Spark Cheatsheet

Contents

[Spark Cheatsheet 1](#_Toc459519507)

[DataFrames (pyspark.sql.dataframe.DataFrame) 3](#_Toc459519508)

[Average of a column 3](#_Toc459519509)

[Cache a DataFrame 3](#_Toc459519510)

[Display a DataFrame as a Chart 4](#_Toc459519511)

[Histogram 4](#_Toc459519512)

[Scatter Plot 4](#_Toc459519513)

[Display Data Frame as json 5](#_Toc459519514)

[Display Spark DataFrame Columns and Types 6](#_Toc459519515)

[Display a Spark DataFrame schema 6](#_Toc459519516)

[Execute a Lambda on a DataFrame 6](#_Toc459519517)

[Join on two DataFrames 7](#_Toc459519518)

[max() or min() of a DataFrame column 7](#_Toc459519519)

[Randomly Split a DataFrame 7](#_Toc459519520)

[Read a DataFrame from a CSV file with Schema 8](#_Toc459519521)

[Read a DataFrame from a Text File 9](#_Toc459519522)

[Remove Punctuation from a DataFrame Column 10](#_Toc459519523)

[Rename a Column 10](#_Toc459519524)

[Select columns or column derivatives from a DataFrame 11](#_Toc459519525)

[Summary Statistics for a DataFrame 11](#_Toc459519526)

[MLLib 11](#_Toc459519527)

[DenseVector 11](#_Toc459519528)

[Dot Product (DenseVector) 11](#_Toc459519529)

[LabeledPoint 11](#_Toc459519530)

[VectorAssembler 12](#_Toc459519531)

[Pair RDDs 13](#_Toc459519532)

[Count the occurrences of a word 13](#_Toc459519533)

[Count the number of unique words 13](#_Toc459519534)

[Extract the Keys and Values from an RDD 13](#_Toc459519535)

[Return as a list 13](#_Toc459519536)

[Find the n most common words 13](#_Toc459519537)

[groupByKey() 14](#_Toc459519538)

[Keep only unique values in a Key-Value PairRDD 14](#_Toc459519539)

[Sorting by Keys and Values 15](#_Toc459519540)

[pyspark 15](#_Toc459519541)

[pyspark.sql 15](#_Toc459519542)

[pyspark.sql.DataFrame 15](#_Toc459519543)

[Resilient Distributed Datasets (RDDs) 16](#_Toc459519544)

[Create an RDD from the items in a list (using flatMap) 16](#_Toc459519545)

[Extract the first n data points from an RDD 16](#_Toc459519546)

[Filter an RDD 16](#_Toc459519547)

[Get Distinct Elements in an RDD 16](#_Toc459519548)

[Number of elements in the RDD 16](#_Toc459519549)

[Read a file into an RDD 17](#_Toc459519550)

[SQL Tables 17](#_Toc459519551)

[Create a DataFrame from a SQL Table 17](#_Toc459519552)

[Register a DataFrame as a SQL Table 17](#_Toc459519553)

# DataFrames (pyspark.sql.dataframe.DataFrame)

## Average of a column

print type(wordCountsDF)

wordCountsDF.show()

<class 'pyspark.sql.dataframe.DataFrame'>

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

| word|count|

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

| cat| 2|

| rat| 2|

|elephant| 1|

averageCount = (wordCountsDF.groupBy()

.mean('count')

.first()[0])

print averageCount

1.66666666667

## Cache a DataFrame

seed = 1800009193L

weights = [0.2, 0.8]

(split20DF, split80DF) = datasetDF.randomSplit(weights, seed)

# Let's cache these datasets for performance

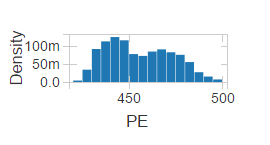
testSetDF = split20DF.cache()

