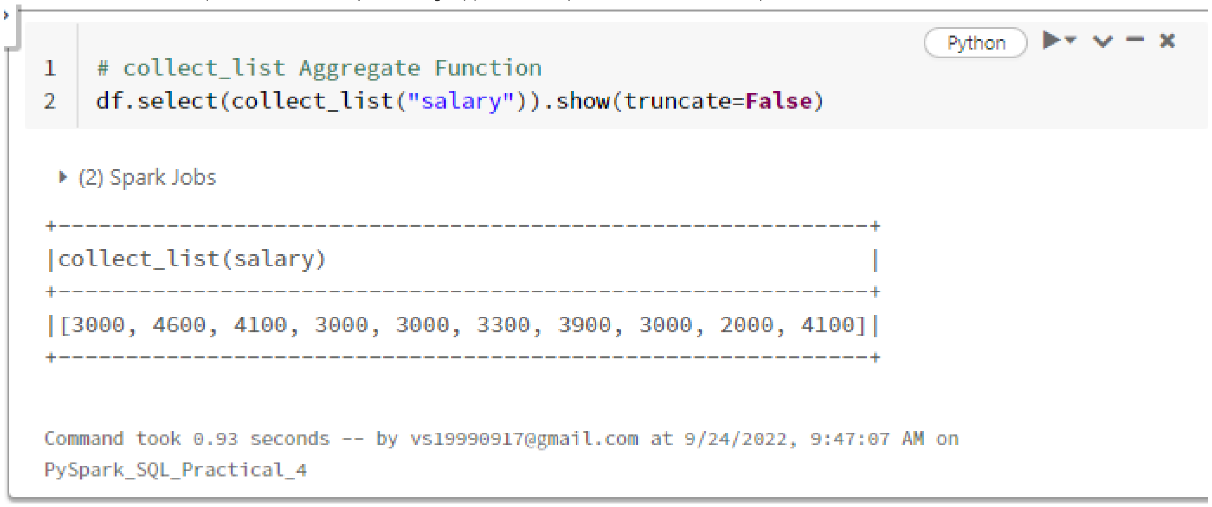


```
# avg (average) Aggregate Function
print("avg: " + str(df.select(avg("salary")).collect()[0][0]))
```

(2) Spark Jobs

```
avg: 3400.0

Command took 0.69 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:45:33 AM on
PySpark_SQL_Practical_4
```

# **Calculate list**
#collect_list() function returns all values from an input column with duplicates.
Code:-df.select(collect_list("salary")).show(truncate=False)

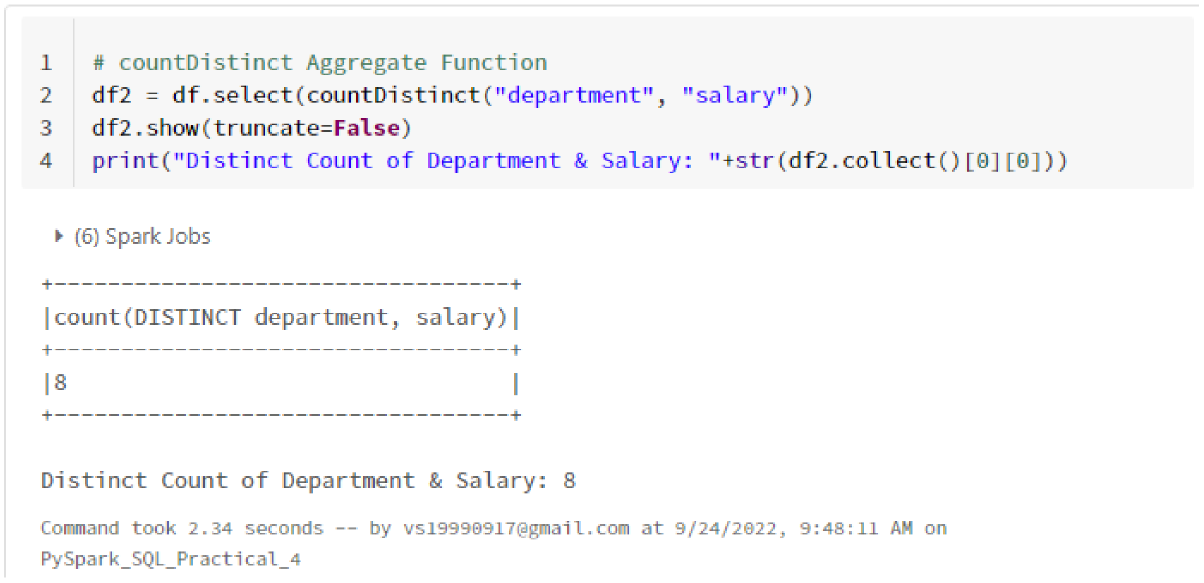


```
# collect_list Aggregate Function
df.select(collect_list("salary")).show(truncate=False)
```

(2) Spark Jobs

```
+--------------------------------------------------------------+
|collect_list(salary)                                          |
+--------------------------------------------------------------+
|[3000, 4600, 4100, 3000, 3000, 3300, 3900, 3000, 2000, 4100]|
+--------------------------------------------------------------+


Command took 0.93 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:47:07 AM on
PySpark_SQL_Practical_4
```

#Calculate countDistinct
#countDistinct() function returns the number of distinct elements in a columns
Code:-df2 = df.select(countDistinct("department", "salary"))
df2.show(truncate=False)
print("Distinct Count of Department & Salary: "+str(df2.collect()[0][0]))



```
# countDistinct Aggregate Function
df2 = df.select(countDistinct("department", "salary"))
df2.show(truncate=False)
print("Distinct Count of Department & Salary: "+str(df2.collect()[0][0]))
```

▸ (6) Spark Jobs

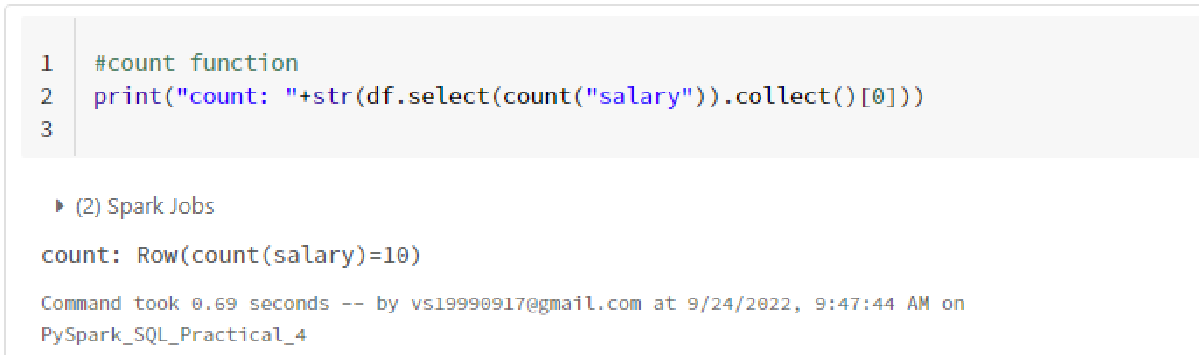
```
+----------------------------------+
|count(DISTINCT department, salary)|
+----------------------------------+
|8                                 |
+----------------------------------+

Distinct Count of Department & Salary: 8
```

Command took 2.34 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:48:11 AM on PySpark_SQL_Practical_4

#Show columns

# count() function returns number of elements in a column.

Code:-print("count: "+str(df.select(count("salary")).collect()[0]))



```
#count function
print("count: "+str(df.select(count("salary")).collect()[0]))
```

▸ (2) Spark Jobs

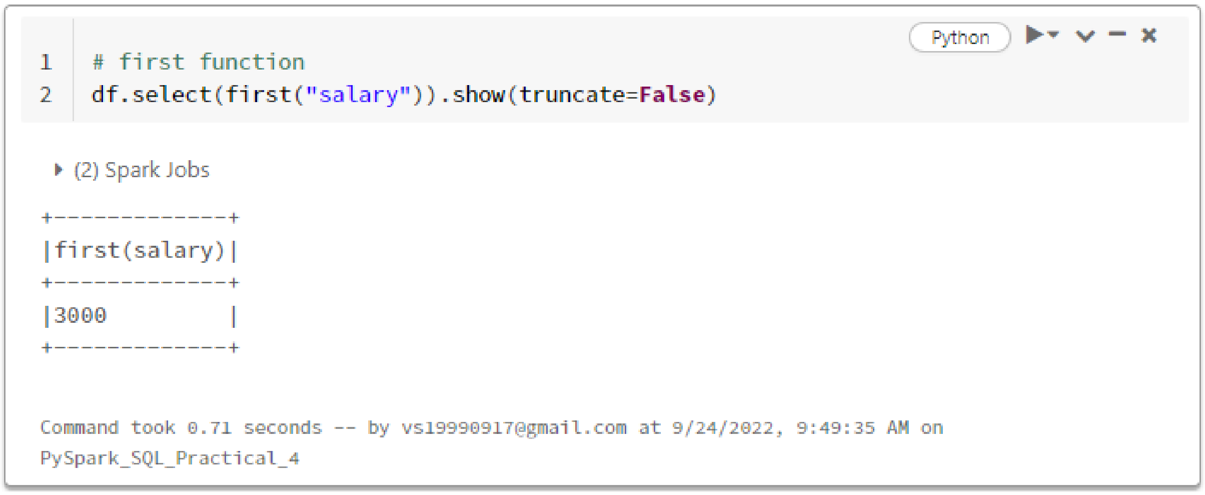
```
count: Row(count(salary)=10)
```

Command took 0.69 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:47:44 AM on PySpark_SQL_Practical_4

#Show First Element in Column

# first() function returns the first element in a column when ignoreNulls is set to true, it returns the first non-null element.

Code:-df.select(first("salary")).show(truncate=False)



```
# first function
df.select(first("salary")).show(truncate=False)
```

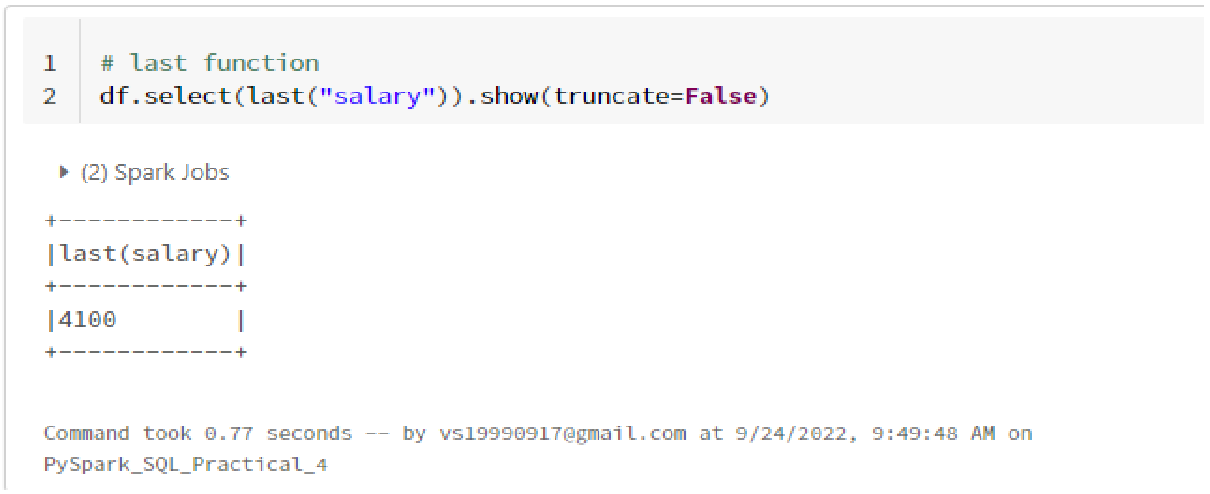
```
+-------------+
|first(salary)|
+-------------+
|3000         |
+-------------+
```

```
Command took 0.71 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:49:35 AM on
PySpark_SQL_Practical_4
```

#Show Last Element in Column

# last() function returns the last element in a column. when ignoreNulls is set to true, it returns the last non-null element.

Code:-df.select(last("salary")).show(truncate=False)



```
# last function
df.select(last("salary")).show(truncate=False)
```

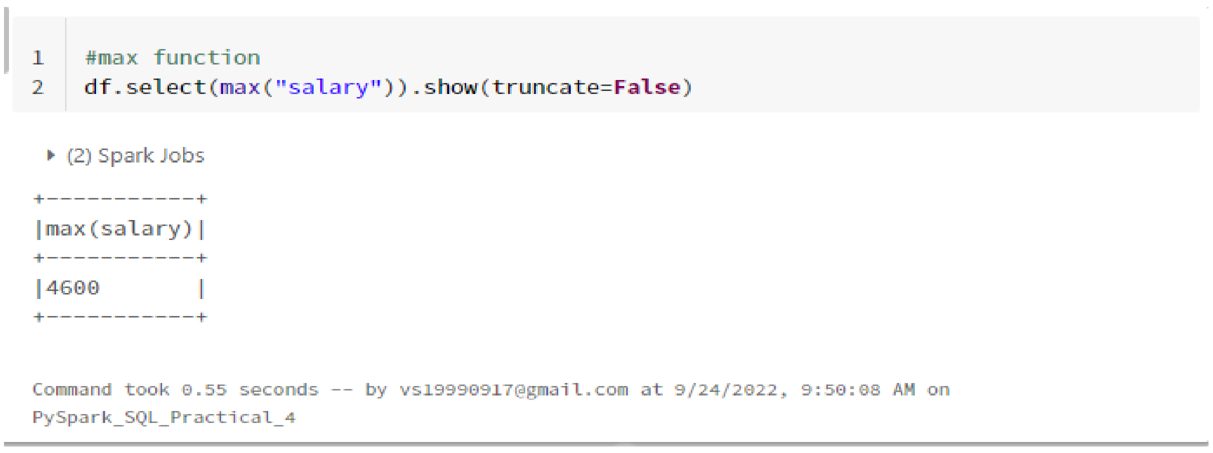
```
+------------+
|last(salary)|
+------------+
|4100        |
+------------+
```

```
Command took 0.77 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:49:48 AM on
PySpark_SQL_Practical_4
```

#Show maximum value in a column.

# max() function returns the maximum value in a column.

Code:-df.select(max("salary")).show(truncate=False)



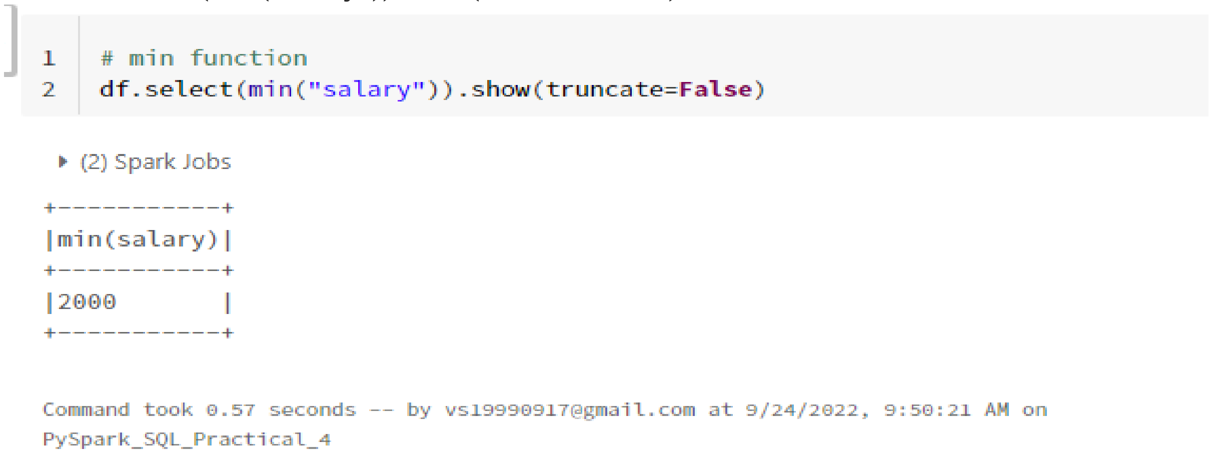
```
#max function
df.select(max("salary")).show(truncate=False)
```

▸ (2) Spark Jobs

```
+-----------+
|max(salary)|
+-----------+
|4600       |
+-----------+
```

```
Command took 0.55 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:50:08 AM on
PySpark_SQL_Practical_4
```

#Show minimum value in a column.
min() function
Code:-df.select(min("salary")).show(truncate=False)



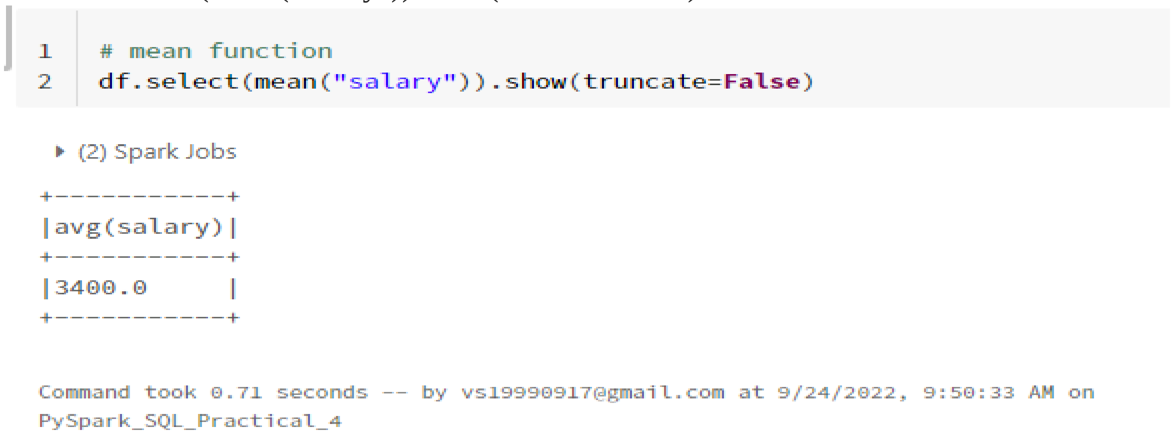
```
# min function
df.select(min("salary")).show(truncate=False)
```

▸ (2) Spark Jobs

```
+-----------+
|min(salary)|
+-----------+
|2000       |
+-----------+
```

```
Command took 0.57 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:50:21 AM on
PySpark_SQL_Practical_4
```

#Show mean value in a column.
# mean() function returns the average of the values in a column. Alias for Avg
Code:-df.select(mean("salary")).show(truncate=False)



```
# mean function
df.select(mean("salary")).show(truncate=False)
```

▸ (2) Spark Jobs

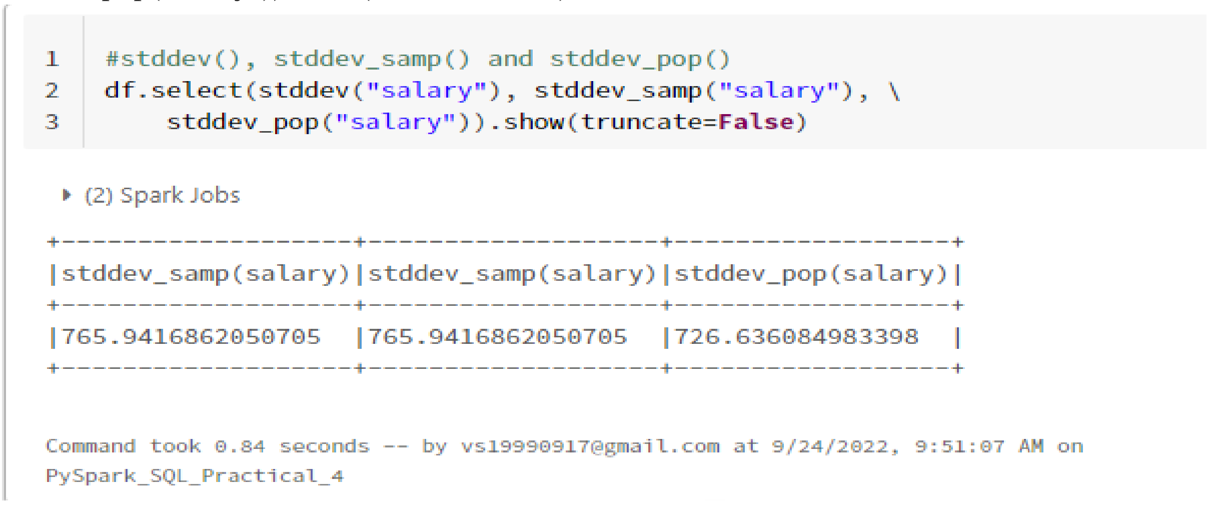
```
+-----------+
|avg(salary)|
+-----------+
|3400.0     |
+-----------+
```

```
Command took 0.71 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:50:33 AM on
PySpark_SQL_Practical_4
```

#Calculate Standard Deviation

# stddev() alias for stddev_samp.
# stddev_samp() function returns the sample standard deviation of values in a column.
# stddev_pop() function returns the population standard deviation of the values in a column.

Code:- df.select(stddev("salary"), stddev_samp("salary"),
stddev_pop("salary")).show(truncate=False)



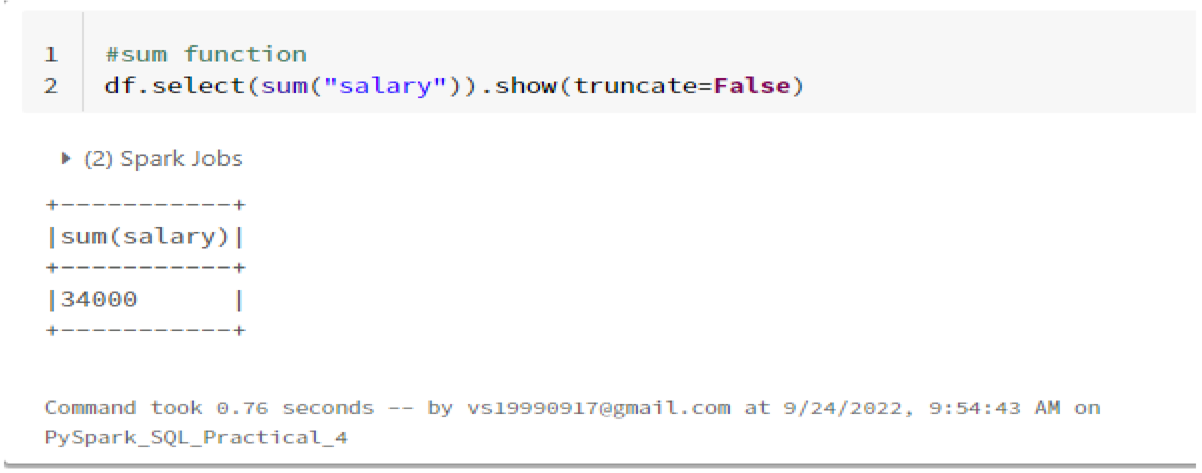
```
#stddev(), stddev_samp() and stddev_pop()
df.select(stddev("salary"), stddev_samp("salary"), \
    stddev_pop("salary")).show(truncate=False)
```

▸ (2) Spark Jobs

```
+-------------------+-------------------+------------------+
|stddev_samp(salary)|stddev_samp(salary)|stddev_pop(salary)|
+-------------------+-------------------+------------------+
|765.9416862050705  |765.9416862050705  |726.636084983398  |
+-------------------+-------------------+------------------+
```

Command took 0.84 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:51:07 AM on PySpark_SQL_Practical_4

# Calculate sum of all values in a column.
# sum() function Returns the sum of all values in a column
Code:-df.select(sum("salary")).show(truncate=False)



```
#sum function
df.select(sum("salary")).show(truncate=False)
```

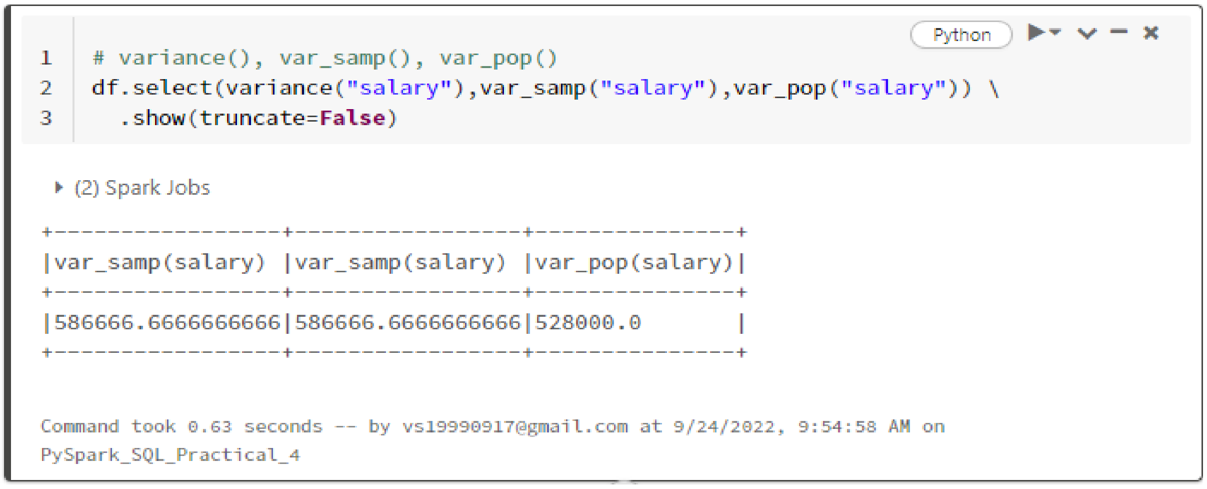
▸ (2) Spark Jobs

```
+-----------+
|sum(salary)|
+-----------+
|34000      |
+-----------+
```

Command took 0.76 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:54:43 AM on PySpark_SQL_Practical_4

# Calculate variance of the values in a column.
# variance() alias for var_samp
# var_samp() function returns the unbiased variance of the values in a column.
# var_pop() function returns the population variance of the values in a column

Code:- df.select(variance("salary"),var_samp("salary"),var_pop("salary"))
.show(truncate=False)



```
# variance(), var_samp(), var_pop()
df.select(variance("salary"),var_samp("salary"),var_pop("salary")) \
  .show(truncate=False)
```

(2) Spark Jobs

```
+-----------------+-----------------+---------------+
|var_samp(salary) |var_samp(salary) |var_pop(salary)|
+-----------------+-----------------+---------------+
|586666.6666666666|586666.6666666666|528000.0       |
+-----------------+-----------------+---------------+
```

```
Command took 0.63 seconds -- by vs19990917@gmail.com at 9/24/2022, 9:54:58 AM on
PySpark_SQL_Practical_4
```

Practical 5 - PySpark Mlib

a) **Basic Statistics** :the measure of central tendency and measure of dispersion.
i)Summary Statistics

check the summary using statistics technique.

