**PYSPARK Datacamp**

**1. Fundamentals of Big Data**

00:00 - 00:10

Welcome to the first video of Big Data fundamentals via PySpark course. My name is Upendra Devisetty and I am a Science Analyst at CyVerse. Let's get started.

**2. What is Big Data?**

00:10 - 00:32

What exactly is Big Data? There is no single definition of Big Data because projects, vendors, practitioners, and business professionals use it quite differently. According to Wikipedia - Big data is a term used to refer to the study and applications of data sets that are too complex for traditional data-processing software. There are three

**3. The 3 V's of Big Data**

00:32 - 00:53

Vs of Big data that are used to describe its characteristics. They are volume, velocity, and variety. Volume refers to the size of data. Variety refers to different sources and formats of data. Velocity is the speed at which data is generated and available for processing. Now let's take a look at some

**4. Big Data concepts and Terminology**

00:53 - 01:30

of the concepts and terminology of Big Data. Clustered computing is the pooling of resources of multiple machines to complete jobs. Parallel computing is a type of computation in which many calculations are carried out simultaneously. A distributed computing involves nodes or networked computers that run jobs in parallel. Batch processing refers to the breaking data into smaller pieces and running each piece on an individual machine. Real-time processing demands that information is processed and made ready immediately. There are two popular

**5. Big Data processing systems**

01:30 - 02:03

frameworks for Big Data processing. The first is the highly successful Hadoop/MapReduce framework. Hadoop/MapReduce framework is open source and scalable framework for batch data. The second is the most popular Apache Spark which is a parallel framework for storing and processing of Big Data across clustered computers. It is also open source and is suited for both batch and real-time data processing. In this course, you'll learn about Apache Spark. Let's talk about the main

**6. Features of Apache Spark framework**

02:03 - 02:40

features of Apache Spark. Spark distributes data and computation across multiple computers executing complex multi-stage applications such as machine learning. Spark runs most computations in memory and thereby provides better performance for applications such as interactive data mining. Spark helps to run an application up to 100 times faster in memory, and 10 times faster when running on disk. Spark is mainly written in Scala language but also have support for Java, Python, R, and SQL. Apache Spark is a

**7. Apache Spark Components**

02:40 - 03:34

powerful alternative to Hadoop MapReduce, with rich features like machine learning, real-time stream processing, and graph computations. At the center of the ecosystem is the Spark Core which contains the basic functionality of Spark. The rest of Spark’s libraries are built on top of it. First is Spark SQL, which is a library for processing structured and semi-structured data in Python, Java, and Scala. The second is MLlib, which is a library of common machine learning algorithms. The third component is GraphX, which is a collection of algorithms and tools for manipulating graphs and performing parallel graph computations. Finally, Spark Streaming is a scalable, high-throughput processing library for real-time data. In this course, you'll learn about SparkSQL and MLlib.

**8. Spark modes of deployment**

03:34 - 04:10

Spark can be run on two modes. The first is the local mode where you can run Spark on a single machine such as your laptop. The local mode is very convenient for testing, debugging and demonstration purposes. The second is the cluster mode where Spark is run on a cluster. The cluster mode is mainly used for production. The development workflow is that you start on local mode and transition to cluster mode. During the transition from local to cluster mode, no code change is necessary. In this course, you'll be using local mode.

**9. Coming up next - PySpark**

04:10 - 04:15

In the next video, you'll learn about PySpark which is the Python API for Spark.

# The 3 V's of Big Data

Which of the following is **NOT** considered as one of the three Vs of Big Data?

##### Answer the question

**50XP**

#### Possible Answers

Volume

Velocity

**Variation** Correct! Variation is not considered as one of the 3 V's of Big Data.

Variety

LECTURE

**1. PySpark: Spark with Python**

00:00 - 00:17

In the last video, you were introduced to Apache Spark which is a fast and general-purpose framework for Big data processing. Apache Spark provides high-level APIs in Scala, Java, Python, and R. In this video, you'll learn about PySpark which is Spark's version of Python.

**2. Overview of PySpark**

00:17 - 00:42

Apache Spark is originally written in Scala programming language. To support Python with Spark, PySpark was developed. Unlike previous versions, the newest version of PySpark provides computation power similar to Scala. APIs in PySpark are similar to Pandas & Scikit-learn Python packages. Thus, the entry level barrier to PySpark is very low for beginners.

**3. What is Spark shell?**

00:42 - 01:26

Spark comes with interactive shells that enable ad-hoc data analysis. Spark shell is an interactive environment through which one can access Spark's functionality quickly and conveniently. Spark shell is particularly helpful for fast interactive prototyping before running the jobs on clusters. Unlike most other shells, Spark shell allow you to interact with data that is distributed on disk or in memory across many machines, and Spark takes care of automatically distributing this processing. Spark provides the shell in three programming languages: spark-shell for Scala, PySpark for Python and sparkR for R. PySpark

**4. PySpark shell**

01:26 - 01:51

shell is the Python-based command line tool to develop Spark's interactive applications in Python. PySpark helps data scientists interface with Spark data structures in Apache Spark and Python. Similar to Scala Shell, Pyspark shell has been augmented to support connecting to a cluster. In this course, you'll use PySpark Shell. In order

**5. Understanding SparkContext**

01:51 - 02:31

to interact with Spark using PySpark shell, you need an entry point. SparkContext is an entry point to interact with underlying Spark functionality. Before understanding SparkContext, let’s understand what an entry point is. An entry point is where control is transferred from the Operating system to the provided program. In simpler terms, it's like a key to your house. Without the key you cannot enter the house, similarly, without an entry point, you cannot run any PySpark jobs. You can access the SparkContext in the PySpark shell as a variable named sc. Now let's take a look at some of the important attributes of SparkContext.

**6. Inspecting SparkContext**

02:31 - 03:20

The first is the version. This attribute shows the version of spark that you are currently running. In this example, sc dot version shows the version of spark that is running in this course's environment. The second is the Python version. This attribute shows the version of Python that Spark is currently using. In this example, sc dot pythonVer shows the version of Python that is running in this course's environment. The final attribute is the Master. Master is the URL of the cluster or “local” string to run in local mode. In this example, sc dot master returns local meaning the SparkContext acts as a master on a local node using all available threads on the computer where it is running. You can load your raw data

**7. Loading data in PySpark**

03:20 - 03:47

into PySpark using SparkContext by two different methods. The first is the SparkContext’s parallelize method on a list. For example, here is how to create parallelize collections holding the numbers 1 to 5. The second is the SparkContext’s textFile method on a file. For example, here’s a way to load a text file named "test-dot-txt" using SparkContext's textFile method. Now that you

**8. Let's practice**

03:47 - 03:53

understand PySpark, let's write your first Spark code in PySpark shell.

## Exercise

# Understanding SparkContext

A SparkContext represents the entry point to Spark functionality. It's like a key to your car. When we run any Spark application, a driver program starts, which has the main function and your SparkContext gets initiated here. PySpark automatically creates a SparkContext for you in the PySpark shell (so you don't have to create it by yourself) and is exposed via a variable sc.

In this simple exercise, you'll find out the attributes of the SparkContext in your PySpark shell which you'll be using for the rest of the course.

## Instructions

* Print the version of SparkContext in the PySpark shell.
* Print the Python version of SparkContext in the PySpark shell.
* What is the master of SparkContext in the PySpark shell?

# Print the version of SparkContext

print("The version of Spark Context in the PySpark shell is", sc.\_\_\_\_)

# Print the Python version of SparkContext

print("The Python version of Spark Context in the PySpark shell is", \_\_\_\_.pythonVer)

# Print the master of SparkContext

print("The master of Spark Context in the PySpark shell is", \_\_\_\_.\_\_\_\_)

Welcome to

\_\_\_\_ \_\_

/ \_\_/\_\_ \_\_\_ \_\_\_\_\_/ /\_\_

\_\ \/ \_ \/ \_ `/ \_\_/ '\_/

/\_\_ / .\_\_/\\_,\_/\_/ /\_/\\_\ version 3.2.0

/\_/

Using Python version 3.9.7 (default, Sep 10 2021 00:03:59)

Spark context Web UI available at http://2aa79d27-14de-4962-a3ac-cc63b390d4d7.sessions.sessions.svc.cluster.local:4040

Spark context available as 'sc' (master = local[\*], app id = local-1688892227500).

SparkSession available as 'spark'.

# Print the version of SparkContext

print("The version of Spark Context in the PySpark shell is", sc.version)

# Print the Python version of SparkContext

print("The Python version of Spark Context in the PySpark shell is", sc.pythonVer)

# Print the master of SparkContext

print("The master of Spark Context in the PySpark shell is", sc.master)

The version of Spark Context in the PySpark shell is 3.2.0

The Python version of Spark Context in the PySpark shell is 3.9

The master of Spark Context in the PySpark shell is local[\*]

# Print the version of SparkContext

print("The version of Spark Context in the PySpark shell is", sc.version)

# Print the Python version of SparkContext

print("The Python version of Spark Context in the PySpark shell is", sc.pythonVer)

# Print the master of SparkContext

print("The master of Spark Context in the PySpark shell is", sc.master)

You just ran your first Spark code. Congratulations!

## Exercise

## Exercise

# Interactive Use of PySpark

Spark comes with an interactive Python shell in which PySpark is already installed in it. PySpark shell is useful for basic testing and debugging and it is quite powerful. The easiest way to demonstrate the power of PySpark’s shell is to start using it. In this example, you'll load a simple list containing numbers ranging from 1 to 100 in the PySpark shell.

The most important thing to understand here is that we are not creating any SparkContext object because PySpark automatically creates the SparkContext object named sc, by default in the PySpark shell.

## Instructions

* Create a Python list named numb containing the numbers 1 to 100.
* Load the list into Spark using Spark Context's parallelize method and assign it to a variable spark\_data.

# Create a Python list of numbers from 1 to 100 numb = range(1,101) # Load the list into PySpark spark\_data = sc.parallelize(numb)

# Create a Python list of numbers from 1 to 100

numb = range(1,101)

# Load the list into PySpark

spark\_data = sc.parallelize(numb)

Good job! For the rest of this course, you'll have a SparkContext called sc available in PySpark shell.

## Exercise

# Loading data in PySpark shell

In PySpark, we express our computation through operations on distributed collections that are automatically parallelized across the cluster. In the previous exercise, you have seen an example of loading a list as parallelized collections and in this exercise, you'll load the data from a local file in PySpark shell.

Remember you already have a SparkContext sc and file\_path variable (which is the path to the README.md file) already available in your workspace.

## Instructions

* Load a local text file README.md in PySpark shell.

# Load a local file into PySpark shell

lines = sc.\_\_\_\_(file\_path)

# Load a local file into PySpark shell lines = sc.textFile(file\_path)

# Load a local file into PySpark shell

lines = sc.textFile(file\_path)

Wonderful! SparkContext's textFile() method is quite powerful for creating distributed collections of unstructured data which you'll see in the next chapter.

**LECTURE**

**1. Use of Lambda function in python - filter()**

00:00 - 00:12

Understanding PySpark becomes a lot easier if we understand functional programming principles in Python. In this video, let's review some of the Python functions such as lambda, map and filter.

**2. What are anonymous functions in Python?**

00:12 - 00:55

Python supports the creation of anonymous functions. That is functions that are not bound to a name at runtime, using a construct called the lambda. lambda functions are very powerful, well integrated into Python, and are often used in conjunction with typical functional concepts like map and filter functions. Like def, the lambda creates a function to be called later in the program. However, it returns the function instead of assigning it to a name. This is why lambdas are known as anonymous functions. In practice, they are used as a way to inline a function definition, or to defer execution of a code. Lambda functions can be used

**3. Lambda function syntax**

00:55 - 01:39

whenever function objects are required. They can have any number of arguments but only one expression and the expression is evaluated and returned. The general syntax of lambda function is shown here. Here is an example of a lambda function. In this example, lambda x: x \* 2, x is the argument and x \* 2 is the expression that gets evaluated and returned. This function has no name. It returns a function object which is assigned to the identifier "double" here. Applying the lambda function to a number such as 3 returns 6 which is the double of the original number. Let's take a look at the differences between

**4. Difference between def vs lambda functions**

01:39 - 02:18

def and lambda. Here is the Python code to illustrate cube of a number showing the difference between normal Python function using def and anonymous function using lambda. As you can see, both def and lambda do exactly the same. The main difference is that the lambda definition does not include a return statement and it always contains an expression which is returned. Also note that we can put a lambda definition anywhere a function is expected, and we don't have to assign it to a variable at all, unlike normal Python function using def. We use lambda functions when we

**5. Use of Lambda function in Python - map()**

02:18 - 02:58

require a nameless function for a short period of time. Most of the times we use lambdas with built-in functions like map and filter. The map function is called with all the items in the list and a new list is returned which contains items returned by that function for each item. The general syntax of map function is shown here. It takes in a function and a list. Here is an example of map function with lambda to add the number 2 to all the items in a list. The result indicates that the number 2 is added to 1, 2, 3, 4 resulting in 3, 4, 5, 6. The filter

**6. Use of Lambda function in python - filter()**

02:58 - 03:39

function in Python takes in a function and a list as arguments. The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True. Here is the general syntax of filter function in Python. Similar to map, it takes a function and a list as arguments. Here is an example use of filter with lambda to filter out only odd numbers from a list. As shown in the example, filtering the items list containing number 1, 2, 3, 4 resulted in 1 and 3 which are the only odd numbers for the input list. Lambda functions

**7. Let's practice**

03:39 - 03:48

are incredibly useful and before going deep into Pyspark, let's practice some lambda functions in PySpark shell.

LECTURE

# Use of lambda() with map()

The map() function in Python returns a list of the results after applying the given function to each item of a given iterable (list, tuple etc.). The general syntax of map() function is map(fun, iter). We can also use lambda functions with map(). The general syntax of map() function with lambda() is map(lambda <argument>:<expression>, iter). Refer to slide 5 of video 1.7 for general help of map() function with lambda().

In this exercise, you'll be using lambda function inside the map() built-in function to square all numbers in the list.

## Instructions

* Print my\_list which is available in your environment.
* Square each item in my\_list using map() and lambda().

Print the result of map function.

# Print my\_list in the console

print("Input list is", \_\_\_\_)

# Square all numbers in my\_list

squared\_list\_lambda = list(\_\_\_\_(lambda x: \_\_\_\_, my\_list))

# Print my\_list in the console

print("Input list is", my\_list)

# Square all numbers in my\_list

squared\_list\_lambda = list(map(lambda x: x\*\*2, my\_list))

# Print the result of the map function

print("The squared numbers are", squared\_list\_lambda)

Input list is [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

The squared numbers are [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

# Print the result of the map function

print("The squared numbers are", \_\_\_\_)

# Print my\_list in the console

print("Input list is", my\_list)

# Square all numbers in my\_list

squared\_list\_lambda = list(map(lambda x: x\*\*2, my\_list))

# Print the result of the map function

print("The squared numbers are", squared\_list\_lambda)

Good start with Python lambdas, way to go! Notice how each number got squared.

## Exercise

# Use of lambda() with filter()

Another function that is used extensively in Python is the filter() function. The filter() function in Python takes in a function and a list as arguments. The general syntax of the filter() function is filter(function, list\_of\_input). Similar to the map(), filter() can be used with lambda function. The general syntax of the filter() function with lambda() is filter(lambda <argument>:<expression>, list). Refer to slide 6 of video 1.7 for general help of the filter() function with lambda().

In this exercise, you'll be using lambda() function inside the filter() built-in function to find all the numbers divisible by 10 in the list.

## Instructions

* Print my\_list2 which is available in your environment.
* Filter the numbers divisible by 10 from my\_list2 using filter() and lambda().
* Print the numbers divisible by 10 from my\_list2.

# Print my\_list2 in the console

print("Input list is:", my\_list2)

# Filter numbers divisible by 10

filtered\_list = list(filter(lambda x: (x%10 == 0), my\_list2))

# Print the numbers divisible by 10

print("Numbers divisible by 10 are:", filtered\_list)

Input list is: [10, 21, 31, 40, 51, 60, 72, 80, 93, 101]

Numbers divisible by 10 are: [10, 40, 60, 80]

# Print my\_list2 in the console

print("Input list is:", my\_list2)

# Filter numbers divisible by 10

filtered\_list = list(filter(lambda x: (x%10 == 0), my\_list2))

# Print the numbers divisible by 10

print("Numbers divisible by 10 are:", filtered\_list)

Lambda functions are indeed simple to use! You'll be using lambda functions with PySpark in the next chapter.

LECTURE

**1. Introduction to PySpark RDD**

00:00 - 00:14

In the first chapter, you have learned about different components of Spark namely, Spark Core, Spark SQL, and Spark MLlib. In this chapter, we will start with RDDs which are Spark’s core abstraction for working with data.

**2. What is RDD?**

00:14 - 00:40

Let's get started. RDD stands for Resilient Distributed Datasets. It is simply a collection of data distributed across the cluster. RDD is the fundamental and backbone data type in PySpark. When Spark starts processing data, it divides the data into partitions and distributes the data across cluster nodes, with each node containing a slice of data. Now, let's take a

**3. Decomposing RDDs**

00:40 - 01:06

look at the different features of RDD. The name RDD captures 3 important properties. Resilient, which means the ability to withstand failures and recompute missing or damaged partitions. Distributed, which means spanning the jobs across multiple nodes in the cluster for efficient computation. Datasets, which is a collection of partitioned data e.g. Arrays, Tables, Tuples or other objects. There are three different

**4. Creating RDDs. How to do it?**

01:06 - 01:48

methods for creating RDDs. You have already seen two methods in the previous chapter even though you are not aware that you are creating RDDs. The simplest method to create RDDs is to take an existing collection of objects (eg. a list, an array or a set) and pass it to SparkContext’s parallelize method. A more common way to create RDDs is to load data from external datasets such as files stored in HDFS or objects in Amazon S3 buckets or from lines in a text file stored locally and pass it to SparkContext's textFile method. Finally, RDDs can also be created from existing RDDs which we will see in the next video. In the first method,

**5. Parallelized collection (parallelizing)**

01:48 - 02:22

RDDs are created from a list or a set using the SparkContext’s parallelize method. Let's try and understand how RDDs are created using this method with a couple of examples. In the first example, an RDD named numRDD is created from a Python list containing numbers 1, 2, 3, and 4. In the second example, an RDD named helloRDD is created from the 'hello world' string. You can confirm the object created is RDD using Python's type method. Creating

**6. From external datasets**

02:22 - 02:49

RDDs from external datasets is by far the most common method in PySpark. In this method, RDDs are created using SparkContext’s textFile method. In this simple example, an RDD named fileRDD is created from the lines of a README-dot-md file stored locally on your computer. Similar to previous method, you can confirm the RDD using the type method. Data

**7. Understanding Partitioning in PySpark**

02:49 - 03:52

partitioning is an important concept in Spark and understanding how Spark deals with partitions allow one to control parallelism. A partition in Spark is the division of the large dataset with each part being stored in multiple locations across the cluster. By default Spark partitions the data at the time of creating RDD based on several factors such as available resources, external datasets etc, however, this behavior can be controlled by passing a second argument called minPartitions which defines the minimum number of partitions to be created for an RDD. In the first example, we create an RDD named numRDD from the list of 10 integers using SparkContext's parallelize method with 6 partitions. In the second example, we create another RDD named fileRDD using SparkContext's textFile method with 6 partitions. The number of partitions in an RDD can always be found by using the getNumPartitions method. In the next

**8. Let's practice**

03:52 - 04:00

video, you'll see the final method of creating RDDs, for now let's create some RDDs like you just learnt.

