SPARK: PROGRAMMING CHALLENGES

MANAGING BIG DATA (MBD)
2017/2018

DOINA BUCUR

These programming challenges will teach you the Spark programming Python API for big-data mining. You will run your code on single Hadoop nodes on the Farm cluster, as a stepping stone to running code cluster-wide. These challenges will also introduce you to some (samples) of the full datasets we store on the larger CTIT cluster; they can lead to ideas for your projects.

Running Spark applications on Hadoop nodes, Farm cluster. Login on the Farm machine assigned to you. For example, if that is `farm01`, do `ssh username@farm01.ewi.utwente.nl`. (NB: If you installed an ssh cryptographic key pair and thus skip typing your password at login, first do a `kinit` and type your password.)

To start Spark Shell, simply type `pyspark` and you obtain the `>>>` prompt where you can execute Spark code interactively. While you're a beginner at Spark, do use the shell – debugging your code will be much easier.

```
bucurd@farm01: ~$ pyspark
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.6.0
      /_/
Using Python version 2.7.6 (default, Oct 26 2016 20:30:19)
SparkContext available as sc, SQLContext available as sqlContext.
>>>
```

To instead execute a Python Spark program from a `.py` file, type in the Unix directory where your file is:

```
bucurd@farm01: ~$ spark-submit yourfile.py 2> /dev/null
```

You may omit the redirection of the `stderr` to `/dev/null`; you will then see some extra Hadoop status information in the terminal.

Spark API documentation. Use the matching API version to what our clusters have installed, 1.6.0:

`https://spark.apache.org/docs/1.6.0/api/python/` (packages: `pyspark`, `pyspark.sql`)

What to submit. Each Python Spark programming challenge has an identifier written in all-caps (e.g., IINDEX). If you've solved a challenge, name the file: `IINDEX-s123456-s234567-RND.py`, where the s-numbers are the student ids in the group, and `RND` is a short random string of your choice (this is needed to prevent others reading your file). *Important*: at the top of the file, add a comment block with: the full names and student numbers of the (pair of) students who wrote this submission.

How to submit. We use a simple drop box on the Farm cluster. First, copy your code (using `scp` or `rsync`) to your Farm machine. Then, in the folder where your code is, do `cp IINDEX-s123456-s234567-RND.py /home/bucurd/MBD`. The latter folder is a write-only folder on the Unix NFS file system, accessible from any Farm machine; if the command exits without error, your code was copied.

Points towards your final grade. Pay attention to the *complexity* of your code (the concepts are the same as in MapReduce: minimize network shuffling, only send small data to one machine). Points will be subtracted for code that is inefficient, obfuscated, undocumented, or terribly inelegant; submit professional-looking code only! Your grade from all programming challenges (MapReduce, Spark) will be "capped" at 2 grade points. Each challenge has grade points attached. No challenge is mandatory; you may choose freely which challenges to attempt.

Plagiarism. In case you haven't studied at the UT before and don't know the rules: **never ever exchange any code**! You may talk about your solutions, but no line of code can be pasted between students. Copied solutions will be very easy to spot here, because (a) web-crawled data is messy, and there are many programmatic ways to clean it; (b) there are also many ways to design a reasonably correct program. It is unlikely that two groups will code the same way, and there's rarely one absolutely best program; even your output will differ, depending on your many data-cleaning / program-design decisions!

(grade points: .3)

An inverted index for a document base (challenge identifier: **IINDEX**)

Given a set of plain-text documents, construct the inverted index for these documents; you've seen this already in the MapReduce framework – this time, on realistic data.

This inverted index is a list, in which each item is the following: a word and the complete *set* of document names in which this word can be found. Each document name appears only once in a set, regardless of how many copies of the same word exist in that document.

Schematically, such an inverted index would look like this:

word1 {`document2, document3, document4`}
word2 {`document1, document2`}
word3 {`document2`}

Input

You have a set of plain-text books in the HDFS folder `/data/doina/Gutenberg-EBooks`. Here is a listing:

```
bucurd@farm01: Spark$ hdfs dfs -ls /data/doina/Gutenberg-EBooks
Found 14 items
Gutenberg-Adventures_of_Huckleberry_Finn.txt
Gutenberg-Alice_in_Wonderland.txt
Gutenberg-Autobiography_of_Benjamin_Franklin.txt
Gutenberg-Beowulf.txt
Gutenberg-Dracula.txt
Gutenberg-Fairy_Tales.txt
Gutenberg-Great_Expectations.txt
Gutenberg-Metamorphosis.txt
Gutenberg-Moby_Dick.txt
Gutenberg-Peter_Pan.txt
Gutenberg-The_Adventures_of_Tom_Sawyer.txt
Gutenberg-The_Importance_of_Being_Earnest.txt
Gutenberg-The_Prince.txt
Gutenberg-War_and_Peace.txt
```

You can look through a book: `hdfs dfs -cat /data/doina/Gutenberg-EBooks/Gutenberg-Dracula.txt`. You'll see that the books contain brief headers and footers which are not part of the books themselves. Ignore this fact, so that your code remains simple.