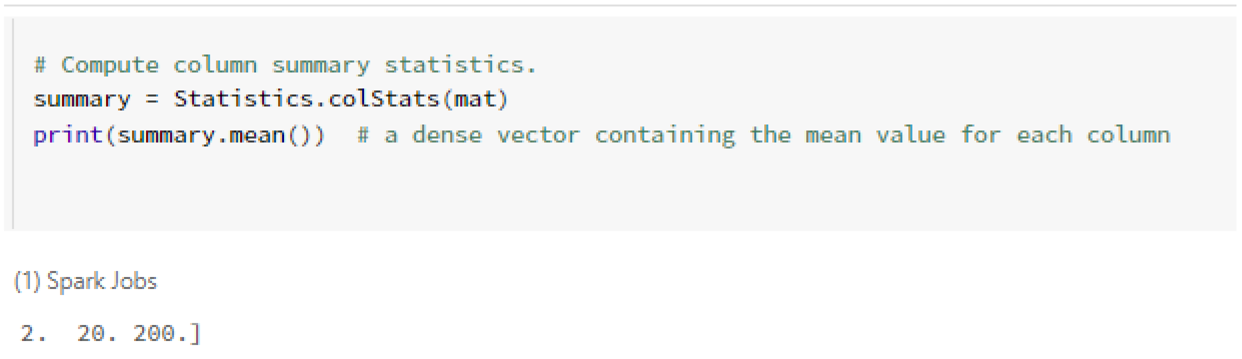
Code:

```
import numpy as np
from pyspark.mllib.stat import Statistics
mat = sc.parallelize(
   [np.array([1.0, 10.0, 100.0]), np.array([2.0, 20.0, 200.0]), np.array([3.0, 3 .0, 300.0])]
)  # an RDD of Vectors
```

1.Check column summary statistics.

```
summary = Statistics.colStats(mat)
print(summary.mean())  # a dense vector containing the mean value for each column
```

Output:



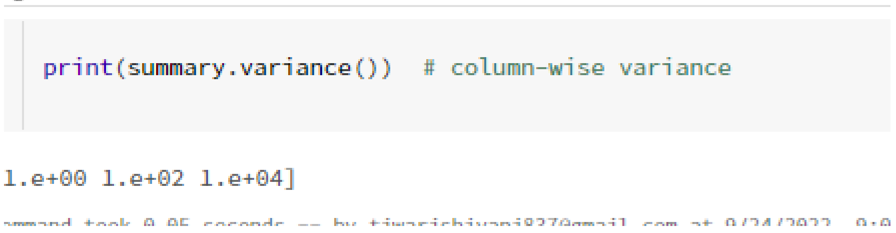
```
# Compute column summary statistics.
summary = Statistics.colStats(mat)
print(summary.mean())  # a dense vector containing the mean value for each column
```

(1) Spark Jobs

```
 2.   20. 200.]
```

2.check column-wise variance

```
print(summary.variance())
```

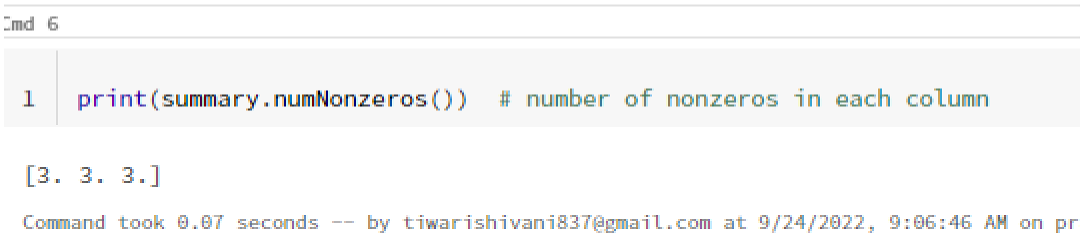


```
print(summary.variance())  # column-wise variance
```

```
1.e+00 1.e+02 1.e+04]
ommand took 0.05 seconds -- by tiwarishivani8270gmail.com at 9/24/2022, 9:0
```

3.count the number of nonzeros in each column

print(summary.numNonzeros())


```
Cmd 6
print(summary.numNonzeros())  # number of nonzeros in each column

[3. 3. 3.]

Command took 0.07 seconds -- by tiwarishivani837@gmail.com at 9/24/2022, 9:06:46 AM on pr
```

ii)Correlations:

Calculating the correlation between two series of data is a common operation in Statistics. In spark.mllib we provide the flexibility to calculate pairwise correlations among many series. The supported correlation methods are currently Pearson's and Spearman's correlation.

Find the correlation of data between two series.

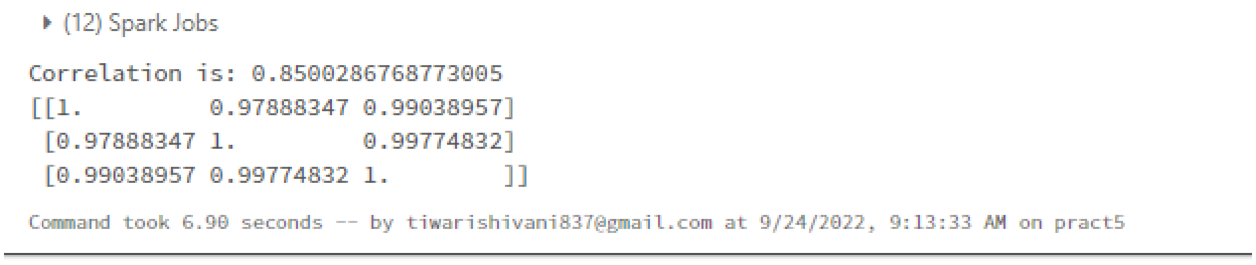
Code:

```
from pyspark.mllib.stat import Statistics
seriesX = sc.parallelize([1.0, 2.0, 3.0, 3.0, 5.0])  # a series
# seriesY must have the same number of partitions and cardinality as seriesX
seriesY = sc.parallelize([11.0, 22.0, 33.0, 33.0, 555.0])

# Compute the correlation using Pearson's method. Enter "spearman" for Spearman's method.
# If a method is not specified, Pearson's method will be used by default.
print("Correlation is: " + str(Statistics.corr(seriesX, seriesY, method="pearson")))

data = sc.parallelize(
   [np.array([1.0, 10.0, 100.0]), np.array([2.0, 20.0, 200.0]), np.array([5.0, 33.0, 366.0])]
)  # an RDD of Vectors
```

# calculate the correlation matrix using Pearson's method. Use "spearman" for Spearman's method.
# If a method is not specified, Pearson's method will be used by default.

```
print(Statistics.corr(data, method="pearson"))
```

Output:



```
Correlation is: 0.8500286768773005
[[1.         0.97888347 0.99038957]
 [0.97888347 1.         0.99774832]
 [0.99038957 0.99774832 1.        ]]
Command took 6.90 seconds -- by tiwarishivani837@gmail.com at 9/24/2022, 9:13:33 AM on pract5
```

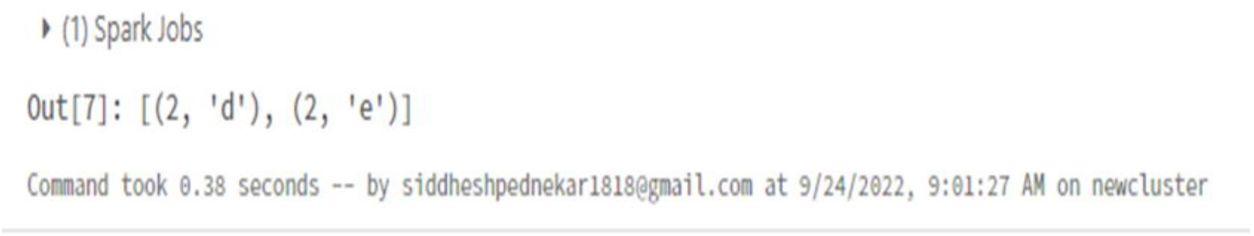
iii)Stratified Sampling : it generates randomly sampled data.

Find random sample of the given data using stratified sampling

code:

```
data = sc.parallelize([(1, 'a'), (1, 'b'), (2, 'c'), (2, 'd'), (2, 'e'), (3, 'f')])
 # specify the exact fraction desired from each key as a dictionary
fractions = {1: 0.1, 2: 0.6, 3: 0.3}
 approxSample = data.sampleByKey(False, fractions)
approxSample.collect()
```

 output:



```
Out[7]: [(2, 'd'), (2, 'e')]
Command took 0.38 seconds -- by siddheshpednekar1818@gmail.com at 9/24/2022, 9:01:27 AM on newcluster
```

iv)Hypothesis Testing:

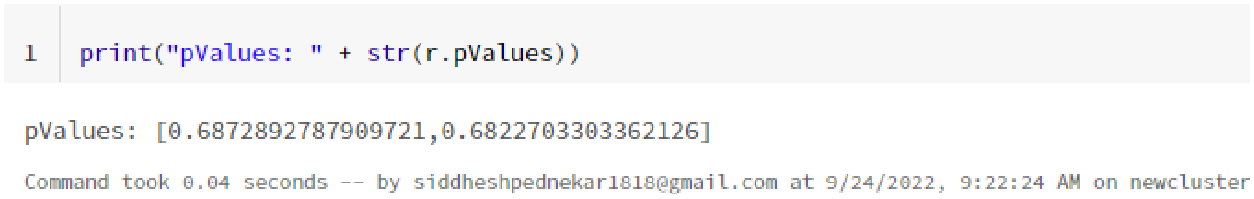
Hypothesis testing is a powerful tool in statistics to determine whether a result is statistically significant, whether this result occurred by chance or not. spark.ml currently supports Pearson's Chi-squared (χ2) tests for independence.

ChiSquareTest
ChiSquareTest conducts Pearson's independence test for every feature against the label. For each feature, the (feature, label) pairs are converted into a contingency matrix for which the Chi-squared statistic is computed. All label and feature values must be categorical.

1.examine the differences between categorical variables from a random sample using ChiSquare Test.
code:
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
.appName("ChiSquareTestExample") \
    .getOrCreate()
from pyspark.ml.linalg import Vectors
from pyspark.ml.stat import ChiSquareTest
data = [(0.0, Vectors.dense(0.5, 10.0)),
        (0.0, Vectors.dense(1.5, 20.0)),
        (1.0, Vectors.dense(1.5, 30.0)),
        (0.0, Vectors.dense(3.5, 30.0)),
        (0.0, Vectors.dense(3.5, 40.0)),
        (1.0, Vectors.dense(3.5, 40.0))]
df = spark.createDataFrame(data, ["label", "features"])
r = ChiSquareTest.test(df, "features", "label").head()

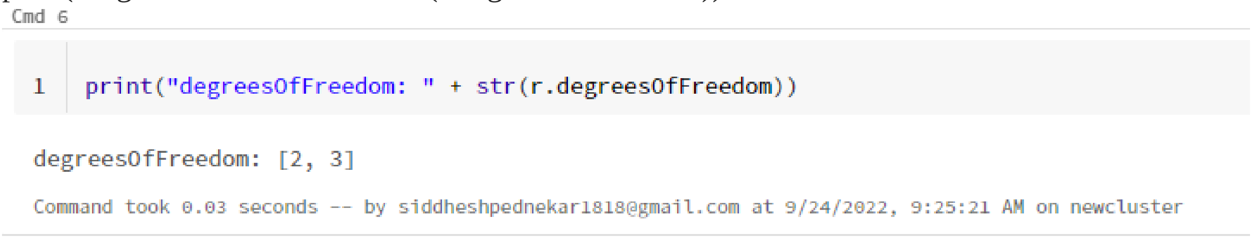
print("pValues: " + str(r.pValues))



```
print("pValues: " + str(r.pValues))
```

```
pValues: [0.6872892787909721,0.6822703303362126]

Command took 0.04 seconds -- by siddheshpednekar1818@gmail.com at 9/24/2022, 9:22:24 AM on newcluster
```

print("degreesOfFreedom: " + str(r.degreesOfFreedom))

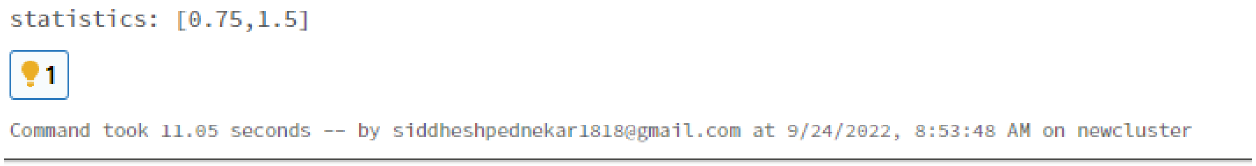


Cmd 6

```
print("degreesOfFreedom: " + str(r.degreesOfFreedom))
```

```
degreesOfFreedom: [2, 3]

Command took 0.03 seconds -- by siddheshpednekar1818@gmail.com at 9/24/2022, 9:25:21 AM on newcluster
```

print("statistics: " + str(r.statistics))



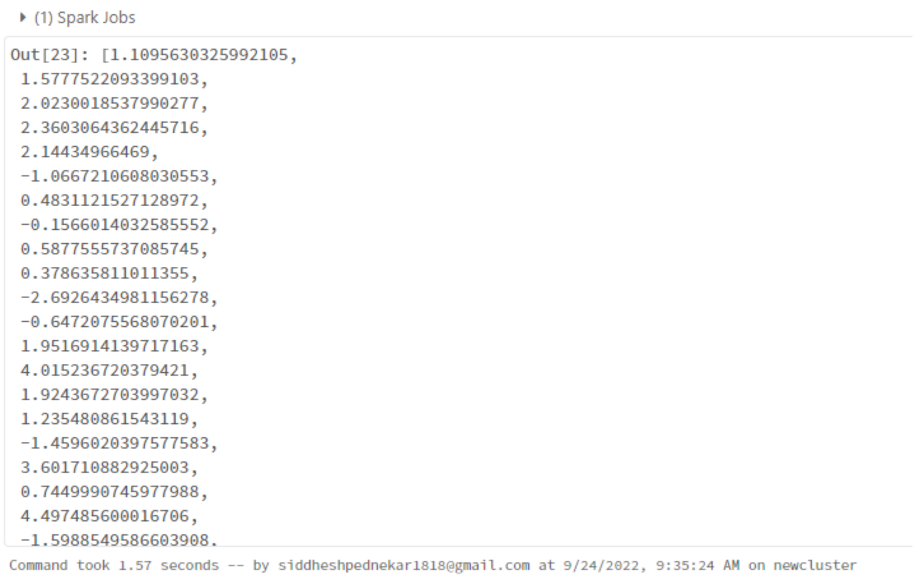
```
statistics: [0.75,1.5]

Command took 11.05 seconds -- by siddheshpednekar1818@gmail.com at 9/24/2022, 8:53:48 AM on newcluster
```

v)Random Data Generation : Random data generation is useful for randomized algorithms, prototyping, and performance testing. spark.mllib supports generating random RDDs with i.i.d. values drawn from a given distribution: uniform, standard normal, or Poisson.

Generate random data using standard normal distribution
code:
from pyspark.context import SparkContext
sc = SparkContext.getOrCreate()
from pyspark.mllib.random import RandomRDDs

```
# Generate a random double RDD that contains 1 million i.i.d. values drawn from the
# standard normal distribution `N(0, 1)`, evenly distributed in 10 partitions.
u = RandomRDDs.normalRDD(sc, 1000000, 10)
# Apply a transform to get a random double RDD following `N(1, 4)`.
v = u.map(lambda x: 1.0 + 2.0 * x)
v.collect()
```

output



```
▸ (1) Spark Jobs
Out[23]: [1.1095630325992105,
 1.5777522093399103,
 2.0230018537990277,
 2.3603064362445716,
 2.14434966469,
 -1.0667210608030553,
 0.4831121527128972,
 -0.1566014032585552,
 0.5877555737085745,
 0.378635811011355,
 -2.6926434981156278,
 -0.6472075568070201,
 1.9516914139717163,
 4.015236720379421,
 1.9243672703997032,
 1.235480861543119,
 -1.4596020397577583,
 3.601710882925003,
 0.7449990745977988,
 4.497485600016706,
 -1.5988549586603908.
Command took 1.57 seconds -- by siddheshpednekar1818@gmail.com at 9/24/2022, 9:35:24 AM on newcluster
```

b) Pipeline

Create text document pipeline on the given text data for tokenizing the text hashing term frequencies and building the logistic regression model for the prediction of a text.

Data list of documents with id,text,labels in tuples.

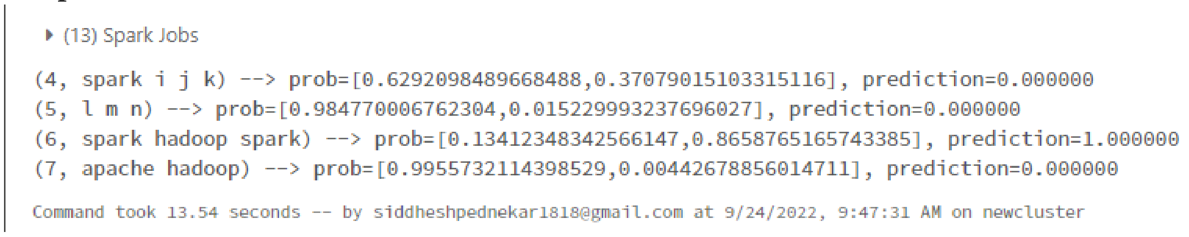
```
 ([(0, "a b c d e spark", 1.0),
    (1, "b d", 0.0),
    (2, "spark f g h", 1.0),
    (3, "hadoop mapreduce", 0.0)
```

Code:

```
from pyspark.ml import Pipeline
from pyspark.ml.classification import LogisticRegression
from pyspark.ml.feature import HashingTF, Tokenizer
# Prepare training documents from a list of (id, text, label) tuples.
training = spark.createDataFrame([
    (0, "a b c d e spark", 1.0),
    (1, "b d", 0.0),
    (2, "spark f g h", 1.0),
    (3, "hadoop mapreduce", 0.0)
], ["id", "text", "label"])
```

```
# Configure an ML pipeline, which consists of three stages: tokenizer, hashingTF, and lr.
tokenizer = Tokenizer(inputCol="text", outputCol="words")
hashingTF = HashingTF(inputCol=tokenizer.getOutputCol(), outputCol="features")
lr = LogisticRegression(maxIter=10, regParam=0.001)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])
# Fit the pipeline to training documents.
model = pipeline.fit(training)

# Prepare test documents, which are unlabeled (id, text) tuples.
test = spark.createDataFrame([
    (4, "spark i j k"),
    (5, "l m n"),
    (6, "spark hadoop spark"),
    (7, "apache hadoop")
], ["id", "text"])
# Make predictions on test documents and print columns of interest.
prediction = model.transform(test)
selected = prediction.select("id", "text", "probability", "prediction")
for row in selected.collect():
    rid, text, prob, prediction = row
    print(
        "(%d, %s) --> prob=%s, prediction=%f" % (
            rid, text, str(prob), prediction   # type: ignore
        )
    )
```

Output:



▸ (13) Spark Jobs

```
(4, spark i j k) --> prob=[0.6292098489668488,0.37079015103315116], prediction=0.000000
(5, l m n) --> prob=[0.984770006762304,0.015229993237696027], prediction=0.000000
(6, spark hadoop spark) --> prob=[0.13412348342566147,0.8658765165743385], prediction=1.000000
(7, apache hadoop) --> prob=[0.9955732114398529,0.00442678856014711], prediction=0.000000

Command took 13.54 seconds -- by siddheshpednekar1818@gmail.com at 9/24/2022, 9:47:31 AM on newcluster
```

Practical 6 - PySpark MLib: Feature extraction, transforming and selecting features

Spark MLlib is a short form of spark machine-learning library. Pyspark MLlib is a wrapper over PySpark Core to do data analysis using machine-learning algorithms. It works on distributed systems and is scalable. We can find implementations of classification, clustering, linear regression, and other machine-learning algorithms in PySpark MLlib.

MLlib is Spark’s scalable machine learning library consisting of common machine learning algorithms and utilities, including classification, regression, clustering, collaborative filtering, and dimensionality reduction, as well as underlying optimization primitives.

Feature Extractors

(Extraction: Extracting features from“raw”data)

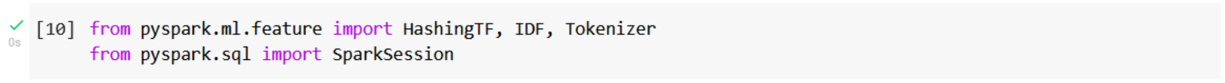
1. TF-IDF
2. Word2Vec
3. CountVectorizer

1)TF-IDF

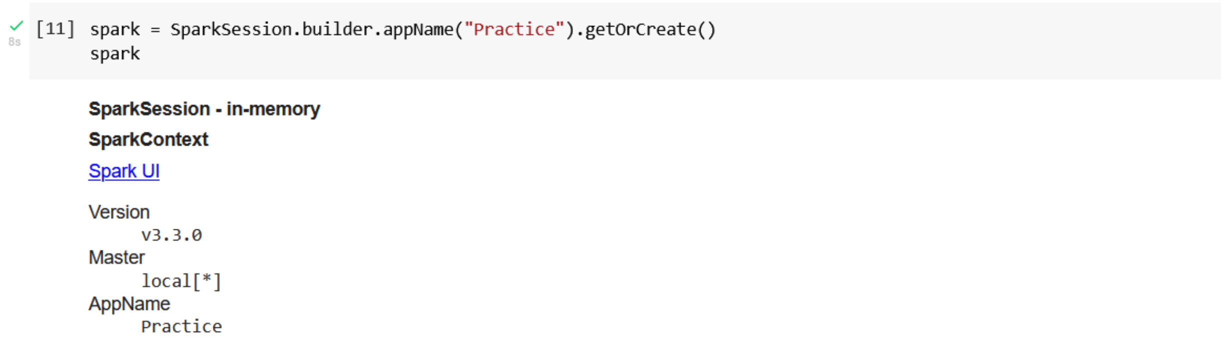
Term frequency-inverse document frequency (TF-IDF) is a feature vectorization method widely used in text mining to reflect the importance of a term to a document in the corpus.
TF: Both HashingTF and CountVectorizer can be used to generate the term frequency vectors.

Example

In the following code segment, we start with a set of sentences. We split each sentence into words using Tokenizer. For each sentence (bag of words), we use HashingTF to hash the sentence into a feature vector. We use IDF to rescale the feature vectors; this generally improves performance when using text as features. Our feature vectors could then be passed to a learning algorithm.