## Understanding SparkContext

# Print the version of SparkContext

print("The version of Spark Context in the PySpark shell is", sc.version)

# Print the Python version of SparkContext

print("The Python version of Spark Context in the PySpark shell is", sc.pythonVer)

# Print the master of SparkContext

print("The master of Spark Context in the PySpark shell is", sc.master)

## Interactive Use of PySpark

# Create a python list of numbers from 1 to 100

numb = range(1, 100)

# Load the list into PySpark

spark\_data = sc.parallelize(numb)

## Loading data in PySpark shell

# Load a local file into PySpark shell

lines = sc.textFile(file\_path)

## Use of lambda() with map()

# Print my\_list in the console

print("Input list is", my\_list)

# Square all numbers in my\_list

squared\_list\_lambda = list(map(lambda x: x\*\*2, my\_list))

# Print the result of the map function

print("The squared numbers are", squared\_list\_lambda)

## Use of lambda() with filter()

# Print my\_list2 in the console

print("Input list is:", my\_list2)

# Filter numbers divisible by 10

filtered\_list = list(filter(lambda x: (x%10 == 0), my\_list2))

# Print the numbers divisible by 10

print("Numbers divisible by 10 are:", filtered\_list)

## RDDs from Parallelized collections

# Create an RDD from a list of words

RDD = sc.parallelize(["Spark", "is", "a", "framework", "for", "Big Data processing"])

# Print out the type of the created object

print("The type of RDD is", type(RDD))

## RDDs from External Datasets

# Print the file\_path

print("The file\_path is", sc.textFile(file\_path))

# Create a fileRDD from file\_path

fileRDD = sc.parallelize(file\_path)

# Check the type of fileRDD

print("The file type of fileRDD is", type(fileRDD))

## Partitions in your data

# Check the number of partitions in fileRDD

print("Number of partitions in fileRDD is", fileRDD.getNumPartitions())

# Create a fileRDD\_part from file\_path with 5 partitions

fileRDD\_part = sc.textFile(file\_path, minPartitions = 5)

# Check the number of partitions in fileRDD\_part

print("Number of partitions in fileRDD\_part is", fileRDD\_part.getNumPartitions())

## Map and Collect

# Create map() transformation to cube numbers

cubedRDD = numbRDD.map(lambda x: x \*\*3)

# Collect the results

numbers\_all = cubedRDD.collect()

# Print the numbers from numbers\_all

for numb in numbers\_all:

print(numb)

## Filter and Count

# Filter the fileRDD to select lines with Spark keyword

fileRDD\_filter = fileRDD.filter(lambda line: 'Spark' in line)

# How many lines are there in fileRDD?

print("The total number of lines with the keyword Spark is", fileRDD\_filter.count())

# Print the first four lines of fileRDD

for line in fileRDD\_filter.take(4):

print(line)

## Filter and Count

# Filter the fileRDD to select lines with Spark keyword

fileRDD\_filter = fileRDD.filter(lambda line: 'Spark' in line)

# How many lines are there in fileRDD?

print("The total number of lines with the keyword Spark is", fileRDD\_filter.count())

# Print the first four lines of fileRDD

for line in fileRDD\_filter.take(4):

print(line)

**## ReduceBykey and Collect**

# Sort the reduced RDD with the key by descending order

Rdd\_Reduced\_Sort = Rdd\_Reduced.sortByKey(ascending=False)

# Iterate over the result and print the output

for num in Rdd\_Reduced\_Sort.collect():

print("Key {} has {} Counts".format(num[0], num[1]))

## CountingBykeys

# Transform the rdd with countByKey()

total = Rdd.countByKey()

# What is the type of total?

print("The type of total is", type(total))

# Iterate over the total and print the output

for k, v in total.items():

print("key", k, "has", v, "counts")

## Create a base RDD and transform it

# Create a baseRDD from the file path

baseRDD = sc.textFile(file\_path)

# Split the lines of baseRDD into words

splitRDD = baseRDD.flatMap(lambda x: x.split())

# Count the total number of words

print("Total number of words in splitRDD:", splitRDD.count())

## Remove stop words and reduce the dataset

# Convert the words in lower case and remove stop words from stop\_words

splitRDD\_no\_stop = splitRDD.filter(lambda x: x.lower() not in stop\_words)

# Create a tuple of the word and 1

splitRDD\_no\_stop\_words = splitRDD\_no\_stop.map(lambda w: (w, 1))

# Count of the number of occurences of each word

resultRDD = splitRDD\_no\_stop\_words.reduceByKey(lambda x, y: x + y)

## Print word frequencies

# Display the first 10 words and their frequencies

for word in resultRDD.take(10):

print(word)

# Swap the keys and values

resultRDD\_swap = resultRDD.map(lambda x: (x[1], x[0]))

# Sort the keys in descending order

resultRDD\_swap\_sort = resultRDD\_swap.sortByKey(ascending=False)

# Show the top 10 most frequent words and their frequencies

for word in resultRDD\_swap\_sort.take(10):

print("{} has {} counts". format(word[1], word[0]))

# Create a list of tuples

sample\_list = [('Mona',20), ('Jennifer',34), ('John',20), ('Jim',26)]

# Create a RDD from the list

rdd = sc.parallelize(sample\_list)

# Create a PySpark DataFrame

names\_df = spark.createDataFrame(rdd, schema=['Name', 'Age'])

# Check the type of names\_df

print("The type of names\_df is", type(names\_df))

## Loading CSV into DataFrame

# Create an DataFrame from file\_path

people\_df = spark.read.csv(file\_path, header=True, inferSchema=True)

# Check the type of people\_df

print("The type of people\_df is", type(people\_df))

## Inspecting data in PySpark DataFrame

# Print the first 10 observations

people\_df.show(10)

# Count the number of rows

print("There are {} rows in the people\_df DataFrame.".format(people\_df.count()))

# Count the number of columns and their names

print("There are {} columns in the people\_df \

DataFrame and their names are {}".format(len(people\_df.columns), people\_df.columns))

## PySpark DataFrame subsetting and cleaning

# Select name, sex and date of birth columns

people\_df\_sub = people\_df.select('name', 'sex', 'date of birth')

# Print the first 10 observations from people\_df\_sub

people\_df\_sub.show(10)

# Remove duplicate entries from people\_df\_sub

people\_df\_sub\_nodup = people\_df\_sub.dropDuplicates()

# Count the number of rows

print("There were {} rows before removing duplicates,\

and {} rows after removing duplicates".format(people\_df\_sub.count(),

people\_df\_sub\_nodup.count()))

## Running SQL Queries Programmatically

# Create a temporary table "people"

people\_df.createOrReplaceTempView("people")

# Construct a query to select the names of the people from the temporary table "people"

query = '''SELECT name FROM people'''

# Assign the result of Spark's query to people\_df\_names

people\_df\_names = spark.sql(query)

# Print the top 10 names of the people

people\_df\_names.show(10)

## SQL queries for filtering Table

# Filter the people table to select female sex

people\_female\_df = spark.sql('SELECT \* FROM people WHERE sex=="female"')

# Filter the people table DataFrame to select male sex

people\_male\_df = spark.sql('SELECT \* FROM people WHERE sex=="male"')

# Count the number of rows in both DataFrames

print("There are {} rows in the people\_female\_df and {} rows \

in the people\_male\_df DataFrames".format(people\_female\_df.count(),

people\_male\_df.count()))

## PySpark DataFrame visualization

# Check the column names of names\_df

print("The column names of names\_df are", names\_df.columns)

# Convert to Pandas DataFrame

df\_pandas = names\_df.toPandas()

# Create a horizontal bar plot

df\_pandas.plot(kind='barh', x='Name', y='Age', colormap='winter\_r')

plt.show()

## Part 1: Create a DataFrame from CSV file

# Load the Dataframe

fifa\_df = spark.read.csv(file\_path,

header=True,

inferSchema=True)

# Check the schema of columns

fifa\_df.printSchema()

# Show the first 10 observations

fifa\_df.show(10)

# Print the total number of rows

print("There are {} rows in the fifa\_df DataFrame".format(fifa\_df.count()))

## Part 2: SQL Queries on DataFrame

# Create a temporary view of fifa\_df

fifa\_df.createOrReplaceTempView('fifa\_df\_table')

# Construct the "query"

query = '''SELECT Age FROM fifa\_df\_table WHERE Nationality == "Germany"'''

# Apply the SQL "query"

fifa\_df\_germany\_age = spark.sql(query)

# Generate basic statistics

fifa\_df\_germany\_age.describe().show()

## Part 3: Data visualization

# Convert fifa\_df to fifa\_df\_germany\_age\_pandas DataFrame

fifa\_df\_germany\_age\_pandas = fifa\_df\_germany\_age.toPandas()

# Plot the 'Age' density of Germany Players

fifa\_df\_germany\_age\_pandas.plot(kind='density')

plt.show()

## PySpark MLlib algorithms

# Import the library for ALS

from pyspark.mllib.recommendation import ALS

# Import the library for Logistic Regression

from pyspark.mllib.classification import LogisticRegressionWithLBFGS

# Import the library for Kmeans

from pyspark.mllib.clustering import KMeans

## Loading Movie Lens dataset into RDDs

# Load the data into RDD

data = sc.textFile(file\_path)

# Split the RDD

ratings = data.map(lambda l: l.split(','))

# Transform the ratings RDD

ratings\_final = ratings.map(lambda line: Rating(int(line[0]), int(line[1]), float(line[2])))

# Split the data into training and test

training\_data, test\_data = ratings\_final.randomSplit([0.8, 0.2])

## Model training and predictions

# Create the ALS model on the training data

model = ALS.train(training\_data, rank=10, iterations=10)

# Drop the ratings column

testdata\_no\_rating = test\_data.map(lambda p: (p[0], p[1]))

# Predict the model

predictions = model.predictAll(testdata\_no\_rating)

# Print the first rows of the RDD

predictions.take(2)

## Model evaluation using MSE

# Prepare ratings data

rates = ratings\_final.map(lambda r: ((r[0], r[1]),

r[2]))

# Prepare predictions data

preds = predictions.map(lambda r: ((r[0], r[1]),

r[2]))

# Join the ratings data with predictions data

rates\_and\_preds = rates.join(preds)

# Calculate and print MSE

MSE = rates\_and\_preds.map(lambda r: (r[1][0] - r[1][1])\*\*2).mean()

print("Mean Squared Error of the model for the test data = {:.2f}".format(MSE))

## Loading spam and non-spam data

# Load the datasets into RDDs

spam\_rdd = sc.textFile(file\_path\_spam)

non\_spam\_rdd = sc.textFile(file\_path\_non\_spam)

# Split the email messages into words

spam\_words = spam\_rdd.flatMap(lambda email: email.split(' '))

non\_spam\_words = non\_spam\_rdd.flatMap(lambda email: email.split(' '))

# Print the first element in the split RDD

print("The first element in spam\_words is", spam\_words.first())

print("The first element in non\_spam\_words is", non\_spam\_words.first())

## Feathre Hashing and LabelPoint

# Create a HashingTf instance with 200 features

tf = HashingTF(numFeatures=200)

# Map each word to one feature

spam\_features = tf.transform(spam\_words)

non\_spam\_features = tf.transform(non\_spam\_words)

# Label the features: 1 for spam, 0 for non-spam

spam\_samples = spam\_features.map(lambda features: LabeledPoint(1, features))

non\_spam\_samples = non\_spam\_features.map(lambda features: LabeledPoint(0, features))

# Combine the two datasets

samples = spam\_samples.join(non\_spam\_samples)

## Logistic Regression model training

# Split the data into training and testing

train\_samples,test\_samples = samples.randomSplit([0.8, 0.2])

# Train the model

model = LogisticRegressionWithLBFGS.train(train\_samples)

# Create a prediction label from the test data

predictions = model.predict(test\_samples.map(lambda x: x.features))

# Combine original labels with the predicted labels

labels\_and\_preds = test\_samples.map(lambda x: x.label).zip(predictions)

# Check the accuracy of the model on the test data

accuracy = labels\_and\_preds.filter(lambda x: x[0] == x[1]).count() / float(test\_samples.count())

print("Model accuracy : {:.2f}".format(accuracy))

## Loading and parsing the 5000 points data

# Load the dataset into a RDD

clusterRDD = sc.textFile(file\_path)

# Split the RDD based on tab

rdd\_split = clusterRDD.map(lambda x: x.split("\t"))

# Transform the split RDD by creating a list of integers

rdd\_split\_int = rdd\_split.map(lambda x: [int(x[0]), int(x[1])])

# Count the number of rows in RDD

print("There are {} rows in the rdd\_split\_int dataset".format(rdd\_split\_int.count()))

## K-means training

# Train the model with clusters from 13 to 16 and compute WSSSE

for clst in range(13, 17):

model = KMeans.train(rdd\_split\_int, clst, seed=1)

WSSSE = rdd\_split\_int.map(lambda point: error(point)).reduce(lambda x, y: x + y)

print("The cluster {} has Within Set Sum of Squared Error {}".format(clst, WSSSE))

# Train the model again with the best k

model = KMeans.train(rdd\_split\_int, k=15, seed=1)

# Get cluster centers

cluster\_centers = model.clusterCenters

## Visualizing clusters

# Convert rdd\_split\_int RDD into Spark DataFrame

rdd\_split\_int\_df = spark.createDataFrame(rdd\_split\_int, schema=["col1", "col2"])

# Convert Spark DataFrame into Pandas DataFrame

rdd\_split\_int\_df\_pandas = rdd\_split\_int\_df.toPandas()

# Convert "cluster\_centers" that you generated earlier into Pandas DataFrame

cluster\_centers\_pandas = pd.DataFrame(cluster\_centers, columns=["col1", "col2"])

# Create an overlaid scatter plot

plt.scatter(rdd\_split\_int\_df\_pandas["col1"], rdd\_split\_int\_df\_pandas["col2"])

plt.scatter(cluster\_centers\_pandas["col1"], cluster\_centers\_pandas["col2"], color="red", marker="x")

plt.show()

## Exercise

# RDDs from Parallelized collections

Resilient Distributed Dataset (RDD) is the basic abstraction in Spark. It is an immutable distributed collection of objects. Since RDD is a fundamental and backbone data type in Spark, it is important that you understand how to create it. In this exercise, you'll create your first RDD in PySpark from a collection of words.

Remember you already have a SparkContext sc available in your workspace.

## Instructions

* Create an RDD named RDD from a list of words.
* Confirm the object created is RDD.

# Create an RDD from a list of words

RDD = sc.\_\_\_\_(["Spark", "is", "a", "framework", "for", "Big Data processing"])

# Print out the type of the created object

print("The type of RDD is", \_\_\_\_(RDD))

# Create an RDD from a list of words

RDD = sc.parallelize(["Spark", "is", "a", "framework", "for", "Big Data processing"])

# Print out the type of the created object

print("The type of RDD is", type(RDD))

The type of RDD is <class 'pyspark.rdd.RDD'>

# Create an RDD from a list of words

RDD = sc.parallelize(["Spark", "is", "a", "framework", "for", "Big Data processing"])

# Print out the type of the created object

print("The type of RDD is", type(RDD))

EXERCISES

# RDDs from External Datasets

PySpark can easily create RDDs from files that are stored in external storage devices such as HDFS (Hadoop Distributed File System), Amazon S3 buckets, etc. However, the most common method of creating RDD's is from files stored in your local file system. This method takes a file path and reads it as a collection of lines. In this exercise, you'll create an RDD from the file path (file\_path) with the file name README.md which is already available in your workspace.

Remember you already have a SparkContext sc available in your workspace.

## Instructions

* Print the file\_path in the PySpark shell.
* Create an RDD named fileRDD from a file\_path.
* Print the type of the fileRDD created.

# Print the file\_path

print("The file\_path is", \_\_\_\_)

# Create a fileRDD from file\_path

fileRDD = sc.\_\_\_\_(file\_path)

# Check the type of fileRDD

print("The file type of fileRDD is", type(\_\_\_\_))

# Print the file\_path

print("The file\_path is", file\_path)

# Create a fileRDD from file\_path

fileRDD = sc.textFile(file\_path)

# Check the type of fileRDD

print("The file type of fileRDD is", type(fileRDD))

The file\_path is /usr/local/share/datasets/README.md

The file type of fileRDD is <class 'pyspark.rdd.RDD'>

# Print the file\_path

print("The file\_path is", file\_path)

# Create a fileRDD from file\_path

fileRDD = sc.textFile(file\_path)

# Check the type of fileRDD

print("The file type of fileRDD is", type(fileRDD))

Wonderful! Now you can create RDDs from Text files too!

## Exercise

# Partitions in your data

SparkContext's textFile() method takes an optional second argument called minPartitions for specifying the minimum number of partitions. In this exercise, you'll create an RDD named fileRDD\_part with 5 partitions and then compare that with fileRDD that you created in the previous exercise. Refer to the "Understanding Partition" slide in video 2.1 to know the methods for creating and getting the number of partitions in an RDD.

Remember, you already have a SparkContext sc, file\_path and fileRDD available in your workspace.

## Instructions

* Find the number of partitions that support fileRDD RDD.
* Create an RDD named fileRDD\_part from the file path but create 5 partitions.
* Confirm the number of partitions in the new fileRDD\_part RDD.