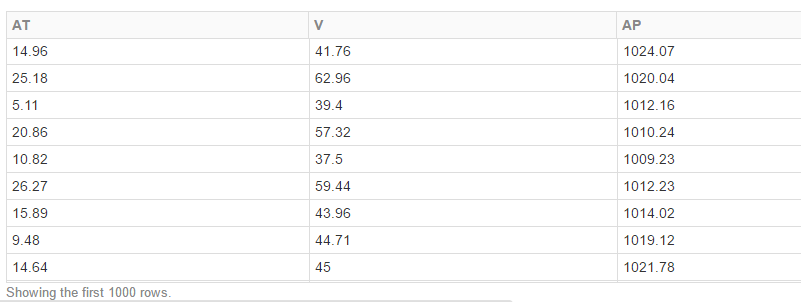
trainingSetDF = split80DF.cache()

## Display a DataFrame as a Chart

### Histogram

display(powerPlantDF)

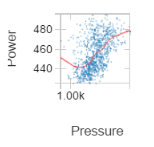




and other chart types, as well…

### Scatter Plot

display(sqlContext.sql("SELECT PE as Power, AP as Pressure FROM power\_plant"))



## Display Data Frame as json

import json

def makeJson(someDf):

aDict = json.loads(someDf)

return json.dumps(aDict, indent=2)

def printResultsAsJson(someDf):

coll = someDf.toJSON().map(makeJson).collect()

for element in coll:

print element

queryResult = sqlContext.sql('''

SELECT smfPayloadData.events.DeliveryLocationArrival

FROM mapped

where smfPayloadData.events.DeliveryLocationArrival is not null

limit 10

''')

# queryResult.show()

printResultsAsJson( queryResult)

## Display Spark DataFrame Columns and Types

print powerPlantDF.dtypes

output:

[('AT', 'double'), ('V', 'double'), ('AP', 'double'), ('RH', 'double'), ('PE', 'double')]

## Display a Spark DataFrame schema

# Print the schema in a tree format

df.printSchema()

## root

## |-- age: long (nullable = true)

## |-- name: string (nullable = true)

## Execute a Lambda on a DataFrame

Suppose you have a DataFrame consisting of a first name and a last name, and you want to add a unique [SHA-256](https://en.wikipedia.org/wiki/Secure_Hash_Algorithm) hash to each row.

df = sqlContext.createDataFrame([("John", "Smith"), ("Ravi", "Singh"), ("Julia", "Jones")], ("first\_name", "last\_name"))

Here's a simple function to calculate such a hash, using Python's built-in hashlib library:

def make\_hash(first\_name, last\_name):

import hashlib

m = hashlib.sha256()

# Join the first name and last name by a blank and hash the resulting

# string.

full\_name = ' '.join((first\_name, last\_name))

m.update(full\_name)

return m.hexdigest()

Okay, that's great. But, how do we use it on our DataFrame? We can use a UDF:

from pyspark.sql.functions import udf

u\_make\_hash = udf(make\_hash)

df2 = df.select(df['\*'], u\_make\_hash(df['first\_name'], df['last\_name']))

# could run df2.show() here to prove it works

Or we can step down to an RDD, use a lambda to call make\_hash and have the lambda return a Row object, which Spark can use to ["infer" a new DataFrame](http://spark.apache.org/docs/latest/sql-programming-guide.html#inferring-the-schema-using-reflection).

from pyspark.sql import Row

def make\_hash\_from\_row(row):

hash = make\_hash(row[0], row[1])

return Row(first\_name=row[0], last\_name=row[1], hash=hash)

df2 = (df.rdd

.map(lambda row: make\_hash\_from\_row(row))

.toDF())

These methods are roughly equivalent. You'll need to do something similar to convert your raw\_data\_df DataFrame into a new DataFrame of LabeledPoint objects.