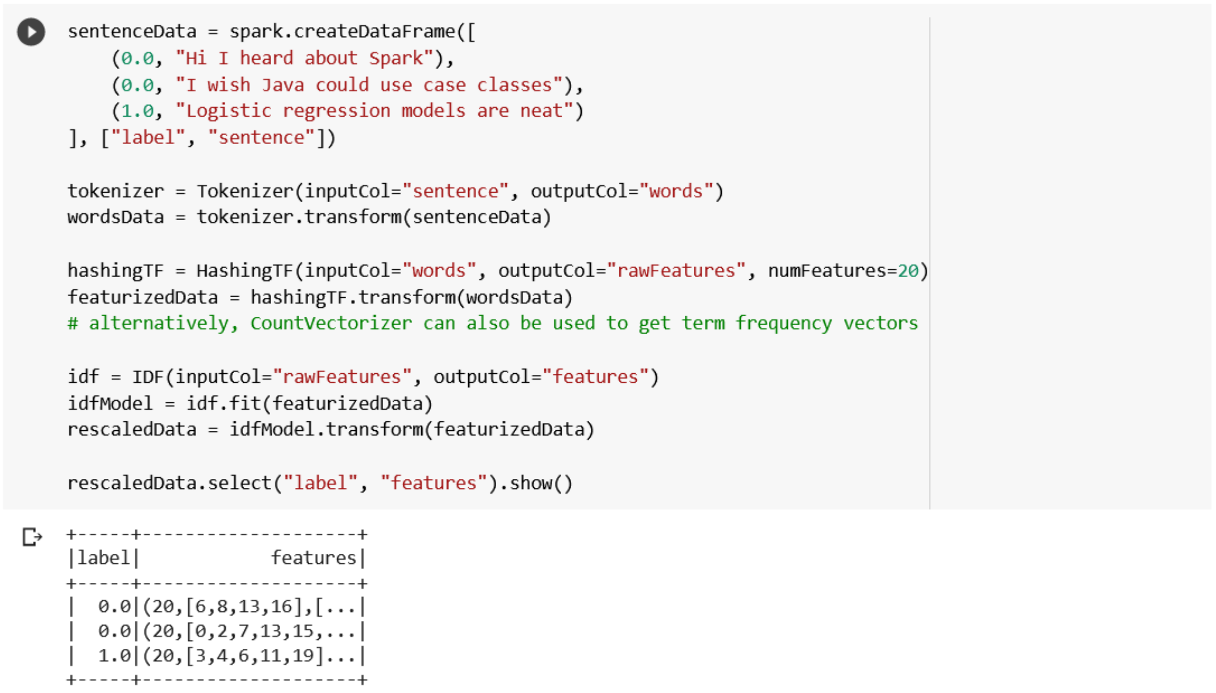


```
[10] from pyspark.ml.feature import HashingTF, IDF, Tokenizer
     from pyspark.sql import SparkSession
```



```
[11] spark = SparkSession.builder.appName("Practice").getOrCreate()
     spark
```

```
SparkSession - in-memory
SparkContext
Spark UI

Version
      v3.3.0
Master
      local[*]
AppName
      Practice
```

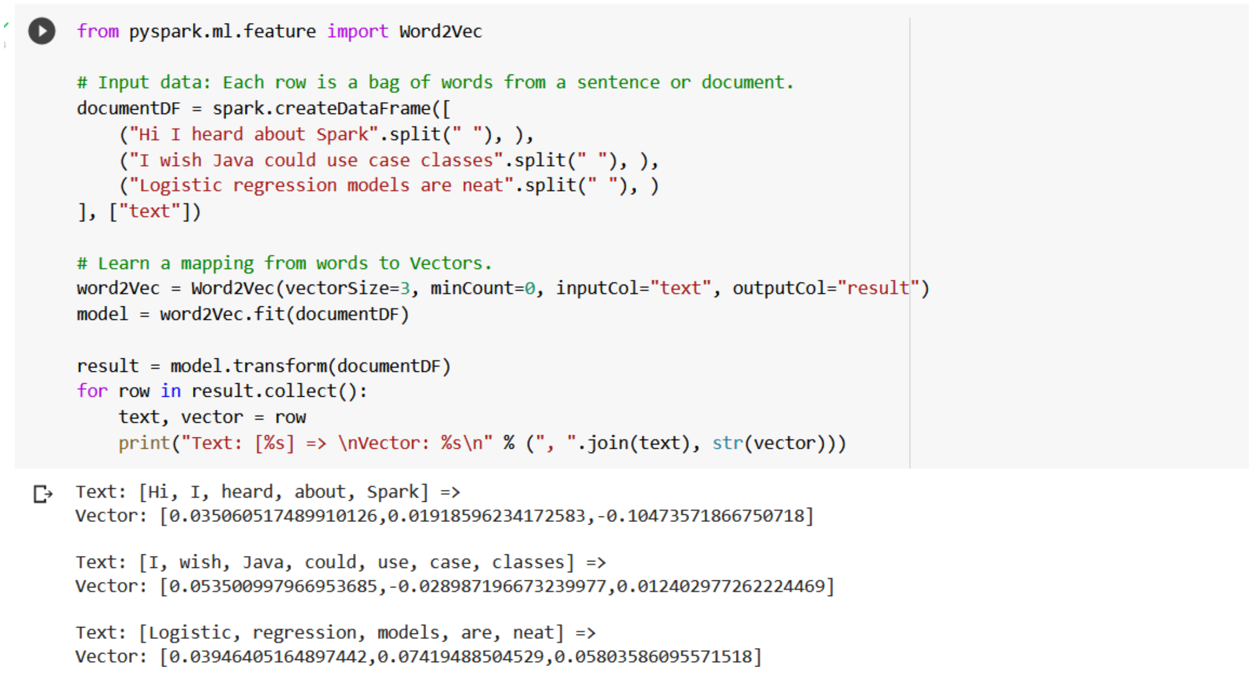


```
sentenceData = spark.createDataFrame([
    (0.0, "Hi I heard about Spark"),
    (0.0, "I wish Java could use case classes"),
    (1.0, "Logistic regression models are neat")
], ["label", "sentence"])

tokenizer = Tokenizer(inputCol="sentence", outputCol="words")
wordsData = tokenizer.transform(sentenceData)

hashingTF = HashingTF(inputCol="words", outputCol="rawFeatures", numFeatures=20)
featurizedData = hashingTF.transform(wordsData)
# alternatively, CountVectorizer can also be used to get term frequency vectors

idf = IDF(inputCol="rawFeatures", outputCol="features")
idfModel = idf.fit(featurizedData)
rescaledData = idfModel.transform(featurizedData)

rescaledData.select("label", "features").show()
```

```
+-----+--------------------+
|label|            features|
+-----+--------------------+
|  0.0|(20,[6,8,13,16],[...|
|  0.0|(20,[0,2,7,13,15,...|
|  1.0|(20,[3,4,6,11,19]...|
+-----+--------------------+
```

2)Word2Vec

Word2Vec is an Estimator which takes sequences of words representing documents and trains a Word2VecModel. The model maps each word to a unique fixed-size vector. The Word2VecModel transforms each document into a vector using the average of all words in the document; this vector can then be used as features for prediction, document similarity calculations, etc.



```
from pyspark.ml.feature import Word2Vec

# Input data: Each row is a bag of words from a sentence or document.
documentDF = spark.createDataFrame([
    ("Hi I heard about Spark".split(" "), ),
    ("I wish Java could use case classes".split(" "), ),
    ("Logistic regression models are neat".split(" "), )
], ["text"])

# Learn a mapping from words to Vectors.
word2Vec = Word2Vec(vectorSize=3, minCount=0, inputCol="text", outputCol="result")
model = word2Vec.fit(documentDF)

result = model.transform(documentDF)
for row in result.collect():
    text, vector = row
    print("Text: [%s] => \nVector: %s\n" % (", ".join(text), str(vector)))
```

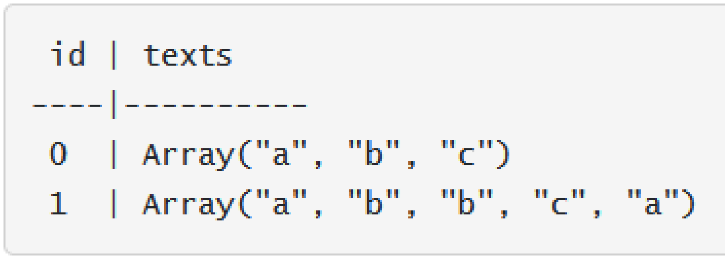
```
Text: [Hi, I, heard, about, Spark] =>
Vector: [0.035060517489910126,0.01918596234172583,-0.10473571866750718]

Text: [I, wish, Java, could, use, case, classes] =>
Vector: [0.053500997966953685,-0.028987196673239977,0.012402977262224469]

Text: [Logistic, regression, models, are, neat] =>
Vector: [0.03946405164897442,0.07419488504529,0.05803586095571518]
```

3)CountVectorizer

CountVectorizer and CountVectorizerModel aim to help convert a collection of text documents to vectors of token counts. When an a-priori dictionary is not available, CountVectorizer can be used as an Estimator to extract the vocabulary, and generates a CountVectorizerModel. The model produces sparse representations for the documents over the vocabulary, which can then be passed to other algorithms like LDA.

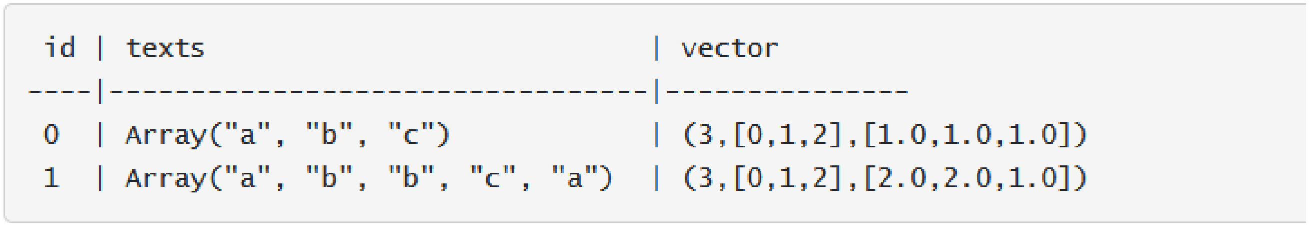
During the fitting process, CountVectorizer will select the top vocabSize words ordered by term frequency across the corpus. An optional parameter minDF also affects the fitting process by specifying the minimum number (or fraction if < 1.0) of documents a term must appear in to be included in the vocabulary. Another optional binary toggle parameter controls the output vector. If set to true all nonzero counts are set to 1. This is especially useful for discrete probabilistic models that model binary, rather than integer, counts.

Example
Assume that we have the following DataFrame with columns id and texts:



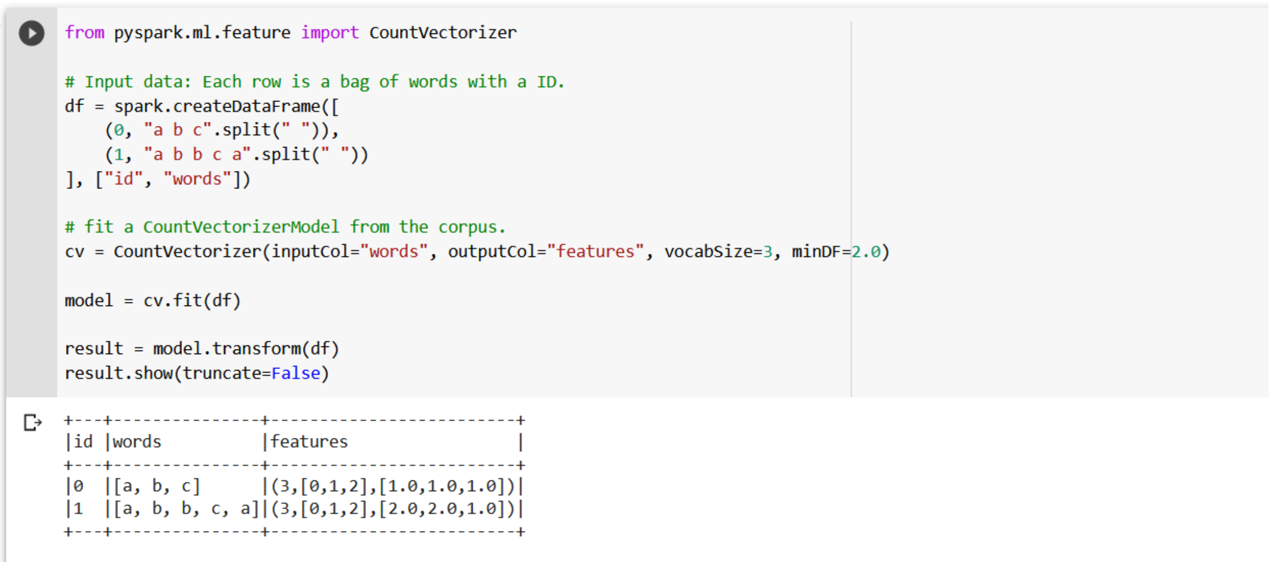
```
 id | texts
----|----------
 0  | Array("a", "b", "c")
 1  | Array("a", "b", "b", "c", "a")
```

each row in texts is a document of type Array[String]. Invoking fit of CountVectorizer produces a CountVectorizerModel with vocabulary (a, b, c). Then the output column “vector” after transformation contains:



```
 id | texts                           | vector
----|---------------------------------|---------------
 0  | Array("a", "b", "c")            | (3,[0,1,2],[1.0,1.0,1.0])
 1  | Array("a", "b", "b", "c", "a")  | (3,[0,1,2],[2.0,2.0,1.0])
```

Each vector represents the token counts of the document over the vocabulary.


```
from pyspark.ml.feature import CountVectorizer

# Input data: Each row is a bag of words with a ID.
df = spark.createDataFrame([
    (0, "a b c".split(" ")),
    (1, "a b b c a".split(" "))
], ["id", "words"])

# fit a CountVectorizerModel from the corpus.
cv = CountVectorizer(inputCol="words", outputCol="features", vocabSize=3, minDF=2.0)

model = cv.fit(df)

result = model.transform(df)
result.show(truncate=False)
```

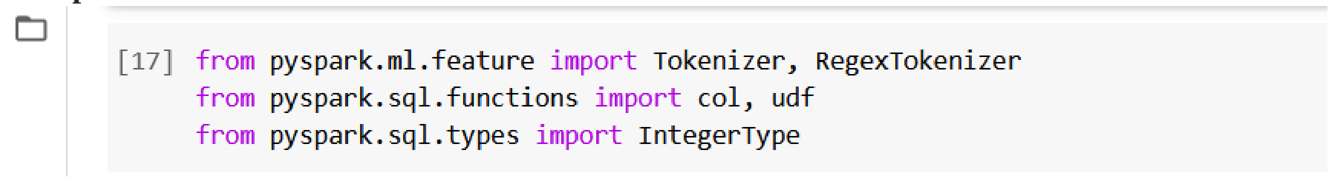
```
+---+---------------+-------------------------+
|id |words          |features                 |
+---+---------------+-------------------------+
|0  |[a, b, c]      |(3,[0,1,2],[1.0,1.0,1.0])|
|1  |[a, b, b, c, a]|(3,[0,1,2],[2.0,2.0,1.0])|
+---+---------------+-------------------------+
```

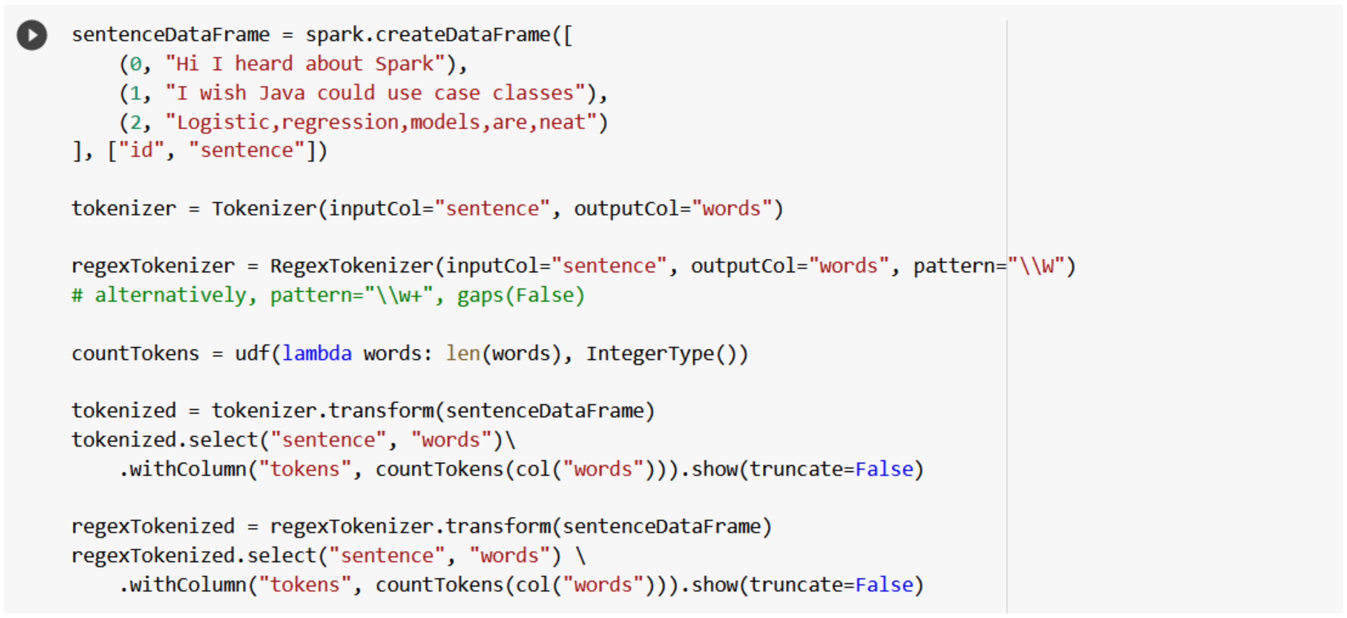
Feature Transformers

Tokenization is the process of taking text (such as a sentence) and breaking it into individual terms (usually words). A simple Tokenizer class provides this functionality. The example below shows how to split sentences into sequences of words.

RegexTokenizer allows more advanced tokenization based on regular expression (regex) matching. By default, the parameter "pattern" (regex, default: "\\s+") is used as delimiters to split the input text. Alternatively, users can set parameter "gaps" to false indicating the regex "pattern" denotes "tokens" rather than splitting gaps, and find all matching occurrences as the tokenization result.

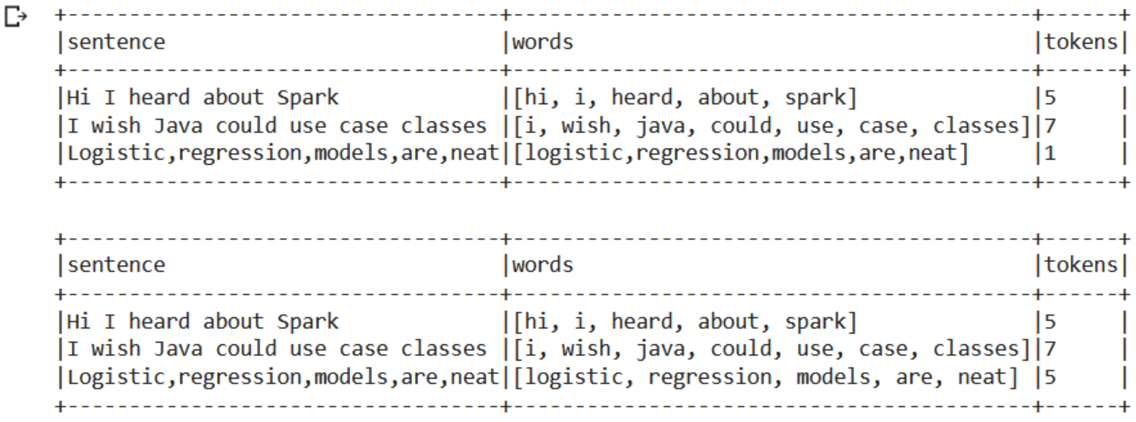
Example


```
[17] from pyspark.ml.feature import Tokenizer, RegexTokenizer
     from pyspark.sql.functions import col, udf
     from pyspark.sql.types import IntegerType
```



```python
sentenceDataFrame = spark.createDataFrame([
    (0, "Hi I heard about Spark"),
    (1, "I wish Java could use case classes"),
    (2, "Logistic,regression,models,are,neat")
], ["id", "sentence"])

tokenizer = Tokenizer(inputCol="sentence", outputCol="words")

regexTokenizer = RegexTokenizer(inputCol="sentence", outputCol="words", pattern="\\W")
# alternatively, pattern="\\w+", gaps(False)

countTokens = udf(lambda words: len(words), IntegerType())

tokenized = tokenizer.transform(sentenceDataFrame)
tokenized.select("sentence", "words")\
    .withColumn("tokens", countTokens(col("words"))).show(truncate=False)

regexTokenized = regexTokenizer.transform(sentenceDataFrame)
regexTokenized.select("sentence", "words") \
    .withColumn("tokens", countTokens(col("words"))).show(truncate=False)
```

OUTPUT:



```
+-----------------------------------+------------------------------------------+------+
|sentence                           |words                                     |tokens|
+-----------------------------------+------------------------------------------+------+
|Hi I heard about Spark             |[hi, i, heard, about, spark]              |5     |
|I wish Java could use case classes |[i, wish, java, could, use, case, classes]|7     |
|Logistic,regression,models,are,neat|[logistic,regression,models,are,neat]     |1     |
+-----------------------------------+------------------------------------------+------+

+-----------------------------------+------------------------------------------+------+
|sentence                           |words                                     |tokens|
+-----------------------------------+------------------------------------------+------+
|Hi I heard about Spark             |[hi, i, heard, about, spark]              |5     |
|I wish Java could use case classes |[i, wish, java, could, use, case, classes]|7     |
|Logistic,regression,models,are,neat|[logistic, regression, models, are, neat] |5     |
+-----------------------------------+------------------------------------------+------+
```

StopWordsRemover

Stop words are words which should be excluded from the input, typically because the words appear frequently and don’t carry as much meaning.
StopWordsRemover takes as input a sequence of strings (e.g. the output of a Tokenizer) and drops all the stop words from the input sequences. The list of stopwords is specified by the stopWords parameter. Default stop words for some languages are accessible by calling StopWordsRemover.loadDefaultStopWords(language), for which available options are “danish”, “dutch”, “english”, “finnish”, “french”, “german”, “hungarian”, “italian”, “norwegian”, “portuguese”, “russian”, “spanish”, “swedish” and “turkish”. A boolean parameter caseSensitive indicates if the matches should be case sensitive (false by default).

