Spark and Data Analytics

Ching-Yung Lin, Ph.D.
Adjunct Professor, Dept. of Electrical Engineering and Computer Science
Mgr., Dept. of Network Science and Big Data Analytics, IBM Watson Research Center
Updated Information on TAs

- TAs:
  - Ghazal Fazelnia <gf2293>, EE; Tuesdays 9:30 - 11:00 am, EE Student Lounge, Mudd 13th floor
  - Lev Givon <leg22>, EE; Wednesdays 6:00 - 7:30 pm, 8LE5 CEPSR Building
Spark Stack

Spark SQL structured data
Spark Streaming real-time
MLib machine learning
GraphX graph processing

Spark Core

Standalone Scheduler
YARN
Mesos
Spark Core

Basic functionality of Spark, including components for:

- Task Scheduling
- Memory Management
- Fault Recovery
- Interacting with Storage Systems
- and more

Home to the API that defines resilient distributed datasets (RDDs) - Spark’s main programming abstraction.

RDD represents a collection of items distributed across many compute nodes that can be manipulated in parallel.
Download Spark

http://spark.apache.org/downloads.html

Download Spark
The latest release of Spark is Spark 1.2.0, released on December 18, 2014 (release notes) (git tag)

1. Chose a Spark release: 1.2.0 (Dec 18 2014)
2. Chose a package type: Pre-built for Hadoop 2.4 and later
3. Chose a download type: Direct Download
4. Download Spark: spark-1.2.0-bin-hadoop2.4.tgz
5. Verify this release using the 1.2.0 signatures and checksums.

Note: Scala 2.11 users should download the Spark source package and build with Scala 2.11 support.

Link with Spark
Spark artifacts are hosted in Maven Central. You can add a Maven dependency with the following coordinates:

<table>
<thead>
<tr>
<th>groupId</th>
<th>artifactId</th>
<th>version</th>
</tr>
</thead>
<tbody>
<tr>
<td>org.apache.spark</td>
<td>spark-core_2.10</td>
<td>1.2.0</td>
</tr>
</tbody>
</table>
First language to use — Python

Python is a programming language that lets you work quickly and integrate systems more effectively. 

```
# Python 3: List comprehensions to n
>>> fruits = ['Banana', 'Apple', 'Lime']
>>> loudfruits = [fruit.upper() for fruit in fruits] while a < n:
    >>> print(loudfruits)
    ['BANANA', 'APPLE', 'LIME']
>>> print()
# list and the enumerate function
0: 1123581321345589144233377610
```

| 1 | 2 | 3 | 4 | 5 |

**Compound Data Types**

Lists (known as arrays in other languages) are one of the compound data types that Python understands. Lists can be indexed, sliced and manipulated with other built-in about functions. More about lists in Python 3.
Spark’s Python Shell (PySpark Shell)

bin/pyspark
Test installation

*Example 2-1. Python line count*

```python
>>> lines = sc.textFile("README.md") # Create an RDD called lines

>>> lines.count() # Count the number of items in this RDD
127

>>> lines.first() # First item in this RDD, i.e. first line of README.md
u'# Apache Spark'
```
Disable logging

You may find the logging statements that get printed in the shell distracting. You can control the verbosity of the logging. To do this, you can create a file in the `conf` directory called `log4j.properties`. The Spark developers already include a template for this file called `log4j.properties.template`. To make the logging less verbose, make a copy of `conf/log4j.properties.template` called `conf/log4j.properties` and find the following line:

```
log4j.rootCategory=INFO, console
```

Then lower the log level so that we show only the WARN messages, and above by changing it to the following:

```
log4j.rootCategory=WARN, console
```
Core Spark Concepts

- At a high level, every Spark application consists of a **driver program** that launches various parallel operations on a cluster.

- The driver program contains your application’s main function and defines distributed databases on the cluster, then applies operations to them.

- In the preceding example, the driver program was the Spark shell itself.

- Driver programs access Spark through a SparkContext object, which represents a connection to a computing cluster.

- In the shell, a SparkContext is automatically created as the variable called sc.
Driver Programs

Driver programs typically manage a number of nodes called **executors**.

