Big Data Analysis using Spark

Spark

Apache Spark is an open source fast and general engine for large-scale data processing, which can run 100 times faster than Hadoop alone in memory or 10 times via disk. It can be built interactively with Python/Scala or with Java, and can be combined with Shark (http://shark.cs.berkeley.edu/) to query via SQL, and MLlib (http://spark.apache.org/mllib/) which is included with it for machine learning purposes.

Additionally it also includes GraphX and Spark Streaming which allow for work with graphs and stream mining. Spark uses the Hadoop core library to talk to HDFS and other Hadoop-supported storage systems. Because the protocols have changed in different versions of Hadoop, you must build Spark against the same version that your cluster runs. For this project, I installed the pre-built package of Spark 1.0.0 (release date May 30, 2014) for Hadoop 2 available at http://spark.apache.org.

Dataset:

“MY World” is a global survey for citizens led by the United Nations and partners. According to the site, its aims is to "capture people's voices, priorities and views, so that global leaders can be informed as they begin the process of defining the new development agenda for the world". Thus far, it contains a little over 1.2 million responses. Each response contains a timestamp, a source (offline,website,etc), partner_code (if any), country, age, gender, education, up to six priorities (non-order, selected from a list of 16),a text box for a suggested priority and a vote reason, a free text box for anything.
The priorities themselves include the following:

100 = "Action taken on climate change"
101 = "Better transport and roads"
102 = "Support for people who can't work"
103 = "Access to clean water and sanitation"
104 = "Better healthcare"
105 = "A good education"
106 = "A responsive government we can trust"
107 = "Phone and internet access"
108 = "Reliable energy at home"
109 = "Affordable and nutritious food"
110 = "Protecting forests, rivers and oceans"
112 = "Political freedoms"
111 = "Protection against crime and violence"
113 = "Freedom from discrimination and persecution"
114 = "Equality between men and women"
115 = "Better job opportunities"

Additionally, a nonprofit group of data analysts in NY named DataKind had an event last November to attempt to analyze the dataset at that point in time, and added country specific information gathered from the World Bank for each row in the set. This added fields for a rule of law, corruption, stability, government effectiveness, and voice (ie, speaking to people's needs) and each has corresponding estimate, rank, raw number, standard deviation, and upper/lower estimates. The final csv file is 307M big.

Why Spark?

I initially tried analysing this data in R alone as it didn't seem excessively large, but at least on my laptop (with 8 gigs of memory), it responded quite slowly and thus I tried using some R specific "big data" libraries that I had heard of (specifically ff and bigmemory) but ran into difficulties with these, and while doing so became aware of Spark which seemed quite interesting.

Getting Started/Benchmark.

I downloaded, installed Spark, and went through the quick start guide (http://spark.apache.org/docs/latest/quick-start.html), running **./bin/pyspark** to start the interactive environment and go through examples things there. I then moved on to using **./bin/spark-submit** to run standalone python files which included a SparkContext within them (whereas the interactive environment includes a SparkContext contained in the sc variable by default)

I first ran a benchmark to see who fast it took to load my file in plain old python vs Spark. This method looks like this:

```python
import csv
import sys
import time

if __name__ == '__main__':
    start_time = time.time()
    f = open(sys.argv[1], 'rb')
    reader = csv.reader(f)
    headers = reader.next()
    column = {}
    for h in headers: column[h] = []

    for row in reader:
        for h, v in zip(headers, row): column[h].append(v)

    print(column.keys())
    end_time = time.time() - start_time
    print("--- Execution time in seconds ---")
    print(end_time)
```

to run: **python UnitedNationsSurveryCluster.py myworld_wgi.csv**

The Spark version was as follows:

```
import csv
import sys
import time
import numpy as np
from pyspark import SparkContext
from pyspark.mllib.clustering import KMeans

def parseVector(line):
        return np.array([x for x in line.split(' ') if type(x) ==
float])

if __name__ == '__main__':
    start_time = time.time()
    sc = SparkContext(appName="UnitedNationsSurveryClustering")
    lines = sc.textFile(sys.argv[1])
    data = lines.map(parseVector).cache()
    end_time = time.time() - start_time
    print("--- Execution time in seconds ---")
    print(end_time)
```

We ran it as follows: **path_to/spark-1.0.0-bin-hadoop2/bin/spark-submit UnitedNationsSurveryCluster.py myworld_wgi.csv**

The first manner had an execution time in seconds of 40.29 the first time and 46.63 the second time whereas Spark took 19.97 seconds the first time and 8.17 seconds the second time!