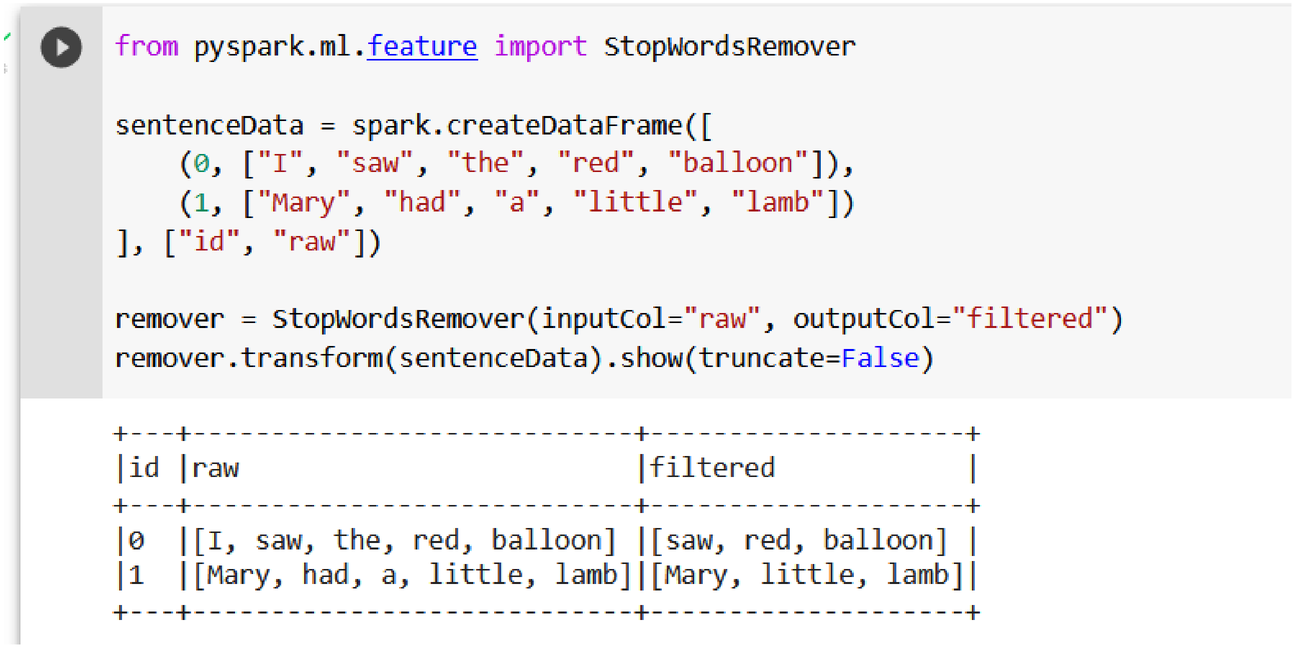
Example

Assume that we have the following DataFrame with columns id and raw:

```
 id | raw
----|----------
 0  | [I, saw, the, red, balloon]
 1  | [Mary, had, a, little, lamb]
```

Applying StopWordsRemover with raw as the input column and filtered as the output column, we should get the following:

```
 id | raw                         | filtered
----|-----------------------------|--------------------
 0  | [I, saw, the, red, balloon]  |  [saw, red, balloon]
 1  | [Mary, had, a, little, lamb]|[Mary, little, lamb]
```



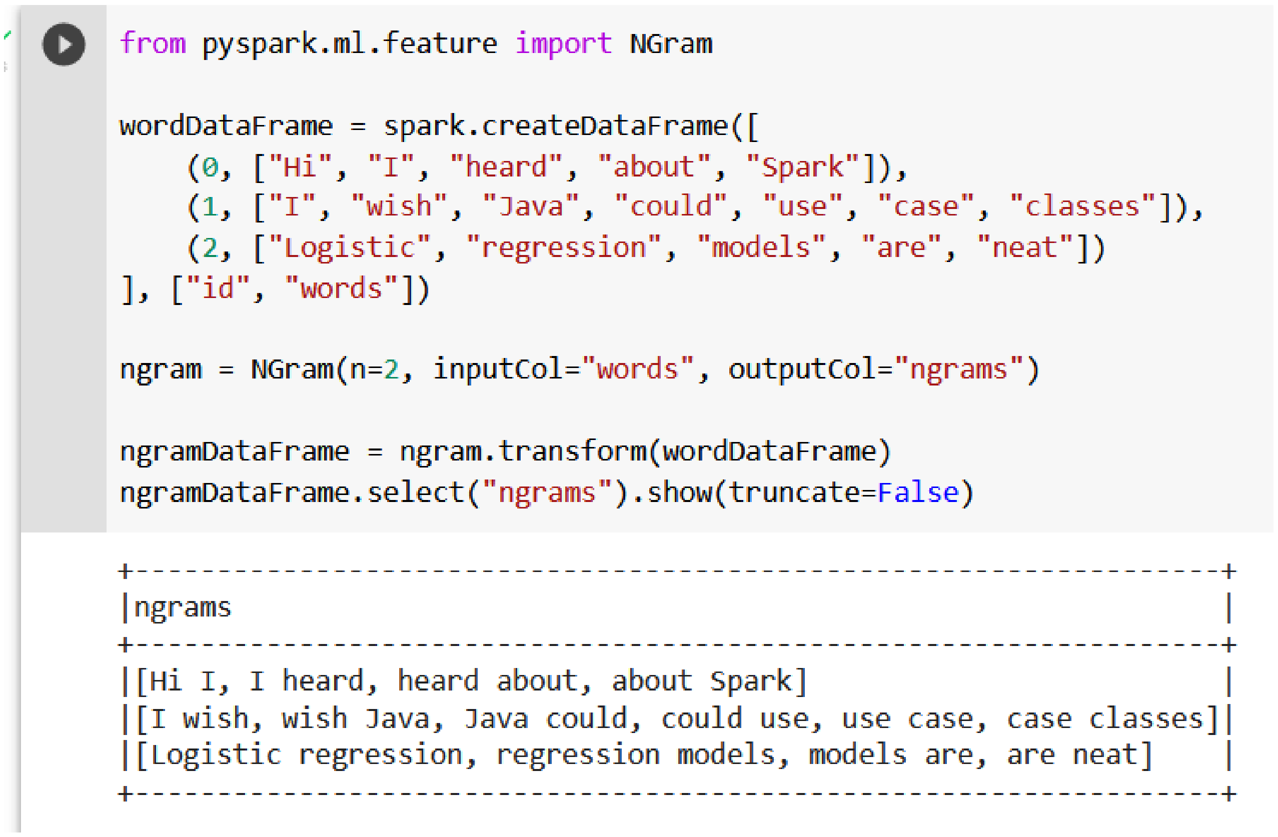
```
from pyspark.ml.feature import StopWordsRemover

sentenceData = spark.createDataFrame([
    (0, ["I", "saw", "the", "red", "balloon"]),
    (1, ["Mary", "had", "a", "little", "lamb"])
], ["id", "raw"])

remover = StopWordsRemover(inputCol="raw", outputCol="filtered")
remover.transform(sentenceData).show(truncate=False)
```

```
+---+----------------------------+--------------------+
|id |raw                         |filtered            |
+---+----------------------------+--------------------+
|0  |[I, saw, the, red, balloon] |[saw, red, balloon] |
|1  |[Mary, had, a, little, lamb]|[Mary, little, lamb]|
+---+----------------------------+--------------------+
```

***n*-gram**

An n-gram is a sequence of n tokens (typically words) for some integern.The NGram class can be used to transform input features into n-grams.NGram takes as input a sequence of strings (e.g. the output of a Tokenizer). The parameter n is used to determine the number of terms in each n-gram. The output will consist of a sequence of n -grams where each n-gram is represented by a space-delimited string of *n* consecutive words. If the input sequence contains fewer than n strings, no output is produced.

Example



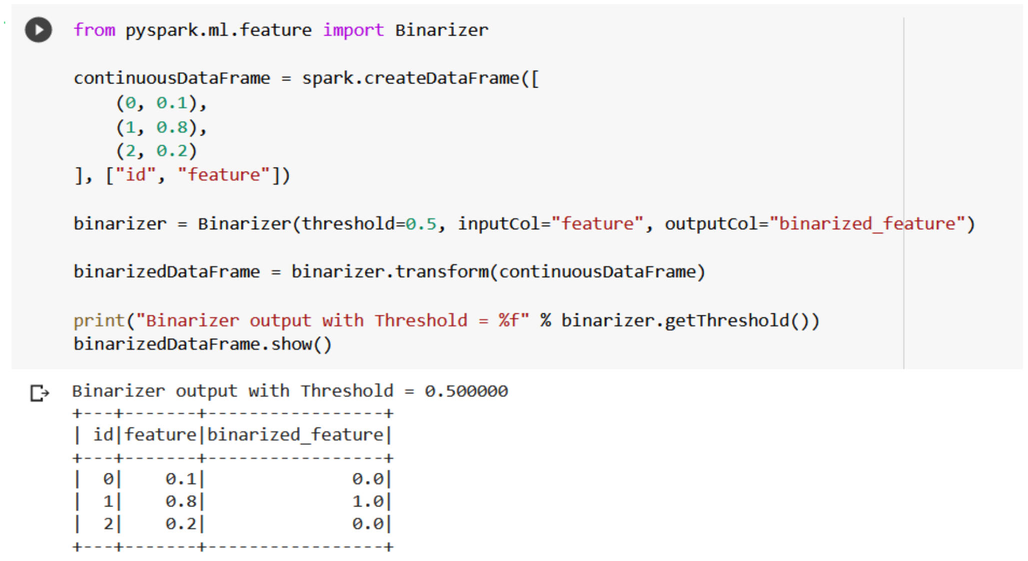
```
from pyspark.ml.feature import NGram

wordDataFrame = spark.createDataFrame([
    (0, ["Hi", "I", "heard", "about", "Spark"]),
    (1, ["I", "wish", "Java", "could", "use", "case", "classes"]),
    (2, ["Logistic", "regression", "models", "are", "neat"])
], ["id", "words"])

ngram = NGram(n=2, inputCol="words", outputCol="ngrams")

ngramDataFrame = ngram.transform(wordDataFrame)
ngramDataFrame.select("ngrams").show(truncate=False)
```

```
+------------------------------------------------------------------+
|ngrams                                                            |
+------------------------------------------------------------------+
|[Hi I, I heard, heard about, about Spark]                         |
|[I wish, wish Java, Java could, could use, use case, case classes]|
|[Logistic regression, regression models, models are, are neat]   |
+------------------------------------------------------------------+
```

Binarizer

Binarization is the process of thresholding numerical features to binary (0/1) features.Binarizer takes the common parameters inputCol and outputCol, as well as the threshold for binarization. Feature values greater than the threshold are binarized to 1.0; values equal to or less than the threshold are binarized to 0.0. Both Vector and Double types are supported for inputCol.

Example



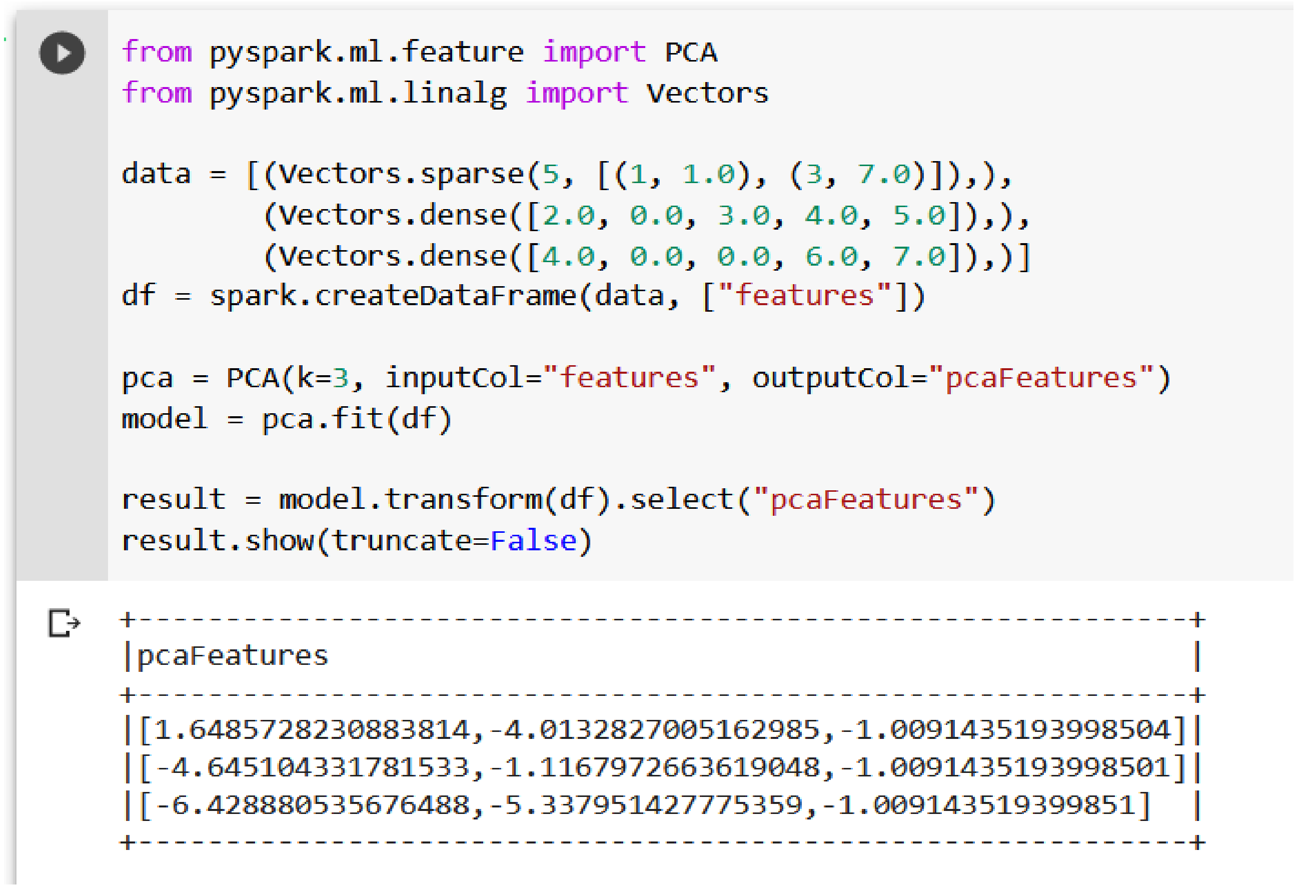
```
from pyspark.ml.feature import Binarizer

continuousDataFrame = spark.createDataFrame([
    (0, 0.1),
    (1, 0.8),
    (2, 0.2)
], ["id", "feature"])

binarizer = Binarizer(threshold=0.5, inputCol="feature", outputCol="binarized_feature")

binarizedDataFrame = binarizer.transform(continuousDataFrame)

print("Binarizer output with Threshold = %f" % binarizer.getThreshold())
binarizedDataFrame.show()
```

```
Binarizer output with Threshold = 0.500000
+---+-------+-----------------+
| id|feature|binarized_feature|
+---+-------+-----------------+
|  0|    0.1|              0.0|
|  1|    0.8|              1.0|
|  2|    0.2|              0.0|
+---+-------+-----------------+
```

PCA

PCA is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components. A PCA class trains a model to project vectors to a low-dimensional space using PCA. The example below shows how to project 5-dimensional feature vectors into 3-dimensional principal components.

Example



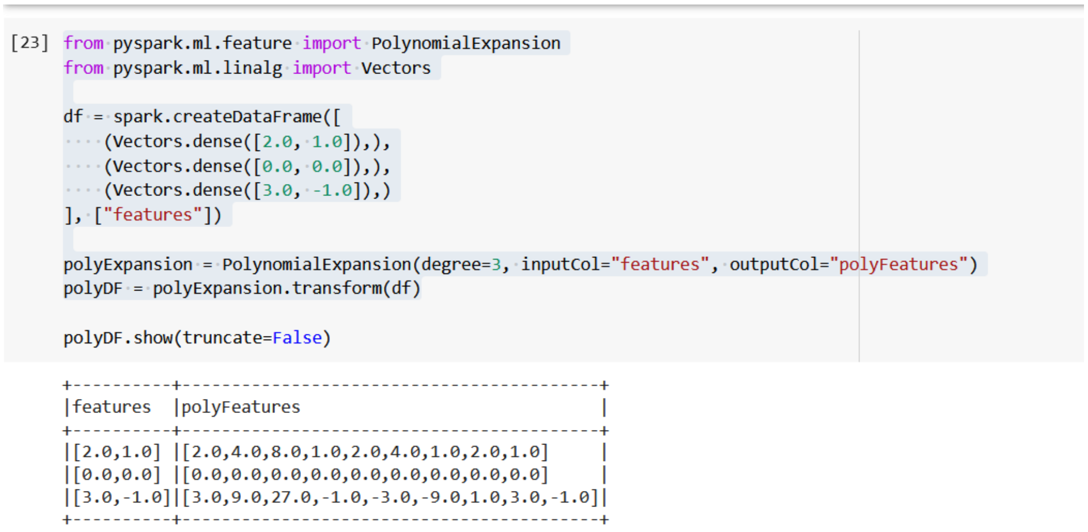
```python
from pyspark.ml.feature import PCA
from pyspark.ml.linalg import Vectors

data = [(Vectors.sparse(5, [(1, 1.0), (3, 7.0)]),),
        (Vectors.dense([2.0, 0.0, 3.0, 4.0, 5.0]),),
        (Vectors.dense([4.0, 0.0, 0.0, 6.0, 7.0]),)]
df = spark.createDataFrame(data, ["features"])

pca = PCA(k=3, inputCol="features", outputCol="pcaFeatures")
model = pca.fit(df)

result = model.transform(df).select("pcaFeatures")
result.show(truncate=False)
```

```
+-----------------------------------------------------------+
|pcaFeatures                                                |
+-----------------------------------------------------------+
|[1.6485728230883814,-4.013282700516296,-1.0091435193998504]|
|[-4.645104331781533,-1.1167972663619048,-1.0091435193998501]|
|[-6.428880535676488,-5.337951427775359,-1.009143519399851]  |
+-----------------------------------------------------------+
```

PolynomialExpansion

Polynomial expansion is the process of expanding your features into a polynomial space, which is formulated by an n-degree combination of original dimensions. A PolynomialExpansion class provides this functionality. The example below shows how to expand your features into a 3-degree polynomial space.

Example



```
[23] from pyspark.ml.feature import PolynomialExpansion
     from pyspark.ml.linalg import Vectors

     df = spark.createDataFrame([
         (Vectors.dense([2.0, 1.0]),),
         (Vectors.dense([0.0, 0.0]),),
         (Vectors.dense([3.0, -1.0]),)
     ], ["features"])

     polyExpansion = PolynomialExpansion(degree=3, inputCol="features", outputCol="polyFeatures")
     polyDF = polyExpansion.transform(df)

     polyDF.show(truncate=False)
```

```
+----------+------------------------------------------+
|features  |polyFeatures                              |
+----------+------------------------------------------+
|[2.0,1.0] |[2.0,4.0,8.0,1.0,2.0,4.0,1.0,2.0,1.0]     |
|[0.0,0.0] |[0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0]     |
|[3.0,-1.0]|[3.0,9.0,27.0,-1.0,-3.0,-9.0,1.0,3.0,-1.0]|
+----------+------------------------------------------+
```

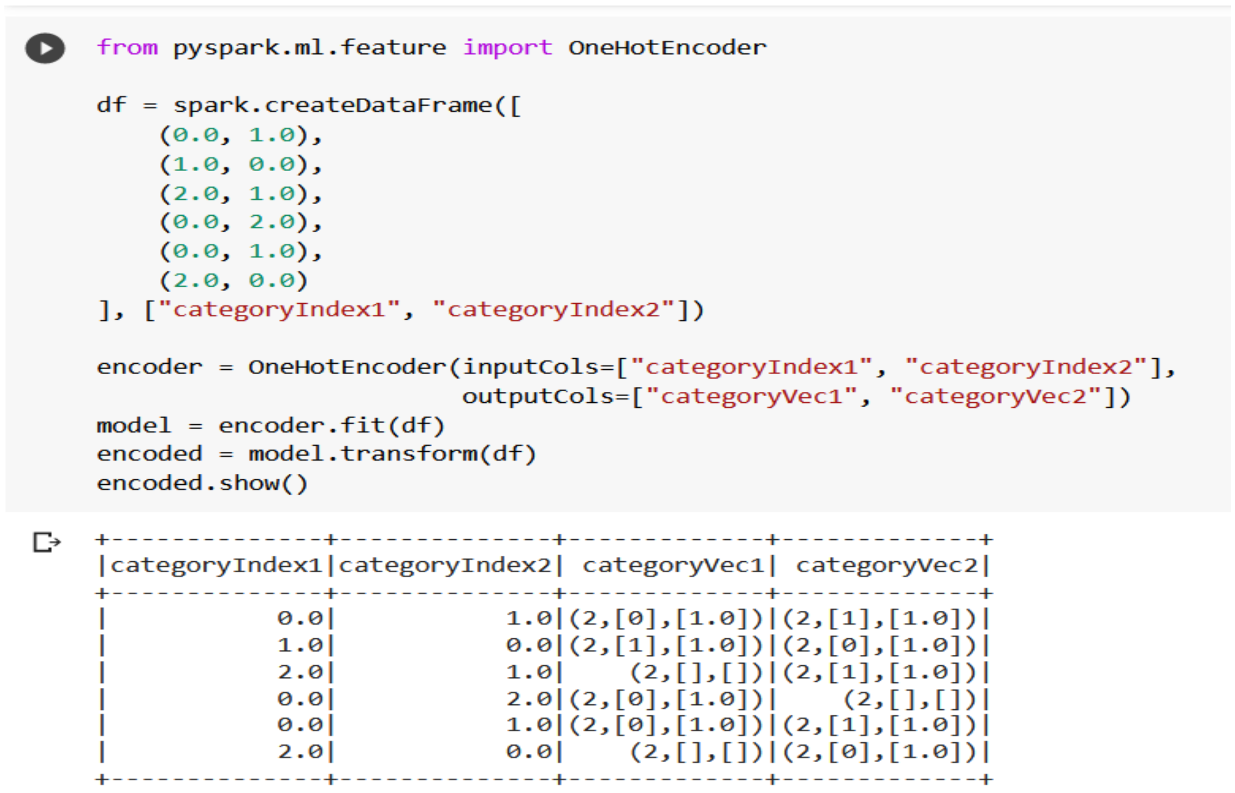
OneHotEncoder

One-hot encoding maps a categorical feature, represented as a label index, to a binary vector with at most a single one-value indicating the presence of a specific feature value from among the set of all feature values. This encoding allows algorithms which expect continuous features, such as Logistic Regression, to use categorical features. For string type input data, it is common to encode categorical features using StringIndexer first.

OneHotEncoder can transform multiple columns, returning an one-hot-encoded output vector column for each input column. It is common to merge these vectors into a single feature vector using VectorAssembler.

OneHotEncoder supports the handleInvalid parameter to choose how to handle invalid input during transforming data. Available options include ‘keep’ (any invalid inputs are assigned to an extra categorical index) and ‘error’ (throw an error).

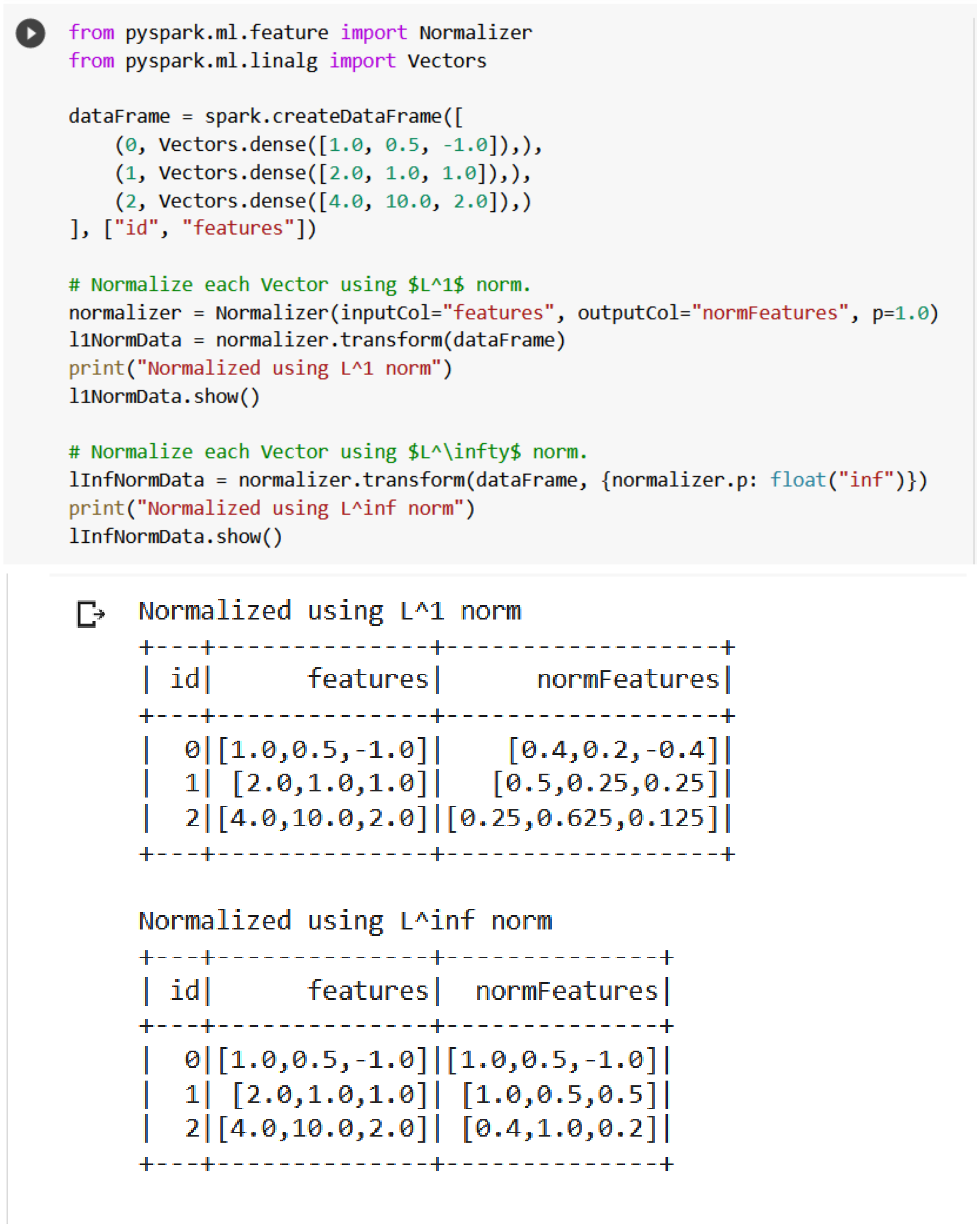
Example



```
from pyspark.ml.feature import OneHotEncoder

df = spark.createDataFrame([
    (0.0, 1.0),
    (1.0, 0.0),
    (2.0, 1.0),
    (0.0, 2.0),
    (0.0, 1.0),
    (2.0, 0.0)
], ["categoryIndex1", "categoryIndex2"])

encoder = OneHotEncoder(inputCols=["categoryIndex1", "categoryIndex2"],
                        outputCols=["categoryVec1", "categoryVec2"])
model = encoder.fit(df)
encoded = model.transform(df)
encoded.show()
```

```
+--------------+--------------+-------------+-------------+
|categoryIndex1|categoryIndex2| categoryVec1| categoryVec2|
+--------------+--------------+-------------+-------------+
|           0.0|           1.0|(2,[0],[1.0])|(2,[1],[1.0])|
|           1.0|           0.0|(2,[1],[1.0])|(2,[0],[1.0])|
|           2.0|           1.0|    (2,[],[])|(2,[1],[1.0])|
|           0.0|           2.0|(2,[0],[1.0])|    (2,[],[])|
|           0.0|           1.0|(2,[0],[1.0])|(2,[1],[1.0])|
|           2.0|           0.0|    (2,[],[])|(2,[0],[1.0])|
+--------------+--------------+-------------+-------------+
```