If we run the `count()` operation on a cluster, different machines might count lines in different ranges of the file.
Example filtering

```python
>>> lines = sc.textFile("README.md")

>>> pythonLines = lines.filter(lambda line: "Python" in line)

>>> pythonLines.first()
'## Interactive Python Shell'
```

lambda —> define functions inline in Python.

```python
def hasPython(line):
    return "Python" in line

pythonLines = lines.filter(hasPython)
```
Running as a Standalone Application

In Python, you simply write applications as Python scripts, but you must run them using the `bin/spark-submit` script included in Spark. The `spark-submit` script includes the Spark dependencies for us in Python. This script sets up the environment for Spark’s Python API to function. Simply run your script with the line given in Example 2-6.

*Example 2-6. Running a Python script*

```
bin/spark-submit my_script.py
```
Example — word count

import sys
from operator import add

from pyspark import SparkContext

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print >> sys.stderr, "Usage: wordcount <file>"
        exit(-1)
    sc = SparkContext(appName="PythonWordCount")
    lines = sc.textFile(sys.argv[1], 1)
    counts = lines.flatMap(lambda x: x.split(' '))
        .map(lambda x: (x, 1))
        .reduceByKey(add)
    output = counts.collect()
    for (word, count) in output:
        print "%s: %i" % (word, count)

    sc.stop()
Resilient Distributed Dataset (RDD) Basics

• An RDD in Spark is an immutable distributed collection of objects.

• Each RDD is split into multiple partitions, which may be computed on different nodes of the cluster.

• Users create RDDs in two ways: by loading an external dataset, or by distributing a collection of objects in their driver program.

• Once created, RDDs offer two types of operations: transformations and actions.

```python
>>> lines = sc.textFile("README.md")  # create RDD

>>> pythonLines = lines.filter(lambda line: "Python" in line)  # transformation

>>> pythonLines.first()  # action
u'## Interactive Python Shell'
```

Transformations and actions are different because of the way Spark computes RDDs. Only computes when something is, the first time, in an action.
Persistance in Spark

• By default, RDDs are computed each time you run an action on them.
• If you like to reuse an RDD in multiple actions, you can ask Spark to persist it using RDD.persist().
• RDD.persist() will then store the RDD contents in memory and reuse them in future actions.
• Persisting RDDs on disk instead of memory is also possible.
• The behavior of not persisting by default seems to be unusual, but it makes sense for big data.

Example 3-4. Persisting an RDD in memory

```python
>>> pythonLines.persist

>>> pythonLines.count()
2

>>> pythonLines.first()
u'## Interactive Python Shell'
```

To summarize, every Spark program and shell session will work as follows:

1. Create some input RDDs from external data.
2. Transform them to define new RDDs using transformations like filter().
3. Ask Spark to persist() any intermediate RDDs that will need to be reused.
4. Launch actions such as count() and first() to kick off a parallel computation, which is then optimized and executed by Spark.
Parallelizing in Spark

The simplest way to create RDDs is to take an existing collection in your program and pass it to SparkContext’s parallelize() method. But, that needs all dataset in memory on one machine.

*Example 3-5. parallelize() method in Python*

```python
lines = sc.parallelize(["pandas", "i like pandas"])
```
## Spark file loading

### Table 5-1. Common supported file formats

<table>
<thead>
<tr>
<th>Format name</th>
<th>Structured</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Text files</td>
<td>No</td>
<td>Plain old text files. Records are assumed to be one per line.</td>
</tr>
<tr>
<td>JSON</td>
<td>Semi</td>
<td>Common text-based format, semistructured; most libraries require one record per line.</td>
</tr>
<tr>
<td>CSV</td>
<td>Yes</td>
<td>Very common text-based format, often used with spreadsheet applications.</td>
</tr>
<tr>
<td>SequenceFiles</td>
<td>Yes</td>
<td>A common Hadoop file format used for key/value data.</td>
</tr>
<tr>
<td>Protocol buffers</td>
<td>Yes</td>
<td>A fast, space-efficient multilanguage format.</td>
</tr>
<tr>
<td>Object files</td>
<td>Yes</td>
<td>Useful for saving data from a Spark job to be consumed by shared code. Breaks if you change your classes, as it relies on Java Serialization.</td>
</tr>
</tbody>
</table>
Loading and Saving