## Grouping Data in a DataFrame (using Aggregations)

### count, average

from pyspark.sql import functions as F

# From ratingsDF, create a movie\_ids\_with\_avg\_ratings\_df that combines the two DataFrames

movie\_ids\_with\_avg\_ratings\_df = ratings\_df.groupBy('movieId').agg(F.count(ratings\_df.rating).alias("count"),

F.avg(ratings\_df.rating).alias("average"))

ratings\_df.show(3, truncate=False)

print 'movie\_ids\_with\_avg\_ratings\_df:'

movie\_ids\_with\_avg\_ratings\_df.show(3, truncate=False)

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

|userId|movieId|rating|

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

|1 |2 |3.5 |

|1 |29 |3.5 |

|1 |32 |3.5 |

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

only showing top 3 rows

movie\_ids\_with\_avg\_ratings\_df:

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

|movieId|count|average |

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

|1831 |7463 |2.5785207021305103|

|431 |8946 |3.695059244355019 |

|631 |2193 |2.7273141814865483|

## Join on two DataFrames

results\_df = sqlContext.sql('''

SELECT mapped.smfPayloadData.paired.shipmentId as shipment\_id,

sap\_idoc.smfPayloadData.SAPShipment.SAPShippingPoint as origin

FROM mapped

INNER JOIN sap\_idoc ON sap\_idoc.smfPayloadData.SAPShipment.SAPShipmentID =

mapped.smfPayloadData.paired.shipmentId

limit 10

''')

print type(results\_df)

results\_df.printSchema()

output:

<class 'pyspark.sql.dataframe.DataFrame'>

root

|-- shipment\_id: string (nullable = true)

|-- origin: string (nullable = true)

movie\_names\_df = movie\_ids\_with\_avg\_ratings\_df.join(movies\_df,

movie\_ids\_with\_avg\_ratings\_df.movieId == movies\_df.ID,

'inner')

movie\_names\_with\_avg\_ratings\_df = movie\_names\_df.select('average', 'title', 'count', 'movieId')

print 'movie\_names\_with\_avg\_ratings\_df:'

movie\_names\_with\_avg\_ratings\_df.show(3, truncate=False)

Output:

movie\_ids\_with\_avg\_ratings\_df:

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

|movieId|count|average |

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

|1831 |7463 |2.5785207021305103|

|431 |8946 |3.695059244355019 |

|631 |2193 |2.7273141814865483|

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

only showing top 3 rows

movie\_names\_with\_avg\_ratings\_df:

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

|average |title |count|movieId|

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

|2.5785207021305103|Lost in Space (1998) |7463 |1831 |

|3.695059244355019 |Carlito's Way (1993) |8946 |431 |

|2.7273141814865483|All Dogs Go to Heaven 2 (1996)|2193 |631 |

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

## max() or min() of a DataFrame column

print parsed\_points\_df.take(2)

content\_stats = (parsed\_points\_df

.selectExpr("min(label)", "max(label)"))

content\_stats.first()

min\_year = content\_stats.first()[0]

max\_year = content\_stats.first()[1]

print min\_year, max\_year

Output:

[Row(features=DenseVector([0.8841, 0.6105, 0.6005, 0.4747, 0.2472, 0.3573, 0.3441, 0.3396, 0.6009, 0.4257, 0.6049, 0.4192]), label=2001.0), Row(features=DenseVector([0.8544, 0.6041, 0.5936, 0.4959, 0.2663, 0.2615, 0.5064, 0.4645, 0.6658, 0.543, 0.5804, 0.4452]), label=2001.0)] 1922.0 2011.0

## Randomly Split a DataFrame

weights = [.8, .1, .1]

seed = 42

parsed\_train\_data\_df, parsed\_val\_data\_df, parsed\_test\_data\_df = parsed\_data\_df.randomSplit(weights, seed)

print n\_train, n\_val, n\_test, n\_train + n\_val + n\_test

print parsed\_data\_df.count()

Output:

5382 672 670 6724

6724

## Read a DataFrame from a CSV file with Schema

from pyspark.sql.types import \*

# Custom Schema for Power Plant

customSchema = StructType([StructField('AT', DoubleType(), True),

StructField('V', DoubleType(), True),

StructField('AP', DoubleType(), True),

StructField('RH', DoubleType(), True),

StructField('PE', DoubleType(), True)])

altPowerPlantDF = sqlContext.read.format('com.databricks.spark.csv').options(header='true',

inferschema='false',

delimiter='\t').load("/databricks-datasets/power-plant/data", schema = customSchema)

## Read a DataFrame from a Text File

fileName = "dbfs:/databricks-datasets/cs100/lab1/data-001/shakespeare.txt"

shakespeareDF = sqlContext.read.text(fileName)

shakespeareDF.show(15)

shakespeareDF = sqlContext.read.text(fileName).select(removePunctuation(col('value')))

shakespeareDF.show(15, truncate=False)

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

| value|

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

| 1609|

| |

| THE SONNETS|

| |

|by William Shakes...|

| |

| |

| |

| ...|

| From fairest cr...|

| That thereby be...|

| But as the ripe...|

| His tender heir...|

| But thou contra...|

| Feed'st thy lig...|

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

## Remove Punctuation from a DataFrame Column

from pyspark.sql.functions import regexp\_replace, trim, col, lower

def removePunctuation(column):

"""Removes punctuation, changes to lower case, and strips leading and trailing spaces.

Note:

Only spaces, letters, and numbers should be retained. Other characters should should be

eliminated (e.g. it's becomes its). Leading and trailing spaces should be removed after

punctuation is removed.

Args:

column (Column): A Column containing a sentence.

Returns:

Column: A Column named 'sentence' with clean-up operations applied.

"""

removed\_column = trim(lower(regexp\_replace(column, "\p{Punct}", ''))).alias('sentence')

return removed\_column

# end removePunctuation

sentenceDF = sqlContext.createDataFrame([('Hi, you!',),

(' No under\_score!',),

(' \* Remove punctuation then spaces \* ',)], ['sentence'])

sentenceDF.show(truncate=False)

testPunctDF = sqlContext.createDataFrame([(" The Elephant's 4 cats. ",)])

testPunctDF.show()

testOutput = testPunctDF.select(removePunctuation(col('\_1')))

Test.assertEquals(testPunctDF.select(removePunctuation(col('\_1'))).first()[0],

'the elephants 4 cats',

'incorrect definition for removePunctuation function')

print type(testOutput)

testOutput.show()

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

| \_1|

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

| The Elephant's 4...|

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

1 test passed.

<class 'pyspark.sql.dataframe.DataFrame'>

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

| sentence|

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

|the elephants 4 cats|

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

## Rename a Column

parsed\_data\_df = (parsed\_points\_df.select(parsed\_points\_df.features,

parsed\_points\_df.label - min\_year)

.withColumnRenamed('(label - 1922.0)', 'label')

)

print '\n{0}'.format(parsed\_data\_df.first())

Output:

Row(features=DenseVector([0.8841, 0.6105, 0.6005, 0.4747, 0.2472, 0.3573, 0.3441, 0.3396, 0.6009, 0.4257, 0.6049, 0.4192]), label=79.0)

## Select columns or column derivatives from a DataFrame

parsed\_data\_df = parsed\_points\_df.select(parsed\_points\_df.features,

parsed\_points\_df.label - min\_year)

print '\n{0}'.format(parsed\_data\_df.first())

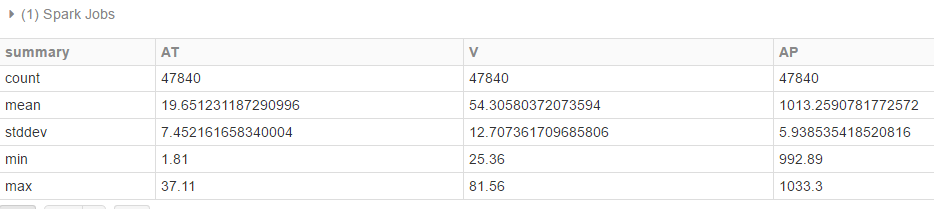
Output:

Row(features=DenseVector([0.8841, 0.6105, 0.6005, 0.4747, 0.2472, 0.3573, 0.3441, 0.3396, 0.6009, 0.4257, 0.6049, 0.4192]), (label - 1922.0)=79.0)

## Summary Statistics for a DataFrame

df = sqlContext.table("power\_plant")

display(df.describe())



# MLLib

## DenseVector

from pyspark.mllib.linalg import DenseVector

## Dot Product (DenseVector)

dv = DenseVector([1,2,3])

dv2 = DenseVector([4,5,6])

dv.dot(dv2)

Output: 32.0

## LabeledPoint

from pyspark.mllib.regression import LabeledPoint

lp = LabeledPoint(0.0, [1,2,3])

print lp.label

print lp.features

output:

0.0

[1.0,2.0,3.0]

## VectorAssembler

from pyspark.mllib.linalg import Vectors

from pyspark.ml.feature import VectorAssembler

test\_dataset = sqlContext.createDataFrame(

[(0, 18, 1.0, Vectors.dense([0.0, 10.0, 0.5]), 1.0)],

["id", "hour", "mobile", "userFeatures", "clicked"])

assembler = VectorAssembler(

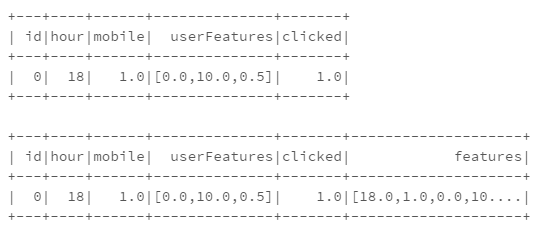
inputCols=["hour", "mobile", "userFeatures"],

outputCol="features")

output = assembler.transform(test\_dataset)

test\_dataset.show()

output.show()



# Pair RDDs

## Count the occurrences of a word

In:

wordsList = ['cat', 'elephant', 'rat', 'rat', 'cat']

wordsRDD = sc.parallelize(wordsList, 4)

Out:

['cat', 'elephant', 'rat', 'rat', 'cat']

In:

wordPairs = wordsRDD.map(lambda x : (x,1))

Out:

[('cat', 1), ('elephant', 1), ('rat', 1), ('rat', 1), ('cat', 1)]

In:

wordCounts = wordPairs.reduceByKey(lambda a,b : a+b)

Out:

[('rat', 2), ('elephant', 1), ('cat', 2)]

## Count the number of unique words

In:

uniqueWords = len(wordCounts.collect())

print uniqueWords

Out:

3

## Extract the Keys and Values from an RDD

keys()

values()

### Return as a list

daysWithHosts = dailyHosts.keys().collect()

## Find the n most common words

endpointCountPairTuple = not200.map(lambda log: (log.endpoint, 1))

endpointSum = endpointCountPairTuple.reduceByKey(lambda a,b : a + b)

# these endpointSum tuples are like

# [(u'/images/NASA-logosmall.gif', 8761), (u'/images/KSC-logosmall.gif', 7236),…]

print endpointSum.takeOrdered(10, lambda s: -1 \* s[1])

## groupByKey()

This outputs a ResultIterable object , which can be cast to a list using a .mapValues() function

dayAndHostTuple = access\_logs.map(lambda log: (log.date\_time.day, log.host))

print dayAndHostTuple.take(3)

groupedByDay = dayAndHostTuple.groupByKey()

print groupedByDay.take(3)

output

[(1, u'in24.inetnebr.com'), (1, u'uplherc.upl.com'), (1, u'uplherc.upl.com')]

[(8, <pyspark.resultiterable.ResultIterable object at 0xb0904bec>), (12, <pyspark.resultiterable.ResultIterable object at 0xb0904a6c>), (4, <pyspark.resultiterable.ResultIterable object at 0xb0902a2c>)]

If you want the value of the PairRDD to be a list, use:

groupedByDay = (dayAndHostTuple.groupByKey()

.mapValues(lambda x: list(x)))

output:

[(8, [u'pc7567.dialup.rwth-aachen.de', u'mtrsppp46.epix.net',…]), (9, […]), …]

## Keep only unique values in a Key-Value PairRDD

Using list(set(x)) converts the iterable result of groupByKey() to a list containing only the distinct values.

dayGroupedHosts = (dayToHostPairTuple.groupByKey()

.mapValues(lambda x: list(set(x))))

## Sorting by Keys and Values

dayHostCount = dayGroupedHosts.mapValues(lambda host\_list: len(host\_list))

# This looks like [(8, 60142), (12, 38070), (4, 59554), (16, 56651), (20, 32963)]

**dailyHosts = dayHostCount.sortByKey()**

print dailyHosts.take(3)

output: [(1, 33996), (3, 41387), (4, 59554)]

Sort by keys (ascending):

RDD.takeOrdered(5, key = lambda x: x[0])

Sort by keys (descending):

RDD.takeOrdered(5, key = lambda x: -x[0])

Sort by values (ascending):

RDD.takeOrdered(5, key = lambda x: x[1])

Sort by values (descending):

RDD.takeOrdered(5, key = lambda x: -x[1])

# pyspark

## pyspark.sql

<https://spark.apache.org/docs/1.6.1/api/python/pyspark.sql.html>

### pyspark.sql.DataFrame

<https://spark.apache.org/docs/1.6.1/api/python/pyspark.sql.html#pyspark.sql.DataFrame>

#### Filter a pyspark.sql.DataFrame

<https://spark.apache.org/docs/1.6.1/api/python/pyspark.sql.html#pyspark.sql.DataFrame>

shipment\_df = mapped\_recent.filter("smfpayloadData.paired.shipmentId = '0305775487'")

print shipment\_df.count()

output: 1

# Resilient Distributed Datasets (RDDs)

## Create an RDD from the items in a list (using flatMap)

shakespeareWordsRDD = (shakespeareRDD.**flatMap(lambda line: line.split(" "))**

.map(lambda word: (word, 1))

.reduceByKey(lambda a,b: a+b)

)

shakespeareWordCount = shakespeareWordsRDD.count()

print shakespeareWordsRDD.top(5)

Output:

[(u'zwaggerd', 1), (u'zounds', 24), (u'zone', 1), (u'zodiacs', 1), (u'zodiac', 1)]

## Extract the first n data points from an RDD

samplePoints = rawData.take(n)

## Filter an RDD

The lambda filters in the rows which evaluate to true.

badRecords = (access\_logs.filter(lambda row: row.response\_code == 404))

print 'Found %d 404 URLs' % badRecords.count()

output: Found 6185 404 URLs

## Get Distinct Elements in an RDD

Returns an RDD containing the unique subset

badUniqueEndpoints = badEndpoints.distinct()

## Number of elements in the RDD

numPoints = rawData.count()

print numPoints

## Read a file into an RDD

from test\_helper import Test

import os.path

baseDir = os.path.join('data')

inputPath = os.path.join('cs190', 'millionsong.txt')

fileName = os.path.join(baseDir, inputPath)

numPartitions = 2

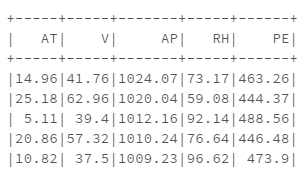
rawData = sc.textFile(fileName, numPartitions)

# SQL Tables

## Create a DataFrame from a SQL Table

datasetDF = sqlContext.sql("SELECT \* from power\_plant")

datasetDF.show()



## Register a DataFrame as a SQL Table

sqlContext.sql("DROP TABLE IF EXISTS power\_plant")

dbutils.fs.rm("dbfs:/user/hive/warehouse/power\_plant", True)

sqlContext.registerDataFrameAsTable(powerPlantDF, "power\_plant")