Normalizer
Normalizer is a Transformer which transforms a dataset of Vector rows, normalizing each Vector to have unit norm. It takes parameter p, which specifies the p-norm used for normalization. (
$p=2$
by default.) This normalization can help standardize your input data and improve the behavior of learning algorithms.
Example



```python
from pyspark.ml.feature import Normalizer
from pyspark.ml.linalg import Vectors

dataFrame = spark.createDataFrame([
    (0, Vectors.dense([1.0, 0.5, -1.0]),),
    (1, Vectors.dense([2.0, 1.0, 1.0]),),
    (2, Vectors.dense([4.0, 10.0, 2.0]),)
], ["id", "features"])

# Normalize each Vector using $L^1$ norm.
normalizer = Normalizer(inputCol="features", outputCol="normFeatures", p=1.0)
l1NormData = normalizer.transform(dataFrame)
print("Normalized using L^1 norm")
l1NormData.show()

# Normalize each Vector using $L^\infty$ norm.
lInfNormData = normalizer.transform(dataFrame, {normalizer.p: float("inf")})
print("Normalized using L^inf norm")
lInfNormData.show()
```

```
Normalized using L^1 norm
+---+--------------+------------------+
| id|      features|      normFeatures|
+---+--------------+------------------+
|  0|[1.0,0.5,-1.0]|    [0.4,0.2,-0.4]|
|  1| [2.0,1.0,1.0]|   [0.5,0.25,0.25]|
|  2|[4.0,10.0,2.0]|[0.25,0.625,0.125]|
+---+--------------+------------------+

Normalized using L^inf norm
+---+--------------+--------------+
| id|      features|  normFeatures|
+---+--------------+--------------+
|  0|[1.0,0.5,-1.0]|[1.0,0.5,-1.0]|
|  1| [2.0,1.0,1.0]| [1.0,0.5,0.5]|
|  2|[4.0,10.0,2.0]| [0.4,1.0,0.2]|
+---+--------------+--------------+
```

Feature Selectors

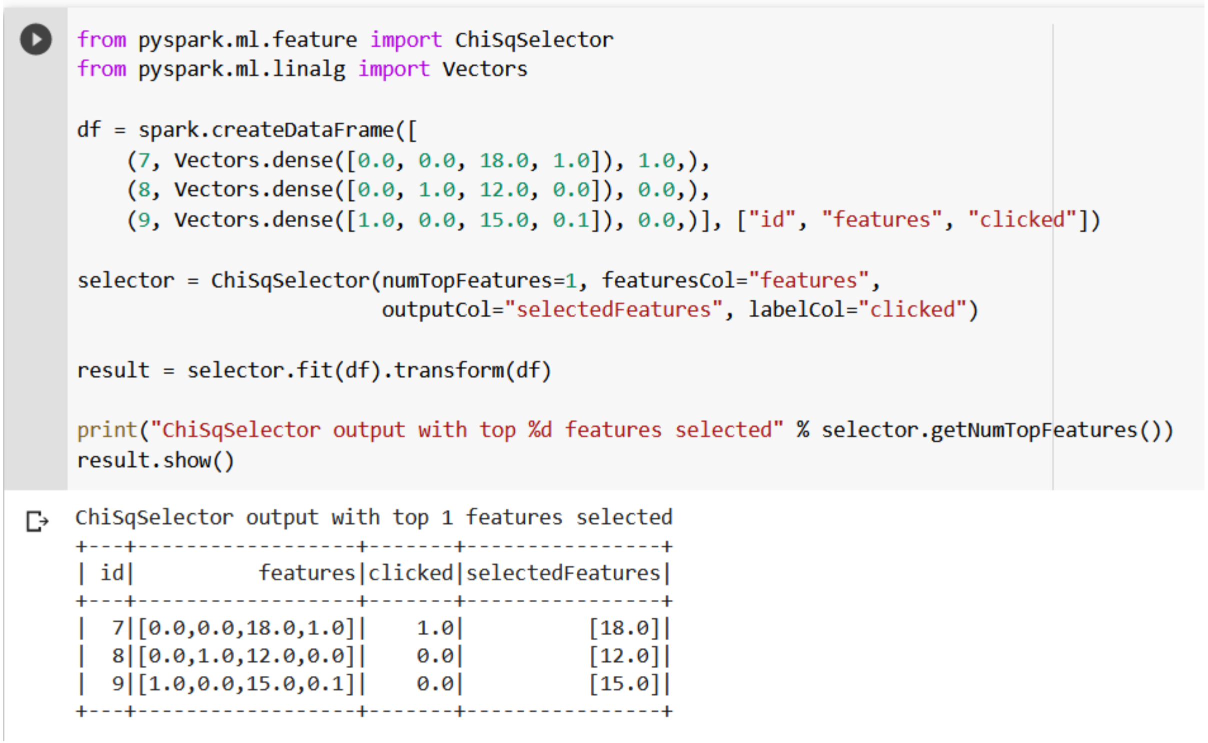
ChiSqSelector

ChiSqSelector stands for Chi-Squared feature selection. It operates on labeled data with categorical features. ChiSqSelector uses the Chi-Squared test of independence to decide which features to choose. It supports five selection methods: numTopFeatures, percentile, fpr, fdr, fwe:

- numTopFeatures chooses a fixed number of top features according to a chi-squared test. This is akin to yielding the features with the most predictive power.

- percentile is similar to numTopFeatures but chooses a fraction of all features instead of a fixed number.
- fpr chooses all features whose p-values are below a threshold, thus controlling the false positive rate of selection.
- fdr uses the Benjamini-Hochberg procedure to choose all features whose false discovery rate is below a threshold.
- fwe chooses all features whose p-values are below a threshold. The threshold is scaled by 1/numFeatures, thus controlling the family-wise error rate of selection. By default, the selection method is numTopFeatures, with the default number of top features set to 50. The user can choose a selection method using setSelectorType.

Example

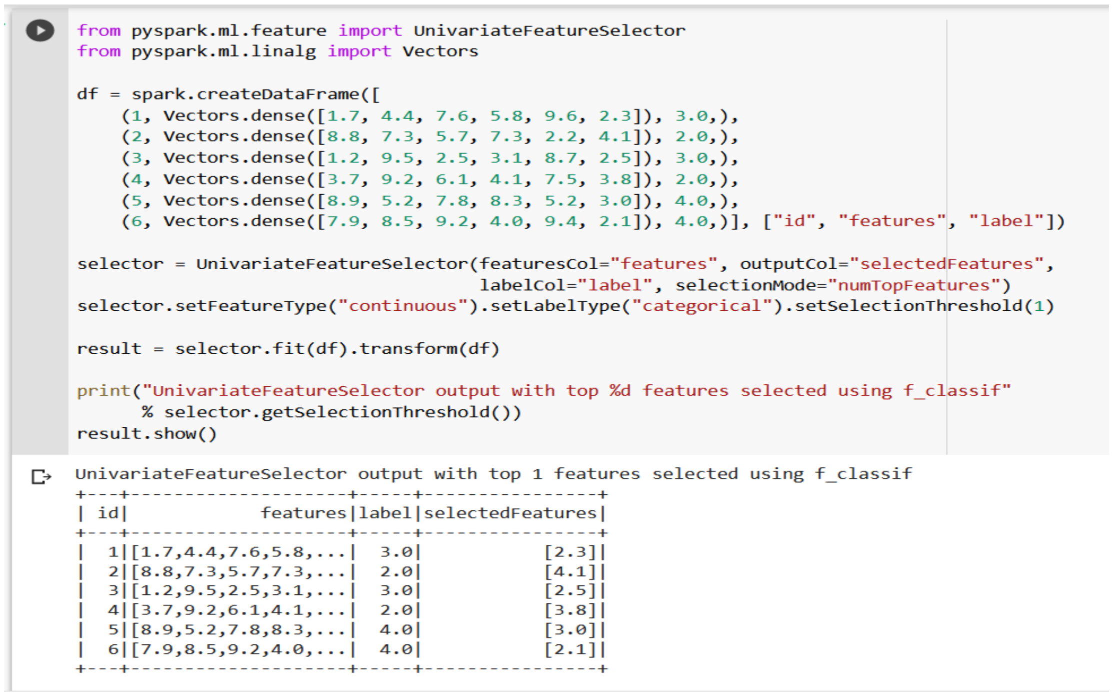


```
from pyspark.ml.feature import ChiSqSelector
from pyspark.ml.linalg import Vectors

df = spark.createDataFrame([
    (7, Vectors.dense([0.0, 0.0, 18.0, 1.0]), 1.0,),
    (8, Vectors.dense([0.0, 1.0, 12.0, 0.0]), 0.0,),
    (9, Vectors.dense([1.0, 0.0, 15.0, 0.1]), 0.0,)], ["id", "features", "clicked"])

selector = ChiSqSelector(numTopFeatures=1, featuresCol="features",
                         outputCol="selectedFeatures", labelCol="clicked")

result = selector.fit(df).transform(df)

print("ChiSqSelector output with top %d features selected" % selector.getNumTopFeatures())
result.show()
```

```
ChiSqSelector output with top 1 features selected
+---+------------------+-------+----------------+
| id|          features|clicked|selectedFeatures|
+---+------------------+-------+----------------+
|  7|[0.0,0.0,18.0,1.0]|    1.0|          [18.0]|
|  8|[0.0,1.0,12.0,0.0]|    0.0|          [12.0]|
|  9|[1.0,0.0,15.0,0.1]|    0.0|          [15.0]|
+---+------------------+-------+----------------+
```

UnivariateFeatureSelector

UnivariateFeatureSelector operates on categorical/continuous labels with categorical/continuous features. User can set featureType and labelType, and Spark will pick the score function to use based on the specified featureType andlabelType.



```
from pyspark.ml.feature import UnivariateFeatureSelector
from pyspark.ml.linalg import Vectors

df = spark.createDataFrame([
    (1, Vectors.dense([1.7, 4.4, 7.6, 5.8, 9.6, 2.3]), 3.0,),
    (2, Vectors.dense([8.8, 7.3, 5.7, 7.3, 2.2, 4.1]), 2.0,),
    (3, Vectors.dense([1.2, 9.5, 2.5, 3.1, 8.7, 2.5]), 3.0,),
    (4, Vectors.dense([3.7, 9.2, 6.1, 4.1, 7.5, 3.8]), 2.0,),
    (5, Vectors.dense([8.9, 5.2, 7.8, 8.3, 5.2, 3.0]), 4.0,),
    (6, Vectors.dense([7.9, 8.5, 9.2, 4.0, 9.4, 2.1]), 4.0,)], ["id", "features", "label"])

selector = UnivariateFeatureSelector(featuresCol="features", outputCol="selectedFeatures",
                                     labelCol="label", selectionMode="numTopFeatures")
selector.setFeatureType("continuous").setLabelType("categorical").setSelectionThreshold(1)

result = selector.fit(df).transform(df)

print("UnivariateFeatureSelector output with top %d features selected using f_classif"
      % selector.getSelectionThreshold())
result.show()
```

```
UnivariateFeatureSelector output with top 1 features selected using f_classif
+---+--------------------+-----+----------------+
| id|            features|label|selectedFeatures|
+---+--------------------+-----+----------------+
|  1|[1.7,4.4,7.6,5.8,...|  3.0|           [2.3]|
|  2|[8.8,7.3,5.7,7.3,...|  2.0|           [4.1]|
|  3|[1.2,9.5,2.5,3.1,...|  3.0|           [2.5]|
|  4|[3.7,9.2,6.1,4.1,...|  2.0|           [3.8]|
|  5|[8.9,5.2,7.8,8.3,...|  4.0|           [3.0]|
|  6|[7.9,8.5,9.2,4.0,...|  4.0|           [2.1]|
+---+--------------------+-----+----------------+
```

Practical 7 - Implementation of Classification & Regression Algorithms Using Big Data.

1) Classification

i) Logistic Regression
ii) Random Forest
iii) Multilayer Perceptron Classifier
Regression
iv) Linear Regression
v) Decision Regression

Solution

1) Classification

1. Logistic Regression:

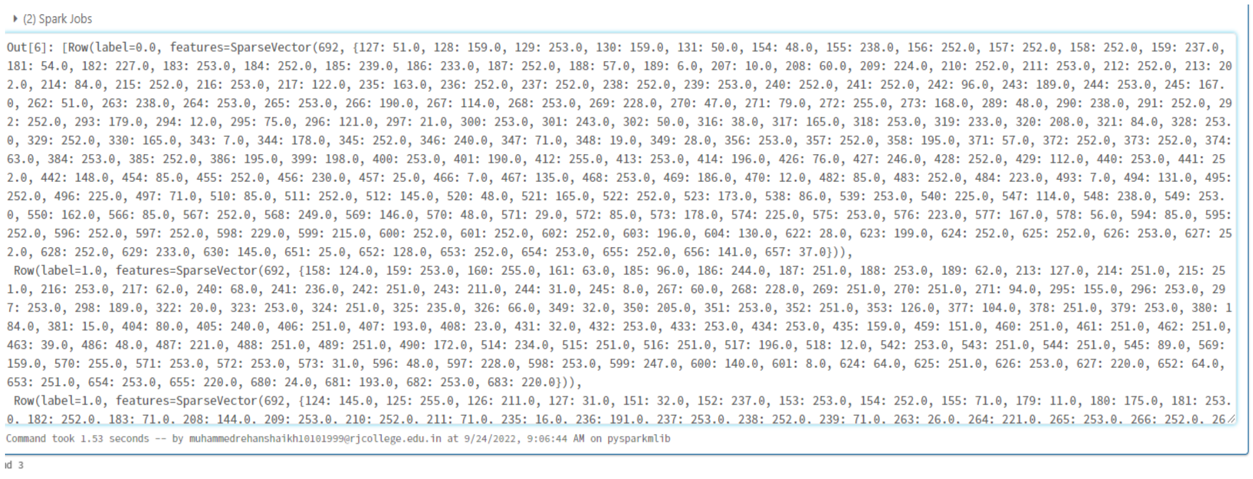
Problem Statement: Performing Decision Tree Classifier on LIBSVM Classification dataset.

1. Binomial Logistic Regression:

#Import
Code:

```
from pyspark.ml.classification import LogisticRegression

# Load training data
training =
spark.read.format('libsvm').load('dbfs:/FileStore/shared_uploads/muhammedrehanshaikh10101
999@rjcollege.edu.in/sample_lib_classification_data.txt')

training.collect()
```

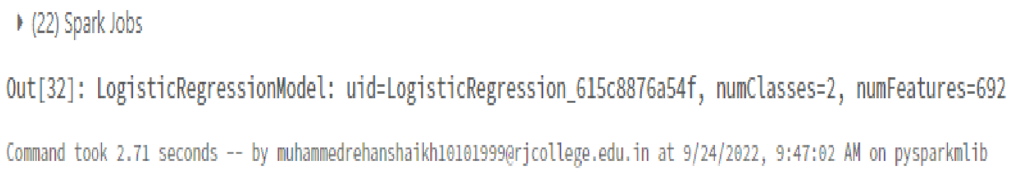
Output:



```
Out[6]: [Row(label=0.0, features=SparseVector(692, {127: 51.0, 128: 159.0, 129: 253.0, 130: 159.0, 131: 50.0, 154: 48.0, 155: 238.0, 156: 252.0, 157: 252.0, 158: 252.0, 159: 237.0,
181: 54.0, 182: 227.0, 183: 253.0, 184: 252.0, 185: 239.0, 186: 233.0, 187: 252.0, 188: 57.0, 189: 6.0, 207: 10.0, 208: 60.0, 209: 224.0, 210: 252.0, 211: 253.0, 212: 252.0, 213: 20
2.0, 214: 84.0, 215: 252.0, 216: 253.0, 217: 122.0, 235: 163.0, 236: 252.0, 237: 252.0, 238: 252.0, 239: 253.0, 240: 252.0, 241: 252.0, 242: 96.0, 243: 189.0, 244: 253.0, 245: 167.
0, 262: 51.0, 263: 238.0, 264: 253.0, 265: 253.0, 266: 190.0, 267: 114.0, 268: 253.0, 269: 228.0, 270: 47.0, 271: 79.0, 272: 255.0, 273: 168.0, 289: 48.0, 290: 238.0, 291: 252.0, 29
2: 252.0, 293: 179.0, 294: 12.0, 295: 75.0, 296: 121.0, 297: 21.0, 300: 253.0, 301: 243.0, 302: 50.0, 316: 38.0, 317: 165.0, 318: 253.0, 319: 233.0, 320: 208.0, 321: 84.0, 328: 253.
0, 329: 252.0, 330: 165.0, 343: 7.0, 344: 178.0, 345: 252.0, 346: 240.0, 347: 71.0, 348: 19.0, 349: 28.0, 356: 253.0, 357: 252.0, 358: 195.0, 371: 57.0, 372: 252.0, 373: 252.0, 374:
63.0, 384: 253.0, 385: 252.0, 386: 195.0, 399: 198.0, 400: 253.0, 401: 190.0, 412: 255.0, 413: 253.0, 414: 196.0, 426: 76.0, 427: 246.0, 428: 252.0, 429: 112.0, 440: 253.0, 441: 25
2.0, 442: 148.0, 454: 85.0, 455: 252.0, 456: 230.0, 457: 25.0, 466: 7.0, 467: 135.0, 468: 253.0, 469: 186.0, 470: 12.0, 482: 85.0, 483: 252.0, 484: 223.0, 493: 7.0, 494: 131.0, 495:
252.0, 496: 225.0, 497: 71.0, 510: 85.0, 511: 252.0, 512: 145.0, 520: 48.0, 521: 165.0, 522: 252.0, 523: 173.0, 538: 86.0, 539: 253.0, 540: 225.0, 547: 114.0, 548: 238.0, 549: 253.
0, 550: 162.0, 566: 85.0, 567: 252.0, 568: 249.0, 569: 146.0, 570: 48.0, 571: 29.0, 572: 85.0, 573: 178.0, 574: 225.0, 575: 253.0, 576: 223.0, 577: 167.0, 578: 56.0, 594: 85.0, 595:
252.0, 596: 252.0, 597: 252.0, 598: 229.0, 599: 215.0, 600: 252.0, 601: 252.0, 602: 252.0, 603: 196.0, 604: 130.0, 622: 28.0, 623: 199.0, 624: 252.0, 625: 252.0, 626: 253.0, 627: 25
2.0, 628: 252.0, 629: 233.0, 630: 145.0, 651: 25.0, 652: 128.0, 653: 252.0, 654: 253.0, 655: 252.0, 656: 141.0, 657: 37.0})),
 Row(label=1.0, features=SparseVector(692, {158: 124.0, 159: 253.0, 160: 255.0, 161: 63.0, 185: 96.0, 186: 244.0, 187: 251.0, 188: 253.0, 189: 62.0, 213: 127.0, 214: 251.0, 215: 25
1.0, 216: 253.0, 217: 62.0, 240: 68.0, 241: 236.0, 242: 251.0, 243: 211.0, 244: 31.0, 245: 8.0, 267: 60.0, 268: 228.0, 269: 251.0, 270: 251.0, 271: 94.0, 295: 155.0, 296: 253.0, 29
7: 253.0, 298: 189.0, 322: 20.0, 323: 253.0, 324: 251.0, 325: 235.0, 326: 66.0, 349: 32.0, 350: 205.0, 351: 253.0, 352: 251.0, 353: 126.0, 377: 104.0, 378: 251.0, 379: 253.0, 380: 1
84.0, 381: 15.0, 404: 80.0, 405: 240.0, 406: 251.0, 407: 193.0, 408: 23.0, 431: 32.0, 432: 253.0, 433: 253.0, 434: 253.0, 435: 159.0, 459: 151.0, 460: 251.0, 461: 251.0, 462: 251.0,
463: 39.0, 486: 48.0, 487: 221.0, 488: 251.0, 489: 251.0, 490: 172.0, 514: 234.0, 515: 251.0, 516: 251.0, 517: 196.0, 518: 12.0, 542: 253.0, 543: 251.0, 544: 251.0, 545: 89.0, 569:
159.0, 570: 255.0, 571: 253.0, 572: 253.0, 573: 31.0, 596: 48.0, 597: 228.0, 598: 253.0, 599: 247.0, 600: 140.0, 601: 8.0, 624: 64.0, 625: 251.0, 626: 253.0, 627: 220.0, 652: 64.0,
653: 251.0, 654: 253.0, 655: 220.0, 680: 24.0, 681: 193.0, 682: 253.0, 683: 220.0})),
 Row(label=1.0, features=SparseVector(692, {124: 145.0, 125: 255.0, 126: 211.0, 127: 31.0, 151: 32.0, 152: 237.0, 153: 253.0, 154: 252.0, 155: 71.0, 179: 11.0, 180: 175.0, 181: 253.
0, 182: 252.0, 183: 71.0, 208: 144.0, 209: 253.0, 210: 252.0, 211: 71.0, 235: 16.0, 236: 191.0, 237: 253.0, 238: 252.0, 239: 71.0, 263: 26.0, 264: 221.0, 265: 253.0, 266: 252.0, 26
Command took 1.53 seconds -- by muhammedrehanshaikh10101999@rjcollege.edu.in at 9/24/2022, 9:06:44 AM on pysparkmlib
```

Code:
#Creating Logistic Regression model
lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)

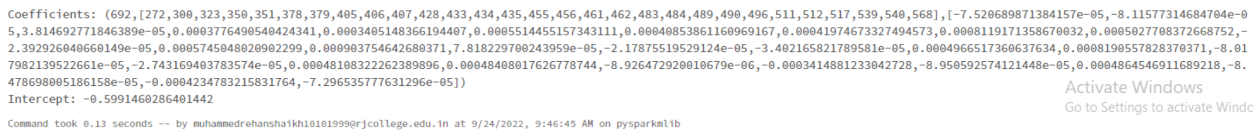
# Fit the model
lrModel = lr.fit(training)
lrModel
Output:



```
Out[32]: LogisticRegressionModel: uid=LogisticRegression_615c8876a54f, numClasses=2, numFeatures=692

Command took 2.71 seconds -- by muhammedrehanshaikh10101999@rjcollege.edu.in at 9/24/2022, 9:47:02 AM on pysparkmlib
```

Code:
# Print the coefficients and intercept for logistic regression
print("Coefficients: " + str(lrModel.coefficients))
print("Intercept: " + str(lrModel.intercept))

Output:

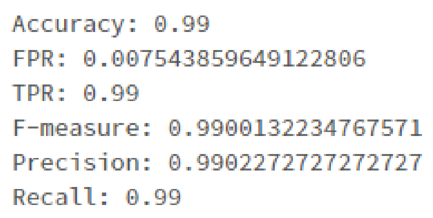


```
Coefficients: (692,[272,300,323,350,351,378,379,405,406,407,428,433,434,435,455,456,461,462,483,484,489,490,496,511,512,517,539,540,568],[-7.520689871384157e-05,-8.11577314684704e-0
5,3.814692771846389e-05,0.0003776490540424341,0.0003405148366194407,0.0005514455157343111,0.0004085386116096916 7,0.00041974673327494573,0.0008119171358670032,0.0005027708372668752,-
2.392926040660149e-05,0.0005745048020902299,0.000903754642680371,7.818229700243959e-05,-2.17875519529124e-05,-3.402165821789581e-05,0.0004966517360637634,0.0008190557828370371,-8.01
7982139522661e-05,-2.743169403783574e-05,0.00048108322262389896,0.00048408017626778744,-8.926472920010679e-06,-0.0003414881233042728,-8.950592574121448e-05,0.0004864546911689218,-8.
478698005186158e-05,-0.0004234783215831764,-7.296535777631296e-05])
Intercept: -0.5991460286401442

Command took 0.13 seconds -- by muhammedrehanshaikh10101999@rjcollege.edu.in at 9/24/2022, 9:46:45 AM on pysparkmlib
```

Code:
trainingSummary = lrModel.summary
accuracy = trainingSummary.accuracy
falsePositiveRate = trainingSummary.weightedFalsePositiveRate
truePositiveRate = trainingSummary.weightedTruePositiveRate
fMeasure = trainingSummary.weightedFMeasure()

```
precision = trainingSummary.weightedPrecision
recall = trainingSummary.weightedRecall
print("Accuracy: %s\nFPR: %s\nTPR: %s\nF-measure: %s\nPrecision: %s\nRecall: %s"
    % (accuracy, falsePositiveRate, truePositiveRate, fMeasure, precision, recall))
```

Output:



```
Accuracy: 0.99
FPR: 0.007543859649122806
TPR: 0.99
F-measure: 0.9900132234767571
Precision: 0.9902272727272727
Recall: 0.99
```

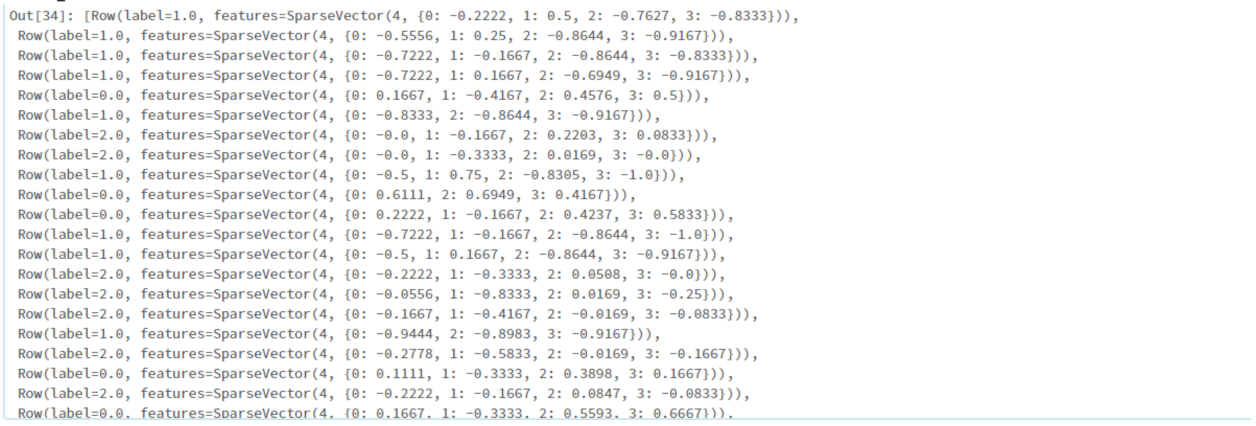
2. **Multinomial Logistic Regression:**

Problem Statement: Performing Decision Tree Classifier on LIBSVM Multi Class Classification dataset.