Initial Code Explained

The first python manner is straightforward to read, but the second Spark version is as follows. The first thing a Spark program must do is to create a SparkContext object, which tells Spark how to access a cluster. Spark revolves around the concept of a **resilient distributed dataset** (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel.

There are two ways to create RDDs:
1)parallelizing an existing collection in your driver program, or
2)referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat. This manner is used in the benchmark code.

1) Parallelized collections are created by calling SparkContext's parallelize() method on an existing iterable or collection in your driver program. The elements of the collection are copied to form a distributed dataset that can be operated on in parallel.

```
    data = [1, 2, 3, 4, 5]
    distData = sc.parallelize(data)
```

Once created, the distributed dataset (distData) can be operated on in parallel. For example, we can run the following to sum up the elements of the list.

```
    distData.reduce(lambda a, b: a + b)
```

*One important parameter for parallel collections is the number of slices to cut the dataset into. Spark will run one task for each slice of the cluster. Typically you want 2-4 slices for each CPU in your cluster. Normally, Spark tries to set the number of slices automatically based on your cluster.
However, you can also set it manually by passing it as a second parameter to parallelize (e.g. sc.parallelize(data, 10)).

2) PySpark can create distributed datasets from any file system supported by Hadoop, including your local file system, HDFS, KFS, Amazon S3, etc. The current API is limited to text files, but support for binary Hadoop InputFormats is expected in future versions.

Text file RDDs can be created using SparkContext's textFile which reads a given URI as a collection of lines. Once created, the RDD (the lines variable in our code) can be acted on by dataset operations (using map reduce paradigm).

* All of Spark's file-based input methods, including textFile, support running on directories, compressed files, and wildcards as well. For example, you can use textFile("/my/directory"), textFile("/my/directory/*.txt"), & textFile("/my/drctry/*.gz").

* The textFile method also takes an optional second argument for controlling the number of slices of the file. By default, Spark creates one slice for each block of the file (blocks being 64MB by default in HDFS), but you can also ask for a higher number of slices by passing a larger value.

An RDD frame of mind

The RDD operations are really the thing that differentiate Spark from normal data analysis in python, both in terms of speed/power, but also because the tools are very different. For instance, simply figuring out how to load in and parse a CSV file as opposed to a plain text file is a big undertaking for the newly initiated to SPARK (or at least for this one it was). There are many tutorials and examples online and a lot of interest, but its still relatively new. See references for more a good primer.

Finding Clusters with Spark

After reading in the data into Spark you can do clustering in the following way (*note, this is currently causing errors in the centeroids its returning being empty. I'm using only continous data from the dataset, but I wasn't able to figure out what was causing the error. It wasn't a runtime one, but rather that the centeroids return appeared empty)

```
K = int(sys.argv[2])
convergeDist = 1000

kPoints = data.takeSample(False, K, 1)
tempDist = 1.0

while tempDist > convergeDist:
 closest = data.map(lambda p: (closestPoint(p, kPoints), (p, 1)))
 pointStats = closest.reduceByKey( lambda (x1, y1), (x2, y2): (x1
+ x2, y1 + y2))
 newPoints = pointStats.map( lambda (x, (y, z)): (x, y /
z)).collect()

 tempDist = sum(np.sum((kPoints[x] - y) ** 2) for (x, y) in
newPoints)

 for (x, y) in newPoints:
  kPoints[x] = y

print "Final centers: " + str(kPoints)
```

Conclusions:

This was my first journey into Spark and although there will be some overheard in learning the subtlities of RDD operations and how to fulfill some common tasks I'm accustomed too in R (and to a lesser extent in Python). Performance wise I was very impressed and how fast its able to do common tasks without leveraging its distributed offerings. Further steps will include learning how to better debug things (as for this initial go, the approach taken was that of printing out commands to figure out class/types expected, etc), completing this actual task and then learning how to leverage this on remote Amazon stored sets.

references/links:

programming guide:
http://spark.apache.org/docs/latest/programming-guide.html
mllib guide: http://spark.apache.org/docs/latest/mllib-guide.html
mllib examples:
http://docs.sigmoidanalytics.com/index.php/MLlib_Examples
2014 summit: http://spark-summit.org/2014
2013 summit: http://spark-summit.org/2013 (includes slides, videos)
un survey dataset: http://54.227.246.164/dataset/
hackpad to UN data: https://nycdatadive2013.hackpad.com/UN-Data-Diving-the-Worlds-Development-Priorities-DK-LVdzneYoGIK
rdd primer: http://spark.apache.org/docs/latest/programming-guide.html#rdd-operations