You can read all files in one go using `wholeTextFiles(path)` from the `pyspark` library.

Output

The inverted index will be large, so you're not asked to print it out entirely. Instead, print out, from the inverted index, only those words which are contained in *all* the books! (You will obtain under half a screenful of words.) You may also ignore issues related to punctuation and capitalization; remove them or not, as you like.

Sample output:

a about above afraid after again against air all almost alone along always am among an and another answer any anyone anything anywhere are arm arms as asked at author away back be beautiful because become been before beg behind being believe below best better between both bring broke brought but by call called calling came can care carried certainly change changed child close come comes coming copy cost could course cut date day dead deal dear decided deep did different do doing done door doubt down each easy eat ebook either else end english enough even ever every everything exactly eyes face fact fall...

(grade points: .4)

Find the most frequent hashtags (challenge identifier: **HASHTAGS**)

Given a large number of tweets stored in a structured `json` format, calculate the most frequent hashtags among these tweets. You've already seen this problem for the MapReduce framework; this time it's on (a sample from) a big structured dataset.

Not all tweets contain hashtags. Some tweets may contain multiple hashtags, including duplicates; you may include duplicates in the count.

Print in the output the most frequent 20 hashtags, and how many times these hashtags appears in tweets. Sort this output descendingly by the hashtag count. This will look something like:

```
(hashtag1, count1)
(hashtag2, count2)
(hashtag3, count3)
...
```

(where `count1` is larger than or equal to `count2`, etc.)

Input

Take one hour of Dutch tweets from right before New Year's Eve 2017. This is stored in the gzipped HDFS file:

`/data/doina/twitterNL/201612/20161231-23.out.gz`.

Each tweet has a very complicated schema; you can see the *first tweet* in its entirety with:

```
hdfs dfs -text /data/doina/twitterNL/201612/20161231-23.out.gz | head -1
```

RDDs are not the most suitable data structures to read structured data into; use the `pyspark.sql` library to read the input into a DataFrame, then look at the schema:

```
from pyspark import SparkContext
from pyspark.sql import SQLContext
sc = SparkContext("local", "Twitter")
sqlc = SQLContext(sc)

df = sqlc.read.json("/data/doina/twitterNL/201612/20161231-23.out.gz")
df.printSchema()
```

You can get the hashtags from the tweets in at least two ways: by splitting into words the `text` column (see the schema printed by the code above), or by enumerating the `entities.hashtags` column of the same schema. Start, for example, with:

```
tweets = df.select("id", "entities.hashtags", "text")
tweets.printSchema()
```

and you will get a simplified, human-readable tweet dataset with the schema:

```
root
 |-- id: long (nullable = true)
 |-- hashtags: array (nullable = true)
 |    |-- element: struct (containsNull = true)
 |    |    |-- indices: array (nullable = true)
 |    |    |    |-- element: long (containsNull = true)
 |    |    |-- text: string (nullable = true)
 |-- text: string (nullable = true)
```

DataFrames can be quickly turned into RDDs when you want to do so (see the library).

Output

If you are foreign, the most frequent hashtag in this dataset will tell you something about a favourite Dutch activity, done before every New Year.

(grade points: .4)

Compute Twitter user activity (challenge identifier: **USERS**)

Given a large number of tweets stored in a structured `json` format, calculate the number of times each user in the dataset has tweeted.

Print in the output the most active 40 users, and how many times these users tweeted. Sort this output descendingly by the tweet count. This will look something like:

```
(user1, count1)
(user2, count2)
(user3, count3)
...
```

(where `count1` is larger than or equal to `count2`, etc.)

Input

Take the hour of Dutch tweets from the gzipped HDFS file:

`/data/doina/twitterNL/201612/20161231-23.out.gz`

or the multi-lingual, global tweets from Jan 1, 2017, one minute of tweets per archive, from the files in the HDFS folder:

`/data/doina/Twitter-Archive.org/01/01/01/`.

The datasets above have similar schemas. If you take the multi-lingual dataset, you can read all archives in one go with the same function as for a single file, `sqlc.read.json(...)`. NB: this dataset contains tweet deletion events (which you may want to filter out first) of the form:

`{"delete":{"status":{"id":...,"user_id":...,"timestamp_ms":...}}}`

You can consider a user to be uniquely defined by either the `user.id` column (see the schema of the dataset), or by the `user.screen_name` column of the same schema; the `user.name` will not be unique.

Sample output

```
(u'watkopenwij', 100)
(u'openomroep', 67)
(u'volkstribunaal', 66)
(u'imkeroose', 54)
...
```

or

```
(u'sports_shinkan', 5)
(u'jon_umi_sif', 5)
(u'diamondbIack', 4)
(u'sheilamaestroz', 4)
...
```

(grade points: .4)

Compute Amazon music sales-rank distribution (challenge identifier: **MUSIC**)

Given a dataset of Amazon products crawled from the public `amazon.com` pages, where most product records contain the *sales rank* of the product (a natural number), show visually the *distribution of the sales ranks* of the products in this crawl.

