CMU MSP 36602: Spark, Part 3

H. Seltman, March 27, 2019

I) pyspark DataFrames

i) These are implemented in **module** pyspark.sql (http://spark.apache.org/docs/2.1.0/api/python/pyspark.sql.html).

ii) Spark has its own **DataFrame** objects separate from Panda. These are like RDDs but are organized into named columns. See details and examples below.

iii) Creating a DataFrame as rows (less used)

(1) DataFrames are made of Rows. After running `from pyspark.sql import Row`, a **Row** can be directly created with `myRow = Row(myPi=3.15, myE=2.72)`. Elements of a row can be accessed with `myRow['myPi']` or `myRow.myPi`. The `list()` function will convert a Row to a list, dropping the column names.

A Row object with names only can be used as a template to create additional Row objects: myTemplate = Row('id', 'gender', 'age') followed by myRow = myTemplate('AB2', 'F', 22).
See below for details and examples.

(2) If `myDf` is a DataFrame, then `myDf.myCol` or `myDf['myCol']` is a **Column** object. See details and example below.

iv) A DataFrame has a "schema" which is like str() in R

```
# Setup to define a schema
from pyspark.sql.types import StructField, StructType, \
                              IntegerType, FloatType, StringType

# Define a data base schema (third argument is "allow null?")
pschema = StructType([StructField('id', StringType(), False),
                      StructField('store', IntegerType(), True),
                      StructField('purchase', FloatType(), True),
                      StructField('debit', IntegerType(), True)])

# Read from csv using a schema (good, but optional)
p2 = sqlContext.read.csv("purchase.dat", pschema, sep=" ",
                         header=True)
type(p2)
# <class 'pyspark.sql.dataframe.DataFrame'>
p2.show(5)
+---------+-----+--------+-----+
|       id|store|purchase|debit|
+---------+-----+--------+-----+
|623910455|   12|   25.53|    1|
|967941424|    7|   58.11|    1|
|517476391|   15|   62.31|    0|
|425127415|   14|   15.03|    1|
|279814400|   11|    91.4|    1|
+---------+-----+--------+-----+
only showing top 5 rows
```

```
# Converting from an RDD to a DataFrame
jnk = sc.parallelize([[23, 'C', 3.2], [25, 'C', 3.8],
                      [28, 'A', 6.9], [33, 'A', 7.5]])
df = jnk.toDF(['id', 'Control or Active', 'value'])

df.show()
+---+-----------------+-----+
| id|Control or Active|value|
+---+-----------------+-----+
| 23|                C|  3.2|
| 25|                C|  3.8|
| 28|                A|  6.9|
| 33|                A|  7.5|
+---+-----------------+-----+

df.printSchema()
root
 |-- id: long (nullable = true)
 |-- Control or Active: string (nullable = true)
 |-- value: double (nullable = true)


# Explore basic DataFrame methods
dir(p2)
p2.columns  # is a list of column names
p2.dtypes   # is a list of tuples as (name, type)
p2.count()
p2.first()
p2.show(4) # neater than .take(4), but with no return value

# Data description
type(p2.describe())  # pyspark.sql.dataframe.DataFrame
p2.describe().show()
p2.describe(['store','debit']).show()

# Data subsets and new columns (usually you assign the result)
p2.head(5) # returns a list of Row objects (not a DataFrame)
p2.select('id', 'store').show(5)
p2.select('id', 'purchase', 'debit',
  (p2.purchase * p2.debit).alias('debit_purchase')).show(5)
p2.filter((p2.purchase>100) & (p2.store==15)).show(5)
p2.drop('debit', 'store').show(5)  # drops a column or columns
p2.dropDuplicates()  # drop full rows
p2.dropna()  # drop full rows
p2.fillna(0.0, subset='purchase')  # (not a "Data subset")
# There is no "easy" way to pull out a column as a list
store_col = [r.store for r in p2.collect()]
store_col = [r.store for r in p2.select('store').collect()]
# Pull out a portion of a column
[p['purchase'] for p in p2.filter(p2.purchase>119). \
   select('purchase').collect()]

# Basic statistics
dir(pyspark.sql.functions)
from pyspark.sql.functions import mean, min, max, stddev
p2.select([mean('purchase'), stddev('purchase'),
            min('purchase'), max('purchase')]).show()
```

```
# A few built-in functions
p2.corr('store', 'debit')
p2.cov('store', 'debit')
p2.freqItems(['store'], 0.20).collect() # values seen >20% of the time
p2.approxQuantile('purchase', [0.25,0.5,0.75],
  relativeError=0.01)
p2.crosstab('store', 'debit').show()
p2.crosstab('store', 'debit').orderBy('store_debit').show()

# Grouping
p2.drop('id').groupBy('debit').avg().show()
temp = p2.drop('id','store').groupBy('debit').avg()
[ap['avg(purchase)'] for ap in temp.collect()][1]

# Using an empty groupBy()
p2.select(['store','purchase']).groupBy().sum().show()

# Joining
c = sc.textFile("cust.dat").map(str).filter(lambda s: s[:2]!='id').\
    map(lambda s: s.split()).\
    map(lambda x: (int(x[0]), 'F' if x[1]=='1' else 'M', int(x[2])))
cdf = c.toDF(['id', 'M0F1', 'origin'])
p2.join(cdf, on='id').show()

# Back to rdd
p2.filter(p2.purchase>119).rdd.collect()
```

II) Spark SQL

```
# Analyze a small built-in data set using SQL
f = "/usr/local/spark/examples/src/main/resources/people.json"
df = spark.read.json(f)
df.show()

# Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")
sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
sqlDF = spark.sql("SELECT *, age*age AS ageSqr FROM people WHERE " +
                  "age IS NOT NULL")
sqlDF.show()
```

III) Spark machine learning

a) Use newer pyspark.ml (using DataFrame), not old pyspark.mllib (using RDD)

b) **Regression Example**:

```
from pyspark.ml.linalg import DenseVector

p.take(5)
PURCHASE = 2; STORE = 1; DEBIT = 3
forReg = p.map(lambda v: (v[PURCHASE],
                          DenseVector((v[STORE], v[DEBIT]))))
forReg.take(5)
forReg = forReg.toDF(['label', 'features'])
forReg.first()

from pyspark.ml.regression import LinearRegression
lr = LinearRegression()
lrModel = lr.fit(forReg)
type(lrModel)
# <class 'pyspark.ml.regression.LinearRegressionModel'>
dir(lrModel)

# Print the coefficients and intercept for linear regression
print("Coefficients:", ",
  ".join([str(round(float(x),2)) for x in lrModel.coefficients]))
print("Intercept: {}".format(round(lrModel.intercept, 2)))
```

c) Figure out a new model family: **regularized logistic regression**

a. Data in regLR.dat

b. Beginning code:

```
from pyspark.ml.classification import LogisticRegression
help(LogisticRegression)

# Load data
train = sc.textFile("regLR.dat").map(lambda s: s.split()).\
    map(lambda x: (int(x[0]),
                   DenseVector([float(z) for z in x[1:]]))).\
    toDF(['label', 'features'])


lr = LogisticRegression(regParam=0.3)

# Fit the model
model = lr.fit(train)

# Print the coefficients and intercept for generalized linear
regression model
print("Coefficients: " + str(model.coefficients))
print("Intercept: " + str(model.intercept))
dir(model)
```

d) Figure out a new model family: **Accelerated failure model**

a. Search and find:
http://spark.apache.org/docs/2.1.0/api/python/pyspark.ml.html#pyspark.ml.regression.AFTSurvivalRegression

b. Data simulated with b_0=0, b_M=-1, b_{Age}=-0.5, shape=2: cdata.csv

c. Beginning code:

```
rdd = sc.textFile("cdata.csv")
rdd.take(2)
MALE = 0; AGE = 1; TIME = 2; OBS = 3  # after dropping
'int'
rdd = rdd.filter(lambda s: s[:3] != 'int').map(lambda s:
s.split(',')).\
    map(lambda v: (int(v[1+MALE]), float(v[1+AGE]),
                   float(v[1+TIME]), float(v[1+OBS])))
rdd.take(5)
weib = rdd.map(lambda v: (v[TIME],
                          DenseVector((v[MALE], v[AGE])),
                                      v[OBS])).\
       toDF(['label', 'features', 'censor'])
weib.take(5)

from pyspark.ml.regression import AFTSurvivalRegression
aftsr = AFTSurvivalRegression()
model = aftsr.fit(weib)
dir(model)
```