

# Getting Started with PySpark

## Introduction

Apache Spark is a fast and powerful framework that provides an API to perform massive distributed processing over resilient sets of data. The main abstraction Spark provides is a resilient distributed dataset (RDD), which is the fundamental and backbone data type of this engine. Spark SQL is Apache Spark's module for working with structured data and MLlib is Apache Spark's scalable machine learning library. Apache Spark is written in Scala programming language. To support Python with Spark, Apache Spark community released a tool, PySpark. PySpark has similar computation speed and power as Scala. PySpark is a parallel and distributed engine for running big data applications. Using PySpark, you can work with RDDs in Python programming language.

This tutorial discusses Big Data via PySpark, a Python package for spark programming. Spark's high level libraries such as SparkSQL, MLlib will be used to interact with two different datasets.

## What is SparkContext?

Spark comes with interactive python shell in which PySpark is already installed in it. PySpark automatically creates a SparkContext for you in the PySpark Shell. SparkContext is an entry point into the world of Spark. An entry point is a way of connecting to Spark cluster. We can use SparkContext using `sc` variable. In the following examples, we retrieve SparkContext version and Python version of SparkContext.

```
In [1]: # to retrieve SparkContext version  
sc.version
```

```
Out[1]: '3.3.1'
```

```
In [2]: # to retrieve Python version of SparkContext  
sc.pythonVer
```

```
Out[2]: '3.9'
```

## Import Packages

```
In [3]: import pandas as pd  
import matplotlib.pyplot as plt  
plt.style.use("seaborn")  
from math import sqrt  
from pyspark.ml.clustering import KMeans
```

## Using map and filter methods with Lambda function in Python

Lambda functions are anonymous functions in Python. Anonymous functions do not bind to any name in runtime and it returns the functions without any name. They are usually used with map and filter methods. Lambda functions create functions to be called later. In the following example, we use lambda function with map and filter methods.

```
In [4]: my_list=[1,2,3,4,5]
squared_my_list=list(map(lambda x: x*x, my_list))
squared_my_list
```

```
Out[4]: [1, 4, 9, 16, 25]
```

```
In [5]: filtered_my_list=list(filter(lambda x: (x%2 !=0), my_list))
filtered_my_list
```

```
Out[5]: [1, 3, 5]
```

## Creating RDD from Object

```
In [6]: numbers=list(range(0,100))
```

```
In [7]: # Load the numbers into PySpark
numbersRDD=sc.parallelize(numbers)
```

```
In [8]: type(numbersRDD)
```

```
Out[8]: pyspark.rdd.RDD
```

## Transformations and Actions on RDD

Transformations and actions are two type of operations in Spark. Transformations create new RDDs. Actions performs computation on the RDDs. Map, filter, flatmap and union are basic RDD transformations. Collect, take, first and count are basic RDD actions. In the following example, we create rdd named numRDD from list and then using map transformation we create a new rdd named cubeRDD from numRDD. Finally, we use collect action to return a list that contains all of the elements in this RDD.

```
In [9]: # map() transformations
numRDD=sc.parallelize(my_list)
cubeRDD=numRDD.map(lambda x: x**3)
numbers_all=cubeRDD.collect()

[print(num) for num in numbers_all];
```

```
1
8
27
64
125
```

## Transformations and Actions on pair RDD

Pair RDD is a special type of RDD to work with datasets with key/value pairs. All regular transformations work on pair RDD. In the following example, we create pair RDD with 4 tuple with two numbers. In each tuple, the first number is key and the second number is value. Then, we apply reduceByKey transformation to pair RDD. ReduceByKey tranformation combine values with the same key. Therefore, this transformation adds the values of tuples with the same key.

```
In [10]: # create pairRdd with key and value pairs
pairRdd=sc.parallelize([(1,2),(3,4),(3,6),(4,5)])

pairRdd_Reduced=pairRdd.reduceByKey(lambda x, y: x+y)

[print(num)for num in pairRdd_Reduced.collect()]

(4, 5)
(1, 2)
(3, 10)
```