# Check the number of partitions in fileRDD

print("Number of partitions in fileRDD is", fileRDD.\_\_\_\_)

# Create a fileRDD\_part from file\_path with 5 partitions

fileRDD\_part = sc.textFile(\_\_\_\_, minPartitions = \_\_\_\_)

# Check the number of partitions in fileRDD\_part

print("Number of partitions in fileRDD\_part is", fileRDD\_part.\_\_\_\_)

# Check the number of partitions in fileRDD

print("Number of partitions in fileRDD is", fileRDD.getNumPartitions())

# Create a fileRDD\_part from file\_path with 5 partitions

fileRDD\_part = sc.textFile(file\_path, minPartitions = 5)

# Check the number of partitions in fileRDD\_part

print("Number of partitions in fileRDD\_part is", fileRDD\_part.getNumPartitions())

**Number of partitions in fileRDD is 1**

**Number of partitions in fileRDD\_part is 5**

# Check the number of partitions in fileRDD

print("Number of partitions in fileRDD is", fileRDD.getNumPartitions())

# Create a fileRDD\_part from file\_path with 5 partitions

fileRDD\_part = sc.textFile(file\_path, minPartitions = 5)

# Check the number of partitions in fileRDD\_part

print("Number of partitions in fileRDD\_part is", fileRDD\_part.getNumPartitions())

**Excellent! Note that modifying the number of partitions may result in faster performance due to parallelization.**

**LECTURE**

**1. RDD operations in PySpark**

00:00 - 00:11

In the last video, you have learned how to load your data into RDDs. In this video, you'll learn about the various operations that support RDDs in PySpark. RDDs

**2. Overview of PySpark operations**

00:11 - 00:27

in PySpark supports two different types of operations - Transformations and Actions. Transformations are operations on RDDs that return a new RDD and Actions are operations that perform some computation on the RDD. The most important

**3. RDD Transformations**

00:27 - 00:59

feature which helps RDDs in fault tolerance and optimizing resource use is the lazy evaluation. So what is lazy evaluation? Spark creates a graph from all the operations you perform on an RDD and execution of the graph starts only when an action is performed on RDD as shown in this figure. This is called lazy evaluation in Spark. The RDD transformations we will look in this video are map, filter, flatMap and union. The map

**4. map() Transformation**

00:59 - 01:42

transformation takes in a function and applies it to each element in the RDD. Say you have an input RDD with elements 1,2,3,4. The map transformation takes in a function and applies it to each element in the RDD with the result of the function being the new value of each element in the resulting RDD. In this example, the square function is applied to each element of the RDD. Let's understand this with an example. We first create an RDD using SparkContext's parallelize method on a list containing elements 1,2,3,4. Next, we apply map transformation for squaring each element of the RDD. The

**5. filter() Transformation**

01:42 - 02:11

filter transformation takes in a function and returns an RDD that only has elements that pass the condition. Suppose we have an input RDD with numbers 1,2,3,4 and we want to select numbers greater than 2, we can apply the filter transformation. Here is an example of the filter transformation wherein we use the same RDD as before to apply the filter transformation to filter out the numbers that are greater than 2. flatMap

**6. flatMap() Transformation**

02:11 - 02:59

is similar to map transformation except it returns multiple values for each element in the source RDD. A simple usage of flatMap is splitting up an input string into words. Here, you have an input RDD with two elements - "hello world" and "how are you". Applying the split function of the flatMap transformation results in 5 elements in the resulting RDD - "hello", "world", "how", "are", "you". As you can see, even though the input RDD has 2 elements, the output RDD now contains 5 elements. In this example, we create an RDD from a list containing the words "hello world" and "how are you". Next, we apply flatmap along with split function on the RDD to split the input string into individual words.

**7. union() Transformation**

02:59 - 03:50

union Transformation returns the union of one RDD with another RDD. In this figure, we are filtering the inputRDD and creating two RDDs - errorsRDD and warningsRDD and next we are combining both the RDDs using union transformation. To illustrate this using PySpark code, let's first create an inputRDD from a local file using SparkContext's textFile method, next we will use two filter transformations to create two RDDs errorRDD and warningsRDD and finally using union transformation we will combine them both. So far you have seen how RDD Transformations but after applying Transformations at some point, you'll want to actually do something with your dataset. This is when Actions come into picture.

**8. RDD Actions**

03:50 - 04:07

Actions are the operations that are applied on RDDs to return a value after running a computation. The four basic actions that you'll learn in this lesson are collect, take, first and count. Collect

**9. collect() and take() Actions**

04:07 - 04:41

action returns complete list of elements from the RDD. Whereas take(N) print an 'N' number of elements from the RDD. Continuing the map transformation example, executing collect returns all elements i.e 1, 4, 9, 16 from the RDD\_map RDD that you created earlier. Similarly here is an example of take(2) action that prints the first 2 elements i.e 1 and 4 from the RDD\_map RDD. Sometimes you just want to print the first element of

**10. first() and count() Actions**

04:41 - 05:17

the RDD. first action returns the first element in an RDD. It is similar to take(1). Here is an example of first action which prints the first element i.e 1 from the RDD\_map RDD. Finally, the count action is used to return the total number of rows/elements in the RDD. Here is an example of count action to count the number of elements in the RDD\_flatmap RDD. The result here indicates that there are 5 elements in the RDD\_flatmap RDD. It's time for you to practice

**11. Let's practice RDD operations**

05:17 - 05:21

RDD operations in PySpark shell now.

## Exercise

# Map and Collect

The main method by which you can manipulate data in PySpark is using map(). The map() transformation takes in a function and applies it to each element in the RDD. It can be used to do any number of things, from fetching the website associated with each URL in our collection to just squaring the numbers. In this simple exercise, you'll use map() transformation to cube each number of the numbRDD RDD that you created earlier. Next, you'll return all the elements to a variable and finally print the output.

Remember, you already have a SparkContext sc, and numbRDD available in your workspace.

## Instructions

* Create map() transformation that cubes all of the numbers in numbRDD.
* Collect the results in a numbers\_all variable.
* Print the output from numbers\_all variable.

# Create map() transformation to cube numbers

cubedRDD = numbRDD.map(lambda x: \_\_\_\_)

# Collect the results

numbers\_all = cubedRDD.\_\_\_\_()

# Print the numbers from numbers\_all

for numb in \_\_\_\_:

    print(\_\_\_\_)

# Create map() transformation to cube numbers

cubedRDD = numbRDD.map(lambda x: x\*\*3)

# Collect the results

numbers\_all = cubedRDD.collect()

# Print the numbers from numbers\_all

for numb in numbers\_all:

print(numb)

1

8

27

64

125

216

343

512

729

1000

# Create map() transformation to cube numbers

cubedRDD = numbRDD.map(lambda x: x\*\*3)

# Collect the results

numbers\_all = cubedRDD.collect()

# Print the numbers from numbers\_all

for numb in numbers\_all:

    print(numb)

**Brilliant! collect() should only be used to retrieve results for small datasets. It shouldn’t be used on large datasets.**

## Exercise

# Filter and Count

The RDD transformation filter() returns a new RDD containing only the elements that satisfy a particular function. It is useful for filtering large datasets based on a keyword. For this exercise, you'll filter out lines containing keyword Spark from fileRDD RDD which consists of lines of text from the README.md file. Next, you'll count the total number of lines containing the keyword Spark and finally print the first 4 lines of the filtered RDD.

Remember, you already have a SparkContext sc, file\_path and fileRDD available in your workspace.

## Instructions

* Create filter() transformation to select the lines containing the keyword Spark.
* How many lines in fileRDD\_filter contains the keyword Spark?
* Print the first four lines of the resulting RDD.

# Filter the fileRDD to select lines with Spark keyword

fileRDD\_filter = fileRDD.filter(lambda line: 'Spark' in \_\_\_\_)

# How many lines are there in fileRDD?

print("The total number of lines with the keyword Spark is", fileRDD\_filter.\_\_\_\_())

# Print the first four lines of fileRDD

for line in fileRDD\_filter.\_\_\_\_(\_\_\_\_):

  print(line)

# Filter the fileRDD to select lines with Spark keyword

fileRDD\_filter = fileRDD.filter(lambda line: 'Spark' in line)

# How many lines are there in fileRDD?

print("The total number of lines with the keyword Spark is", fileRDD\_filter.count())

# Print the first four lines of fileRDD

for line in fileRDD\_filter.take(4):

  print(line)

# Filter the fileRDD to select lines with Spark keyword

fileRDD\_filter = fileRDD.filter(lambda line: 'Spark' in line)

# How many lines are there in fileRDD?

print("The total number of lines with the keyword Spark is", fileRDD\_filter.count())

# Print the first four lines of fileRDD

for line in fileRDD\_filter.take(4):

print(line)

The total number of lines with the keyword Spark is 7

Examples for Learning Spark

Examples for the Learning Spark book. These examples require a number of libraries and as such have long build files. We have also added a stand alone example with minimal dependencies and a small build file

These examples have been updated to run against Spark 1.3 so they may

be slightly different than the versions in your copy of "Learning Spark".

**Well done! Note that the filter() operation does not mutate the existing fileRDD. Instead, it returns a pointer to an entirely new RDD.**

**LECTURE**

**1. Working with Pair RDDs in PySpark**

00:00 - 00:14

In the last video, you were introduced to some basic RDD operations and in this video, you'll learn how to work with RDDs of key/value pairs, which are a common data type required for many operations in Spark

**2. Introduction to pair RDDs in PySpark**

00:14 - 00:46

Most of the real world datasets are generally key/value pairs. An example of this kind of dataset has the team name as key and the list of players as values. The typical pattern of this kind of dataset is each row is a key that maps to one or more values. In order to deal with this kind of dataset, PySpark provides a special data structure called pair RDDs. In pair RDDs, the key refers to the identifier, whereas value refers to the data.

**3. Creating pair RDDs**

00:46 - 01:38

There are a number of ways to create pair RDDs. The two most common ways are creating from a list of the key-value tuple or from a regular RDD. Irrespective of the method, the first step in creating pair RDDs is to get the data into key/value form. Here is an example of creating pair RDD from a list of the key-value tuple that contains the names as key and age as the value using SparkContext's parallelize method. And here is an example of creating pair RDD from regular RDDs. In this example, a regular RDD is created from a list that contains strings using SparkContext's parallelize method. Next, we create a pair RDD using map function which returns tuple with key/value pairs with key being the name and age being the value.

**4. Transformations on pair RDDs**

01:38 - 02:04

Pair RDDs are still RDDs and thus use all the transformations available to regular RDDs. Since pair RDDs contain tuples, we need to pass functions that operate on key-value pairs. A few special operations are available for this kind such as reduceByKey, groupByKey, sortByKey and join. Let's take a look at each of these four pair RDD transformations in detail now.

**5. reduceByKey() transformation**

02:04 - 02:48

The reduceByKey transformation is the most popular pair RDD transformation which combines values with the same key using a function. reduceByKey runs several parallel operations, one for each key in the dataset. Because datasets can have very large numbers of keys, reduceByKey is not implemented as an action. Instead, it returns a new RDD consisting of each key and the reduced value for that key. Here is an example of reducebykey transformation that uses a function to combine all the goals scored by each of the players. The result shows that player as key and total number of goals scored as value.

**6. sortByKey() transformation**

02:48 - 03:14

Sorting of data is necessary for many downstream applications. We can sort pair RDD as long as there is an ordering defined in the key. The sortByKey transformation returns an RDD sorted by key in ascending or descending order. Continuing our reduceByKey example, here is an example that sorts the data based on the number of goals scored by each player. A common use case of

**7. groupByKey() transformation**

03:14 - 03:47

pair RDDs is grouping the data by key. For example, viewing all of the airports for a particular country together. If the data is already keyed in the way that we want, the groupByKey operation groups all the values with the same key in the pair RDD. Here is an example of groupByKey transformation that groups all the airports for a particular country from an input list that contains list of tuples. Each tuple consists of country code and the corresponding airport code. Join transformation

**8. join() transformation**

03:47 - 04:25

joins two pair RDDs based on their key. Let's demonstrate this with an example. First, we create two RDDs. RDD1 contains the list of tuples with each tuple consisting of name and age and RDD2 contains the list of tuples with each tuple consisting of name and income. Applying join transformation on RDD1 and RDD2 merges two RDDs together by grouping elements with the same key. Here is an example that shows the result of join transformation of RDD1 and RDD2.

**9. Let's practice**

04:25 - 04:30

Now that you have learned all about pair RDDs, it's time for you to practice.

## Exercise

# ReduceBykey and Collect

One of the most popular pair RDD transformations is reduceByKey() which operates on key, value (k,v) pairs and merges the values for each key. In this exercise, you'll first create a pair RDD from a list of tuples, then combine the values with the same key and finally print out the result.

Remember, you already have a SparkContext sc available in your workspace.

## Instructions

* Create a pair RDD named Rdd with tuples (1,2),(3,4),(3,6),(4,5).
* Transform the Rdd with reduceByKey() into a pair RDD Rdd\_Reduced by adding the values with the same key.
* Collect the contents of pair RDD Rdd\_Reduced and iterate to print the output.

# Create PairRDD Rdd with key value pairs

Rdd = sc.parallelize([\_\_\_\_])

# Apply reduceByKey() operation on Rdd

Rdd\_Reduced = Rdd.reduceByKey(lambda x, y: \_\_\_\_)

# Iterate over the result and print the output

for num in Rdd\_Reduced.\_\_\_\_:

  print("Key {} has {} Counts".format(\_\_\_\_, num[1]))

# Create PairRDD Rdd with key value pairs

Rdd = sc.parallelize([(1,2),(3,4),(3,6),(4,5)])

# Apply reduceByKey() operation on Rdd

Rdd\_Reduced = Rdd.reduceByKey(lambda x, y: x+y)

# Iterate over the result and print the output

for num in Rdd\_Reduced.collect():

print("Key {} has {} Counts".format(num[0], num[1]))

Key 1 has 2 Counts

Key 3 has 10 Counts

Key 4 has 5 Counts

# Create PairRDD Rdd with key value pairs

Rdd = sc.parallelize([(1,2),(3,4),(3,6),(4,5)])

# Apply reduceByKey() operation on Rdd

Rdd\_Reduced = Rdd.reduceByKey(lambda x, y: x+y)

# Iterate over the result and print the output

for num in Rdd\_Reduced.collect():

  print("Key {} has {} Counts".format(num[0], num[1]))

**Good job! reduceByKey() transformation merges the values for each key using an associative reduce function.**

## Exercise

# SortByKey and Collect

Many times it is useful to sort the pair RDD based on the key (for example word count which you'll see later in the chapter). In this exercise, you'll sort the pair RDD Rdd\_Reduced that you created in the previous exercise into descending order and print the final output.

Remember, you already have a SparkContext sc and Rdd\_Reduced available in your workspace.

## Instructions

* Sort the Rdd\_Reduced RDD using the key in descending order.
* Collect the contents and iterate to print the output.

# Sort the reduced RDD with the key by descending order

Rdd\_Reduced\_Sort = Rdd\_Reduced.\_\_\_\_(ascending=False)

# Iterate over the result and retrieve all the elements of the RDD

for num in Rdd\_Reduced\_Sort.\_\_\_\_():

  print("Key {} has {} Counts".format(\_\_\_\_, num[1]))

# Sort the reduced RDD with the key by descending order

Rdd\_Reduced\_Sort = Rdd\_Reduced.sortByKey(ascending=False)

# Iterate over the result and retrieve all the elements of the RDD

for num in Rdd\_Reduced\_Sort.collect():

print("Key {} has {} Counts".format(num[0], num[1]))

ERROR! Session/line number was not unique in database. History logging moved to new session 3

Key 4 has 5 Counts

Key 3 has 10 Counts

Key 1 has 2 Counts

# Sort the reduced RDD with the key by descending order

Rdd\_Reduced\_Sort = Rdd\_Reduced.sortByKey(ascending=False)

# Iterate over the result and retrieve all the elements of the RDD

for num in Rdd\_Reduced\_Sort.collect():

  print("Key {} has {} Counts".format(num[0], num[1]))

**Congratulations! You'll see how you can use sortByKey() with real world data at the end of this chapter.**

**LECTURE**

**1. More actions**

00:00 - 00:12

Previously you learned about advanced RDD Transformations for key/value datasets. Similar to advanced RDD Transformations there are advanced RDD Actions which you'll see in this video.

**2. reduce() action**

00:12 - 00:53

Reduce action takes in a function which operates on two elements of the same type of RDD and returns a new element of the same type. The function should be commutative and associative so that it can be computed correctly in parallel. A simple example of such a function is +, which we can use to sum our RDD. Here is an example of reduce action that calculates the sum of all the elements in an RDD. In this example, input RDD is first created using SparkContext's parallelize method on a list consisting of numbers 1,3,4,6. Eexcuting reduce action results in 14 which is the sum of 1,3,4,6.

**3. saveAsTextFile() action**

00:53 - 01:46

In many cases, it is not advisable to run collect action on RDDs because of the huge size of the data. In these cases, it’s common to write data out to a distributed storage systems such as HDFS or Amazon S3. saveAsTextFile action can be used to save RDD as a text file inside a particular directory. By default, saveAsTextFile saves RDD with each partition as a separate file inside a directory. Here is an example of saveAsTextFile that saves an RDD with each partition as a separate file inside a directory. However, you can change it to return a new RDD that is reduced into a single partition using the coalesce method. Here is an example of saveAsTextFile that saves RDD as a single file inside a directory. Similar to

**4. Action Operations on pair RDDs**

01:46 - 02:10

pair RDD Transformations, there are also RDD Actions available for pair RDDs. However, pair RDDs also attain some additional actions of PySpark especially those that leverage the advantage of data which is of key-value nature. Let’s take a look at two pair RDD actions - countByKey and collectAsMap in this video.

**5. countByKey() action**

02:10 - 02:53

countByKey is only available on RDDs of type (Key, Value). With the countByKey operation, we can count the number of elements for each key. Here is an example of counting the number of values for each key in the dataset. In this example, we first create a pair RDD named rdd using SparkContext's parallelize method. Since countByKey generates a dictionary, next we iterate over the dictionary to print the each unique and number of values associated with each key as shown here. One thing to note is that countByKey should only be used on a dataset whose size is small enough to fit in memory. collectAsMap

**6. collectAsMap() action**

02:53 - 03:26

returns the key-value pairs in the RDD to the as a dictionary. Here is an example of collectAsMap on a pair RDD. As before we create a pair RDD using SparkContext's parallelize method and next use collectAsMap action. collectAsMap produces the key-value pairs in the RDD as a dictionary which can be used for downstream analysis. Similar to countByKey, this action should only be used if the resulting data is expected to be small, as all the data is loaded into the memory. Let's practice

**7. Let's practice**

03:26 - 03:32

some of these advanced Actions on some test data in PySpark shell.

