# basicDescriptiveStatistics

February 8, 2026

[1]:
```
from pyspark.sql import SparkSession

spark=SparkSession.builder.appName("Descriptive Statistics").getOrCreate()

sc=spark.sparkContext
```

Basic Descriptive Statistics on RDD

[7]:
```
# combine multiple rdd data with same key
# we use 'cogroup' function
p1=sc.parallelize([
    ("a",1),("b",4)
])
p2=sc.parallelize([
    ("a",2)
])
print(p1.collect())
print(p2.collect())

p1.cogroup(p2).collect()
```

```
[('a', 1), ('b', 4)]
[('a', 2)]
```

[7]:
```
[('a',
  (<pyspark.resultiterable.ResultIterable at 0x22c6438a9d0>,
   <pyspark.resultiterable.ResultIterable at 0x22c645caad0>)),
 ('b',
  (<pyspark.resultiterable.ResultIterable at 0x22c50e6a910>,
   <pyspark.resultiterable.ResultIterable at 0x22c645c86d0>))]
```

[21]:
```
# the result of cogroup function can be realized by combining
# for loop with tuple & map
p1 = sc.parallelize([("a", 1), ("b", 4)])
p2 = sc.parallelize([("a", 2)])

result = [(x, tuple(map(list, y))) for x, y in p1.cogroup(p2).collect()]

print(result)
```

```
[('a', ([1], [2])), ('b', ([4], []))]
```

[5]:
```
# combining multiple rdd (more than 2rdd's)
# we can use groupWith Function
data1=sc.parallelize([
    ("setosa",5),("non-setosa",6)
])
data2 = sc.parallelize([("setosa", 1), ("non-setosa", 4)])
data3 = sc.parallelize([("setosa", 2)])
data4 = sc.parallelize([("non-setosa", 7)])
groupwith1 = data1.groupWith(data2, data3, data4).collect()

for x,y in groupwith1:
    print(x,tuple(map(list,y)))
```

```
non-setosa ([6], [4], [], [7])
setosa ([5], [1], [2], [])
```

Count Occurence

[9]:
```
p1=sc.parallelize([
    ("setosa",1),("non-setosa",1),("setosa",1)
])
print(p1.countByKey().items())
```

```
dict_items([('setosa', 2), ('non-setosa', 1)])
```

[13]:
```
iris1 = sc.textFile("iris/iris_site.csv")
iris1_split = iris1.map(lambda var1: var1.split(","))
iris1_mod = iris1_split.map(lambda var1: (var1[4], 1))
print(iris1_mod.countByKey().items())
```

```
dict_items([('setosa', 50), ('versicolor', 50), ('virginica', 50)])
```

[14]:
```
# count total occurences of values in a particular column can be calculated
# by countByValue function

print(iris1_split.map(lambda col:col[4]).countByValue().items())
```

```
dict_items([('setosa', 50), ('versicolor', 50), ('virginica', 50)])
```

[15]:
```
# count total number of values
print(iris1.count())
```

```
150
```

[16]:
```
# Distinct

iris1_mod=iris1_split.map(lambda col:col[4])
distinct1=iris1_mod.distinct()
print(distinct1.collect())
```

```
['setosa', 'versicolor', 'virginica']
```

[17]:
```
# generate sequence of values
range1=sc.range(start=1,end=10,step=2)
print(range1.collect())
```

```
[1, 3, 5, 7, 9]
```

[19]:
```
# Apply function for each key
# to apply some function to the values of a particular key we should use␣
 ↪mapValues function
data1 = sc.parallelize(
    [("Sepal_Length", [2, 1, 3, 5, 4]), ("Sepal_Width", [3, 1, 2, 4, 2])]
)

def func(para1):
    return sum(para1)

data1.mapValues(func).collect()
```

[19]:
```
[('Sepal_Length', 15), ('Sepal_Width', 12)]
```

## 0.1 Grouping and Aggregation

fold is an action that aggregates (combines) all elements of an RDD into ONE value.

rdd.fold(zeroValue,func)

start with an initial value then combine all elements using the same function

foldByKey() –> works on pair RDD's

(key,value)

rdd.foldByKey(zeroValue, func) –> For each key, start with zeroValue and combine all its values using the function.