We can sort keys of tuples using sortByKey transformation like in the following example.

```
In [11]: pairRdd_Reduced_Sort=pairRdd_Reduced.sortByKey(ascending=True)

[print(num) for num in pairRdd_Reduced_Sort.collect()]

(1, 2)
(3, 10)
(4, 5)
```

We can count number of tuples with the same key. In the following example, we see (3,2) because there are two tuple with key 3 in pairRdd.

```
In [12]: total=pairRdd.countByKey()

In [13]: total.items()

Out[13]: dict_items([(1, 1), (3, 2), (4, 1)])

In [14]: [print("key", k, "has", v, "counts") for k,v in total.items()]

key 1 has 1 counts
key 3 has 2 counts
key 4 has 1 counts
```

## What is SparkSession?

SparkContext is the main entry point for creating RDDs while SparkSession provides a single point of entry to interact with Spark Dataframes. SparkSession is used to create DataFrame, register DataFrames, execute SQL queries. We can access SparkSession in PySpark using spark variable. In the following examples, we retrieve SparkSession version and other informations about it.

In [15]: `spark`

Out[15]: **SparkSession - in-memory**

### SparkContext

#### Spark UI

<b>Version</b>	v3.3.1
<b>Master</b>	spark://jkg-deployment-f010097f-1181-40fb-a814-6107be479b2b-85b789tjqtg:7077
<b>AppName</b>	python3.9

## Creating PySpark DataFrame from RDD

Spark SQL which is a Spark module for structured data processing provides a programming abstraction called DataFrames and can also act as a distributed SQL query engine. In the following example, we create rdd from list then we create PySpark dataframe using SparkSession's createDataFrame method. When we look at the type of dataframe, we can see `pyspark.sql.DataFrame` as an output. Furthermore, we can use `show` method to print out the dataframe.

```
In [16]: # Create a PySpark Dataframe from rdd
sample_list=[("Mona", 23), ("Lisa", 29), ('Leonardo', 37), ('Piero', 41)]
rdd=sc.parallelize(sample_list)

df_names=spark.createDataFrame(rdd, schema=["Name", "Age"])

type(df_names)
```

Out[16]: `pyspark.sql.DataFrame`

In [17]: `df_names.show()`

```
+-----+---+
|    Name|Age|
+-----+---+
|    Mona| 23|
|    Lisa| 29|
| Leonardo| 37|
|    Piero| 41|
+-----+---+
```

## Add Datasets

You can delete df\_data\_1.take(5) part and then copy cos.url('file\_name', 'bucket\_name') above it then assign cos.url('file\_name', 'bucket\_name') to path\_people variable and comment out this variable. cos.url('file\_name', 'bucket\_name') is path to your file you can access the file by using this path.

```
In [18]: import ibmos2spark, os
# @hidden_cell
metadata = {
    'endpoint': 'https://s3.private.us.cloud-object-storage.appdomain.cloud',
    'service_id': 'iam-ServiceId-9ab6fa3e-5fb3-4866-89cd-514409e8f87e',
    'iam_service_endpoint': 'https://iam.cloud.ibm.com/oidc/token',
    'api_key': 'GK718cWpFgdMGpD3d5T6qSQP3CLKdD2sLxMSR8l0WzIN'
}

configuration_name = 'os_d5994ea9631c4ef79eeff2447a43cbecd_configs'
cos = ibmos2spark.CloudObjectStorage(sc, metadata, configuration_name, 'bluemix_cos')

from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
df_data_1 = spark.read\
    .format('org.apache.spark.sql.execution.datasources.csv.CSVFileFormat')\
    .option('header', 'true')\
    .load(cos.url('people.csv', 'advancedatasciencewithibmsspeciali-donotdelete-pr-xd5sk2ea28xxtp'))

path_people = cos.url('people.csv', 'advancedatasciencewithibmsspeciali-donotdelete-pr-xd5sk2ea28xxtp')
```