## Exercise

# CountingBykeys

For many datasets, it is important to count the number of keys in a key/value dataset. For example, counting the number of countries where the product was sold or to show the most popular baby names. In this simple exercise, you'll use the Rdd that you created earlier and count the number of unique keys in that pair RDD.

Remember, you already have a SparkContext sc and Rdd available in your workspace.

## Instructions

* Count the unique keys and assign the result to a variable total.
* What is the type of total?
* Iterate over the total and print the keys and their counts.

# Count the unique keys

total = Rdd.\_\_\_\_()

# What is the type of total?

print("The type of total is", \_\_\_\_(total))

# Iterate over the total and print the output

for k, v in total.\_\_\_():

  print("key", \_\_\_\_, "has", \_\_\_\_, "counts")

# Count the unique keys

total = Rdd.countByKey()

# What is the type of total?

print("The type of total is", type(total))

# Iterate over the total and print the output

for k, v in total.items():

print("key", k, "has", v, "counts")

The type of total is <class 'collections.defaultdict'>

key 1 has 1 counts

key 3 has 2 counts

key 4 has 1 counts

# Count the unique keys

total = Rdd.countByKey()

# What is the type of total?

print("The type of total is", type(total))

# Iterate over the total and print the output

for k, v in total.items():

  print("key", k, "has", v, "counts")

**Good job! Remember unlike reduceByKey() and sortByKey(), countByKey() is an action and not a transformation on the pair RDD.**

## Exercise

# Create a base RDD and transform it

The volume of unstructured data (log lines, images, binary files) in existence is growing dramatically, and PySpark is an excellent framework for analyzing this type of data through RDDs. In this 3 part exercise, you will write code that calculates the most common words from [Complete Works of William Shakespeare](http://www.gutenberg.org/ebooks/100).

Here are the brief steps for writing the word counting program:

* Create a base RDD from Complete\_Shakespeare.txt file.
* Use RDD transformation to create a long list of words from each element of the base RDD.
* Remove stop words from your data.
* Create pair RDD where each element is a pair tuple of ('w', 1)
* Group the elements of the pair RDD by key (word) and add up their values.
* Swap the keys (word) and values (counts) so that keys is count and value is the word.
* Finally, sort the RDD by descending order and print the 10 most frequent words and their frequencies.

In this first exercise, you'll create a base RDD from Complete\_Shakespeare.txt file and transform it to create a long list of words.

Remember, you already have a SparkContext sc already available in your workspace. A file\_path variable (which is the path to the Complete\_Shakespeare.txt file) is also loaded for you.

## Instructions

* Create an RDD called baseRDD that reads lines from file\_path.
* Transform the baseRDD into a long list of words and create a new splitRDD.
* Count the total words in splitRDD.

# Create a baseRDD from the file path

baseRDD = sc.\_\_\_\_(file\_path)

# Split the lines of baseRDD into words

splitRDD = baseRDD.\_\_\_\_(lambda x: x.split())

# Count the total number of words

print("Total number of words in splitRDD:", splitRDD.\_\_\_\_())

# Create a baseRDD from the file path

baseRDD = sc.textFile(file\_path)

# Split the lines of baseRDD into words

splitRDD = baseRDD.flatMap(lambda x: x.split())

# Count the total number of words

print("Total number of words in splitRDD:", splitRDD.count())

**# Create a baseRDD from the file path**

**baseRDD = sc.textFile(file\_path)**

**# Split the lines of baseRDD into words**

**splitRDD = baseRDD.flatMap(lambda x: x.split())**

**# Count the total number of words**

**print("Total number of words in splitRDD:", splitRDD.count())**

**Total number of words in splitRDD: 904061**

Good start! You have succesfully created and transformed RDD from unstructured data.

## Exercise

# Remove stop words and reduce the dataset

After splitting the lines in the file into a long list of words in the previous exercise, in the next step, you'll remove stop words from your data. Stop words are common words that are often uninteresting. For example "I", "the", "a" etc., are stop words. You can remove many obvious stop words with a list of your own. But for this exercise, you will just remove the stop words from a curated list stop\_words provided to you in your environment.

After removing stop words, you'll next create a pair RDD where each element is a pair tuple (k, v) where k is the key and v is the value. In this example, pair RDD is composed of (w, 1) where w is for each word in the RDD and 1 is a number. Finally, you'll combine the values with the same key from the pair RDD.

Remember you already have a SparkContext sc and splitRDD available in your workspace.

## Instructions

* Convert the words in splitRDD in lower case and then remove stop words from stop\_words curated list.
* Create a pair RDD tuple containing the word and the number 1 from each word element in splitRDD.
* Get the count of the number of occurrences of each word (word frequency) in the pair RDD.

# Convert the words in lower case and remove stop words from the stop\_words curated list

splitRDD\_no\_stop = splitRDD.\_\_\_\_(lambda x: x.lower() not in \_\_\_\_)

# Create a tuple of the word and 1

splitRDD\_no\_stop\_words = splitRDD\_no\_stop.map(lambda w: (\_\_\_\_, \_\_\_\_))

# Count of the number of occurences of each word

resultRDD = splitRDD\_no\_stop\_words.\_\_\_\_(lambda x, y: x + y)

# Convert the words in lower case and remove stop words from the stop\_words curated list

splitRDD\_no\_stop = splitRDD.filter(lambda x: x.lower() not in stop\_words)

# Create a tuple of the word and 1

splitRDD\_no\_stop\_words = splitRDD\_no\_stop.map(lambda w: (w,1))

# Count of the number of occurences of each word

resultRDD = splitRDD\_no\_stop\_words.reduceByKey(lambda x, y: x + y)

# Convert the words in lower case and remove stop words from the stop\_words curated list splitRDD\_no\_stop = splitRDD.filter(lambda x: x.lower() not in stop\_words) # Create a tuple of the word and 1 splitRDD\_no\_stop\_words = splitRDD\_no\_stop.map(lambda w: (w,1)) # Count of the number of occurences of each word resultRDD = splitRDD\_no\_stop\_words.reduceByKey(lambda x, y: x + y)

Good job! You are nearly ready to print the words and their frequencies.

## Exercise

# Print word frequencies

After combining the values (counts) with the same key (word), in this exercise, you'll return the first 10 word frequencies. You could have retrieved all the elements at once using collect() but it is bad practice and not recommended. RDDs can be huge: you may run out of memory and crash your computer..

What if we want to return the top 10 words? For this, first you'll need to swap the key (word) and values (counts) so that keys is count and value is the word. After you swap the key and value in the tuple, you'll sort the pair RDD based on the key (count). This way it is easy to sort the RDD based on the key rather than the key using sortByKey operation in PySpark. Finally, you'll return the top 10 words from the sorted RDD.

You already have a SparkContext sc and resultRDD available in your workspace.

## Instructions

* Print the first 10 words and their frequencies from the resultRDD RDD.
* Swap the keys and values in the resultRDD.
* Sort the keys according to descending order.
* Print the top 10 most frequent words and their frequencies from the sorted RDD.

# Display the first 10 words and their frequencies from the input RDD

for word in resultRDD.\_\_\_\_(10):

    print(word)

# Swap the keys and values from the input RDD

resultRDD\_swap = resultRDD.\_\_\_\_(lambda x: (x[1], x[\_\_\_\_]))

# Sort the keys in descending order

resultRDD\_swap\_sort = resultRDD\_swap.\_\_\_\_(ascending=False)

# Show the top 10 most frequent words and their frequencies from the sorted RDD

for word in resultRDD\_swap\_sort.\_\_\_\_(\_\_\_\_):

    print("{},{}". format(\_\_\_\_, word[0]))

# Display the first 10 words and their frequencies from the input RDD

for word in resultRDD.take(10):

    print(word)

# Swap the keys and values from the input RDD

resultRDD\_swap = resultRDD.map(lambda x: (x[1], x[0]))

# Sort the keys in descending order

resultRDD\_swap\_sort = resultRDD\_swap.sortByKey(ascending=False)

# Show the top 10 most frequent words and their frequencies from the sorted RDD

for word in resultRDD\_swap\_sort.take(10):

    print("{},{}". format(word[1], word[0]))

# Display the first 10 words and their frequencies from the input RDD

for word in resultRDD.take(10):

print(word)

# Swap the keys and values from the input RDD

resultRDD\_swap = resultRDD.map(lambda x: (x[1], x[0]))

# Sort the keys in descending order

resultRDD\_swap\_sort = resultRDD\_swap.sortByKey(ascending=False)

# Show the top 10 most frequent words and their frequencies from the sorted RDD

for word in resultRDD\_swap\_sort.take(10):

print("{},{}". format(word[1], word[0]))

ERROR! Session/line number was not unique in database. History logging moved to new session 5

('Project', 85)

('Gutenberg', 26)

('EBook', 2)

('Complete', 4)

('Works', 5)

('William', 67)

('Shakespeare,', 2)

('Shakespeare', 45)

('eBook', 9)

('use', 266)

thou,4247

thy,3630

shall,3018

good,2046

would,1974

Enter,1926

thee,1780

I'll,1737

hath,1614

like,1452

Congratulations! You have sucessfully created a word count program using RDD in PySpark.

**LECTURE**

**1. Introduction to PySpark DataFrames**

00:00 - 00:13

In the previous chapter, you looked at RDDs which is Spark’s core abstraction for working with data. In this chapter, we will explore PySpark SQL which is Spark's high level API for working with structured data.

**2. What are PySpark DataFrames?**

00:13 - 01:06

PySpark SQL is a Spark library for structured data. Unlike the PySpark RDD API, PySpark SQL provides more information about the structure of data and the computation being performed. PySpark SQL provides a programming abstraction called DataFrames. A DataFrame is an immutable distributed collection of data with named columns. It is similar to a table in SQL. DataFrames are designed to process a large collection of structured data such as relational database and semi-structured data such as JSON (JavaScript Object Notation). DataFrame API currently supports several languages such as Python, R, Scala, and Java. DataFrames allows PySpark to query data using SQL, for example (SELECT \* from table) or using the expression method for example (df-dot-select).

**3. SparkSession - Entry point for DataFrame API**

01:06 - 01:47

Previously you have learned about SparkContext which is the main entry point for creating RDDs. Similarly, SparkSession provides a single point of entry to interact with underlying Spark functionality and allows programming Spark with DataFrame API. The SparkSession does for DataFrames what the SparkContext does for RDDs. A SparkSession can be used to create DataFrame, register DataFrame as tables, execute SQL over tables, cache tables etc., Similar to SparkContext, SparkSession is exposed to the PySpark shell as variable spark. DataFrames in

**4. Creating DataFrames in PySpark**

01:47 - 02:27

Pyspark can be created in two main ways. From an existing RDD using SparkSession's createDataFrame method and From different data sources such as CSV, JSON, TXT using SparkSession's read method. Before going into the details of creating DataFrames, let's understand what schema is. Schema is the structure of data in DataFrame and helps Spark to optimize queries on the data more efficiently. A schema provides informational detail such as the column name, the type of data in that column, and whether null or empty values are allowed in the column. To create a DataFrame

**5. Create a DataFrame from RDD**

02:27 - 03:20

from an RDD we will need to pass an RDD and a schema into SparkSession's createDataFrame method. In this example, we will first create an RDD named iphones\_RDD from a list of iphones using SparkContext's parallelize method. Next, we will create a DataFrame using SparkSession's createDataFrame method using iphones\_RDD and the list of column names such as Model, Year, Height, Width and Weight as schema. The type of object created can be confirmed using type method, which shows that it is a PySpark DataFrame. A thing to note here is when the schema is a list of column names, the type of each column will be inferred from data as shown above. However when the schema is None, it will try to infer the schema from data. To create a

**6. Create a DataFrame from reading a CSV/JSON/TXT**

03:20 - 04:33

DataFrame from CSV/JSON/TXT files, we will make use of the SparkSession's spark-dot-read property. Here is an example of creating df\_csv DataFrame from people-dot-csv file using spark-dot-read-dot-csv method. Similarly here is an example for creating df\_json DataFrame from people-dot-json file using spark-dot-read-dot-json method. Finally here is an example for creating df\_txt DataFrame from people-dot-txt file using spark-dot-read-dot-txt method. Irrespective of the file type, this method requires the path to the file and two optional parameters. The first optional parameter, header=True may be passed to make sure that the method treats the first row as column names. The second optional parameter, inferSchema=True may be passed to instruct the DataFrame reader to infer the schema from the data and by doing so, it will attempt to assign the right datatype to each column based on the content. Now let's

**7. Let's practice**

04:33 - 04:38

practice creating some DataFrames in PySpark shell.

## Exercise

# RDD to DataFrame

Similar to RDDs, DataFrames are immutable and distributed data structures in Spark. Even though RDDs are a fundamental data structure in Spark, working with data in DataFrame is easier than RDD, and so understanding of how to convert RDD to DataFrame is necessary.

In this exercise, you'll first make an RDD using the sample\_list that is already provided to you. This RDD contains the list of tuples ('Mona',20), ('Jennifer',34),('John',20), ('Jim',26) with each tuple contains the name of the person and their age. Next, you'll create a DataFrame using the RDD and the schema (which is the list of 'Name' and 'Age') and finally confirm the output as PySpark DataFrame.

Remember, you already have a SparkContext sc and SparkSession spark available in your workspace.

## Instructions

* Create an RDD from the sample\_list.
* Create a PySpark DataFrame using the above RDD and schema.
* Confirm the output as PySpark DataFrame.

# Create an RDD from the list

rdd = sc.\_\_\_\_(sample\_list)

# Create a PySpark DataFrame

names\_df = spark.createDataFrame(\_\_\_\_, \_\_\_\_=['Name', 'Age'])

# Check the type of names\_df

print("The type of names\_df is", \_\_\_\_(names\_df))

# Create an RDD from the list

rdd = sc.parallelize(sample\_list)

# Create a PySpark DataFrame

names\_df = spark.createDataFrame(rdd, schema=['Name', 'Age'])

# Check the type of names\_df

print("The type of names\_df is", type(names\_df))

The type of names\_df is <class 'pyspark.sql.dataframe.DataFrame'>

# Create an RDD from the list

rdd = sc.parallelize(sample\_list)

# Create a PySpark DataFrame

names\_df = spark.createDataFrame(rdd, schema=['Name', 'Age'])

# Check the type of names\_df

print("The type of names\_df is", type(names\_df))

Brilliant! Creating DataFrames from RDDs may not be a common practise but helps in certain circumstances.

## Exercise

# Loading CSV into DataFrame

In the previous exercise, you have seen a method for creating a DataFrame from an RDD. Generally, loading data from CSV file is the most common method of creating DataFrames. In this exercise, you'll create a PySpark DataFrame from a people.csv file that is already provided to you as a file\_path and confirm the created object is a PySpark DataFrame.

Remember, you already have a SparkSession spark and a variable file\_path (the path to the people.csv file) available in your workspace.

## Instructions

* Create a DataFrame from file\_path variable which is the path to the people.csv file.
* Confirm the output as PySpark DataFrame.

# Create an DataFrame from file\_path

people\_df = spark.\_\_\_\_(file\_path, header=True, inferSchema=True)

# Check the type of people\_df

print("The type of people\_df is", \_\_\_\_(people\_df))

**# Create an DataFrame from file\_path**

**people\_df = spark.read.csv(file\_path, header=True, inferSchema=True)**

**# Check the type of people\_df**

**print("The type of people\_df is", type(people\_df))**

**ERROR! Session/line number was not unique in database. History logging moved to new session 3**

**The type of people\_df is <class 'pyspark.sql.dataframe.DataFrame'>**

# Create an DataFrame from file\_path

people\_df = spark.read.csv(file\_path, header=True, inferSchema=True)

# Check the type of people\_df

print("The type of people\_df is", type(people\_df))

**Nice! You'll be using spark.read.csv() method a lot in the next several exercises.**

**LECTURE**

**1. Interacting with PySpark DataFrames**

00:00 - 00:10

Just like RDDs, DataFrames also support both transformations and actions. In this video, you'll learn some DataFrame operations in PySpark.

**2. DataFrame operators in PySpark**

00:10 - 00:43

Similar to RDD operations, the DataFrame operations in PySpark can be divided into Transformations and Actions. PySpark DataFrame provides operations to filter, group, or compute aggregates, and can be used with PySpark SQL. Let's explore some of the most common DataFrame Transformations such as select, filter, groupby, orderby, dropDuplicates, withColumnRenamed and some common DataFrame Actions such as printSchema, show, count, columns and describe in this video.

**3. select() and show() operations**

00:43 - 01:24

Let's start with select and show operations. The select Transformation is used to extract one or more columns from a DataFrame. We need to pass the column name inside select operation. As an example, let’s select ‘Age’ columns from a test DataFrame. select is a Transformation and so it creates a new DataFrame and in order to print the rows from df\_id\_age DataFrame, we need to execute an Action. show is an Action that prints the first 20 rows by default. Let’s apply show(3) on df\_id\_age DataFrame and print the first 3 rows as shown in this example.

**4. filter() and show() operations**

01:24 - 01:56

Unlike select, the filter Transformation selects only rows that pass the condition specified. The parameter you pass is the column name and the value of what you want to filter that column on. For example, if we want to filter out the rows with 'Age' greater than 21, we pass the column expression (new\_df-dot-Age) and the condition (greater than 21) as shown in here. We can use show(3) action to print out the first 3 rows from the new DataFrame.

**5. groupby() and count() operations**

01:56 - 02:28

The groupby Transformation groups the DataFrame using the specified columns, so we can run aggregation on them. To better understand, we will first group the 'Age' column and create another DataFrame. Then we will use count action that returns the total number of rows in the DataFrame and finally use show(3) operation to print the first 3 rows in the DataFrame. The result is a table that shows the first 3 Age groups and the corresponding number of members in each group.

**6. orderby() Transformations**

02:28 - 02:53

Orderby transformation returns a DataFrame sorted by the given columns. Let’s sort the test\_df\_age\_group-dot-count that we obtained in the previous example based on ‘Age’ column and print out the first 3 rows of the DataFrame using show(3) action. As you can see the age groups have been sorted in ascending order now.