Example 5-1. Loading a text file in Python

```python
input = sc.textFile("file:///home/holden/repos/spark/README.md")
```

Example 5-5. Saving as a text file in Python

```python
result.saveAsTextFile(outputFile)
```

Example 5-6. Loading unstructured JSON in Python

```python
import json
data = input.map(lambda x: json.loads(x))
```

Example 5-9. Saving JSON in Python

```python
(data.filter(lambda x: x['lovesPandas']).map(lambda x: json.dumps(x))
  .saveAsTextFile(outputFile))
```
Loading and Saving (II)

Example 5-12. Loading CSV with textFile() in Python

```python
import csv
import StringIO
...
def loadRecord(line):
    '''Parse a CSV line'''
    input = StringIO.StringIO(line)
    reader = csv.DictReader(input, fieldnames=['name', 'favouriteAnimal'])
    return reader.next()
input = sc.textFile(inputFile).map(loadRecord)
```

Example 5-18. Writing CSV in Python

```python
def writeRecords(records):
    '''Write out CSV lines'''
    output = StringIO.StringIO()
    writer = csv.DictWriter(output, fieldnames=['name', 'favoriteAnimal'])
    for record in records:
        writer.writerow(record)
    return [output.getvalue()]
pandaLovers.mapPartitions(writeRecords).saveAsTextFile(outputFile)
```
Example 5-20. Loading a SequenceFile in Python

```scala
val data = sc.sequenceFile(inFile,
```
### Table 5-3. Compression options

<table>
<thead>
<tr>
<th>Format</th>
<th>Splittable</th>
<th>Average compression speed</th>
<th>Effectiveness on text</th>
<th>Hadoop compression codec</th>
<th>Pure Java</th>
<th>Native</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>N</td>
<td>Fast</td>
<td>High</td>
<td>org.apache.hadoop.io.compress.GzipCodec</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>lzo</td>
<td>Y(^6)</td>
<td>Very fast</td>
<td>Medium</td>
<td>com.hadoop.compression.lzo.LzoCodec</td>
<td>Y</td>
<td>Y</td>
<td>LZ0 requires installation on every worker node</td>
</tr>
<tr>
<td>bzip2</td>
<td>Y</td>
<td>Slow</td>
<td>Very high</td>
<td>org.apache.hadoop.io.compress.BZip2Codec</td>
<td>Y</td>
<td>Y</td>
<td>Uses pure Java for splittable version</td>
</tr>
<tr>
<td>zlib</td>
<td>N</td>
<td>Slow</td>
<td>Medium</td>
<td>org.apache.hadoop.io.compress.DefaultCodec</td>
<td>Y</td>
<td>Y</td>
<td>Default compression codec for Hadoop</td>
</tr>
<tr>
<td>Snappy</td>
<td>N</td>
<td>Very Fast</td>
<td>Low</td>
<td>org.apache.hadoop.io.compress.SnappyCodec</td>
<td>N</td>
<td>Y</td>
<td>There is a pure Java port of Snappy but it is not yet available in Spark/Hadoop</td>
</tr>
</tbody>
</table>
Example of Spark Programming

Example 6-1. Sample call log entry in JSON, with some fields removed

```json
{"address":"address here", "band":"40m","callsign":"KK6JLK","city":"SUNNYVALE", 
"contactlat":"37.384733","contactlong":"-122.032164", 
"county":"Santa Clara","dxcc":"291","fullname":"MATTHEW McPherrin", 
"id":57779,"mode":"FM","mylat":"37.751952821","mylong":"-122.4208688735",...}
```

Example 6-2. Accumulator empty line count in Python

```python
file = sc.textFile(inputFile)
# Create Accumulator[Int] initialized to 0
blanklines = sc.accumulator(0)

def extractCallSigns(line):
    global blanklines  # Make the global variable accessible
    if (line == ""):
        blankLines += 1
    return line.split(" ")

callSigns = file.flatMap(extractCallSigns)
callSigns.saveAsTextFile(outputDir + "/callsigns")
print "Blank lines: %d" % blankLines.value
```
### Numeric RDD Operations