Code:

```
training =
spark.read.format('libsvm').load('dbfs:/FileStore/shared_uploads/muhammedrehanshaikh10101
999@rjcollege.edu.in/sample_multiclass_classification_data.txt')
training.collect()
```

Output:



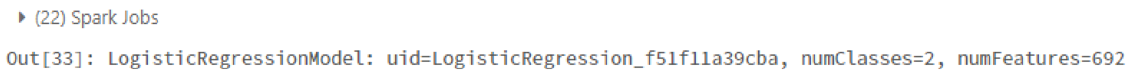
```
Out[34]: [Row(label=1.0, features=SparseVector(4, {0: -0.2222, 1: 0.5, 2: -0.7627, 3: -0.8333})),
 Row(label=1.0, features=SparseVector(4, {0: -0.5556, 1: 0.25, 2: -0.8644, 3: -0.9167})),
 Row(label=1.0, features=SparseVector(4, {0: -0.7222, 1: -0.1667, 2: -0.8644, 3: -0.8333})),
 Row(label=1.0, features=SparseVector(4, {0: -0.7222, 1: 0.1667, 2: -0.6949, 3: -0.9167})),
 Row(label=0.0, features=SparseVector(4, {0: 0.1667, 1: -0.4167, 2: 0.4576, 3: 0.5})),
 Row(label=1.0, features=SparseVector(4, {0: -0.8333, 2: -0.8644, 3: -0.9167})),
 Row(label=2.0, features=SparseVector(4, {0: -0.0, 1: -0.1667, 2: 0.2203, 3: 0.0833})),
 Row(label=2.0, features=SparseVector(4, {0: -0.0, 1: -0.3333, 2: 0.0169, 3: -0.0})),
 Row(label=1.0, features=SparseVector(4, {0: -0.5, 1: 0.75, 2: -0.8305, 3: -1.0})),
 Row(label=0.0, features=SparseVector(4, {0: 0.6111, 2: 0.6949, 3: 0.4167})),
 Row(label=0.0, features=SparseVector(4, {0: 0.2222, 1: -0.1667, 2: 0.4237, 3: 0.5833})),
 Row(label=1.0, features=SparseVector(4, {0: -0.7222, 1: -0.1667, 2: -0.8644, 3: -1.0})),
 Row(label=1.0, features=SparseVector(4, {0: -0.5, 1: 0.1667, 2: -0.8644, 3: -0.9167})),
 Row(label=2.0, features=SparseVector(4, {0: -0.2222, 1: -0.3333, 2: 0.0508, 3: -0.0})),
 Row(label=2.0, features=SparseVector(4, {0: -0.0556, 1: -0.8333, 2: 0.0169, 3: -0.25})),
 Row(label=2.0, features=SparseVector(4, {0: -0.1667, 1: -0.4167, 2: -0.0169, 3: -0.0833})),
 Row(label=1.0, features=SparseVector(4, {0: -0.9444, 2: -0.8983, 3: -0.9167})),
 Row(label=2.0, features=SparseVector(4, {0: -0.2778, 1: -0.5833, 2: -0.0169, 3: -0.1667})),
 Row(label=0.0, features=SparseVector(4, {0: 0.1111, 1: -0.3333, 2: 0.3898, 3: 0.1667})),
 Row(label=2.0, features=SparseVector(4, {0: -0.2222, 1: -0.1667, 2: 0.0847, 3: -0.0833})),
 Row(label=0.0, features=SparseVector(4, {0: 0.1667, 1: -0.3333, 2: 0.5593, 3: 0.6667})),
```

Code:

```
mlr = LogisticRegression(maxIter=10,elasticNetParam=0.8,regParam=0.3)
```

# Fit the model
mlrModel = mlr.fit(training)

Output:

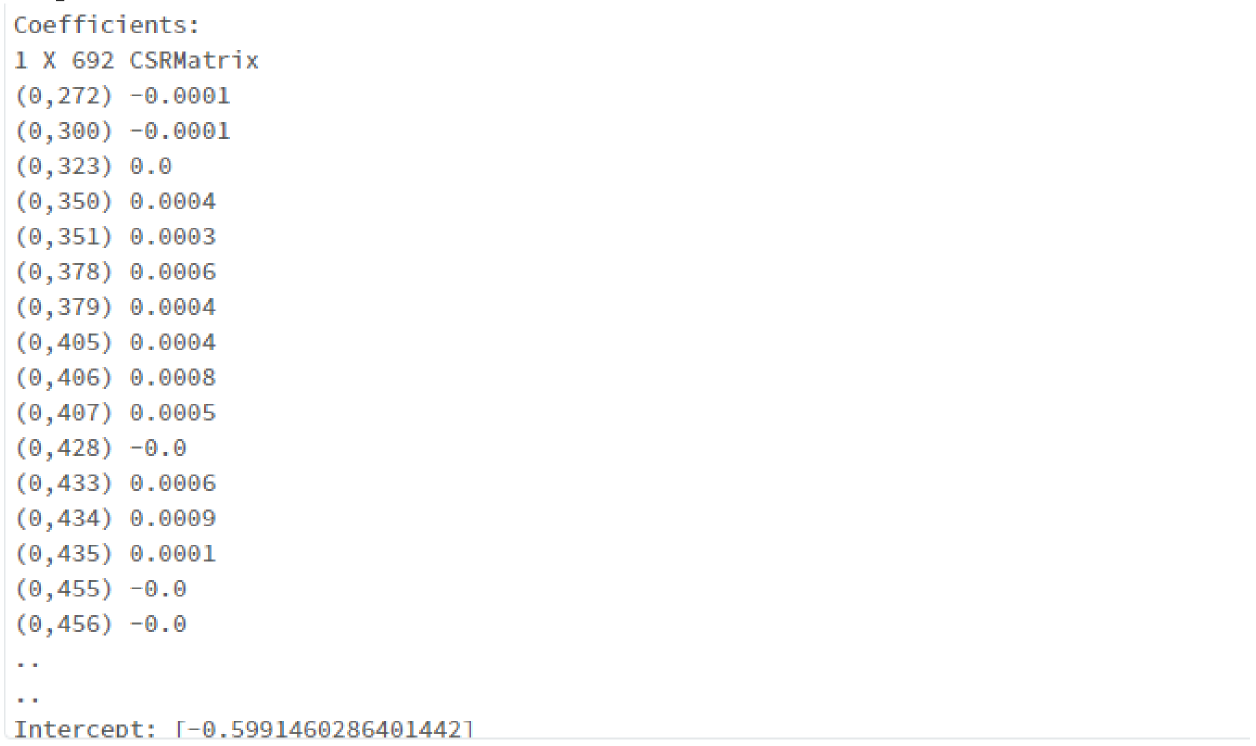


```
▸ (22) Spark Jobs
Out[33]: LogisticRegressionModel: uid=LogisticRegression_f51f11a39cba, numClasses=2, numFeatures=692
```

Code:

# Print the coefficients and intercept for multinomial logistic regression
print("Coefficients: \n" + str(mlrModel.coefficientMatrix))
print("Intercept: " + str(mlrModel.interceptVector))
training.collect()

Output:

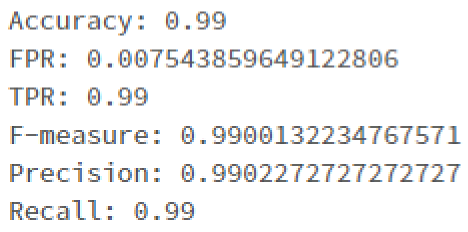


```
Coefficients:
1 X 692 CSRMatrix
(0,272) -0.0001
(0,300) -0.0001
(0,323) 0.0
(0,350) 0.0004
(0,351) 0.0003
(0,378) 0.0006
(0,379) 0.0004
(0,405) 0.0004
(0,406) 0.0008
(0,407) 0.0005
(0,428) -0.0
(0,433) 0.0006
(0,434) 0.0009
(0,435) 0.0001
(0,455) -0.0
(0,456) -0.0
..
..
Intercept: [-0.5991460286401442]
```

Code:
#Summary of the model
trainingSummary = mlrModel.summary
accuracy = trainingSummary.accuracy
falsePositiveRate = trainingSummary.weightedFalsePositiveRate
truePositiveRate = trainingSummary.weightedTruePositiveRate
fMeasure = trainingSummary.weightedFMeasure()

```
precision = trainingSummary.weightedPrecision
recall1 = trainingSummary.weightedRecall
print("Accuracy: %s\nFPR: %s\nTPR: %s\nF-measure: %s\nPrecision: %s\nRecall: %s"
    % (accuracy, falsePositiveRate, truePositiveRate, fMeasure, precision, recall1))
```

Output:



```
Accuracy: 0.99
FPR: 0.007543859649122806
TPR: 0.99
F-measure: 0.9900132234767571
Precision: 0.9902272727272727
Recall: 0.99
```

ii) Decision Tree

Problem Statement:
Performing Decision Tree Classifier on LIBSVM Classification dataset

1. **Import**

   Code:

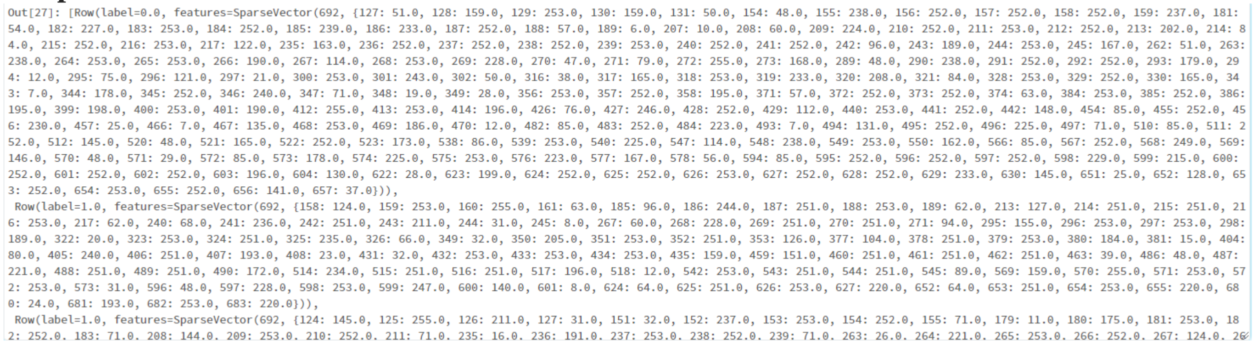
```
from pyspark.ml import Pipeline
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.feature import StringIndexer , VectorIndexerModel
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
```

2. **Load the data stored in LIBSVM format as a DataFrame.**

   Code:

```
data =
spark.read.format("libsvm").load("dbfs:/FileStore/shared_uploads/amandeepverma145@
gmail.com/sample_libsvm_data-3.txt")
data.collect()
```

Output:



```
Out[27]: [Row(label=0.0, features=SparseVector(692, {127: 51.0, 128: 159.0, 129: 253.0, 130: 159.0, 131: 50.0, 154: 48.0, 155: 238.0, 156: 252.0, 157: 252.0, 158: 252.0, 159: 237.0, 181:
54.0, 182: 227.0, 183: 253.0, 184: 252.0, 185: 239.0, 186: 233.0, 187: 252.0, 188: 57.0, 189: 6.0, 207: 10.0, 208: 60.0, 209: 224.0, 210: 252.0, 211: 253.0, 212: 252.0, 213: 202.0, 214: 8
4.0, 215: 252.0, 216: 253.0, 217: 122.0, 235: 163.0, 236: 252.0, 237: 252.0, 238: 252.0, 239: 253.0, 240: 252.0, 241: 252.0, 242: 96.0, 243: 189.0, 244: 253.0, 245: 167.0, 262: 51.0, 263:
238.0, 264: 253.0, 265: 253.0, 266: 190.0, 267: 114.0, 268: 253.0, 269: 228.0, 270: 47.0, 271: 79.0, 272: 255.0, 273: 168.0, 289: 48.0, 290: 238.0, 291: 252.0, 292: 252.0, 293: 179.0, 29
4: 12.0, 295: 75.0, 296: 121.0, 297: 21.0, 300: 253.0, 301: 243.0, 302: 50.0, 316: 38.0, 317: 165.0, 318: 253.0, 319: 233.0, 320: 208.0, 321: 84.0, 328: 253.0, 329: 252.0, 330: 165.0, 34
3: 7.0, 344: 178.0, 345: 252.0, 346: 240.0, 347: 71.0, 348: 19.0, 349: 28.0, 356: 253.0, 357: 252.0, 358: 195.0, 371: 57.0, 372: 252.0, 373: 252.0, 374: 63.0, 384: 253.0, 385: 252.0, 386:
195.0, 399: 198.0, 400: 253.0, 401: 190.0, 412: 255.0, 413: 253.0, 414: 196.0, 426: 76.0, 427: 246.0, 428: 252.0, 429: 112.0, 440: 253.0, 441: 252.0, 442: 148.0, 454: 85.0, 455: 252.0, 45
6: 230.0, 457: 25.0, 466: 7.0, 467: 135.0, 468: 253.0, 469: 186.0, 470: 12.0, 482: 85.0, 483: 252.0, 484: 223.0, 493: 7.0, 494: 131.0, 495: 252.0, 496: 225.0, 497: 71.0, 510: 85.0, 511: 2
52.0, 512: 145.0, 520: 48.0, 521: 165.0, 522: 252.0, 523: 173.0, 538: 86.0, 539: 253.0, 540: 225.0, 547: 114.0, 548: 238.0, 549: 253.0, 550: 162.0, 566: 85.0, 567: 252.0, 568: 249.0, 569:
146.0, 570: 48.0, 571: 29.0, 572: 85.0, 573: 178.0, 574: 225.0, 575: 253.0, 576: 223.0, 577: 167.0, 578: 56.0, 594: 85.0, 595: 252.0, 596: 252.0, 597: 252.0, 598: 229.0, 599: 215.0, 600:
252.0, 601: 252.0, 602: 252.0, 603: 196.0, 604: 130.0, 622: 28.0, 623: 199.0, 624: 252.0, 625: 252.0, 626: 253.0, 627: 252.0, 628: 252.0, 629: 233.0, 630: 145.0, 651: 25.0, 652: 128.0, 65
3: 252.0, 654: 253.0, 655: 252.0, 656: 141.0, 657: 37.0})),
 Row(label=1.0, features=SparseVector(692, {158: 124.0, 159: 253.0, 160: 255.0, 161: 63.0, 185: 96.0, 186: 244.0, 187: 251.0, 188: 253.0, 189: 62.0, 213: 127.0, 214: 251.0, 215: 251.0, 21
6: 253.0, 217: 62.0, 240: 68.0, 241: 236.0, 242: 251.0, 243: 211.0, 244: 31.0, 245: 8.0, 267: 60.0, 268: 228.0, 269: 251.0, 270: 251.0, 271: 94.0, 295: 155.0, 296: 253.0, 297: 253.0, 298:
189.0, 322: 20.0, 323: 253.0, 324: 251.0, 325: 235.0, 326: 66.0, 349: 32.0, 350: 205.0, 351: 253.0, 352: 251.0, 353: 126.0, 377: 104.0, 378: 251.0, 379: 253.0, 380: 184.0, 381: 15.0, 404:
80.0, 405: 240.0, 406: 251.0, 407: 193.0, 408: 23.0, 431: 32.0, 432: 253.0, 433: 253.0, 434: 253.0, 435: 159.0, 459: 151.0, 460: 251.0, 461: 251.0, 462: 251.0, 463: 39.0, 486: 48.0, 487:
221.0, 488: 251.0, 489: 251.0, 490: 172.0, 514: 234.0, 515: 251.0, 516: 251.0, 517: 196.0, 518: 12.0, 542: 253.0, 543: 251.0, 544: 251.0, 545: 89.0, 569: 159.0, 570: 255.0, 571: 253.0, 57
2: 253.0, 573: 31.0, 596: 48.0, 597: 228.0, 598: 253.0, 599: 247.0, 600: 140.0, 601: 8.0, 624: 64.0, 625: 251.0, 626: 253.0, 627: 220.0, 652: 64.0, 653: 251.0, 654: 253.0, 655: 220.0, 68
0: 24.0, 681: 193.0, 682: 253.0, 683: 220.0})),
 Row(label=1.0, features=SparseVector(692, {124: 145.0, 125: 255.0, 126: 211.0, 127: 31.0, 151: 32.0, 152: 237.0, 153: 253.0, 154: 252.0, 155: 71.0, 179: 11.0, 180: 175.0, 181: 253.0, 18
2: 252.0, 183: 71.0, 208: 144.0, 209: 253.0, 210: 252.0, 211: 71.0, 235: 16.0, 236: 191.0, 237: 253.0, 238: 252.0, 239: 71.0, 263: 26.0, 264: 221.0, 265: 253.0, 266: 252.0, 267: 124.0, 26
```

3. **Index labels, adding metadata to the label column.Automatically identify categorical features, and index them.**

   Code :

   labelIndexer = StringIndexer(inputCol="label", outputCol="indexedLabel").fit(data)
   featureIndexer =\
    VectorIndexer(inputCol="features", outputCol="indexedFeatures", maxCategories=4).fit(data)

4. **Split the data into training and test sets (30% held out for testing)**

   Code:

   (trainingData, testData) = data.randomSplit([0.7, 0.3])

5. **Train a DecisionTree model**

   Code:

   dt = DecisionTreeClassifier(labelCol="indexedLabel",featuresCol="indexedFeatures")

6. **Chain indexers and tree in a Pipeline**

   Code :

   pipeline = Pipeline(stages=[labelIndexer, featureIndexer, dt])

7. **Train model. This also runs the indexers**

   Code :

model = pipeline.fit(trainingData)

8. **Make predictions.**

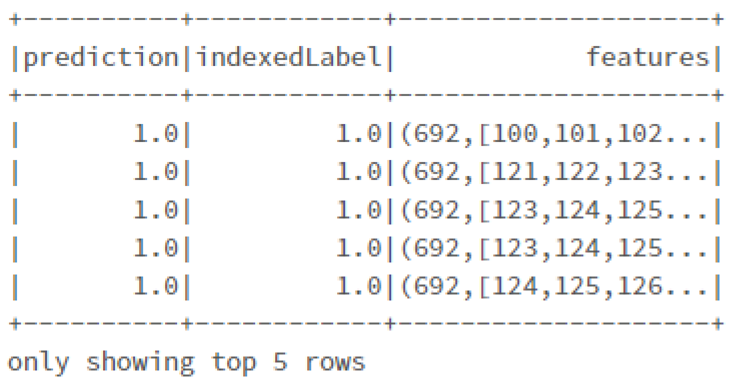
   Code :

   predictions = model.transform(testData)

9. **Select example rows to display.**

   Code :
   predictions.select("prediction", "indexedLabel", "features").show(5)

   Output :



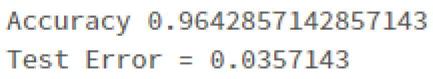
```
+----------+------------+--------------------+
|prediction|indexedLabel|            features|
+----------+------------+--------------------+
|       1.0|         1.0|(692,[100,101,102...|
|       1.0|         1.0|(692,[121,122,123...|
|       1.0|         1.0|(692,[123,124,125...|
|       1.0|         1.0|(692,[123,124,125...|
|       1.0|         1.0|(692,[124,125,126...|
+----------+------------+--------------------+
only showing top 5 rows
```

10. **Select (prediction, true label) and compute test error and Accuracy**

   Code :

```
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction", metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print("Accuracy",accuracy)
print("Test Error = %g " % (1.0 - accuracy))
```

   Output :



```
Accuracy 0.9642857142857143
Test Error = 0.0357143
```

11. **Summary**

    Code :

    ```
    treeModel = model.stages[2]
    # summary only
    print(treeModel)
    ```

i) Random Forest

Problem Statement
Performing Decision Tree Classifier on LIBSVM ClassiFication dataset

1. **Import library**

   Code-
   ```
   from pyspark.ml import Pipeline
   from pyspark.ml.classification import RandomForestClassifier
   from pyspark.ml.feature import IndexToString, StringIndexer, VectorIndexer
   from pyspark.ml.evaluation import MulticlassClassificationEvaluator
   ```

2. **Load and parse the data file, converting it to a DataFrame.**

   Code-
   ```
   data =
   spark.read.format("libsvm").load("dbfs:/FileStore/shared_uploads/seenivasanharis
   h786@gmail.com/sample_lib_classification_data.txt")
   ```

3. **Index labels, adding metadata to the label column. Fit on whole dataset to include all labels in index.**

   Code-
   ```
   labelIndexer = StringIndexer(inputCol="label",
   outputCol="indexedLabel").fit(data)
   ```

4. **Automatically identify categorical features, and index them.Set maxCategories so features with > 4 distinct values are treated as continuous.**

   Code-
   ```
   featureIndexer =\
   ```

```
    VectorIndexer(inputCol="features", outputCol="indexedFeatures",
maxCategories=4).fit(data)
```

5. **Split the data into training and test sets (30% held out for testing)**

   Code-
   ```
   (trainingData, testData) = data.randomSplit([0.7, 0.3])
   ```

6. **Train a RandomForest model.**

   Code-
   ```
   rf = RandomForestClassifier(labelCol="indexedLabel",
   featuresCol="indexedFeatures", numTrees=10)
   ```

7. **Convert indexed labels back to original labels.**

   Code-
   ```
   labelConverter = IndexToString(inputCol="prediction",
   outputCol="predictedLabel",labels=labelIndexer.labels)
   ```

8. **Chain indexers and forest in a Pipeline**

   Code-
   ```
   pipeline = Pipeline(stages=[labelIndexer, featureIndexer, rf, labelConverter])
   ```

9. **Train model. This also runs the indexers.**

   Code-
   ```
   model = pipeline.fit(trainingData)
   ```

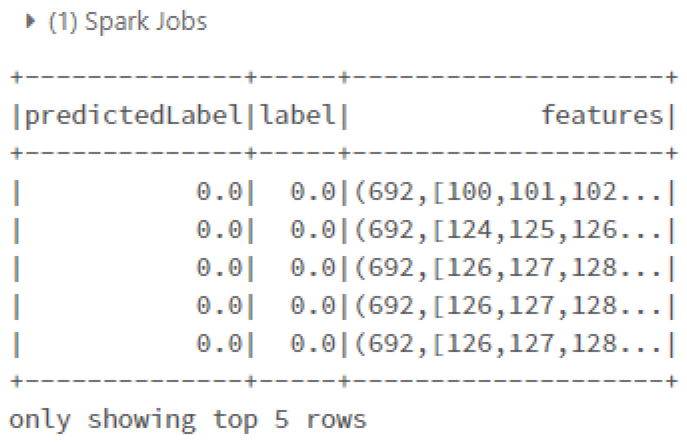
10. **Make predictions.**

    Code-
    ```
    predictions = model.transform(testData)
    ```

11. **Select example rows to display.**

    Code-
    ```
    predictions.select("predictedLabel", "label", "features").show(5)
    ```

    Output-

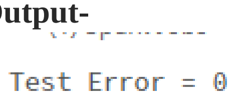


```
▸ (1) Spark Jobs

+--------------+-----+--------------------+
|predictedLabel|label|            features|
+--------------+-----+--------------------+
|           0.0|  0.0|(692,[100,101,102...|
|           0.0|  0.0|(692,[124,125,126...|
|           0.0|  0.0|(692,[126,127,128...|
|           0.0|  0.0|(692,[126,127,128...|
|           0.0|  0.0|(692,[126,127,128...|
+--------------+-----+--------------------+
only showing top 5 rows
```

12. **Select (prediction, true label) and compute test error.**

Code-
```
evaluator = MulticlassClassificationEvaluator(
    labelCol="indexedLabel", predictionCol="prediction",
metricName="accuracy")
accuracy = evaluator.evaluate(predictions)
print("Test Error = %g" % (1.0 - accuracy))
```

Output-

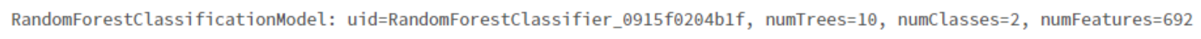


```
Test Error = 0
```

13. **summary only**

Code-
```
rfModel = model.stages[2]
print(rfModel)
```

Output-

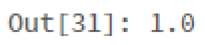


```
RandomForestClassificationModel: uid=RandomForestClassifier_0915f0204b1f, numTrees=10, numClasses=2, numFeatures=692
```

14. **Final**

Code-
Accuracy

Output-



```
Out[31]: 1.0
```

IV)Multilayer Perceptron Classifier

Problem Statement: Performing Decision Tree Classifier on LIBSVM Multi Class Classification dataset.