**7. dropDuplicates()**

02:53 - 03:18

The dropDuplicates transformation returns a new DataFrame with duplicate rows removed. Here is an example, where dropDuplicates transformation is used to remove duplicate rows in 'User\_ID' 'Age' and 'Gender' columns and finally creating a new DataFrame. You can execute count action on this new DataFrame to print the number of non-duplicate rows.

**8. withColumnRenamed Transformations**

03:18 - 03:43

The withColumnRenamed transformation returns a new DataFrame by renaming an existing column. It takes two arguments: the names of the old and new columns. In this example, we rename the column name "Gender" to "Sex" and create a test\_df\_sex. We can use show(3) Action to print out the first 3 rows from the new DataFrame.

**9. printSchema()**

03:43 - 04:10

To check the types of columns in the DataFrame, we can use the printSchema action. Here is an example of printSchema action on test\_df DataFrame that we used previously. printSchema prints out the schema in the tree format as shown here and helps to spot the issues with the schema of the data. As an example product\_ID is shown as string even though it is supposed to be an integer.

**10. columns actions**

04:10 - 04:28

The columns operation returns the names of all the columns in the DataFrame as an array of string. Let’s print the column names in the test\_df DataFrame. In this example, the test\_df DataFrame has three columns 'User\_ID', 'Gender' and 'Age'.

**11. describe() actions**

04:28 - 04:45

describe operation is used to calculate the summary statistics of the numerical columns in the DataFrame. If we don’t specify the name of columns it will calculate summary statistics for all numerical columns present in the DataFrame as shown in this example.

**12. Let's practice**

04:45 - 04:54

Now that you are familiar with DataFrame operations, let's practice using some these operations on a real world data.

## Exercise

# Inspecting data in PySpark DataFrame

Inspecting data is very crucial before performing analysis such as plotting, modeling, training etc., In this simple exercise, you'll inspect the data in the people\_df DataFrame that you have created in the previous exercise using basic DataFrame operators.

Remember, you already have a SparkSession spark and a DataFrame people\_df available in your workspace.

## Instructions

* Print the first 10 observations in the people\_df DataFrame.
* Count the number of rows in the people\_df DataFrame.
* How many columns does people\_df DataFrame have and what are their names?

# Print the first 10 observations

people\_df.\_\_\_\_(10)

# Count the number of rows

print("There are {} rows in the people\_df DataFrame.".format(people\_df.\_\_\_\_()))

# Count the number of columns and their names

print("There are {} columns in the people\_df DataFrame and their names are {}".format(len(people\_df.\_\_\_\_), people\_df.\_\_\_\_))

# Print the first 10 observations

people\_df.show(10)

# Count the number of rows

print("There are {} rows in the people\_df DataFrame.".format(people\_df.count()))

# Count the number of columns and their names

print("There are {} columns in the people\_df DataFrame and their names are {}".format(len(people\_df.columns), people\_df.columns))

+---+---------+----------------+------+-------------+

|\_c0|person\_id| name| sex|date of birth|

+---+---------+----------------+------+-------------+

| 0| 100| Penelope Lewis|female| 1990-08-31|

| 1| 101| David Anthony| male| 1971-10-14|

| 2| 102| Ida Shipp|female| 1962-05-24|

| 3| 103| Joanna Moore|female| 2017-03-10|

| 4| 104| Lisandra Ortiz|female| 2020-08-05|

| 5| 105| David Simmons| male| 1999-12-30|

| 6| 106| Edward Hudson| male| 1983-05-09|

| 7| 107| Albert Jones| male| 1990-09-13|

| 8| 108|Leonard Cavender| male| 1958-08-08|

| 9| 109| Everett Vadala| male| 2005-05-24|

+---+---------+----------------+------+-------------+

only showing top 10 rows

There are 100000 rows in the people\_df DataFrame.

There are 5 columns in the people\_df DataFrame and their names are ['\_c0', 'person\_id', 'name', 'sex', 'date of birth']

# Print the first 10 observations

people\_df.show(10)

# Count the number of rows

print("There are {} rows in the people\_df DataFrame.".format(people\_df.count()))

# Count the number of columns and their names

print("There are {} columns in the people\_df DataFrame and their names are {}".format(len(people\_df.columns), people\_df.columns))

**That is nicely done. PySpark DataFrames make it easy for performing exploratory data analysis with easy to use operators.**

## Exercise

# PySpark DataFrame subsetting and cleaning

After data inspection, it is often necessary to clean the data which mainly involves subsetting, renaming the columns, removing duplicated rows etc., PySpark DataFrame API provides several operators to do this. In this exercise, your job is to subset 'name', 'sex' and 'date of birth' columns from people\_df DataFrame, remove any duplicate rows from that dataset and count the number of rows before and after duplicates removal step.

Remember, you already have a SparkSession spark and a DataFrame people\_df available in your workspace.

## Instructions

* Select 'name', 'sex' and 'date of birth' columns from people\_df and create people\_df\_sub DataFrame.
* Print the first 10 observations in the people\_df DataFrame.
* Remove duplicate entries from people\_df\_sub DataFrame and create people\_df\_sub\_nodup DataFrame.
* How many rows are there before and after duplicates are removed?

# Select name, sex and date of birth columns

people\_df\_sub = people\_df.\_\_\_\_('name', \_\_\_\_, \_\_\_\_)

# Print the first 10 observations from people\_df\_sub

people\_df\_sub.\_\_\_\_(\_\_\_\_)

# Remove duplicate entries from people\_df\_sub

people\_df\_sub\_nodup = people\_df\_sub.\_\_\_\_()

# Count the number of rows

print("There were {} rows before removing duplicates, and {} rows after removing duplicates".format(people\_df\_sub.\_\_\_\_(), people\_df\_sub\_nodup.\_\_\_\_()))

# Select name, sex and date of birth columns

people\_df\_sub = people\_df.select('name', 'sex', 'date of birth')

# Print the first 10 observations from people\_df\_sub

people\_df\_sub.show(10)

# Remove duplicate entries from people\_df\_sub

people\_df\_sub\_nodup = people\_df\_sub.dropDuplicates()

# Count the number of rows

print("There were {} rows before removing duplicates, and {} rows after removing duplicates".format(people\_df\_sub.count(), people\_df\_sub\_nodup.count()))

+----------------+------+-------------+

| name| sex|date of birth|

+----------------+------+-------------+

| Penelope Lewis|female| 1990-08-31|

| David Anthony| male| 1971-10-14|

| Ida Shipp|female| 1962-05-24|

| Joanna Moore|female| 2017-03-10|

| Lisandra Ortiz|female| 2020-08-05|

| David Simmons| male| 1999-12-30|

| Edward Hudson| male| 1983-05-09|

| Albert Jones| male| 1990-09-13|

|Leonard Cavender| male| 1958-08-08|

| Everett Vadala| male| 2005-05-24|

+----------------+------+-------------+

only showing top 10 rows

There were 100000 rows before removing duplicates, and 99998 rows after removing duplicates

# Select name, sex and date of birth columns

people\_df\_sub = people\_df.select('name', 'sex', 'date of birth')

# Print the first 10 observations from people\_df\_sub

people\_df\_sub.show(10)

# Remove duplicate entries from people\_df\_sub

people\_df\_sub\_nodup = people\_df\_sub.dropDuplicates()

# Count the number of rows

print("There were {} rows before removing duplicates, and {} rows after removing duplicates".format(people\_df\_sub.count(), people\_df\_sub\_nodup.count()))

Good job! dropDuplicates() is very useful operator in PySpark DataFrames.

## Exercise

# Filtering your DataFrame

In the previous exercise, you have subset the data using select() operator which is mainly used to subset the DataFrame column-wise. What if you want to subset the DataFrame based on a condition (for example, select all rows where the sex is Female). In this exercise, you will filter the rows in the people\_df DataFrame in which 'sex' is female and male and create two different datasets. Finally, you'll count the number of rows in each of those datasets.

Remember, you already have a SparkSession spark and a DataFrame people\_df available in your workspace.

## Instructions

* Filter the people\_df DataFrame to select all rows where sex is female into people\_df\_female DataFrame.
* Filter the people\_df DataFrame to select all rows where sex is male into people\_df\_male DataFrame.
* Count the number of rows in people\_df\_female and people\_df\_male DataFrames.

# Filter people\_df to select females

people\_df\_female = people\_df.\_\_\_\_(people\_df.\_\_\_\_ == "female")

# Filter people\_df to select males

people\_df\_male = people\_df.\_\_\_\_(\_\_\_\_ == "\_\_\_\_")

# Count the number of rows

print("There are {} rows in the people\_df\_female DataFrame and {} rows in the people\_df\_male DataFrame".format(people\_df\_female.\_\_\_\_(), people\_df\_male.\_\_\_\_()))

# Filter people\_df to select females

people\_df\_female = people\_df.filter(people\_df.sex == "female")

# Filter people\_df to select males

people\_df\_male = people\_df.filter(people\_df.sex == "male")

# Count the number of rows

print("There are {} rows in the people\_df\_female DataFrame and {} rows in the people\_df\_male DataFrame".format(people\_df\_female.count(), people\_df\_male.count()))

There are 49014 rows in the people\_df\_female DataFrame and 49066 rows in the people\_df\_male DataFrame

# Filter people\_df to select females

people\_df\_female = people\_df.filter(people\_df.sex == "female")

# Filter people\_df to select males

people\_df\_male = people\_df.filter(people\_df.sex == "male")

# Count the number of rows

print("There are {} rows in the people\_df\_female DataFrame and {} rows in the people\_df\_male DataFrame".format(people\_df\_female.count(), people\_df\_male.count()))

That is nicely done. PySpark DataFrames make it easy for performing exploratory data analysis with easy to use operators.

LECTURE

**1. Interacting with DataFrames using PySpark SQL**

00:00 - 00:12

Previously, you have seen how to interact with PySparkSQL using DataFrame API. In this video, you'll learn how to interact with PySparkSQL using SQL query.

**2. DataFrame API vs SQL queries**

00:12 - 01:05

In addition to DataFrame API, PySpark SQL allows you to manipulate DataFrames with SQL queries. What you can do using DataFrames API, can be done using SQL queries and vice versa. So what are the differences between DataFrames API and SQL queries? The DataFrames API provides a programmatic interface – basically a domain-specific language (DSL) for interacting with data. DataFrame queries are much easier to construct programmatically. Plain SQL queries can be significantly more concise and easier to understand. They are also portable and can be used without any modifications with every supported language. Many of the DataFrame operations that you have seen in the previous chapter, can be done using SQL queries.

**3. Executing SQL Queries**

01:05 - 02:02

The SparkSession provides a method called sql which can be used to execute a SQL query. The sql method takes a SQL statement as an argument and returns a DataFrame representing the result of the given query. Unfortunately, SQL queries cannot be run directly against a DataFrame. To issue SQL queries against an existing DataFrame we can leverage the createOrReplaceTempView function to build a temporary table as shown in this example. After creating the temporary table, we can simply use the sql method, which allows us to write SQL code to manipulate data within a DataFrame. In this example, we simply extract two columns field1 and field2 from the table using SELECT. Since the result is a DataFrame, you can run DataFrame actions such as collect, first, show etc. An example of collect action is shown here. In the previous

**4. SQL query to extract data**

02:02 - 02:40

lesson, you have seen how to use select operation to subset the data from a DataFrame. Here is an example of how you can do the same with a SQL query. In this example, we will first construct the query for selecting the Product\_ID column form the temporary table. Next we will pass the query to the SparkSession's sql method to create a new DataFrame. Because the result of SQL query returns a DataFrame, all the usual DataFrame operations are available. Here we can use show(5) action to print the first 5 rows of the DataFrame. The SQL queries are

**5. Summarizing and grouping data using SQL queries**

02:40 - 03:12

not limited to extracting data as seen in the previous slide. We can also create SQL queries to run aggregations. In this example, we first construct a query for selecting 'Age' and 'Purchase' columns, then aggregate the total of all the purchases, the maximum per Age group. We can then provide the query to the SparkSession's sql method and use show(5) action to print out the first 5 rows as seen in here. In addition to extracting and summarizing the data,

**6. Filtering columns using SQL queries**

03:12 - 03:43

Spark SQL queries can also be constructed for filtering the rows from a DataFrame. Suppose you want to filter out the rows of Age, Purchase and Gender columns where the Gender is Female and purchase is greater than 20000, you can construct a query as shown in this example. You can confirm whether or not query worked by providing the query to the SparkSession's sql method and using show(5) action to print out the first 5 rows as shown in this example. Let's practice some SQL

**7. Time to practice!**

03:43 - 03:47

within PySpark shell now!

## Exercise

# Running SQL Queries Programmatically

DataFrames can easily be manipulated using SQL queries in PySpark. The sql() function on a SparkSession enables applications to run SQL queries programmatically and returns the result as another DataFrame. In this exercise, you'll create a temporary table of the people\_df DataFrame that you created previously, then construct a query to select the names of the people from the temporary table and assign the result to a new DataFrame.

Remember, you already have a SparkSession spark and a DataFrame people\_df available in your workspace.

## Instructions

* Create a temporary table people that's a pointer to the people\_df DataFrame.
* Construct a query to select the names of the people from the temporary table people.
* Assign the result of Spark's query to a new DataFrame - people\_df\_names.
* Print the top 10 names of the people from people\_df\_names DataFrame.

 Create a temporary table "people"

people\_df.\_\_\_\_("people")

# Construct a query to select the names of the people from the temporary table "people"

query = '''SELECT name FROM \_\_\_\_'''

# Assign the result of Spark's query to people\_df\_names

people\_df\_names = spark.sql(\_\_\_\_)

# Print the top 10 names of the people

people\_df\_names.\_\_\_\_(\_\_\_\_)

# Create a temporary table "people"

people\_df.createOrReplaceTempView("people")

# Construct a query to select the names of the people from the temporary table "people"

query = '''SELECT name FROM people'''

# Assign the result of Spark's query to people\_df\_names

people\_df\_names = spark.sql(query)

# Print the top 10 names of the people

people\_df\_names.show(10)

ERROR! Session/line number was not unique in database. History logging moved to new session 14

+----------------+

| name|

+----------------+

| Penelope Lewis|

| David Anthony|

| Ida Shipp|

| Joanna Moore|

| Lisandra Ortiz|

| David Simmons|

| Edward Hudson|

| Albert Jones|

|Leonard Cavender|

| Everett Vadala|

+----------------+

only showing top 10 rows

# Create a temporary table "people"

people\_df.createOrReplaceTempView("people")

# Construct a query to select the names of the people from the temporary table "people"

query = '''SELECT name FROM people'''

# Assign the result of Spark's query to people\_df\_names

people\_df\_names = spark.sql(query)

# Print the top 10 names of the people

people\_df\_names.show(10)

**Good job on running your first SQL query successfully!!! Spark SQL operations generally return DataFrames. This means you can freely mix DataFrames and SQL.**

## Exercise

# SQL queries for filtering Table

In the previous exercise, you have run a simple SQL query on a DataFrame. There are more sophisticated queries you can construct to obtain the result that you want and use it for downstream analysis such as data visualization and Machine Learning. In this exercise, we will use the temporary table people that you created previously and filter out the rows where the "sex" is male and female and create two DataFrames.

Remember, you already have a SparkSession spark and a temporary table people available in your workspace.

## Instructions

* Filter the people table to select all rows where sex is female into people\_female\_df DataFrame.
* Filter the people table to select all rows where sex is male into people\_male\_df DataFrame.
* Count the number of rows in both people\_female and people\_male DataFrames

# Filter the people table to select female sex

people\_female\_df = spark.\_\_\_\_('SELECT \* FROM \_\_\_\_ WHERE sex=="\_\_\_\_"')

# Filter the people table DataFrame to select male sex

people\_male\_df = spark.\_\_\_\_('SELECT \* \_\_\_\_ people \_\_\_\_ \_\_\_\_=="\_\_\_\_"')

# Count the number of rows in both DataFrames

print("There are {} rows in the people\_female\_df and {} rows in the people\_male\_df DataFrames".format(people\_female\_df.\_\_\_\_(), people\_male\_df.\_\_\_\_()))

# Filter the people table to select female sex

people\_female\_df = spark.sql('SELECT \* FROM people WHERE sex=="female"')

# Filter the people table DataFrame to select male sex

people\_male\_df = spark.sql('SELECT \* FROM people WHERE sex=="male"')

# Count the number of rows in both DataFrames

print("There are {} rows in the people\_female\_df and {} rows in the people\_male\_df DataFrames".format(people\_female\_df.count(), people\_male\_df.count()))

There are 49014 rows in the people\_female\_df and 49066 rows in the people\_male\_df DataFrames

# Filter the people table to select female sex

people\_female\_df = spark.sql('SELECT \* FROM people WHERE sex=="female"')

# Filter the people table DataFrame to select male sex

people\_male\_df = spark.sql('SELECT \* FROM people WHERE  sex=="male"')

# Count the number of rows in both DataFrames

print("There are {} rows in the people\_female\_df and {} rows in the people\_male\_df DataFrames".format(people\_female\_df.count(), people\_male\_df.count()))

That is nicely done! Now that you have used filter() with both Dataframe API and SQL query. Which one do you prefer?

LECTURE

**1. Data Visualization in PySpark using DataFrames**

00:00 - 00:12

Visualization is an essential part of data analysis. In this video, we will explore some visualization methods that can help us make sense of our data in PySpark DataFrames.

**2. What is Data visualization?**

00:12 - 00:50

Data visualization is the way of representing your data in form of graphs or charts. It is considered a crucial component of Exploratory Data Analysis (EDA). Several open source tools exist to aid visualization in Python such as matplotlib, Seaborn, Bokeh etc. However, none of these visualization tools can be used directly with PySpark's DataFrames. Currently, there are three different methods available to create charts using PySpark DataFrames - pyspark\_dist\_explore library, toPandas method, and HandySpark toPandas. Let's understand each of these methods with examples.