**Table 6-2. Summary statistics available from StatsCounter**

<table>
<thead>
<tr>
<th>Method</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>count()</td>
<td>Number of elements in the RDD</td>
</tr>
<tr>
<td>mean()</td>
<td>Average of the elements</td>
</tr>
<tr>
<td>sum()</td>
<td>Total</td>
</tr>
<tr>
<td>max()</td>
<td>Maximum value</td>
</tr>
<tr>
<td>min()</td>
<td>Minimum value</td>
</tr>
<tr>
<td>variance()</td>
<td>Variance of the elements</td>
</tr>
<tr>
<td>sampleVariance()</td>
<td>Variance of the elements, computed for a sample</td>
</tr>
<tr>
<td>stdev()</td>
<td>Standard deviation</td>
</tr>
<tr>
<td>sampleStddev()</td>
<td>Sample standard deviation</td>
</tr>
</tbody>
</table>
Removing outliers

**Example 6-19. Removing outliers in Python**

```python
# Convert our RDD of strings to numeric data so we can compute stats and remove the outliers.
distanceNumerics = distances.map(lambda string: float(string))
stats = distanceNumerics.stats()
stddev = std.stdev()
mean = stats.mean()
reasonableDistances = distanceNumerics.filter(
    lambda x: math.fabs(x - mean) < 3 * stddev)
print reasonableDistances.collect()
```
Machine Learning Library in Spark — MLlib

An example of using MLlib for text classification task, e.g., identifying spammy emails.

1. Start with an RDD of strings representing your messages.
2. Run one of MLlib’s feature extraction algorithms to convert text into numerical features (suitable for learning algorithms); this will give back an RDD of vectors.
3. Call a classification algorithm (e.g., logistic regression) on the RDD of vectors; this will give back a model object that can be used to classify new points.
4. Evaluate the model on a test dataset using one of MLlib’s evaluation functions.
Example: Spam Detection

Example 11.1. Spam classifier in Python

```python
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.feature import HashingTF
from pyspark.mllib.classification import LogisticRegressionWithSGD

spam = sc.textFile("spam.txt")
normal = sc.textFile("normal.txt")

# Create a HashingTF instance to map email text to vectors of 10,000 features.
# Each email is split into words, and each word is mapped to one feature.
tf = HashingTF(numFeatures = 10000)
spamFeatures = spam.map(lambda email: tf.transform(email.split(" ")))
normalFeatures = normal.map(lambda email: tf.transform(email.split(" ")))

# Create LabeledPoint datasets for positive (spam) and negative (normal) examples.
positiveExamples = spamFeatures.map(lambda features: LabeledPoint(1, features))
negativeExamples = normalFeatures.map(lambda features: LabeledPoint(0, features))
trainingData = positiveExamples.union(negativeExamples)
trainingData.cache() # Cache since Logistic Regression is an iterative algorithm.

# Run Logistic Regression using the SGD algorithm.
model = LogisticRegressionWithSGD.train(trainingData)

# Test on a positive example (spam) and a negative one (normal). We first apply
# the same HashingTF feature transformation to get vectors, then apply the model.
posTest = tf.transform("0 M G GET cheap stuff by sending money to ...".split(" "))
negTest = tf.transform("Hi Dad, I started studying Spark the other ...".split(" "))
print "Prediction for positive test example: %g" % model.predict(posTest)
print "Prediction for negative test example: %g" % model.predict(negTest)
```
Feature Extraction Example — TF-IDF

Example 11-7. Using HashingTF in Python

```python
>>> from pyspark.mllib.feature import HashingTF

>>> sentence = "hello hello world"
>>> words = sentence.split()  # Split sentence into a list of terms
>>> tf = HashingTF(10000)  # Create vectors of size S = 10,000
>>> tf.transform(words)
SparseVector(10000, [3065: 1.0, 6861: 2.0])

>>> rdd = sc.wholeTextFiles("data").map(lambda (name, text): text.split())
>>> tfVectors = tf.transform(rdd)  # Transforms an entire RDD
```

Example 11-8. Using TF-IDF in Python

```python
from pyspark.mllib.feature import HashingTF, IDF

# Read a set of text files as TF vectors
rdd = sc.wholeTextFiles("data").map(lambda (name, text): text.split())
tf = HashingTF()
tfVectors = tf.transform(rdd).cache()

# Compute the IDF, then the TF-IDF vectors
idf = IDF()
idfModel = idf.fit(tfVectors)
tfIdfVectors = idfModel.transform(tfVectors)
```
Linear Regression

Example 11-10. Linear regression in Python

```python
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.regression import LinearRegressionWithSGD

points = # (create RDD of LabeledPoint)
model = LinearRegressionWithSGD.train(points, iterations=200, intercept=True)
print "weights: %s, intercept: %s" % (model.weights, model.intercept)
```
Classifiers

**Logistic regression**

Logistic regression is a binary classification method that identifies a linear separating plane between positive and negative examples. In MLlib, it takes LabeledPoints with label 0 or 1 and returns a LogisticRegressionModel that can predict new points.