You can also add 5000\_points.txt dataset by applying same procedure but click insert to code then click insert credentials then write "file" and "bucket" values inside "path\_5000 = cos.url('file\_name', 'bucket\_name')" expression and comment out path\_5000.

```
# @hidden_cell
# The following code contains the credentials for a file in your IBM Cloud Object Storage.
# You might want to remove those credentials before you share your notebook.
metadata_1 = {
    'IAM_SERVICE_ID': 'iam-ServiceId-9ab6fa3e-5fb3-4866-89cd-514409e8f87e',
    'IBM_API_KEY_ID': 'GK718cWpFgdMGpD3d5T6qSQP3CLKdD2sLxMSR8l0WzIN',
    'ENDPOINT': 'https://s3.private.us.cloud-object-storage.appdomain.cloud',
    'IBM_AUTH_ENDPOINT': 'https://iam.cloud.ibm.com/oidc/token',
    'BUCKET': 'advancedatasciencewithibmsspeciali-donotdelete-pr-xd5sk2ea28xxtp',
    'FILE': '5000_points.txt'
}

path_5000 = cos.url('5000_points.txt', 'advancedatasciencewithibmsspeciali-donotdelete-pr-xd5sk2ea28xxtp')
```

## Create PySpark DataFrame from external file

We can create PySpark DataFrame by using SparkSession's read.csv method. To do this, we should give path of csv file as an argument to the method. Show action prints first 20 rows of DataFrame. Count action prints number of rows in DataFrame. Columns attribute prints the list of columns in DataFrame. PrintSchema action prints the types of columns in the Dataframe and it gives information about whether there is null values in columns or not.

```
In [20]: df_people = spark.read.csv(path_people, header=True, inferSchema=True)
```

```
In [21]: df_people.show()
```

_c0	person_id	name	sex	date of birth
0	100	Penelope Lewis	female	1990-08-31 00:00:00
1	101	David Anthony	male	1971-10-14 00:00:00
2	102	Ida Shipp	female	1962-05-24 00:00:00
3	103	Joanna Moore	female	2017-03-10 00:00:00
4	104	Lisandra Ortiz	female	2020-08-05 00:00:00
5	105	David Simmons	male	1999-12-30 00:00:00
6	106	Edward Hudson	male	1983-05-09 00:00:00
7	107	Albert Jones	male	1990-09-13 00:00:00
8	108	Leonard Cavender	male	1958-08-08 00:00:00
9	109	Everett Vadala	male	2005-05-24 00:00:00
10	110	Freddie Claridge	male	2002-05-07 00:00:00
11	111	Annabelle Rousseau	female	1989-07-13 00:00:00
12	112	Eulah Emanuel	female	1976-01-19 00:00:00
13	113	Shaun Love	male	1970-05-26 00:00:00
14	114	Alejandro Brennan	male	1980-12-22 00:00:00
15	115	Robert Mcreynolds	male	1973-12-27 00:00:00
16	116	Carla Spickard	female	1985-06-13 00:00:00
17	117	Florence Eberhart	female	2024-06-01 00:00:00
18	118	Tina Gaskins	female	1966-12-05 00:00:00
19	119	Florence Mulhern	female	1959-05-31 00:00:00