Code:

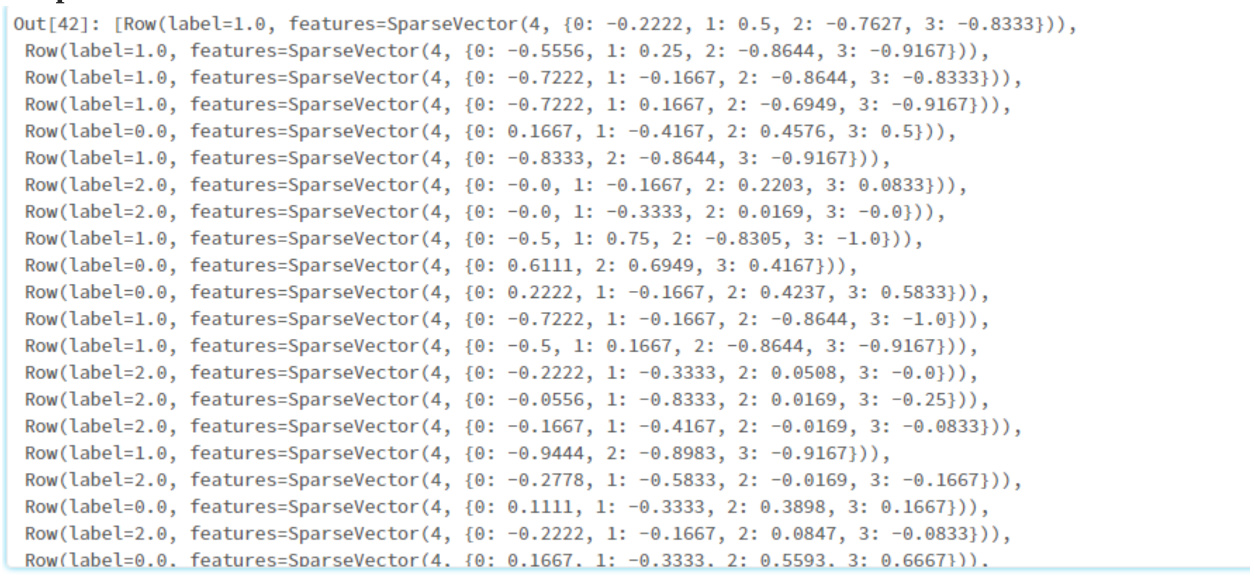
#Import

```
from pyspark.ml.classification import MultilayerPerceptronClassifier
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
```

# Load training data

```
data =
spark.read.format("libsvm").load('dbfs:/FileStore/shared_uploads/muhammedrehanshaikh10101
999@rjcollege.edu.in/sample_multiclass_classification_data.txt')
data.collect()
```

Output:



```
Out[42]: [Row(label=1.0, features=SparseVector(4, {0: -0.2222, 1: 0.5, 2: -0.7627, 3: -0.8333})),
 Row(label=1.0, features=SparseVector(4, {0: -0.5556, 1: 0.25, 2: -0.8644, 3: -0.9167})),
 Row(label=1.0, features=SparseVector(4, {0: -0.7222, 1: -0.1667, 2: -0.8644, 3: -0.8333})),
 Row(label=1.0, features=SparseVector(4, {0: -0.7222, 1: 0.1667, 2: -0.6949, 3: -0.9167})),
 Row(label=0.0, features=SparseVector(4, {0: 0.1667, 1: -0.4167, 2: 0.4576, 3: 0.5})),
 Row(label=1.0, features=SparseVector(4, {0: -0.8333, 2: -0.8644, 3: -0.9167})),
 Row(label=2.0, features=SparseVector(4, {0: -0.0, 1: -0.1667, 2: 0.2203, 3: 0.0833})),
 Row(label=2.0, features=SparseVector(4, {0: -0.0, 1: -0.3333, 2: 0.0169, 3: -0.0})),
 Row(label=1.0, features=SparseVector(4, {0: -0.5, 1: 0.75, 2: -0.8305, 3: -1.0})),
 Row(label=0.0, features=SparseVector(4, {0: 0.6111, 2: 0.6949, 3: 0.4167})),
 Row(label=0.0, features=SparseVector(4, {0: 0.2222, 1: -0.1667, 2: 0.4237, 3: 0.5833})),
 Row(label=1.0, features=SparseVector(4, {0: -0.7222, 1: -0.1667, 2: -0.8644, 3: -1.0})),
 Row(label=1.0, features=SparseVector(4, {0: -0.5, 1: 0.1667, 2: -0.8644, 3: -0.9167})),
 Row(label=2.0, features=SparseVector(4, {0: -0.2222, 1: -0.3333, 2: 0.0508, 3: -0.0})),
 Row(label=2.0, features=SparseVector(4, {0: -0.0556, 1: -0.8333, 2: 0.0169, 3: -0.25})),
 Row(label=2.0, features=SparseVector(4, {0: -0.1667, 1: -0.4167, 2: -0.0169, 3: -0.0833})),
 Row(label=1.0, features=SparseVector(4, {0: -0.9444, 2: -0.8983, 3: -0.9167})),
 Row(label=2.0, features=SparseVector(4, {0: -0.2778, 1: -0.5833, 2: -0.0169, 3: -0.1667})),
 Row(label=0.0, features=SparseVector(4, {0: 0.1111, 1: -0.3333, 2: 0.3898, 3: 0.1667})),
 Row(label=2.0, features=SparseVector(4, {0: -0.2222, 1: -0.1667, 2: 0.0847, 3: -0.0833})),
 Row(label=0.0, features=SparseVector(4, {0: 0.1667, 1: -0.3333, 2: 0.5593, 3: 0.6667})),
```

Code:

# Split the data into train and test

```
splits = data.randomSplit([0.6, 0.4], 1234)
train = splits[0]
test = splits[1]
```

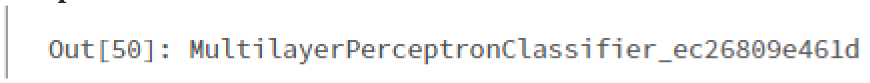
Code:

```
# specify layers for the neural network:
# input layer of size 4 (features), two intermediate of size 5 and 4
# and output of size 3 (classes)
layers = [4, 5, 4, 3]
```

Code:

```
# create the trainer and set its parameters
trainer = MultilayerPerceptronClassifier(maxIter=100, layers=layers, blockSize=128,
seed=1234)
trainer
```

Output:

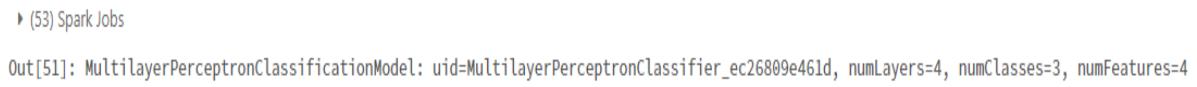


```
Out[50]: MultilayerPerceptronClassifier_ec26809e461d
```

Code:

```
# train the model
model = trainer.fit(train)
model
```

Output:

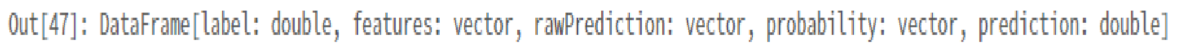


```
▸ (53) Spark Jobs

Out[51]: MultilayerPerceptronClassificationModel: uid=MultilayerPerceptronClassifier_ec26809e461d, numLayers=4, numClasses=3, numFeatures=4
```

Code:

```
# compute accuracy on the test set
result = model.transform(test)
result
```

Output:

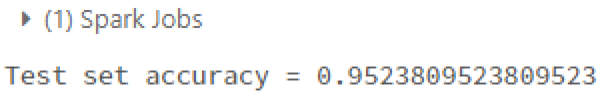


```
Out[47]: DataFrame[label: double, features: vector, rawPrediction: vector, probability: vector, prediction: double]
```

Code:

```
#Make Prediction
predictionAndLabels = result.select("prediction", "label")
evaluator = MulticlassClassificationEvaluator(metricName="accuracy")
print("Test set accuracy = " + str(evaluator.evaluate(predictionAndLabels)))
```

Output:



```
▸ (1) Spark Jobs

Test set accuracy = 0.9523809523809523
```

2) Regression
    i) Linear Regression

PROBLEM STATEMENT:

Performing Linear Regression on Sample Regression dataset.

```
from pyspark.ml.regression import LinearRegression
```

Load Training Data

```
training = spark.read.format("libsvm")\
    .load("data/mllib/sample_linear_regression_data.txt")



lr = LinearRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)
```

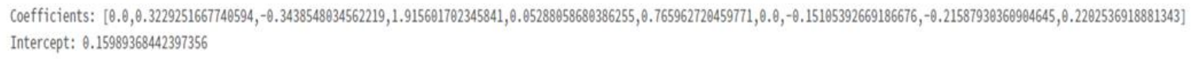
Fit the model

```
lrModel = lr.fit(training)
```

Print the coefficients & intercept for linear regression

```
print("Coefficients: %s" % str(lrModel.coefficients))
print("Intercept: %s" % str(lrModel.intercept))
```

OUTPUT :



```
Coefficients: [0.0,0.3229251667740594,-0.3438548034562219,1.915601702345841,0.05288058680386255,0.765962720459771,0.0,-0.15105392669186676,-0.21587930360904645,0.2202536918881343]
Intercept: 0.15989368442397356
```

Summarize the model over the training set and print out some metrics

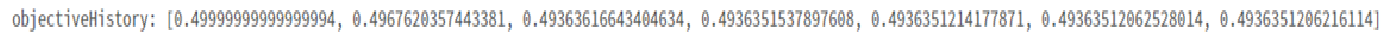
```
trainingSummary = lrModel.summary

print("numIterations: %d" % trainingSummary.totalIterations)
```

OUTPUT :

```
numIterations: 6
```

```
print("objectiveHistory: %s" % str(trainingSummary.objectiveHistory))
trainingSummary.residuals.show()
```

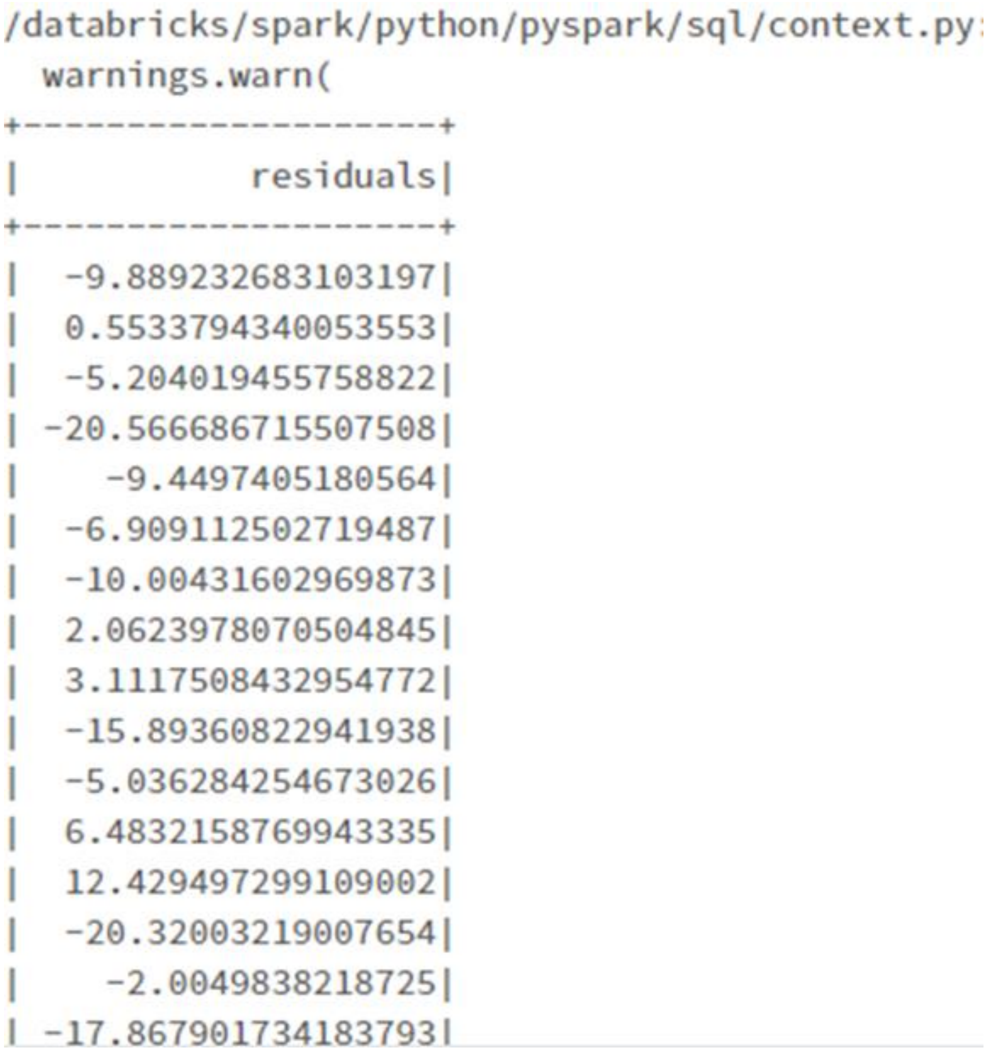
OUTPUT :



```
objectiveHistory: [0.49999999999999994, 0.4967620357443381, 0.49363616643404634, 0.4936351537897608, 0.4936351214177871, 0.49363512062528014, 0.4936351206216114]
```

```
trainingSummary.residuals.show()
```

OUTPUT :

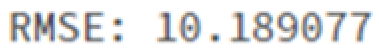


```
/databricks/spark/python/pyspark/sql/context.py
  warnings.warn(
+--------------------+
|           residuals|
+--------------------+
|  -9.889232683103197|
|  0.5533794340053553|
|  -5.204019455758822|
| -20.566686715507508|
|    -9.4497405180564|
|  -6.909112502719487|
|  -10.00431602969873|
|  2.0623978070504845|
|  3.1117508432954772|
|  -15.89360822941938|
|  -5.036284254673026|
|  6.4832158769943335|
|  12.429497299109002|
|  -20.32003219007654|
|    -2.0049838218725|
| -17.867901734183793|
```

Print Root Mean Squared Error

```
print("RMSE: %f" % trainingSummary.rootMeanSquaredError)
```

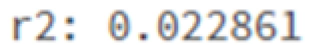
OUTPUT :



```
RMSE: 10.189077
```

```
print("r2: %f" % trainingSummary.r2)
```

OUTPUT :



```
r2: 0.022861
```

ii) Decision Regression :

PROBLEM STATEMENT:

Performing Decision Regression on Sample Regression dataset.

Importing the libraries

```
from pyspark.ml import Pipeline
from pyspark.ml.regression import DecisionTreeRegressor
from pyspark.ml.feature import VectorIndexer
from pyspark.ml.evaluation import RegressionEvaluator
```

Load the data stored in LIBSVM format as a DataFrame

```
data = spark.read.format("libsvm").load("data/mllib/sample_linear_regression_data.txt")
```

Automatically identify categorical features, and index them.
We specify maxCategories so features with > 4 distinct values are treated as continuous.

```
featureIndexer =\
    VectorIndexer(inputCol="features", outputCol="indexedFeatures", maxCategories=4).fit(data)
```

Split the data into training and test sets (30% held out for testing)

```
(trainingData, testData) = data.randomSplit([0.7, 0.3])
```

Train a DecisionTree model.

```
dt = DecisionTreeRegressor(featuresCol="indexedFeatures")
```

Chain indexer and tree in a Pipeline

```
pipeline = Pipeline(stages=[featureIndexer, dt])
```

Train model. This also runs the indexer.

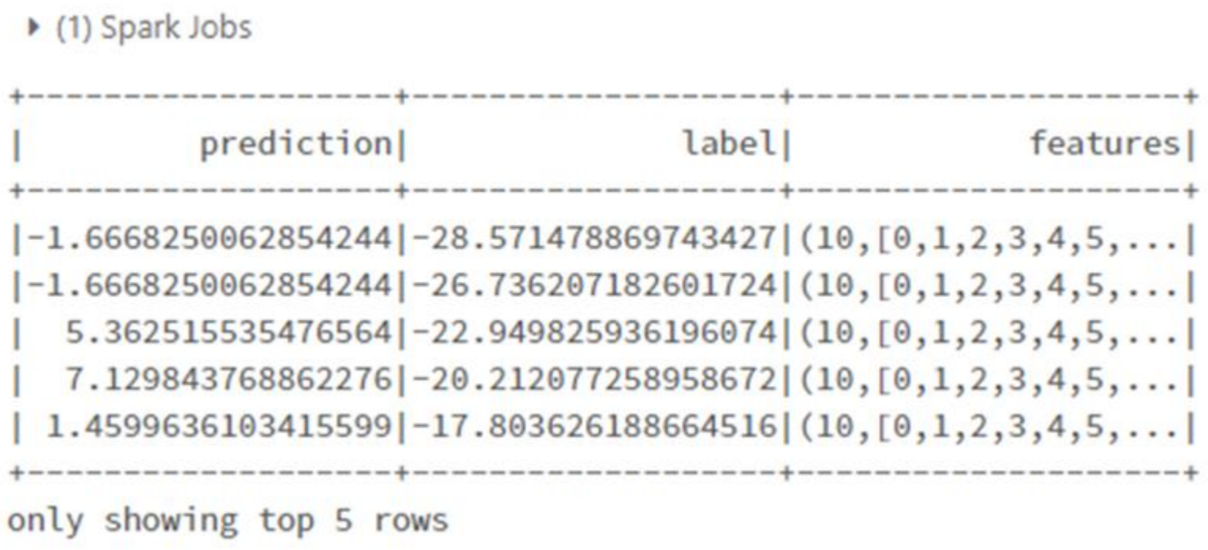
```
model = pipeline.fit(trainingData)
```

Make predictions.

```
predictions = model.transform(testData)
```

Select example rows to display.
```
predictions.select("prediction", "label", "features").show(5)
```

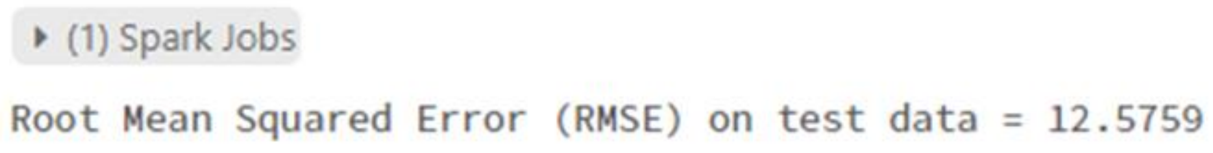
OUTPUT :



```
▸ (1) Spark Jobs

+------------------+-------------------+--------------------+
|        prediction|              label|            features|
+------------------+-------------------+--------------------+
|-1.6668250062854244|-28.571478869743427|(10,[0,1,2,3,4,5,...|
|-1.6668250062854244|-26.736207182601724|(10,[0,1,2,3,4,5,...|
|  5.362515535476564|-22.949825936196074|(10,[0,1,2,3,4,5,...|
|  7.129843768862276|-20.212077258958672|(10,[0,1,2,3,4,5,...|
| 1.4599636103415599|-17.803626188664516|(10,[0,1,2,3,4,5,...|
+------------------+-------------------+--------------------+
only showing top 5 rows
```

Select (prediction, true label) and compute test error
```
evaluator = RegressionEvaluator(
    labelCol="label", predictionCol="prediction", metricName="rmse")
rmse = evaluator.evaluate(predictions)
print("Root Mean Squared Error (RMSE) on test data = %g" % rmse)
```

OUTPUT :

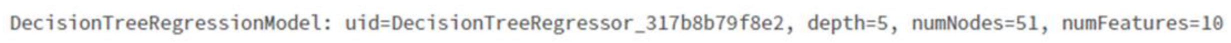


```
▸ (1) Spark Jobs

Root Mean Squared Error (RMSE) on test data = 12.5759
```

```
treeModel = model.stages[1]
```

Summary only
```
print(treeModel)
```

OUTPUT :



```
DecisionTreeRegressionModel: uid=DecisionTreeRegressor_317b8b79f8e2, depth=5, numNodes=51, numFeatures=10
```

Practical 8 - Implementation of Clustering Algorithms Using Big Data.

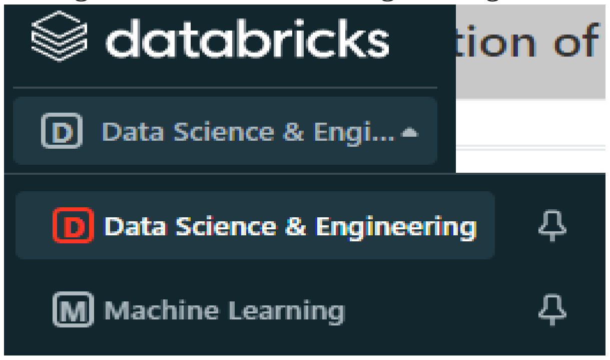
a) K-Means
b) LDA

Implementation of Clustering Algorithms Using Big Data
Documentation referred for this pracs:"https://spark.apache.org/docs/latest/ml-clustering.html"

Description
#"k-means algorithm for clustering"
#k-means is one of the most commonly used clustering algorithms that clusters the data points into a predefined number of clusters. The MLlib implementation includes a parallelized variant of the k-means++ method called kmeans||.
#KMeans is implemented as an Estimator and generates a KMeansModel as the base model.

In this practical we have basically used two algorithms i.e K-means and LDA

Step1:Create cluster and notebook
The notebook need not be in machine learning i.e it will work in the default setting of data science and engineering.

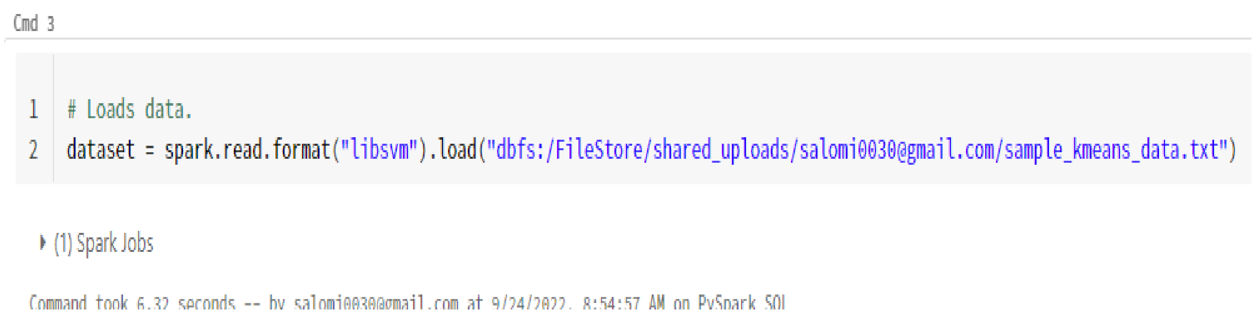


Step2:after creating notebook
Import required libraries for k-means

```
from pyspark.ml.clustering import KMeans
from pyspark.ml.evaluation import ClusteringEvaluator
```

```
Command took 0.03 seconds -- by salomi0030@gmail.com at 9/24/2022, 9:09:00 AM on PySpark SQL
```

Step3:Loading dataset
Search for:
/sample_kmeans_data.txt
Copy the data content from :
https://github.com/apache/spark/blob/master/data/mllib/sample_kmeans_data.txt

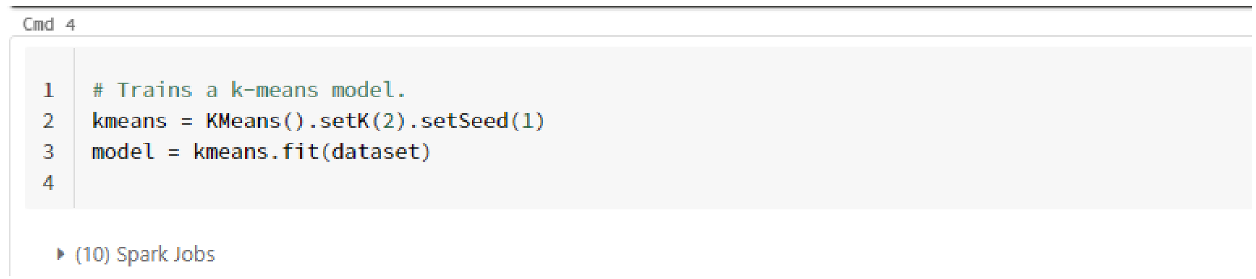

Cmd 3

```
# Loads data.
dataset = spark.read.format("libsvm").load("dbfs:/FileStore/shared_uploads/salomi0030@gmail.com/sample_kmeans_data.txt")
```

▸ (1) Spark Jobs

```
Command took 6.32 seconds -- by salomi0030@gmail.com at 9/24/2022, 8:54:57 AM on PySpark SQL
```

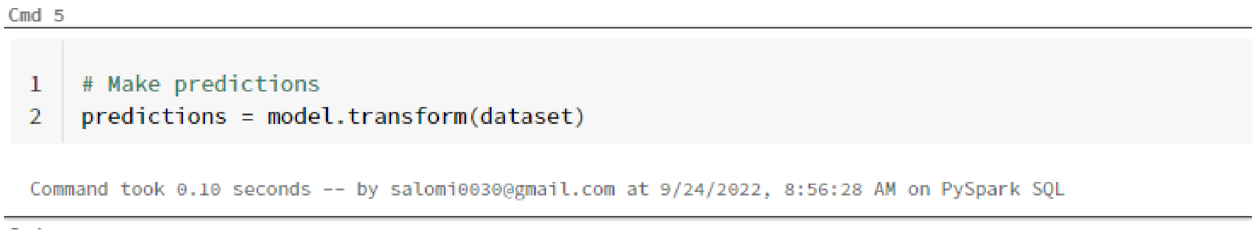
Step3:Training K-means model
kmeans= KMeans().setK(2).setSeed(1)
model=Kmeans.fit(dataset)


Cmd 4

```
# Trains a k-means model.
kmeans = KMeans().setK(2).setSeed(1)
model = kmeans.fit(dataset)

```

▸ (10) Spark Jobs

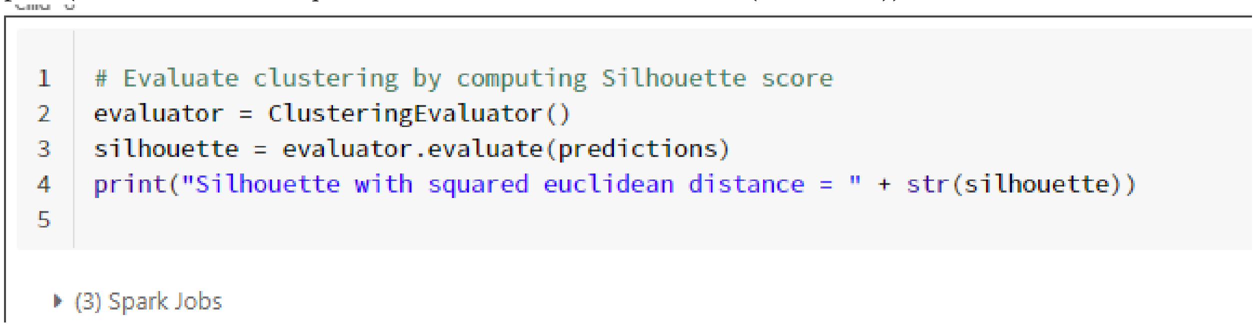
Step4:# Make predictions
predictions = model.transform(dataset)


Cmd 5

```
# Make predictions
predictions = model.transform(dataset)
```

```
Command took 0.10 seconds -- by salomi0030@gmail.com at 9/24/2022, 8:56:28 AM on PySpark SQL
```

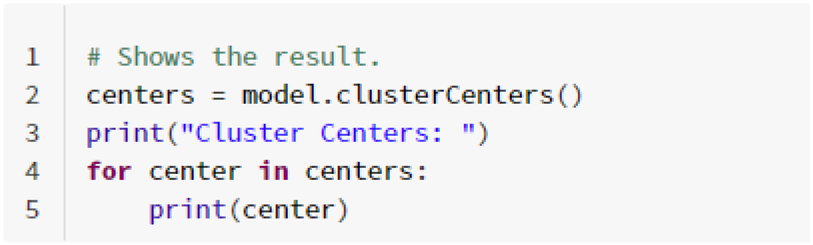
Step5:## Evaluate clustering by computing Silhouette score
evaluator = ClusteringEvaluator()
silhouette = evaluator.evaluate(predictions)
print("Silhouette with squared euclidean distance = " + str(silhouette))



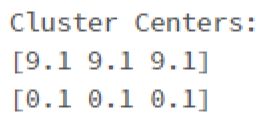
```
# Evaluate clustering by computing Silhouette score
evaluator = ClusteringEvaluator()
silhouette = evaluator.evaluate(predictions)
print("Silhouette with squared euclidean distance = " + str(silhouette))

```

▸ (3) Spark Jobs

Step5:## Shows the result.
centers = model.clusterCenters()
print("Cluster Centers: ")
for center in centers:
    print(center)



```
# Shows the result.
centers = model.clusterCenters()
print("Cluster Centers: ")
for center in centers:
    print(center)
```

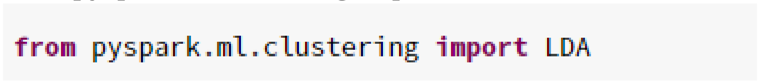


```
Cluster Centers:
[9.1 9.1 9.1]
[0.1 0.1 0.1]
```

#performing clustering now using LDA

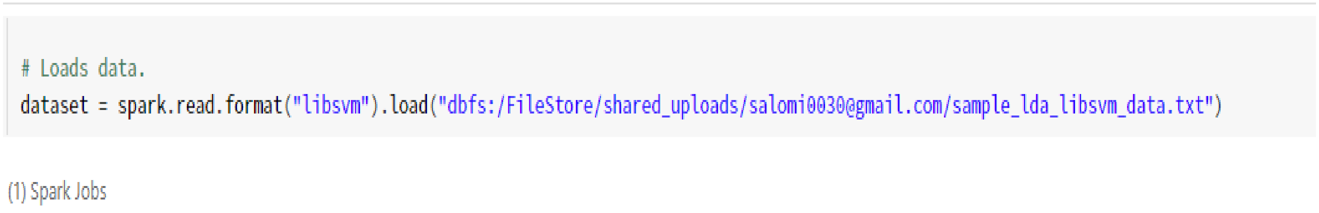
Latent Dirichlet allocation (LDA):
LDA is implemented as an Estimator that supports both EMLDAOptimizer and OnlineLDAOptimizer, and generates a LDAModel as the base model. Expert users may cast a LDAModel generated by EMLDAOptimizer to a DistributedLDAModel if needed.

Step1:#importing
from pyspark.ml.clustering import LDA



```
from pyspark.ml.clustering import LDA
```

Step2:# Loads data.
dataset=spark.read.format("libsvm").load("dbfs:/FileStore/shared_uploads/salomi0030@gmail.com/sample_lda_libsvm_data.txt")



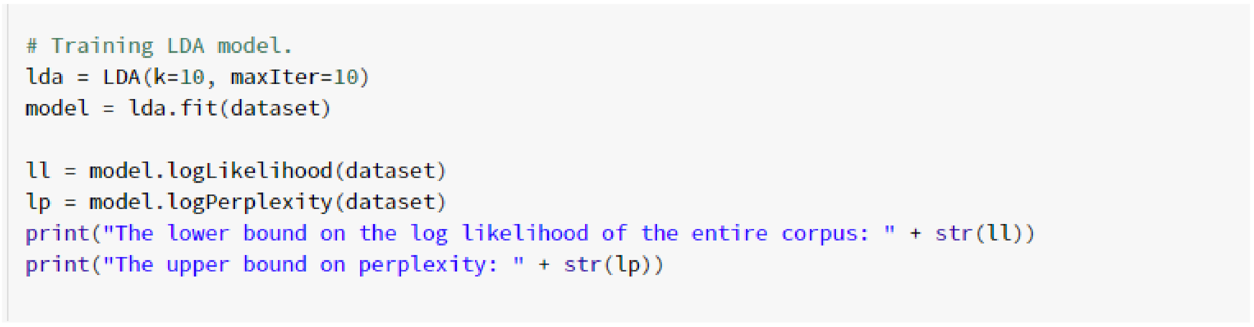
```
# Loads data.
dataset = spark.read.format("libsvm").load("dbfs:/FileStore/shared_uploads/salomi0030@gmail.com/sample_lda_libsvm_data.txt")
```

(1) Spark Jobs

Step3:# Training LDA model.
lda = LDA(k=10, maxIter=10)
model = lda.fit(dataset)

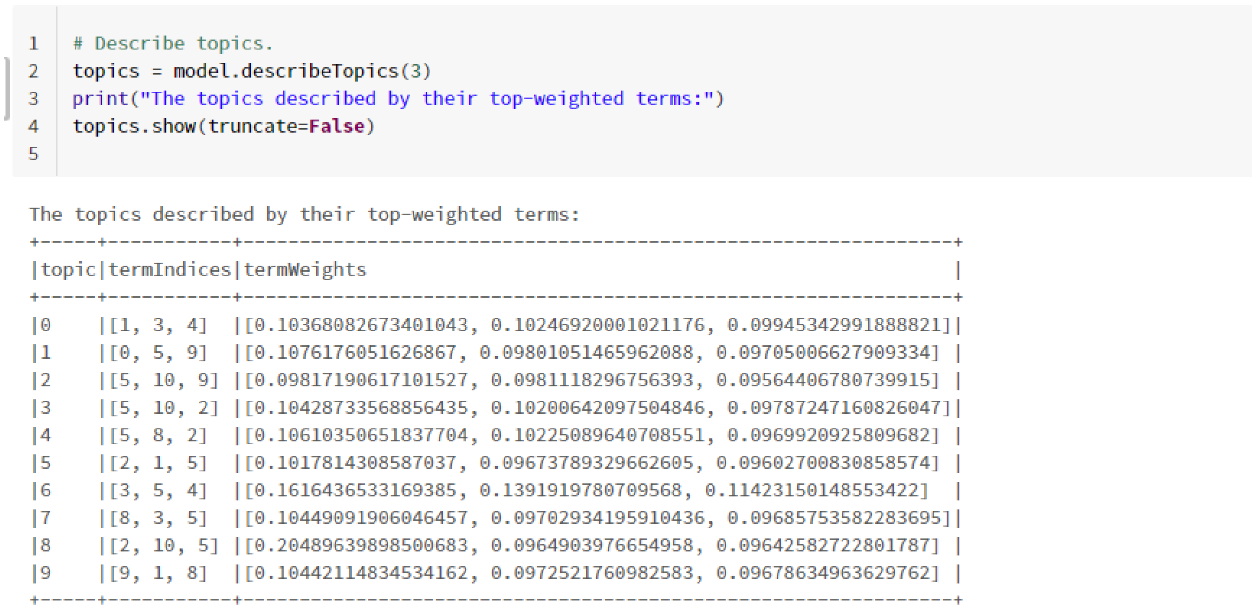
ll = model.logLikelihood(dataset)
lp = model.logPerplexity(dataset)
print("The lower bound on the log likelihood of the entire corpus: " + str(ll))

print("The upper bound on perplexity: " + str(lp))



```
# Training LDA model.
lda = LDA(k=10, maxIter=10)
model = lda.fit(dataset)

ll = model.logLikelihood(dataset)
lp = model.logPerplexity(dataset)
print("The lower bound on the log likelihood of the entire corpus: " + str(ll))
print("The upper bound on perplexity: " + str(lp))
```

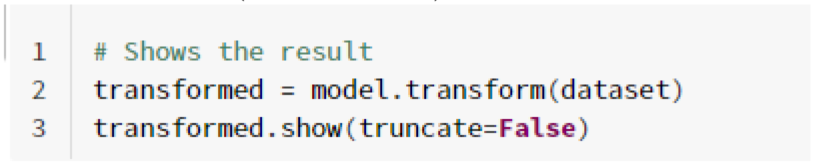
Step4:# Describe topics.
topics = model.describeTopics(3)
print("The topics described by their top-weighted terms:")
topics.show(truncate=False)



```
# Describe topics.
topics = model.describeTopics(3)
print("The topics described by their top-weighted terms:")
topics.show(truncate=False)

```

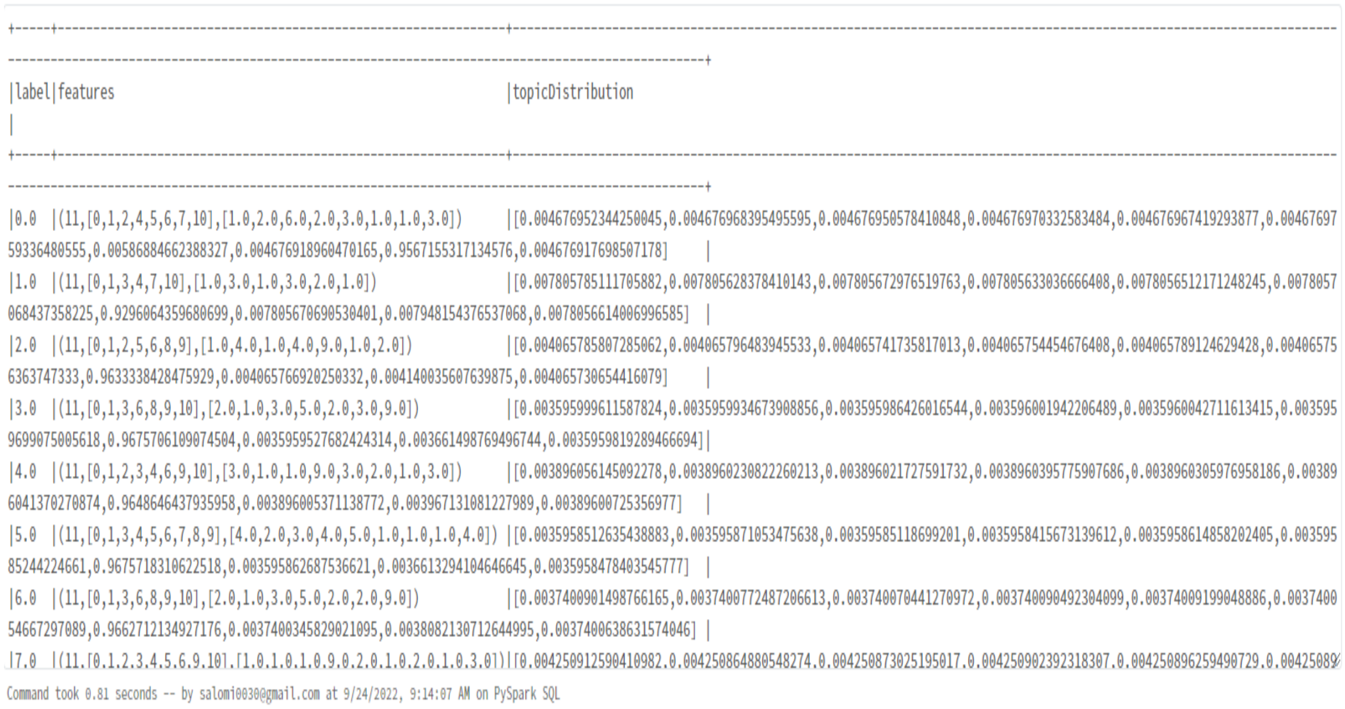
```
The topics described by their top-weighted terms:
+-----+-----------+---------------------------------------------------------------+
|topic|termIndices|termWeights                                                    |
+-----+-----------+---------------------------------------------------------------+
|0    |[1, 3, 4]  |[0.10368082673401043, 0.10246920001021176, 0.09945342991888821]|
|1    |[0, 5, 9]  |[0.1076176051626867, 0.09801051465962088, 0.09705006627909334] |
|2    |[5, 10, 9] |[0.09817190617101527, 0.098111829675 6393, 0.09564406780739915] |
|3    |[5, 10, 2] |[0.10428733568856435, 0.10200642097504846, 0.09787247160826047]|
|4    |[5, 8, 2]  |[0.10610350651837704, 0.10225089640708551, 0.09699209258096 82] |
|5    |[2, 1, 5]  |[0.1017814308587037, 0.09673789329662605, 0.09602700830858574] |
|6    |[3, 5, 4]  |[0.1616436533169385, 0.1391919780709568, 0.11423150148553422]  |
|7    |[8, 3, 5]  |[0.10449091906046457, 0.09702934195910436, 0.09685753582283695]|
|8    |[2, 10, 5] |[0.20489639898500683, 0.0964903976654958, 0.09642582722801787] |
|9    |[9, 1, 8]  |[0.10442114834534162, 0.0972521760982583, 0.09678634963629762] |
+-----+-----------+---------------------------------------------------------------+
```

Step5:# Shows the result
transformed = model.transform(dataset)
transformed.show(truncate=False)



```
# Shows the result
transformed = model.transform(dataset)
transformed.show(truncate=False)
```

Result for LDA:



```
+-----+------------------------------------------------------------+--------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------+
|label|features                                                    |topicDistribution
|
+-----+------------------------------------------------------------+--------------------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------+
|0.0  |(11,[0,1,2,4,5,6,7,10],[1.0,2.0,6.0,2.0,3.0,1.0,1.0,3.0])      |[0.004676952344250045,0.004676968395495595,0.004676950578410848,0.004676970332583484,0.004676967419293877,0.00467697
59336480555,0.005868846623883 27,0.004676918960470165,0.9567155317134576,0.004676917698507178]      |
|1.0  |(11,[0,1,3,4,7,10],[1.0,3.0,1.0,3.0,2.0,1.0])                  |[0.007805785111705882,0.007805628378410143,0.007805672976519763,0.007805633036666408,0.007805651217124824 5,0.0078057
068437358225,0.9296064359680699,0.007805670690530401,0.007948154376537068,0.007805661400699 6585]  |
|2.0  |(11,[0,1,2,5,6,8,9],[1.0,4.0,1.0,4.0,9.0,1.0,2.0])             |[0.004065785807285062,0.004065796483945533,0.004065741735817013,0.004065754454676408,0.004065789124629428,0.00406575
6363747333,0.9633338428475929,0.004065766920250332,0.004140035607639875,0.004065730654416079]      |
|3.0  |(11,[0,1,3,6,8,9,10],[2.0,1.0,3.0,5.0,2.0,3.0,9.0])            |[0.003595999611587824,0.003595993467390885 6,0.003595986426016544,0.003596001942206489,0.003596004271161341 5,0.003595
9699075005618,0.9675706109074504,0.003595952768242431 4,0.003661498769496744,0.003595981928946669 4]|
|4.0  |(11,[0,1,2,3,4,6,9,10],[3.0,1.0,1.0,9.0,3.0,2.0,1.0,3.0])      |[0.003896056145092278,0.0038960230822260213,0.003896021727591732,0.003896039577590768 6,0.0038960305976958186,0.00389
6041370270874,0.964864643793595 8,0.003896005371138772,0.003967131081227989,0.00389600725356977]    |
|5.0  |(11,[0,1,3,4,5,6,7,8,9],[4.0,2.0,3.0,4.0,5.0,1.0,1.0,1.0,4.0]) |[0.003595851263543888 3,0.003595871053475638,0.003595851186992 01,0.003595841567313961 2,0.003595861485820240 5,0.003595
85244224661,0.9675718310622518,0.003595862687536621,0.0036613294104646645,0.0035958478403545777]  |
|6.0  |(11,[0,1,3,6,8,9,10],[2.0,1.0,3.0,5.0,2.0,2.0,9.0])            |[0.003740090149876616 5,0.003740077248720661 3,0.003740070441270972,0.003740090492304099,0.003740091990488 86,0.0037400
54667297089,0.9662712134927176,0.0037400345829021095,0.003808213071264499 5,0.003740063863157404 6] |
|7.0  |(11,[0,1,2,3,4,5,6,9,10],[1.0,1.0,1.0,9.0,2.0,1.0,2.0,1.0,3.0])|[0.004250912590410982,0.004250864880548274,0.004250873025195017,0.004250902392318307,0.004250896259490729,0.00425089
```

Command took 0.81 seconds -- by salomi0030@gmail.com at 9/24/2022, 9:14:07 AM on PySpark SQL

Result from LDA:

Practical 9 - PySpark Structured Streaming

Aim:- How to perform live data streaming using Pyspark
Structured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine. You can express your streaming computation the same way you would express a batch computation on static data. The Spark SQL engine will take care of running it incrementally and continuously and updating the final result as streaming data continues to arrive.

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import explode
from pyspark.sql.functions import split

spark = SparkSession \
    .builder \
    .appName("StructuredNetworkWordCount") \
    .getOrCreate()
# Create DataFrame representing the stream of input lines from connection to localhost:9999
lines = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()

# Split the lines into words
words = lines.select(
   explode(
       split(lines.value, " ")
   ).alias("word")
)

# Generate running word count
wordCounts = words.groupBy("word").count()
# Start running the query that prints the running counts to the console
query = wordCounts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

query.awaitTermination()
```

Practical 10 - Spark Graphx

Bhavesh singh, sunil yadav, Akshay Waje

a) Understanding Spark Graphx
b) Social Networking Analysis using PySpark DataFrame

Solution:

a)Understanding Spark Graphx :

```
#Import GraphFrame Packages
from functools import reduce
from pyspark.sql.functions import col, lit, when
from graphframes import *

vertices = sqlContext.createDataFrame([
 ("a", "Alice", 34),
 ("b", "Bob", 36),
 ("c", "Charlie", 30),
 ("d", "David", 29),
 ("e", "Esther", 32),
 ("f", "Fanny", 36),
 ("g", "Gabby", 60)], ["id", "name", "age"])

#Create Edges

 edges = sqlContext.createDataFrame([
 ("a", "b", "friend"),
 ("b", "c", "follow"),
 ("c", "b", "follow"),
 ("f", "c", "follow"),
 ("e", "f", "follow"),
 ("e", "d", "friend"),
 ("d", "a", "friend"),
 ("a", "e", "friend")
], ["src", "dst", "relationship"])

#Create Graph from above Vertices and Edges

g = GraphFrame(vertices, edges)
print(g)
```

```
# This example graph also comes with the GraphFrames package.
from graphframes.examples import Graphs
same_g = Graphs(sqlContext).friends()
print(same_g)


#Display List of Vertices
display(g.vertices)


#Display Edges

display(g.edges)

#Display the incoming degree of the vertices
display(g.inDegrees)

#Display the outgoing degree of the vertices:
display(g.outDegrees)

#Display the degree of vertices
display(g.degrees)


#You can run queries directly on the vertices DataFrame.
#For example, we can find the age of the youngest person in the graph:
youngest = g.vertices.groupBy().min("age")
display(youngest)

#Likewise, you can run queries on the edges of DataFrame.
#For example, let's count the number of 'follow' relationships in the graph:
numFollows = g.edges.filter("relationship = 'follow'").count()
print("The number of follow edges is", numFollows)

numFollows = g.edges.filter("relationship = 'friend'").count()
print("The number of friends edges is", numFollows)

motifs = g.find("(a)-[e]->(b); (b)-[e2]->(a)")
display(motifs)

filtered = motifs.filter("b.age > 30 or a.age > 30")

# Find chains of 4 vertices.
```

```
        chain4 = g.find("(a)-[ab]->(b); (b)-[bc]->(c); (c)-[cd]->(d)")
        display(chain4)

        # Query on sequence, with state (cnt)
        #  (a) Define method for updating state given the next element of the motif.
        def cumFriends(cnt, edge):
         relationship = col(edge)["relationship"]
         return when(relationship == "friend", cnt + 1).otherwise(cnt)
        #  (b) Use sequence operation to apply method to sequence of elements in motif.
        #   In this case, the elements are the 3 edges.
        edges = ["ab", "bc", "cd"]
        numFriends = reduce(cumFriends, edges, lit(0))


        chainWith2Friends2 = chain4.withColumn("num_friends",
numFriends).where(numFriends >= 2)
        display(chainWith2Friends2)

        g2 = g.filterEdges("relationship = 'friend'").filterVertices("age >
30").dropIsolatedVertices()

        display(g2.vertices)
        display(g2.edges)
```