[ ]:
```
# Measure data in  an RDD can be aggregated using fold function


from operator import add

iris1 = sc.textFile("iris/iris_site.csv")
iris1_split = iris1.map(lambda var1: var1.split(","))
iris1_mod = iris1_split.map(lambda col: col[1])
iris1_split.map(lambda col: float(col[1])).fold(0, add)

round(iris1_split.map(lambda col: float(col[1])).fold(0, add), 2)
```

[ ]:
```
458.6
```

[27]:
```
rdd = sc.parallelize([("a", 1), ("a", 2), ("b", 3)])
rdd.foldByKey(0,add).collect()

# here 1st for the key 'a' ---> 0+1+2
# for the key 'b' --> 0+3
# o/p: a-3,b-3
```

[27]:
```
[('a', 3), ('b', 3)]
```

[ ]:
```
# Aggregate by key
# to aggregate values in each column we use foldByKey
# fold is an action that aggregates (combines) all elements of an RDD into ONE␣
 ↪value.
from operator import add

iris1 = sc.textFile("iris/iris_site.csv")
iris1_split = iris1.map(lambda var1: var1.split(","))
iris1_mod = iris1_split.flatMap(
    lambda var1: (
        ("Sepal.Length", float(var1[0])),
        ("Sepal.Width", float(var1[1])),
        ("Petal.Length", float(var1[2])),
        ("Petal.Width", float(var1[3])),
    )
)
print(iris1_mod.foldByKey(0, add).collect())
```

```
[('Sepal.Width', 458.6000000000001), ('Petal.Width', 179.89999999999998),
('Sepal.Length', 876.4999999999998), ('Petal.Length', 563.7000000000002)]
```

[31]:
```
# Reduce --> used to reduce elements of a RDD (usually used for aggregation)
from operator import add
iris1_mod=iris1_split.map(lambda var1:float(var1[0]))

print(iris1_mod.fold(0,add))
print(iris1_mod.reduce(add)) # sum of values of 'Sepal_length' Column
```

```
876.4999999999998
876.4999999999998
```

[32]:
```
# reduce By key (similar to reduce function, except the function passed as an␣
 ↪argument
# to reduceByKey Function)

iris1_mod = iris1_split.flatMap(
    lambda var1: (
        ("Sepal.Length", float(var1[0])),
        ("Sepal.Width", float(var1[1])),
```

```
        ("Petal.Length", float(var1[2])),
        ("Petal.Width", float(var1[3])),
    )

)

print(iris1_mod.reduceByKey(add).collect())
```

```
[('Sepal.Width', 458.6000000000001), ('Petal.Width', 179.89999999999998),
('Sepal.Length', 876.4999999999998), ('Petal.Length', 563.7000000000002)]
```