only showing top 20 rows

```
In [22]: df_people.count()
```

```
Out[22]: 100000
```

```
In [23]: df_people.columns
```

```
Out[23]: ['_c0', 'person_id', 'name', 'sex', 'date of birth']
```

```
In [24]: df_people.printSchema()
```

```
root
|-- _c0: integer (nullable = true)
|-- person_id: integer (nullable = true)
|-- name: string (nullable = true)
|-- sex: string (nullable = true)
|-- date of birth: timestamp (nullable = true)
```

```
In [25]: df_people_sub=df_people.select('name', "sex", 'date of birth')
```

```
In [26]: df_people_sub.show(10)
```

```
+-----+-----+
|      name|    sex|      date of birth|
+-----+-----+
| Penelope Lewis|female|1990-08-31 00:00:00|
| David Anthony| male|1971-10-14 00:00:00|
| Ida Shipp|female|1962-05-24 00:00:00|
| Joanna Moore|female|2017-03-10 00:00:00|
| Lisandra Ortiz|female|2020-08-05 00:00:00|
| David Simmons| male|1999-12-30 00:00:00|
| Edward Hudson| male|1983-05-09 00:00:00|
| Albert Jones| male|1990-09-13 00:00:00|
| Leonard Cavender| male|1958-08-08 00:00:00|
| Everett Vadala| male|2005-05-24 00:00:00|
+-----+-----+
only showing top 10 rows
```

In [27]: `df_people_sub_nodup=df_people_sub.dropDuplicates()`

In [28]: `print("Before removing duplicates:", df_people_sub.count())  
print("After removing duplicates:", df_people_sub_nodup.count())`

Before removing duplicates: 100000

After removing duplicates: 99998

We can filter out the rows based on a condition by using filter transformation as in the following example.

In [29]: `df_people_female=df_people.filter(df_people.sex == "female")  
df_people_male=df_people.filter(df_people.sex == "male")`

In [30]: `df_people_female.count()`

Out[30]: 49014

In [31]: `df_people_female.show(5)`

```
+-----+-----+-----+
| _c0|person_id|      name|    sex|      date of birth|
+-----+-----+-----+
|  0|     100|Penelope Lewis|female|1990-08-31 00:00:00|
|  2|     102|     Ida Shipp|female|1962-05-24 00:00:00|
|  3|     103| Joanna Moore|female|2017-03-10 00:00:00|
|  4|     104|Lisandra Ortiz|female|2020-08-05 00:00:00|
| 11|     111|Annabelle Rousseau|female|1989-07-13 00:00:00|
+-----+-----+-----+
only showing top 5 rows
```

We can group columns based on their values by using groupby transformation as in the following example.

In [32]: `df_people_sex=df_people.groupby("sex")  
df_people_sex.count().show()`

```
+----+----+
|   sex|count|
+----+----+
| null| 1920|
|female|49014|
| male|49066|
+----+----+
```

We can sort the DataFrame based on one or more columns by using orderBy transformation.

In [33]: `df_people.orderBy("date of birth").show(5)`

```
+-----+-----+-----+
| _c0|person_id|      name|    sex| date of birth|
+-----+-----+-----+
|57359|    57459| Sharon Perez|female|1899-08-28 00:00:00|
|62233|    62333|Martina Morison|female|1901-04-21 00:00:00|
|96318|    96418| Lisa Garrett|female|1901-05-09 00:00:00|
|39703|    39803| Naomi Davis|female|1902-04-25 00:00:00|
|64563|    64663| Brenda French|female|1902-07-27 00:00:00|
+-----+-----+-----+
only showing top 5 rows
```

We can rename a column in DataFrame by using withColumnRenamed transformation.

In [34]: `df_people=df_people.withColumnRenamed("date of birth", "birth")  
df_people.show(5)`

```
+-----+-----+-----+
| _c0|person_id|      name|    sex|        birth|
+-----+-----+-----+
|  0|     100|Penelope Lewis|female|1990-08-31 00:00:00|
|  1|     101| David Anthony| male|1971-10-14 00:00:00|
|  2|     102| Ida Shipp|female|1962-05-24 00:00:00|
|  3|     103| Joanna Moore|female|2017-03-10 00:00:00|
|  4|     104|Lisandra Ortiz|female|2020-08-05 00:00:00|
+-----+-----+-----+
only showing top 5 rows
```

## Using SQL queries with DataFrames by using Spark SQL module

We can also use SQL queries to achieve the same things with DataFrames. Firstly, we should create temporary table by using `createOrReplaceTempView` method. We should give the name of temporary table as an argument to the method. Then, we can give any query we want to execute to `SparkSession's sql` method as an argument. Look at the following example.