**Support Vector Machines**

Support Vector Machines, or SVMs, are another binary classification method with linear separating planes, again expecting labels of 0 or 1. They are available through the SVMWithSGD class, with similar parameters to linear and logistic regression. The returned SVMModel uses a threshold for prediction like LogisticRegressionModel.

**Naive Bayes**

Naive Bayes is a multiclass classification algorithm that scores how well each point belongs in each class based on a linear function of the features. It is commonly used in text classification with TF-IDF features, among other applications. MLlib implements Multinomial Naive Bayes, which expects nonnegative frequencies (e.g., word frequencies) as input features.
Decision trees and Random Forests

Figure 11-2. An example decision tree predicting whether a user might buy a product

In MLlib, you can train trees using the mllib.tree.DecisionTree class, through the static methods trainClassifier() and trainRegressor().
Clustering

**K-means**

MLlib includes the popular K-means algorithm for clustering, as well as a variant called K-means|| that provides better initialization in parallel environments. K-means|| is similar to the K-means++ initialization procedure often used in single-node settings.

- **initializationMode**
  The method to initialize cluster centers, which can be either “k-means||” or “random”; k-means|| (the default) generally leads to better results but is slightly more expensive.

- **maxIterations**
  Maximum number of iterations to run (default: 100).

- **runs**
  Number of concurrent runs of the algorithm to execute. MLlib’s K-means supports running from multiple starting positions concurrently and picking the best result, which is a good way to get a better overall model (as K-means runs can stop in local minima).

Like other algorithms, you invoke K-means by creating a `mllib.clustering.KMeans` object (in Java/Scala) or calling `KMeans.train` (in Python).
Collaborative Filtering

Collaborative Filtering and Recommendation

Collaborative filtering is a technique for recommender systems wherein users’ ratings and interactions with various products are used to recommend new ones. Collaborative filtering is attractive because it only needs to take in a list of user/product interactions: either “explicit” interactions (i.e., ratings on a shopping site) or “implicit” ones (e.g., a user browsed a product page but did not rate the product).

Alternating Least Squares

MLlib includes an implementation of Alternating Least Squares (ALS), a popular algorithm for collaborative filtering that scales well on clusters.\textsuperscript{6} It is located in the mllib.recommendation.ALS class.
Dimensionality Reduction

Principal component analysis

Given a dataset of points in a high-dimensional space, we are often interested in reducing the dimensionality of the points so that they can be analyzed with simpler tools. For example, we might want to plot the points in two dimensions, or just reduce the number of features to train models more effectively.

Singular value decomposition

MLlib also provides the lower-level singular value decomposition (SVD) primitive. The SVD factorizes an \( m \times n \) matrix \( A \) into three matrices \( A \approx U \Sigma V^T \), where:

- \( U \) is an orthonormal matrix, whose columns are called left singular vectors.
- \( \Sigma \) is a diagonal matrix with nonnegative diagonals in descending order, whose diagonals are called singular values.
- \( V \) is an orthonormal matrix, whose columns are called right singular vectors.
Homework #1

1. Download and Install Spark. Learn how to use it.

2. Download Wikipedia dataset. Extract about 1,000 pages (items) based on your own interest. Create TF-IDF of each page.

3. Use Twitter Streaming API to receive real-time twitter data. Collect 30 mins of Twitter data on 5 companies using keyword=xxx (e.g., ibm). Consider all Twitter data from a company is one document. Create TF-IDF of each company’s tweets in that 30 minutes.

4. Use Yahoo Finance to receive the Stock price data. Collect 30 mins of Finance data on 5 companies, one value per minute. Use the outlier function to display outliers that are large than two standard deviation.