[ ]:
```
# group by key
iris1_mod = iris1_split.flatMap(
    lambda var1: (
        ("Sepal.Length", float(var1[0])),
        ("Sepal.Width", float(var1[1])),
        ("Petal.Length", float(var1[2])),
        ("Petal.Width", float(var1[3])),
    )
)

# print(iris1_mod.groupByKey().collect()) # prints in iterable objects
# to conver this to list or any other object form we use mapValues
group1=iris1_mod.groupByKey()
print(group1.mapValues(list).take(2))
```

```
[('Sepal.Width', [3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2.9, 3.1, 3.7, 3.4,
3.0, 3.0, 4.0, 4.4, 3.9, 3.5, 3.8, 3.8, 3.4, 3.7, 3.6, 3.3, 3.4, 3.0, 3.4, 3.5,
3.4, 3.2, 3.1, 3.4, 4.1, 4.2, 3.1, 3.2, 3.5, 3.6, 3.0, 3.4, 3.5, 2.3, 3.2, 3.5,
3.8, 3.0, 3.8, 3.2, 3.7, 3.3, 3.2, 3.2, 3.1, 2.3, 2.8, 2.8, 3.3, 2.4, 2.9, 2.7,
2.0, 3.0, 2.2, 2.9, 2.9, 3.1, 3.0, 2.7, 2.2, 2.5, 3.2, 2.8, 2.5, 2.8, 2.9, 3.0,
2.8, 3.0, 2.9, 2.6, 2.4, 2.4, 2.7, 2.7, 3.0, 3.4, 3.1, 2.3, 3.0, 2.5, 2.6, 3.0,
2.6, 2.3, 2.7, 3.0, 2.9, 2.9, 2.5, 2.8, 3.3, 2.7, 3.0, 2.9, 3.0, 3.0, 2.5, 2.9,
2.5, 3.6, 3.2, 2.7, 3.0, 2.5, 2.8, 3.2, 3.0, 3.8, 2.6, 2.2, 3.2, 2.8, 2.8, 2.7,
3.3, 3.2, 2.8, 3.0, 2.8, 3.0, 2.8, 3.8, 2.8, 2.8, 2.6, 3.0, 3.4, 3.1, 3.0, 3.1,
3.1, 3.1, 2.7, 3.2, 3.3, 3.0, 2.5, 3.0, 3.4, 3.0]), ('Petal.Width', [0.2, 0.2,
0.2, 0.2, 0.2, 0.4, 0.3, 0.2, 0.2, 0.1, 0.2, 0.2, 0.1, 0.1, 0.2, 0.4, 0.4, 0.3,
0.3, 0.3, 0.2, 0.4, 0.2, 0.5, 0.2, 0.2, 0.4, 0.2, 0.2, 0.2, 0.2, 0.4, 0.1, 0.2,
0.2, 0.2, 0.2, 0.1, 0.2, 0.2, 0.3, 0.3, 0.2, 0.6, 0.4, 0.3, 0.2, 0.2, 0.2, 0.2,
1.4, 1.5, 1.5, 1.3, 1.5, 1.3, 1.6, 1.0, 1.3, 1.4, 1.0, 1.5, 1.0, 1.4, 1.3, 1.4,
1.5, 1.0, 1.5, 1.1, 1.8, 1.3, 1.5, 1.2, 1.3, 1.4, 1.4, 1.7, 1.5, 1.0, 1.1, 1.0,
1.2, 1.6, 1.5, 1.6, 1.5, 1.3, 1.3, 1.3, 1.2, 1.4, 1.2, 1.0, 1.3, 1.2, 1.3, 1.3,
1.1, 1.3, 2.5, 1.9, 2.1, 1.8, 2.2, 2.1, 1.7, 1.8, 1.8, 2.5, 2.0, 1.9, 2.1, 2.0,
2.4, 2.3, 1.8, 2.2, 2.3, 1.5, 2.3, 2.0, 2.0, 1.8, 2.1, 1.8, 1.8, 1.8, 2.1, 1.6,
1.9, 2.0, 2.2, 1.5, 1.4, 2.3, 2.4, 1.8, 1.8, 2.1, 2.4, 2.3, 1.9, 2.3, 2.5, 2.3,
1.9, 2.0, 2.3, 1.8])]
```

[37]:
```
# groupByKey & mapValues
iris1_mod = iris1_split.flatMap(
    lambda var1: (
        ("Sepal.Length", float(var1[0])),
        ("Sepal.Width", float(var1[1])),
        ("Petal.Length", float(var1[2])),
        ("Petal.Width", float(var1[3])),
    )
)
print(iris1_mod.groupByKey().mapValues(sum).collect())
```

```
[('Sepal.Width', 458.60000000000014), ('Petal.Width', 179.90000000000012),
('Sepal.Length', 876.5000000000002), ('Petal.Length', 563.7000000000004)]
```

[38]:
```
# Calculating Minimum, Maximum and Mean of data
iris1_mod=iris1_split.map(lambda var1:float(var1[0]))

print(iris1_mod.min())
print(iris1_mod.max())
print(iris1_mod.mean())
```

```
4.3
7.9
5.843333333333332
```

[39]:
```
# measuring standard deviation & variance
print(iris1_mod.stdev())
print(iris1_mod.variance())
```

```
0.8253012917851412
0.6811222222222227
```

[40]:
```
# statistical Summary of an RDD
iris1_mod.stats()
```

[40]: (count: 150, mean: 5.843333333333332, stdev: 0.8253012917851412, max: 7.9, min: 4.3)