In [35]: `df_people.createOrReplaceTempView("people")  
query='SELECT name, birth from people where sex="male" order by birth'`

```
df_people_names=spark.sql(query)
df_people_names.show(10)

+-----+-----+
|      name|      birth|
+-----+-----+
| Tyler Walton|1903-07-14 00:00:00|
| Daniel Naiman|1903-11-07 00:00:00|
| John Merritt|1906-11-04 00:00:00|
| Roger Watkin|1907-12-08 00:00:00|
| Tim Makris|1909-07-11 00:00:00|
| Jeremy Jost|1910-04-14 00:00:00|
| Fredrick Nass|1911-01-12 00:00:00|
| Shaun King|1911-03-27 00:00:00|
| Mitchell Martin|1911-07-06 00:00:00|
| Daniel Rutan|1911-08-09 00:00:00|
+-----+-----+
only showing top 10 rows
```

## Create RDD from external file

The second and mostly used way to create RDD is from external dataset. To do this, we can use SparkContext's `textFile` method. In the following example, we use `5000_points.txt` dataset. To do this, we use path to dataset as an argument to `textFile` method.

```
In [36]: # Load the 5000_points dataset into a rdd named clusterRDD
clusterRDD=sc.textFile(path_5000)
```

Default number of partition of rdd is two. We can see that using `getNumPartitions` method. If we want to increase number of partition, we can give `minPartitions` argument to `textFile` method.

```
In [37]: clusterRDD.getNumPartitions()
```

```
Out[37]: 2
```

```
In [38]: clusterRDD=sc.textFile(path_5000, minPartitions=5)
```

```
In [39]: clusterRDD.getNumPartitions()
```

```
Out[39]: 5
```

```
In [40]: clusterRDD.take(5)
```

```
Out[40]: ['664159\t550946',
          '665845\t557965',
          '597173\t575538',
          '618600\t551446',
          '635690\t608046']
```

We can transform this rdd by splitting the lines based on the tab.

```
In [41]: rdd_split=clusterRDD.map(lambda x: x.split("\t"))
```

```
In [42]: rdd_split.take(5)
```

```
Out[42]: [['664159', '550946'],
           ['665845', '557965'],
           ['597173', '575538'],
           ['618600', '551446'],
           ['635690', '608046']]
```

We can also further transform the splitted RDD to create a list of integers for the two columns.

```
In [43]: rdd_split_int=rdd_split.map(lambda x: [int(x[0]), int(x[1])])
```

```
In [44]: rdd_split_int.take(5)
```

```
Out[44]: [[664159, 550946],
           [665845, 557965],
           [597173, 575538],
           [618600, 551446],
           [635690, 608046]]
```

## Machine Learning with PySpark MLlib

PySpark MLlib is the Apache Spark's scalable machine learning library in Python consisting of common learning algorithms and utilities. We use Kmeans algorithm of MLlib library to cluster data in 5000\_points.txt dataset. First, we should define error method to calculate distance from every point to center of its clusters which the points belong to.

```
In [45]: def error(point):
    center=model.centers[model.predict(point)]
    return sqrt(sum([x**2 for x in (point - center)]))
```