**3. Data Visualization using Pyspark\_dist\_explore**

00:50 - 01:41

Pyspark\_dist\_explore is a plotting library to get quick insights on data in PySpark DataFrames. There are 3 functions available in Pyspark\_dist\_explore to create matplotlib graphs while minimizing the amount of computation needed - hist, distplot and pandas\_histogram. Here is an example of creating a histogram using the Pyspark\_dist\_explore package on the test\_df data. First, the CSV file is loaded into Spark DataFrame using the SparkSession's read-dot-csv method. Then we select the age column from the test\_df DataFrame using the select operation. Finally we use the hist function of the Pyspark\_dist\_explore package to plot a histogram of 'Age' in the test\_df\_age dataset. The second method

**4. Using Pandas for plotting DataFrames**

01:41 - 02:24

of creating charts is by using toPandas on PySpark DataFrames which converts the PySpark DataFrame into a Pandas DataFrame. After conversion, it's easy to create charts from pandas DataFrames using matplotlib or seaborn plotting tools. In this example, first, the CSV is loaded in Spark DataFrame using read-dot-csv method. Next, using toPandas method, we will convert the Spark DataFrame into Pandas DataFrame. Finally, we will create a histogram of the "Age" column using matplotlib's hist method. Before we look at the third method, let's take a look at the differences between Pandas

**5. Pandas DataFrame vs PySpark DataFrame**

02:24 - 03:14

vs Spark DataFrames. But, Pandas won’t work in every case. It is a single machine tool and constrained by single machine limits. So their size is limited by your server memory, and you will process them with the power of a single server. In contrast, operations on Pyspark DataFrames run parallel on different nodes in the cluster. In pandas DataFrames, we get the result as soon as we apply any operation Whereas operations in PySpark DataFrames are lazy in nature. You can change a Pandas DataFrame using methods. We can’t change a PySpark DataFrame due to its immutable property. Finally, the Pandas API supports more operations than PySpark DataFrames. The final method of

**6. HandySpark method of visualization**

03:14 - 04:14

creating charts is using HandySpark libary, which is a relatively a new package. HandySpark is designed to improve PySpark user experience, especially when it comes to exploratory data analysis, including visualization capabilities. It makes fetching data or computing statistics for columns really easy, returning pandas objects straight away. It brings the long-missing capability of plotting data while retaining the advantage of performing the distributed computation. Here is an example of the HandySpark method for creating a histogram. Just like before, we load the CSV into a PySpark DataFrame using SparkSession's read-dot-csv method. After creating the DataFrame, we convert the DataFrame to a HandySpark DataFrame using the toHandy method. Finally, we create a histogram of the Age column using the hist function of HandySpark library. We have learned three exciting

**7. Let's visualize DataFrames**

04:14 - 04:24

methods of visualizing PySpark DataFrames and let's practice creating some charts with them now on real-world datasets now.

## Exercise

# PySpark DataFrame visualization

Graphical representations or visualization of data is imperative for understanding as well as interpreting the data. In this simple data visualization exercise, you'll first print the column names of names\_df DataFrame that you created earlier, then convert the names\_df to Pandas DataFrame and finally plot the contents as horizontal bar plot with names of the people on the x-axis and their age on the y-axis.

Remember, you already have a SparkSession spark and a DataFrame names\_df available in your workspace.

## Instructions

* Print the names of the columns in names\_df DataFrame.
* Convert names\_df DataFrame to df\_pandas Pandas DataFrame.
* Use matplotlib's plot() method to create a horizontal bar plot with 'Name' on x-axis and 'Age' on y-axis.

# Check the column names of names\_df

print("The column names of names\_df are", names\_df.\_\_\_\_)

# Convert to Pandas DataFrame

df\_pandas = names\_df.\_\_\_\_()

# Create a horizontal bar plot

\_\_\_\_.plot(kind='barh', x='\_\_\_\_', y='\_\_\_\_', colormap='winter\_r')

plt.show()

# Check the column names of names\_df

print("The column names of names\_df are", names\_df.columns)

# Convert to Pandas DataFrame

df\_pandas = names\_df.toPandas()

# Create a horizontal bar plot

df\_pandas.plot(kind='barh', x='Name', y='Age', colormap='winter\_r')

plt.show()

# Check the column names of names\_df

print("The column names of names\_df are", names\_df.columns)

# Convert to Pandas DataFrame

df\_pandas = names\_df.toPandas()

# Create a horizontal bar plot

df\_pandas.plot(kind='barh', x='Name', y='Age', colormap='winter\_r')

plt.show()

The column names of names\_df are ['Name', 'Age']

Awesome! You'll get to use other PySpark DataFrame packages for visualization in the next few exercises.

## Exercise

# Part 1: Create a DataFrame from CSV file

Every 4 years, the soccer fans throughout the world celebrates a festival called “Fifa World Cup” and with that, everything seems to change in many countries. In this 3 part exercise, you'll be doing some exploratory data analysis (EDA) on the "FIFA 2018 World Cup Player" dataset using PySpark SQL which involve DataFrame operations, SQL queries and visualization.

In the first part, you'll load FIFA 2018 World Cup Players dataset (Fifa2018\_dataset.csv) which is in CSV format into a PySpark's dataFrame and inspect the data using basic DataFrame operations.

Remember, you already have a SparkSession spark and a variable file\_path available in your workspace.

## Instructions

* Create a PySpark DataFrame from file\_path (which is the path to the Fifa2018\_dataset.csv file).
* Print the schema of the DataFrame.
* Print the first 10 observations.
* How many rows are in there in the DataFrame?
* # Load the Dataframe
* fifa\_df = spark.read.csv(file\_path, header=True, inferSchema=True)
* # Check the schema of columns
* fifa\_df.printSchema()
* # Show the first 10 observations
* fifa\_df.show(10)
* # Print the total number of rows
* print("There are {} rows in the fifa\_df DataFrame".format(fifa\_df.count()))
* root
* |-- \_c0: integer (nullable = true)
* |-- Name: string (nullable = true)
* |-- Age: integer (nullable = true)
* |-- Photo: string (nullable = true)
* |-- Nationality: string (nullable = true)
* |-- Flag: string (nullable = true)
* |-- Overall: integer (nullable = true)
* |-- Potential: integer (nullable = true)
* |-- Club: string (nullable = true)
* |-- Club Logo: string (nullable = true)
* |-- Value: string (nullable = true)
* |-- Wage: string (nullable = true)
* |-- Special: integer (nullable = true)
* |-- Acceleration: string (nullable = true)
* |-- Aggression: string (nullable = true)
* |-- Agility: string (nullable = true)
* |-- Balance: string (nullable = true)
* |-- Ball control: string (nullable = true)
* |-- Composure: string (nullable = true)
* |-- Crossing: string (nullable = true)
* |-- Curve: string (nullable = true)
* |-- Dribbling: string (nullable = true)
* |-- Finishing: string (nullable = true)
* |-- Free kick accuracy: string (nullable = true)
* |-- GK diving: string (nullable = true)
* |-- GK handling: string (nullable = true)
* |-- GK kicking: string (nullable = true)
* |-- GK positioning: string (nullable = true)
* |-- GK reflexes: string (nullable = true)
* |-- Heading accuracy: string (nullable = true)
* |-- Interceptions: string (nullable = true)
* |-- Jumping: string (nullable = true)
* |-- Long passing: string (nullable = true
* There are 17981 rows in the fifa\_df DataFrame

# Load the Dataframe

fifa\_df = spark.read.csv(file\_path, header=True, inferSchema=True)

# Check the schema of columns

fifa\_df.printSchema()

# Show the first 10 observations

fifa\_df.show(10)

# Print the total number of rows

print("There are {} rows in the fifa\_df DataFrame".format(fifa\_df.count()))

**Good job! That's a Big Data. Get ready to do some exciting stuff on this Big Data.**

## Exercise

# Part 2: SQL Queries on DataFrame

The fifa\_df DataFrame that we created has additional information about datatypes and names of columns associated with it. This additional information allows PySpark SQL to run SQL queries on DataFrame. SQL queries are concise and easy to run compared to DataFrame operations. But in order to apply SQL queries on DataFrame first, you need to create a temporary view of DataFrame as a table and then apply SQL queries on the created table (Running SQL Queries Programmatically).

In the second part, you'll create a temporary table of fifa\_df DataFrame and run SQL queries to extract the 'Age' column of players from Germany.

You already have a SparkContext spark and fifa\_df available in your workspace.

## Instructions

* Create temporary table fifa\_df from fifa\_df\_table DataFrame.
* Construct a "query" to extract the "Age" column from Germany players.
* Apply the SQL "query" to the temporary view table and create a new DataFrame.
* Computes basic statistics of the created DataFrame.

# Create a temporary view of fifa\_df

fifa\_df.\_\_\_\_('fifa\_df\_table')

# Construct the "query"

query = '''SELECT \_\_\_\_ FROM \_\_\_\_ WHERE Nationality == "Germany"'''

# Apply the SQL "query"

fifa\_df\_germany\_age = spark.\_\_\_\_(\_\_\_\_)

# Generate basic statistics

fifa\_df\_germany\_age.\_\_\_\_().show()

# Create a temporary view of fifa\_df

fifa\_df.createOrReplaceTempView('fifa\_df\_table')

# Construct the "query"

query = '''SELECT Age FROM fifa\_df\_table WHERE Nationality == "Germany"'''

# Apply the SQL "query"

fifa\_df\_germany\_age = spark.sql(query)

# Generate basic statistics

fifa\_df\_germany\_age.describe().show()

+-------+-----------------+

|summary| Age|

+-------+-----------------+

| count| 1140|

| mean|24.20263157894737|

| stddev|4.197096712293756|

| min| 16|

| max| 36|

+-------+-----------------+

# Create a temporary view of fifa\_df

fifa\_df.createOrReplaceTempView('fifa\_df\_table')

# Construct the "query"

query = '''SELECT Age FROM fifa\_df\_table WHERE Nationality == "Germany"'''

# Apply the SQL "query"

fifa\_df\_germany\_age = spark.sql(query)

# Generate basic statistics

fifa\_df\_germany\_age.describe().show()

**Excellent! Notice how consise SQL queries are compared to DataFrame operations.**

# Part 3: Data visualization

Data visualization is important for exploratory data analysis (EDA). PySpark DataFrame is a perfect for data visualization compared to RDDs because of its inherent structure and schema.

In this third part, you'll create a histogram of the ages of all the players from Germany from the DataFrame that you created in the previous exercise. For this, you'll first convert the PySpark DataFrame into Pandas DataFrame and use matplotlib's plot() function to create a density plot of ages of all players from Germany.

Remember, you already have a SparkSession spark, a temporary table fifa\_df\_table and a DataFrame fifa\_df\_germany\_age available in your workspace.

## Instructions

* Convert fifa\_df\_germany\_age to fifa\_df\_germany\_age\_pandas Pandas DataFrame.
* Generate a density plot of the 'Age' column from the fifa\_df\_germany\_age\_pandas Pandas DataFrame.

# Convert fifa\_df to fifa\_df\_germany\_age\_pandas DataFrame

fifa\_df\_germany\_age\_pandas = fifa\_df\_germany\_age.\_\_\_\_()

# Plot the 'Age' density of Germany Players

\_\_\_\_.plot(kind='density')

plt.show()

# Convert fifa\_df to fifa\_df\_germany\_age\_pandas DataFrame

fifa\_df\_germany\_age\_pandas = fifa\_df\_germany\_age.toPandas()

# Plot the 'Age' density of Germany Players

fifa\_df\_germany\_age\_pandas.plot(kind='density')

plt.show()

# Convert fifa\_df to fifa\_df\_germany\_age\_pandas DataFrame fifa\_df\_germany\_age\_pandas = fifa\_df\_germany\_age.toPandas() # Plot the 'Age' density of Germany Players fifa\_df\_germany\_age\_pandas.plot(kind='density') plt.show()

**Great job on successfully performing Exploratory Data Analysis on PySpark DataFrame using DataFrame operations, SQL queries and visualization!**

**LECTURE**

**1. Overview of PySpark MLlib**

00:00 - 00:17

In the last chapter, you learned about PySpark SQL which is one of the high-level API built on top of Spark Core for structured data. In this chapter, you'll learn about PySpark MLlib which is a built-in library for scalable machine learning.

**2. What is PySpark MLlib?**

00:17 - 01:11

Before diving deep into PySpark MLlib, let's quickly define what machine learning is. According to Wikipedia, Machine learning is a scientific discipline that explores the construction and study of algorithms that can learn from data. PySpark MLlib is a machine-learning library. Its goal is to make practical machine learning scalable and easy. At a high level, PySpark MLlib provides tools such as: Machine learning algorithms which include collaborative filtering, classification, and clustering. Featurization which include feature extraction, transformation, dimensionality reduction, and selection. Pipelines which include constructing, evaluating, and tuning ML Pipelines. In this chapter, we will explore Machine Learning algorithms - collaborative filtering, classification, and clustering.

**3. Why PySpark MLlib?**

01:11 - 02:05

Many of you have heard about Scikit-learn, which is a very popular and easy to use Python library for machine learning. Then what is the need for PySpark MLlib? Scikit-learn algorithms work well for small to medium-sized datasets that can be processed on a single machine, but not for large datasets that require the power of parallel processing. On the other hand, PySpark MLlib only contains algorithms in which operations can be applied in parallel across nodes in a cluster. Unlike Scikit-learn, MLlib supports several other higher languages such as Scala, Java, and R in addition to Python. MLlib also provides a high-level API to build machine-learning pipelines. A machine learning pipeline is a complete workflow combining multiple machine learning algorithms together. PySpark is good

**4. PySpark MLlib Algorithms**

02:05 - 02:59

for iterative algorithms and using iterative algorithms, many machine-learning algorithms have been implemented in PySpark MLlib. PySpark MLlib currently supports various methods for binary classification, multiclass classification, and regression analysis. Some of the algorithms include linear SVMs, logistic regression, decision trees, random forests, gradient-boosted trees, naive Bayes, linear least squares, Lasso, ridge regression, isotonic regression. Collaborative filtering is commonly used for recommender systems and PySpark MLlib uses the alternating least squares (ALS) algorithm for collaborative filtering. Clustering algorithms consist of k-means, gaussian mixture, Power iteration clustering, Bisecting k-means and Streaming k-means. While PySpark MLlib includes several machine

**5. The three C's of machine learning in PySpark MLlib**

02:59 - 03:37

learning algorithms, we will specifically focus on the three key areas, often referred to as the three Cs of machine learning - Collaborative filtering, Classification, and Clustering. Collaborative filtering produces recommendations based on past behavior, preferences, or similarities to known entities/users. Classification is the problem of identifying to which of a set of categories a new observation belongs. Clustering is grouping of data into clusters based on similar characteristics. We'll go in more detail in the next few lessons. Now that you learned the 3 C's of the machine

**6. PySpark MLlib imports**

03:37 - 04:24

learning, let's quickly understand how we can import these PySpark MLlib libraries in the PySpark shell environment. Let's start with PySpark's collaborative filtering which is available in the pyspark-dot-mllib-dot-recommendation submodule. Here is how you import the ALS (Alternating Least Squares) class in PySpark shell. For binary classification, here is an example of how you import LogisticRegressionWithLBFGS class in the pyspark-dot-mllib-dot-classification submodule inside the PySpark shell. Similarly, for clustering, here is an example of importing the KMeans class in PySpark shell using the pyspark-dot-mllib-dot-clustering submodule. Let's practice

**7. Let's practice**

04:24 - 04:30

how well you understand the different Machine learning algorithms by importing them in PySpark shell.

# PySpark ML libraries

What kind of data structures does pyspark.mllib built-in library support in Spark?

##### Answer the question

#### Possible Answers

DataFrames

**RDDs Correct!!! pyspark.mllib is the builtin library for RDD-based API.**

Datasets

All

## Exercise

# PySpark MLlib algorithms

Before using any Machine learning algorithms in PySpark shell, you'll have to import the submodules of pyspark.mllib library and then choose the appropriate class that is needed for a specific machine learning task.

In this simple exercise, you'll learn how to import the different submodules of pyspark.mllib along with the classes that are needed for performing Collaborative filtering, Classification and Clustering algorithms.

## Instructions

* Import pyspark.mllib recommendation submodule and Alternating Least Squares class.
* Import pyspark.mllib classification submodule and Logistic Regression with LBFGS class.
* Import pyspark.mllib clustering submodule and kmeans class

# Import the library for ALS

from pyspark.mllib.\_\_\_\_ import \_\_\_\_

# Import the library for Logistic Regression

from \_\_\_\_.\_\_\_\_.\_\_\_\_ import \_\_\_\_

# Import the library for Kmeans

from \_\_\_\_.\_\_\_\_.\_\_\_\_ \_\_\_\_ \_\_\_\_

# Import the library for ALS

from pyspark.mllib.recommendation import ALS

# Import the library for Logistic Regression

from pyspark.mllib.classification import LogisticRegressionWithLBFGS

# Import the library for Kmeans

from pyspark.mllib.clustering import KMeans

**Sweet. You don't have to import these submodules and classes for the rest of the chapter as they will be imported to you in the environment**

**LECTURE**

**1. Introduction to Collaborative filtering**

00:00 - 00:14

In the previous video, you have been introduced with the three C'S of machine learning. In this video, we will start with the 1st C which is Collaborative filtering, and gain a basic understanding of Recommender Systems in Spark. Let's get started.

**2. What is Collaborative filtering?**

00:14 - 01:03

Collaborative filtering is a method of making automatic predictions about the interests of a user by collecting preferences or taste information from many users. Collaborative filtering is one of the most commonly used algorithms in recommender systems. Collaborative filtering has two approaches: The User-User approach and Item-Item approach. The User-User approach finds users that are similar to the target user and uses their collaborative ratings to make recommendations for the target user. Item-Item approach finds and recommends items that are similar or related to items associated with the target user. Now let's take a look at different components that are needed to build a recommendation system in PySpark.

**3. Rating class in pyspark.mllib.recommendation submodule**

01:03 - 01:48

The Rating class in pyspark-dot-mllib-dot-recommendation submodule is a wrapper around tuple (user, product and rating). The Rating class is very useful for parsing the RDD and creating a tuple of user, product and rating. Here is a simple example of how you can create an instance of Rating class "r" with the values of user equals 1, product equals 2 and rating equals 5-point-0. Once the Rating class is created, you can extract the user, product and rating value using the index of "r" instance. In this example, r[0], r[1] and r[2] shows the userId, ProductID and rating for the "r" instance. Splitting the data

**4. Splitting the data using randomSplit()**

01:48 - 02:36

into training and testing sets is an integral part of machine learning. The training portion will be used to train the model, while the testing data is used to evaluate the model’s performance. Typically, a larger portion of the data is assigned for training and a smaller portion for testing. PySpark's randomSplit function can be used to randomly split the data with the provided weights and returns multiple RDDs. In this example, we first create an RDD which consists of numbers 1 to 10 and using randomSplit function we create two RDDs with 60:40 ratio. The output of the randomSplit function shows training RDDs contains 6 element whereas test RDD contains 4 elements.

**5. Alternating Least Squares (ALS)**

02:36 - 03:27

The alternating least squares (ALS) algorithm available in spark-dot-mllib helps to find products that the customers might like, based on their previous purchases or ratings. The ALS-dot-train method requires that we represent Rating objects as (UserId, ItemId, Rating) tuples along with training parameters rank and iterations. rank represents the number of features. Iterations represent the number of iterations to run the least squares computation. Here is an example of running the ALS model. First, we create an RDD from a list or Rating objects and print out the contents of the RDD using collect action. Next, we use ALS-dot-train to train the training data as shown in this example.

**6. predictAll() – Returns RDD of Rating Objects**

03:27 - 04:09

model, the next step is predicting the ratings for the user and product pairs. The predictAll method takes an RDD of user id and product id pair and returns a prediction for each pair. In order to get the example to work, let's create an RDD from a list of tuples containing userId and productId using Spark Context's parallelize method. Next, we apply the predictAll method on the unrated\_RDD. Running collect Action on predictions shows a list of predicted ratings generated by ALS model for the userId 1 and productIds 1 and 2. For evaluating the model

**7. Model evaluation using MSE**

04:09 - 04:50

trained using ALS, we can use the Mean Squared Error (MSE). The MSE measures the average of the squares of the errors between what is estimated and the existing data. Continuing on our previous example, we'll first organize our ratings and prediction data to make (user, product) the rating. Next, we will join the ratings RDD with the prediction RDD and the result looks as follows. Finally, we apply a squared difference function to the map transformation of the rates\_preds RDD and then use the mean to get the MSE. Now it's your turn

**8. Let's practice!**

04:50 - 04:55

to try your hand at collaborative filtering in PySpark

## Exercise

# Loading Movie Lens dataset into RDDs

Collaborative filtering is a technique for recommender systems wherein users' ratings and interactions with various products are used to recommend new ones. With the advent of Machine Learning and parallelized processing of data, Recommender systems have become widely popular in recent years, and are utilized in a variety of areas including movies, music, news, books, research articles, search queries, social tags. In this 3-part exercise, your goal is to develop a simple movie recommendation system using PySpark MLlib using a subset of [MovieLens 100k dataset](https://grouplens.org/datasets/movielens/100k/).

In the first part, you'll first load the MovieLens data (ratings.csv) into RDD and from each line in the RDD which is formatted as userId,movieId,rating,timestamp, you'll need to map the MovieLens data to a Ratings object (userID, productID, rating) after removing timestamp column and finally you'll split the RDD into training and test RDDs.

Remember, you have a SparkContext sc available in your workspace. Also file\_path variable (which is the path to the ratings.csv file), and ALS class are already available in your workspace.

## Instructions

* Load the ratings.csv dataset into an RDD.
* Split the RDD using , as a delimiter.
* For each line of the RDD, using Rating() class create a tuple of userID, productID, rating.
* Randomly split the data into training data and test data (0.8 and 0.2).

# Load the data into RDD

data = sc.\_\_\_\_(file\_path)

# Split the RDD

ratings = data.\_\_\_\_(lambda l: l.split('\_\_\_\_'))

# Transform the ratings RDD

ratings\_final = ratings.\_\_\_\_(lambda line: Rating(int(line[0]), int(\_\_\_\_), float(\_\_\_\_)))

# Split the data into training and test

training\_data, test\_data = ratings\_final.\_\_\_\_([0.8, 0.2])

# Load the data into RDD

data = sc.textFile(file\_path)

# Split the RDD

ratings = data.map(lambda l: l.split(','))

# Transform the ratings RDD

ratings\_final = ratings.map(lambda line: Rating(int(line[0]), int(line[1]), float(line[2])))

# Split the data into training and test

training\_data, test\_data = ratings\_final.randomSplit([0.8, 0.2])

# Load the data into RDD data = sc.textFile(file\_path) # Split the RDD ratings = data.map(lambda l: l.split(',')) # Transform the ratings RDD ratings\_final = ratings.map(lambda line: Rating(int(line[0]), int(line[1]), float(line[2]))) # Split the data into training and test training\_data, test\_data = ratings\_final.randomSplit([0.8, 0.2])

Good job with preprocessing the data. It's time to train the ALS model!

## Exercise

# Model training and predictions

After splitting the data into training and test data, in the second part of the exercise, you'll train the ALS algorithm using the training data. PySpark MLlib's ALS algorithm has the following mandatory parameters - rank (the number of latent factors in the model) and iterations (number of iterations to run). After training the ALS model, you can use the model to predict the ratings from the test data. For this, you will provide the user and item columns from the test dataset and finally return the list of 2 rows of predictAll() output.

Remember, you have SparkContext sc, training\_data and test\_data are already available in your workspace.

## Instructions

* Train ALS algorithm with training data and configured parameters (rank = 10 and iterations = 10).
* Drop the rating column in the test data.
* Test the model by predicting the rating from the test data.
* Return a list of two rows of the predicted ratings.

# Create the ALS model on the training data

model = ALS.\_\_\_\_(\_\_\_\_, rank=10, iterations=10)

# Drop the ratings column

testdata\_no\_rating = test\_data.\_\_\_(lambda p: (p[0], \_\_\_\_))

# Predict the model

predictions = model.\_\_\_\_(testdata\_no\_rating)

# Return the first 2 rows of the RDD

predictions.\_\_\_\_(2)

# Create the ALS model on the training data

model = ALS.train(training\_data, rank=10, iterations=10)

# Drop the ratings column

testdata\_no\_rating = test\_data.map(lambda p: (p[0], p[1]))

# Predict the model

predictions = model.predictAll(testdata\_no\_rating)

# Return the first 2 rows of the RDD

predictions.take(2)

ERROR! Session/line number was not unique in database. History logging moved to new session 6

[Rating(user=390, product=667, rating=3.3667792093442332),

Rating(user=551, product=667, rating=2.540360491480073)]

# Create the ALS model on the training data

model = ALS.train(training\_data, rank=10, iterations=10)

# Drop the ratings column

testdata\_no\_rating = test\_data.map(lambda p: (p[0], p[1]))

# Predict the model

predictions = model.predictAll(testdata\_no\_rating)

# Return the first 2 rows of the RDD

predictions.take(2)

Model training and predictions are neatly done. Let's find out how successful your model is in the next part.

## Exercise

# Model evaluation using MSE

After generating the predicted ratings from the test data using ALS model, in this final part of the exercise, you'll prepare the data for calculating Mean Square Error (MSE) of the model. The MSE is the average value of (original rating – predicted rating)\*\*2 for all users and indicates the absolute fit of the model to the data. To do this, first, you'll organize both the ratings and prediction RDDs to make a tuple of ((user, product), rating)), then join the ratings RDD with prediction RDD and finally apply a squared difference function along with mean() to get the MSE.

Remember, you have a SparkContext sc available in your workspace. Also, ratings\_final and predictions RDD are already available in your workspace.

## Instructions

* Organize ratings RDD to make ((user, product), rating).
* Organize predictions RDD to make ((user, product), rating).
* Join the prediction RDD with the ratings RDD.
* Evaluate the model using MSE between original rating and predicted rating and print it.

# Prepare ratings data

rates = ratings\_final.\_\_\_\_(lambda r: ((r[0], r[1]), \_\_\_\_))

# Prepare predictions data

preds = predictions.\_\_\_\_(lambda r: ((\_\_\_\_, \_\_\_\_), \_\_\_\_))

# Join the ratings data with predictions data

rates\_and\_preds = rates.\_\_\_\_(preds)

# Calculate and print MSE

MSE = rates\_and\_preds.\_\_\_\_(lambda r: (r[1][0] - r[1]\_\_\_\_)\*\*2).mean()

print("Mean Squared Error of the model for the test data = {:.2f}".format(\_\_\_\_))

**# Prepare ratings data**

**rates = ratings\_final.map(lambda r: ((r[0], r[1]), r[2]))**

**# Prepare predictions data**

**preds = predictions.map(lambda r: ((r[0], r[1]), r[2]))**

**# Join the ratings data with predictions data**

**rates\_and\_preds = rates.join(preds)**

**# Calculate and print MSE**

**MSE = rates\_and\_preds.map(lambda r: (r[1][0] - r[1][1])\*\*2).mean()**

**print("Mean Squared Error of the model for the test data = {:.2f}".format(MSE))**

**# Prepare ratings data**

**rates = ratings\_final.map(lambda r: ((r[0], r[1]), r[2]))**

**# Prepare predictions data**

**preds = predictions.map(lambda r: ((r[0], r[1]), r[2]))**

**# Join the ratings data with predictions data**

**rates\_and\_preds = rates.join(preds)**

**# Calculate and print MSE**

**MSE = rates\_and\_preds.map(lambda r: (r[1][0] - r[1][1])\*\*2).mean()**

**print("Mean Squared Error of the model for the test data = {:.2f}".format(MSE))**

**Mean Squared Error of the model for the test data = 1.31**

Hurray! You have successfully created a Movie Recommendation System using Apache Spark.

LECTURE

**1. Classification**

00:00 - 00:13

In the previous video, you learned about Collaborative filtering which is the 1st C of Machine learning algorithms in PySpark MLlib. In this video, you'll learn about the 2nd C of Machine Learning which is Classification.

**2. Classification using PySpark MLlib**

00:13 - 01:29

Classification is a popular machine learning algorithm that identifies which category an item belongs to. For example, whether an email is spam or non-spam, based on labeled examples of other items. Classification takes a set of data with known labels and pre-determined features and learns how to label new records based on that information. That is why Classification comes under a supervised learning technique. Classifications can be divided into two different types - Binary Classification and Multiclass Classification. In Binary classification, we want to classify entities into two distinct categories. For example, determining whether a cancer type is malignant or not. PySpark MLlib supports various methods for binary classification such as linear SVMs, logistic regression, decision trees, random forests, gradient-boosted trees, naive Bayes. In multiclass classification, we want to classify entities into more than two categories. For example, determining what category a news article belong to. PySpark MLlib supports various methods for multiclass classification such as logistic regression, decision trees, random forests, naive Bayes. Let's focus on Logistic regression which is the

**3. Introduction to Logistic Regression**

01:29 - 02:04

most popular supervised machine learning method. Logistic regression is a classification method to predict a binary response given some independent variable. It measures the relationship between the “Label” on the Y-axis and "Features" on the X-axis using a logistic function as shown in this figure. In logistic regression, the output must be 0 or 1. The convention is if the probability is greater than 50% then the logistic regression output is 1 otherwise, it is 0. PySpark MLlib contains a few specific data types such

**4. Working with Vectors**

02:04 - 02:56

as Vectors and LabelledPoint. Let's understand each of these data types. Vectors in PySpark MLlib comes in two flavors: dense and sparse. Dense vectors store all their entries in an array of floating point numbers. For examples, a vector of 100 will contain 100 double values. In contrast, sparse vectors store only the nonzero values and their indices. Here is an example of creating a dense vector of 1-point-0, 2-point-0, 3-point-0 using Vectors dense method. And here is an example of creating a sparse vector with size of the vector equal to 4 and Non-zero entries 1: 1-point-0, 3: 5-point-5, as a dictionary using Vectors sparse method.

**5. LabelledPoint() in PySpark MLlib**

02:56 - 03:29

A Labeledpoint is a wrapper around the input features and predicted value. LabeledPoint includes a label and a feature vector. The label is a floating-point value and in the case of binary classification, it is either 1 (positive) or 0 (negative). This example shows a positive LabeledPoint with label “1” and a feature vector (1-point-0, 0-point-0, 3-point-0) and negative LabeledPoint with label “0” and a feature vector (2-point-0, 1-point-0, 1-point-0). PySpark MLlib has an

**6. HashingTF() in PySpark MLlib**

03:29 - 04:07

algorithm called HashingTF that computes a term frequency vector of a given size from a document. Let's illustrate this with an example. In this simple example, first, we will split the sentence "hello hello world" into a list of words using the split method and we will create vectors of size 10000. Finally we compute the term frequency vector by using tf's transform method on the words. As you can see the sentence is turned into a sparse vector holding feature number and occurrences of each word. Among several algorithms, the

**7. Logistic Regression using LogisticRegressionWithLBFGS**

04:07 - 04:55

popular algorithm available for Logistic Regression in PySpark MLlib is LBFGS. The minimum requirement for LogisticRegressionWithLBFGS is an RDD of LabeledPoint. To understand how LogisticRegressionworks, let's see a simple example. We first create a list of LabelPoints with labels 0 and 1 and then using SparkContext's parallelize method we will create an RDD. Then we will use LogisticRegressionWithLBFGS-dot-train method to train a logistic regression model on the RDD. Once the model is trained from LogisticRegressionWithLBFGS algorithm, the predict method computes a score between 0 and 1 for each point as shown here. Now it's your turn to

**8. Final Slide**

04:55 - 05:00

practice Classification using PySpark MLlib!

## Exercise

# Loading spam and non-spam data

Logistic Regression is a popular method to predict a categorical response. Probably one of the most common applications of the logistic regression is the message or email spam classification. In this 3-part exercise, you'll create an email spam classifier with logistic regression using Spark MLlib. Here are the brief steps for creating a spam classifier.

* Create an RDD of strings representing email.
* Run MLlib’s feature extraction algorithms to convert text into an RDD of vectors.
* Call a classification algorithm on the RDD of vectors to return a model object to classify new points.
* Evaluate the model on a test dataset using one of MLlib’s evaluation functions.

In the first part of the exercise, you'll load the 'spam' and 'ham' (non-spam) files into RDDs, split the emails into individual words and look at the first element in each of the RDD.

Remember, you have a SparkContext sc available in your workspace. Also file\_path\_spam variable (which is the path to the 'spam' file) and file\_path\_non\_spam (which is the path to the 'non-spam' file) is already available in your workspace.

## Instructions

* Create two RDDS, one for 'spam' and one for 'non-spam (ham)'.
* Split each email in 'spam' and 'non-spam' RDDs into words.
* Print the first element in the split RDD of both 'spam' and 'non-spam'.

# Load the datasets into RDDs

spam\_rdd = sc.\_\_\_\_(file\_path\_spam)

non\_spam\_rdd = sc.\_\_\_\_(file\_path\_non\_spam)

# Split the email messages into words

spam\_words = spam\_rdd.\_\_\_\_(lambda email: email.split(' '))

non\_spam\_words = non\_spam\_rdd.\_\_\_\_(lambda email: \_\_\_\_.\_\_\_\_(' '))

# Print the first element in the split RDD

print("The first element in spam\_words is", spam\_words.\_\_\_\_())

print("The first element in non\_spam\_words is", \_\_\_\_.\_\_\_\_())

**# Load the datasets into RDDs**

**spam\_rdd = sc.textFile(file\_path\_spam)**

**non\_spam\_rdd = sc.textFile(file\_path\_non\_spam)**

**# Split the email messages into words**

**spam\_words = spam\_rdd.flatMap(lambda email: email.split(' '))**

**non\_spam\_words = non\_spam\_rdd.flatMap(lambda email: email.split(' '))**

**# Print the first element in the split RDD**

**print("The first element in spam\_words is", spam\_words.first())**

**print("The first element in non\_spam\_words is", non\_spam\_words.first())**

**The first element in spam\_words is You**

**The first element in non\_spam\_words is Rofl.**

**<script.py> output:**

**The first element in spam\_words is You**

**The first element in non\_spam\_words is Rofl.**

# Load the datasets into RDDs

spam\_rdd = sc.textFile(file\_path\_spam)

non\_spam\_rdd = sc.textFile(file\_path\_non\_spam)

# Split the email messages into words

spam\_words = spam\_rdd.flatMap(lambda email: email.split(' '))

non\_spam\_words = non\_spam\_rdd.flatMap(lambda email: email.split(' '))

# Print the first element in the split RDD

print("The first element in spam\_words is", spam\_words.first())

print("The first element in non\_spam\_words is", non\_spam\_words.first())

Good job! The words in spam and non-spam RDDs may look familiar to you from your emails.

## Exercise

# Feature hashing and LabelPoint

After splitting the emails into words, our raw data set of 'spam' and 'non-spam' is currently composed of 1-line messages consisting of spam and non-spam messages. In order to classify these messages, we need to convert text into features.

In the second part of the exercise, you'll first create a HashingTF() instance to map text to vectors of 200 features, then for each message in 'spam' and 'non-spam' files you'll split them into words, and each word is mapped to one feature. These are the features that will be used to decide whether a message is 'spam' or 'non-spam'. Next, you'll create labels for features. For a valid message, the label will be 0 (i.e. the message is not spam) and for a 'spam' message, the label will be 1 (i.e. the message is spam). Finally, you'll combine both the labeled datasets.

Remember, you have a SparkContext sc available in your workspace. Also spam\_words and non\_spam\_words variables are already available in your workspace.

## Instructions

* Create a HashingTF() instance to map email text to vectors of 200 features.
* Each message in 'spam' and 'non-spam' datasets are split into words, and each word is mapped to one feature.
* Label the features: 1 for spam, 0 for non-spam.
* Combine both the spam and non-spam samples into a single dataset.

# Create a HashingTf instance with 200 features

tf = \_\_\_\_(numFeatures=200)

# Map each word to one feature

spam\_features = tf.\_\_\_\_(spam\_words)

non\_spam\_features = tf.\_\_\_\_(\_\_\_\_)

# Label the features: 1 for spam, 0 for non-spam

spam\_samples = spam\_features.map(lambda features:LabeledPoint(\_\_\_\_, features))

non\_spam\_samples = non\_spam\_features.map(lambda features:\_\_\_\_\_(\_\_\_\_, features))

# Combine the two datasets

samples = spam\_samples.\_\_\_\_(non\_spam\_samples)

# Create a HashingTf instance with 200 features tf = HashingTF(numFeatures=200) # Map each word to one feature spam\_features = tf.transform(spam\_words) non\_spam\_features = tf.transform(non\_spam\_words) # Label the features: 1 for spam, 0 for non-spam spam\_samples = spam\_features.map(lambda features:LabeledPoint(1, features)) non\_spam\_samples = non\_spam\_features.map(lambda features:LabeledPoint(0, features)) # Combine the two datasets samples = spam\_samples.join(non\_spam\_samples)

# Create a HashingTf instance with 200 features

tf = HashingTF(numFeatures=200)

# Map each word to one feature

spam\_features = tf.transform(spam\_words)

non\_spam\_features = tf.transform(non\_spam\_words)

# Label the features: 1 for spam, 0 for non-spam

spam\_samples = spam\_features.map(lambda features:LabeledPoint(1, features))

non\_spam\_samples = non\_spam\_features.map(lambda features:LabeledPoint(0, features))

# Combine the two datasets

samples = spam\_samples.join(non\_spam\_samples)

Feature hashing and LabeledPoints are quite powerful for text based classification in PySpark MLlib!

## Exercise

# Logistic Regression model training

After creating labels and features for the data, we’re ready to build a model that can learn from it (training). But before you train the model, in this final part of the exercise, you'll split the data into training and test, run Logistic Regression model on the training data, and finally check the accuracy of the model trained on training data.

Remember, you have a SparkContext sc available in your workspace, as well as the samples variable.

## Instructions

* Split the combined data into training and test datasets in 80:20 ratio.
* Train the Logistic Regression model with the training dataset.
* Create a prediction label from the trained model on the test dataset.
* Combine the labels in the test dataset with the labels in the prediction dataset.
* Calculate the accuracy of the trained model using original and predicted labels.

# Split the data into training and testing

train\_samples,test\_samples = samples.\_\_\_\_([0.8, 0.2])

# Train the model

model = LogisticRegressionWithLBFGS.train(\_\_\_\_)

# Create a prediction label from the test data

predictions = model.\_\_\_\_(test\_samples.map(lambda x: x.features))

# Combine original labels with the predicted labels

labels\_and\_preds = test\_samples.map(lambda x: x.label).zip(\_\_\_\_)

# Check the accuracy of the model on the test data

accuracy = labels\_and\_preds.filter(lambda x: x[0] == x[\_\_\_\_]).count() / float(test\_samples.count())

print("Model accuracy : {:.2f}".format(\_\_\_\_))

# Split the data into training and testing

train\_samples,test\_samples = samples.randomSplit([0.8, 0.2])

# Train the model

model = LogisticRegressionWithLBFGS.train(train\_samples)

# Create a prediction label from the test data

predictions = model.predict(test\_samples.map(lambda x: x.features))

# Combine original labels with the predicted labels

labels\_and\_preds = test\_samples.map(lambda x: x.label).zip(predictions)

# Check the accuracy of the model on the test data

accuracy = labels\_and\_preds.filter(lambda x: x[0] == x[1]).count() / float(test\_samples.count())

print("Model accuracy : {:.2f}".format(accuracy))

**# Split the data into training and testing**

**train\_samples,test\_samples = samples.randomSplit([0.8, 0.2])**

**# Train the model**

**model = LogisticRegressionWithLBFGS.train(train\_samples)**

**# Create a prediction label from the test data**

**predictions = model.predict(test\_samples.map(lambda x: x.features))**

**# Combine original labels with the predicted labels**

**labels\_and\_preds = test\_samples.map(lambda x: x.label).zip(predictions)**

**# Check the accuracy of the model on the test data**

**accuracy = labels\_and\_preds.filter(lambda x: x[0] == x[1]).count() / float(test\_samples.count())**

**print("Model accuracy : {:.2f}".format(accuracy))**

**Model accuracy : 0.76**

Congratulations! You sucessfully created a spam classifier in just a few steps. Your classifier predicted about 80% of the labels correctly. Can you think of a way to improve this accuracy?

LECTURE

**1. Introduction to Clustering**

00:00 - 00:16

In the previous video, you learned about Classification, a type of supervised learning method. But what if we want to make sense of unlabeled data? In this video, you'll learn about Clustering which is a type of unsupervised learning method to group unlabeled data together.

**2. What is Clustering?**

00:16 - 00:59

So what exactly is Clustering? Clustering is the unsupervised learning task that involves grouping objects into clusters of high similarity with no labels. Unlike the supervised learning methods that you have seen before such as Collaborative filtering and Classification, where data is labeled, Clustering can be used to make sense of unlabeled data. PySpark MLlib library offers a handful of clustering models such as K-means clustering, Gaussian mixture clustering, Power iteration clustering (PIC), Bisecting k-means clustering and Streaming k-means clustering. In this video, we will focus on K-means clustering because of its simplicity and popularity.

**3. K-means Clustering**

00:59 - 01:37

K-means is an unsupervised method that takes data points in an input data and will identify which data points belong to each one of the clusters. As shown in the left side of the figure, we provide 'n' data points and a predefined number of 'k' clusters. The K-means algorithm through a series of iterations creates clusters as shown on the right side of the figure. The K-means clustering minimally requires that the data is a set of numerical features and that we specify the target number of 'K' clusters ahead. The first step in implementing the

**4. K-means with Spark MLLib**

01:37 - 02:20

K-means clustering algorithm using PySpark MLlib is loading the numerical data into an RDD, and then parsing the data based on a delimiter. Here is an example of how you load a CSV file into an RDD using SparkContext's textFile method, then parsing the RDD based on comma delimiter and finally converting the floats to integers. The contents of the first five lines of RDD can be printed using take(5). As you can see, the dataset contains 2 columns, each column indicating a feature loaded into an RDD. Like other algorithms, you invoke K-means by calling KMeans-dot-train method

**5. Train a K-means clustering model**

02:20 - 03:00

which takes an RDD, the number of clusters we expect and the maximum number of iterations allowed. Continuing our previous example, first, we can import the KMeans class from pyspark-dot-mllib-dot-clustering submodule. Next, we call KMeans-dot-train method on RDD and the two parameters k equals 2, and maxIterations equals 10. KMeans-dot-train returns a KMeansModel that lets you access the cluster centers using the model-dot-clusterCenters attribute. An example of cluster centers for k equals 2 is shown here. The next step in K-means clustering is to

**6. Evaluating the K-means Model**

03:00 - 03:34

evaluate the model by computing the error function. Unfortunately, PySpark K-means algorithm doesn't have a method already, so we have to write a function by ourselves as shown here. We will next apply the error function on the RDD and calculate Within Set Sum of Squared Error. Continuing our previous example, we apply map transformation of error function to our input RDD to calculate Within Set Sum of Squared Error which is 77-point-96 in this example. An optional but highly

**7. Visualizing K-means clusters**

03:34 - 04:13

recommended step in K-means clustering is cluster visualization. Continuing from our previous example, let's first create a scatter plot of the two feature columns in the sample data. Next, overlay it with the cluster centers from the KMeans model which are indicated by colored "x"'s in this figure. The purple and yellow colors here represent the labels created from the model based on the K which is 2 in this example. As you can see, the overlaid scatter plot shows a reasonable clustering with the 2 centroids placed in the center of the each of the cluster. Now let's quickly

**8. Visualizing clusters**

04:13 - 04:45

take a look at the code to generate the previous plot. As seen previously, plotting libraries doesn't work directly on RDDs and DataFrames. As shown here, we first convert RDD to Spark DataFrame and then to Pandas DataFrame. We also convert cluster centers from KMeans model into a Pandas DataFrame. Finally, we use plt function in matplotlib library to create a overlaid scatter plot as shown in the previous slide. Let's use a real world

**9. Clustering practice**

04:45 - 04:52

data and generate some nice clusters using PySpark's MLlib KMeans clustering algorithm.

## Exercise

# Loading and parsing the 5000 points data

Clustering is the unsupervised learning task that involves grouping objects into clusters of high similarity. Unlike the supervised tasks, where data is labeled, clustering can be used to make sense of unlabeled data. PySpark MLlib includes the popular K-means algorithm for clustering. In this 3 part exercise, you'll find out how many clusters are there in a dataset containing 5000 rows and 2 columns. For this you'll first load the data into an RDD, parse the RDD based on the delimiter, run the KMeans model, evaluate the model and finally visualize the clusters.

In the first part, you'll load the data into RDD, parse the RDD based on the delimiter and convert the string type of the data to an integer.

Remember, you have a SparkContext sc available in your workspace. Also file\_path variable (which is the path to the 5000\_points.txt file) is already available in your workspace.

## Instructions

* Load the 5000\_points dataset into an RDD named clusterRDD.
* Transform the clusterRDD by splitting the lines based on the tab ("\t").
* Transform the split RDD to create a list of integers for the two columns.
* Confirm that there are 5000 rows in the dataset.

# Load the dataset into an RDD

clusterRDD = sc.\_\_\_\_(file\_path)

# Split the RDD based on tab

rdd\_split = clusterRDD.\_\_\_\_(lambda x: \_\_\_\_.split(\_\_\_\_))

# Transform the split RDD by creating a list of integers

rdd\_split\_int = rdd\_split.\_\_\_\_(lambda x: [int(\_\_\_\_), int(x[1])])

# Count the number of rows in RDD

print("There are {} rows in the rdd\_split\_int dataset".format(\_\_\_\_.\_\_\_\_()))

# Load the dataset into an RDD

clusterRDD = sc.textFile(file\_path)

# Split the RDD based on tab

rdd\_split = clusterRDD.map(lambda x: x.split("\t"))

# Transform the split RDD by creating a list of integers

rdd\_split\_int = rdd\_split.map(lambda x: [int(x[0]), int(x[1])])

# Count the number of rows in RDD

print("There are {} rows in the rdd\_split\_int dataset".format(rdd\_split\_int.count()))

**# Load the dataset into an RDD**

**clusterRDD = sc.textFile(file\_path)**

**# Split the RDD based on tab**

**rdd\_split = clusterRDD.map(lambda x: x.split("\t"))**

**# Transform the split RDD by creating a list of integers**

**rdd\_split\_int = rdd\_split.map(lambda x: [int(x[0]), int(x[1])])**

**# Count the number of rows in RDD**

**print("There are {} rows in the rdd\_split\_int dataset".format(rdd\_split\_int.count()))**

**ERROR! Session/line number was not unique in database. History logging moved to new session 6**

**There are 5000 rows in the rdd\_split\_int dataset**

Good start! You have succesfully loaded the data into RDD for kmeans clustering.

## Exercise

# K-means training

Now that the RDD is ready for training, in this 2nd part, you'll test with k's from 13 to 16 (to save computation time) and use the [elbow](https://bl.ocks.org/rpgove/0060ff3b656618e9136b) method to chose the correct k. The idea of the elbow method is to run K-means clustering on the dataset for different values of k, calculate Within Set Sum of Squared Error (WSSSE) and select the best k based on the sudden drop in WSSSE. Next, you'll retrain the model with the best k and finally, get the centroids (cluster centers).

Remember, you already have a SparkContext sc and rdd\_split\_int RDD available in your workspace.

## Instructions

* Train the KMeans model with clusters from 13 to 16 and print the WSSSE for each cluster.
* Train the KMeans model again with the best k.
* Get the Cluster Centers (centroids) of KMeans model trained with the best k.

# Train the model with clusters from 13 to 16 and compute WSSSE

for clst in range(13, 17):

    model = KMeans.\_\_\_\_(rdd\_split\_int, clst, seed=1)

    WSSSE = rdd\_split\_int.\_\_\_\_(lambda point: error(point)).reduce(lambda x, y: x + y)

    print("The cluster {} has Within Set Sum of Squared Error {}".format(clst, \_\_\_\_))

# Train the model again with the best k

model = KMeans.train(rdd\_split\_int, k=\_\_\_\_, seed=1)

# Get cluster centers

cluster\_centers = model.\_\_\_\_

# Train the model with clusters from 13 to 16 and compute WSSSE

for clst in range(13, 17):

    model = KMeans.train(rdd\_split\_int, clst, seed=1)

    WSSSE = rdd\_split\_int.map(lambda point: error(point)).reduce(lambda x, y: x + y)

    print("The cluster {} has Within Set Sum of Squared Error {}".format(clst, WSSSE))

# Train the model again with the best k

model = KMeans.train(rdd\_split\_int, k=15, seed=1)

# Get cluster centers

cluster\_centers = model.clusterCenters

**# Train the model with clusters from 13 to 16 and compute WSSSE**

**for clst in range(13, 17):**

**model = KMeans.train(rdd\_split\_int, clst, seed=1)**

**WSSSE = rdd\_split\_int.map(lambda point: error(point)).reduce(lambda x, y: x + y)**

**print("The cluster {} has Within Set Sum of Squared Error {}".format(clst, WSSSE))**

**# Train the model again with the best k**

**model = KMeans.train(rdd\_split\_int, k=15, seed=1)**

**# Get cluster centers**

**cluster\_centers = model.clusterCenters**

**The cluster 13 has Within Set Sum of Squared Error 251787626.51713783**

**The cluster 14 has Within Set Sum of Squared Error 257469943.64057225**

**The cluster 15 has Within Set Sum of Squared Error 215235374.39950493**

**The cluster 16 has Within Set Sum of Squared Error 167785881.85891667**

Great job on finding the best K with K-Means algorithm! For real world data, you should train the model with a wide range of K values.

## Exercise

# Visualizing clusters

You just trained the k-means model with an optimum k value (k=15) and generated cluster centers (centroids). In this final exercise, you will visualize the clusters and the centroids by overlaying them. This will indicate how well the clustering worked (ideally, the clusters should be distinct from each other and centroids should be at the center of their respective clusters).

To achieve this, you will first convert the rdd\_split\_int RDD into a Spark DataFrame, and then into Pandas DataFrame which can be used for plotting. Similarly, you will convert cluster\_centers into a Pandas DataFrame. Once both the DataFrames are created, you will create scatter plots using Matplotlib.

The SparkContext sc as well as the variables rdd\_split\_int and cluster\_centers are available in your workspace.

## Instructions

* Convert the rdd\_split\_int RDD to a Spark DataFrame, then to a pandas DataFrame.
* Create a pandas DataFrame from the cluster\_centers list.
* Create a scatter plot from the pandas DataFrame of raw data (rdd\_split\_int\_df\_pandas) and overlay that with a scatter plot from the Pandas DataFrame of centroids (cluster\_centers\_pandas).

# Convert rdd\_split\_int RDD into Spark DataFrame and then to Pandas DataFrame

rdd\_split\_int\_df\_pandas = spark.\_\_\_\_(rdd\_split\_int, schema=["col1", "col2"]).toPandas()

# Convert cluster\_centers to a pandas DataFrame

cluster\_centers\_pandas = pd.DataFrame(\_\_\_\_, columns=["col1", "col2"])

# Create an overlaid scatter plot of clusters and centroids

plt.scatter(rdd\_split\_int\_df\_pandas["col1"], rdd\_split\_int\_df\_pandas["col2"])

plt.scatter(\_\_\_\_["col1"], \_\_\_\_["col2"], color="red", marker="x")

plt.show()

# Convert rdd\_split\_int RDD into Spark DataFrame and then to Pandas DataFrame

rdd\_split\_int\_df\_pandas = spark.createDataFrame(rdd\_split\_int, schema=["col1", "col2"]).toPandas()

# Convert cluster\_centers to a pandas DataFrame

cluster\_centers\_pandas = pd.DataFrame(cluster\_centers, columns=["col1", "col2"])

# Create an overlaid scatter plot of clusters and centroids

plt.scatter(rdd\_split\_int\_df\_pandas["col1"], rdd\_split\_int\_df\_pandas["col2"])

plt.scatter(cluster\_centers\_pandas["col1"], cluster\_centers\_pandas["col2"], color="red", marker="x")

plt.show()

# Convert rdd\_split\_int RDD into Spark DataFrame and then to Pandas DataFrame rdd\_split\_int\_df\_pandas = spark.createDataFrame(rdd\_split\_int, schema=["col1", "col2"]).toPandas() # Convert cluster\_centers to a pandas DataFrame cluster\_centers\_pandas = pd.DataFrame(cluster\_centers, columns=["col1", "col2"]) # Create an overlaid scatter plot of clusters and centroids plt.scatter(rdd\_split\_int\_df\_pandas["col1"], rdd\_split\_int\_df\_pandas["col2"]) plt.scatter(cluster\_centers\_pandas["col1"], cluster\_centers\_pandas["col2"], color="red", marker="x") plt.show()

Fantastic job visualizing the k-means model clusters and centroids. How do you feel about your results? Are the centroids coherent with the clusters?

LECTURE

**1. Congratulations!**

00:00 - 00:25

Congratulations on successfully completing Fundamentals of BigData via PySpark course. Our goal through this course was to equip you with a basic understanding of Big Data and show how Apache Spark can be used to perform powerful data analysis at scale. Let's quickly review what you have learned so far in this course and recommend you few courses that you can take next.

**2. Fundamentals of BigData and Apache Spark**

00:25 - 01:28

Analyzing BigData is equivalent to conducting both descriptive and inferential analyses using distributed computing techniques such as Spark, with the hopes that the volume, variety, and velocity of BigData that makes distributed computing necessary will lead to deeper or more targeted insights. Chapter 1 started with the fundamentals of BigData and introduced Apache Spark as an open source distributed BigData processing engine, as well as its different components namely Spark Core, Spark SQL, Spark MLlib, Graphx, and Spark Streaming. Because Python is one of the most popular languages for data science, we looked specifically at how you might use PySpark which is Spark’s Python API to execute Spark jobs, and PySpark shell to develop Spark's interactive applications in Python. Finally you learned about the two different modes of running Spark namely local mode and cluster mode.

**3. Spark components**

01:28 - 02:33

Chapter 2 introduced PySpark RDD which is the main API in Spark Core for processing unstructured data. We learned about the different features of RDDs, different methods of creating RDDs and finally, RDD operations namely Transformations and Actions. Chapter 3 explored PySpark SQL which is Spark's high-level API for working with structured data. PySpark SQL creates DataFrames which provides more information about the structure of data and the computation being performed. We looked at the different methods of creating DataFrames, DataFrame operations and finally different methods of visualizing Big Data using DataFrames. Chapter 4 delved deep into PySpark MLlib, Spark's built-in library for machine learning, and discussed how PySpark MLlib makes practical machine learning scalable and easy. This chapter introduced the three C's of MLlib - Collaborative filtering, Classification, and Clustering. The ecosystem of

**4. Where to go next?**

02:33 - 03:25

Apache Spark is vast and ever-expanding, but throughout the course, we’ve discussed the essential underlying concepts. Where you choose to go from here, whether that be experimenting and applying some of these tools and patterns on your own, or investigating Spark components such as Spark SQL or Spark MLlib more deeply, is up to you. But we hope that the concepts, tools, and techniques that we’ve introduced in this course have provided a well-informed starting point, and can continue to serve as a basis for you to refer back to throughout your distributed data analysis journey. With this general understanding of PySpark, we would encourage you to look at other DataCamp PySpark courses focused on feature engineering and recommendation engines to further your knowledge.