Input

Take an archived `json` dataset of over 1 quarter million music products on sale:

`/data/doina/UCSD-Amazon-Data/meta_Digital_Music.json.gz`.

The products have a simple, easy-to-read schema; you may want to read the data into Spark DataFrames rather than RDDs.

See the first product in the file with:

```
hdfs dfs -text /data/doina/UCSD-Amazon-Data/meta_Digital_Music.json.gz | head -1
```

and print the schema with:

```
df = sqlc.read.json(filename)
df.printSchema()
```

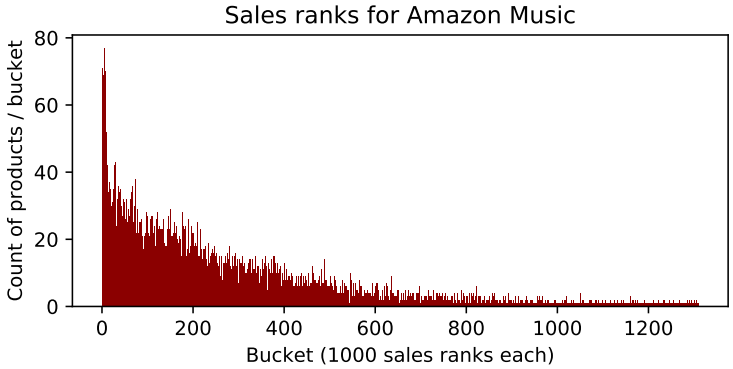
A record is not always complete, because not all product pages at Amazon are; for example, not all product records have sales ranks.

This dataset may be modest in size, but do write your program for *big data* inputs, because you may not get this lucky with the datasets you'll use in your project.

Output

Plot the distribution as a histogram, where you "bucket" N sales ranks per data point on the x axis, and show the count of products having those sales ranks on the y axis. Choose your bucket size so that the data needed for this plot is very *small data*. Submit this picture alongside your code, with the same filename, but different file extension (for example, `MUSIC-s123456-s234567-RND.pdf`).

I obtained the plot below for $N = 1000$, which answers the question whether this Amazon crawl sampled the sales ranks uniformly or not.



Also print in the output the bucket size on one line, and the data points in the histogram (the y axis values), in order, on the second line, e.g.:

```
1000
[67, 71, 69, 59, 77, 57, 70, 49, 50, 52, 39, 40, ...]
```

This will serve as a more fine-grained output, useful when testing the program.

(grade points: .5)

Compute the most also-bought book on Amazon (challenge identifier: **BOOK**)

Mine a dataset of Amazon books crawled from the public `amazon.com` pages, where most records for a book B contain a list of other books that were frequently *also_bought* with B, by the same customer.

Find that book which was most frequently *also_bought* with the other books.

Input

Take an archived `json` dataset of a quarter million books on sale:

`/data/doina/UCSD-Amazon-Data/meta_Books_sample.json.gz`.

(This is a *sample* from a larger dataset of just under 2.5 million books. You may also use the *full* dataset, from the same folder, but the runtime on a single Farm machine may become annoyingly long.)

See the first product in the file with:

```
hdfs dfs -text /data/doina/UCSD-Amazon-Data/meta_Books_sample.json.gz | head -1
```

and print the schema with:

```
df = sqlc.read.json(filename)
df.printSchema()
```

The `asin` string in the schema is the unique identifier for a book; this is identical to the ISBN-10 number, if any is allocated to the book. The dataset includes Kindle books, whose `asin` starts in the letter `B`, and which don't have ISBN-10 numbers allocated. Many of the fields in the schema can be missing in any given record. Here's a typical book record:

```
{'asin': '0000913154',
 'title': 'The Way Things Work: An Illustrated Encyclopedia of Technology',
 'price': 23.26,
 'imUrl': 'http://ecx.images-amazon.com/images/I/7113akhDnTL.jpg',
 'related':
     {'also_bought': ['0590429892', '1114119512', 'B000HW2YQE', '0395938473', '0395428572', ...],
      'buy_after_viewing': ['0395938473', 'B000GP0CT8', '0590429892', '1114119512']},
 'salesRank': {'Books': 455782},
 'categories': [['Books']]}
```

A book B is frequently *also_bought* with other books if many other books have the `asin` of B in their `'also_bought'` lists.

Output

Print in the output, for that book B which was most frequently *also_bought* with other books:

count: the number of other books also-bought with B

asin: the `asin` of B

record: (if a full record for B exists in the dataset; it may not!) print also that record

If there are multiple books with the same count, print any one of them.

For the sample dataset, I obtained a top book which won a fiction prize. For the full dataset, it was a very inexpensive, non-fiction Kindle book.