We train the model with 4 different number of clusters from 13 to 16 and then calculate the error for all of them. As you see in the output, 16 clusters give minimum error. We retrain the model with the number of cluster with the smallest error. We then use clusterCenters attribute to see the center of all clusters.

```
# Train the model with clusters from 13 to 16 and compute SSE
temp=0
cluster=0
for clst in range(13, 17):
    model=KMeans.train(rdd_split_int,clst, seed=1)
    SSE=rdd_split_int.map(lambda point: error(point)).reduce(lambda x,y: x+y)
    while temp> SSE or temp == 0:
        temp=SSE
        cluster=clst

    print("The Cluster",clst, "has some of squared Error", SSE)

print("\ncluster",cluster)
print("\nSSE", temp)

# Train the model again with the best k
```

```

model=KMeans.train(rdd_split_int, k=cluster, seed=1)

# Get cluster centers
cluster_centers=model.clusterCenters

The Cluster 13 has some of squared Error 251839908.5434932
The Cluster 14 has some of squared Error 211629260.76746103
The Cluster 15 has some of squared Error 169394691.52639443
The Cluster 16 has some of squared Error 167648251.99743205

```

cluster 16

SSE 167648251.99743205

We can again use SparkSession's createDataFrame method to create DataFrame from rdd. We must convert PySpark DataFrame to Pandas DataFrame in order to visualise data. To do this, we can use toPandas method. We create another Pandas DataFrame from cluster centers list. Then, using matplotlib's scatter method, we can make plot for clusters and their centers.

```

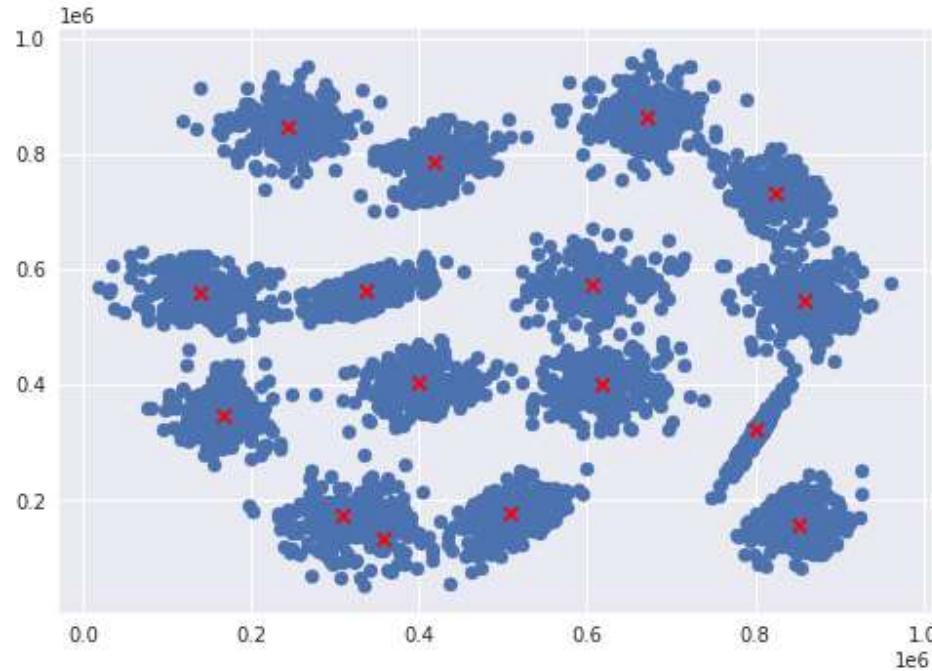
In [47]: rdd_split_int_df = spark.createDataFrame(rdd_split_int, schema=["col1", "col2"])

rdd_split_int_df_pandas = rdd_split_int_df.toPandas()

cluster_centers_pandas = pd.DataFrame(cluster_centers, columns=["col1", "col2"])

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", m
plt.show()

```



## Conclusion

This tutorial discusses Big Data via PySpark, a Python package for spark programming. In this tutorial, I explained SparkContext, using map and filter methods with Lambda functions in Python, creating

rdd from object and external file, transformations and actions on rdd and pair rdd, SparkSession, creating PySpark Dataframe from rdd and external file, using sql queries with Dataframes by using Spark SQL module and finally, machine learning with PySpark MLlib library.

In [ ]: