SPARK ASSIGNMENTS

MANAGING BIG DATA (MBD)
2021/2022

DOINA BUCUR

You now practice the Spark Python API for big-data processing. To start with, you run your code on single cluster machine, as a stepping stone to running code cluster-wide.

Running Spark applications on Hadoop nodes. See the demo log of Lecture 2; read that very carefully before solving the assignments. In essence: Login on the machine assigned to you, `ssh snumber@ctit007.ewi.utwente.nl`. (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 the interactive Spark Shell, type `pyspark` and you obtain the `>>>` prompt. While you're a beginner, do use the shell — debugging your code will be much easier.

To instead execute a Python Spark program from a `.py` file stored on the NFS file system, type:

```
snumber@ctit007: ~$ time spark-submit yourfile.py 2> /dev/null
```

You may omit the redirection of the `stderr` to `/dev/null`; you will then see Hadoop information, warnings, and also program errors in the terminal. You may also omit timing your program.

Spark API documentation. Use the matching Spark API version to what we have installed, 2.4.7:

`https://spark.apache.org/docs/2.4.7/api/python/` (packages: `pyspark`, then later `pyspark.sql`)

What to submit. Each assignment has an identifier (e.g., IINDEX). Name the file: `IINDEX-s123456-s234567-RNDSTR.py`, where the s-numbers are the student ids in the group (one or two), and `RNDSTR` is a short, secret string of your choice (useful to prevent others from accidentally overwriting your file).

Files submitted will be assumed to work with the default Python2. If you worked in Python3, no problem: just add a 3 right after the assignment identifier, so name the file: `IINDEX3-s123456-s234567-RNDSTR.py`. This will tell me to execute your file with the right arguments.

Important: at the top of the file, add a comment with: the names and student numbers, and the *time* your code took to execute!

How to submit. We use a simple "drop box" on the cluster itself. Since your code is already on some cluster machine, do

```
cp IINDEX-s123456-s234567-RNDSTR.py /home/bucurd/MBD
```

The latter is a write-only folder on the NFS file system; if the command completes without error, your code was copied! (You cannot also read that folder to verify.) If you are the user who copied this file, you (but only you) will also be able to overwrite it.

Points towards your final grade. Points will be subtracted for code that is slow, obfuscated, undocumented, or terribly inelegant; submit professional-looking code only, not the first draft that works!

Each assignment has grade points attached. **No assignment is mandatory**; you may choose which to attempt. Try to gain **1 grade point** from them (10% of your course grade). If you do more of them, the extra points will be awarded to you as a bonus. There is an upper bound to the bonus: your points from assignments, plus your points from the project, will be capped at 5 (50% of your course grade).

Plagiarism. In case you haven't studied at the UT and don't know the rules: **never ever send anyone your code, nor look at someone else's code, even if you find a way to do it**! All suspicious cases will be reported to the Examination Board, which will delay your finishing this course. You may talk about your solutions, but no line of code should be shared between groups.

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

grade points: .15 (deadline: Wed Dec 1, 23:59)

Given a set of plain-text documents, construct the inverted index for these documents.

This inverted index is a list, in which each item is the following: a **word** and the collection of **document names** in which this word can be found. Each document name should appear only once in a collection, 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`:

```
snumber@ctit007: Spark$ hdfs dfs -ls /data/doina/Gutenberg-EBooks
Found 14 items
Gutenberg-Adventures_of_Huckleberry_Finn.txt
Gutenberg-Alice_in_Wonderland.txt
...
```

Assume every single file is small data, but all together the folder could be big data.

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; you may remove those, or not. You may ignore punctuation and capitalization, or not, as you prefer.

You can read all files in one go using `wholeTextFiles(path)`, as in the demo of Lecture 2. You are allowed to hardcode the number of files present (in practice, you would first obtain this number by some simple means on the command line, then use it in the Spark code), but you can also obtain it in your code.

Output

The inverted index in the output could be large. Your program must compute all of it. (But it shouldn't print it.)

To print something small in the output, `filter` and print from the inverted index only the very frequent words: those words which are contained in **at least 13** (so, all or almost all) of these books. Omit the document names in this print, to keep it small.

Sample output:

a able about above across act advantage afraid after again against age ago air alarm all allow allowed almost alone along already also always am among an and another answer anxious any anyone anything anywhere are arm arms as ask asked at author away back be beautiful because become bed been before beg began begin beginning begun behind being believe below best better between beyond bit blow book both bound break bring broke broken brother brought but by call called calling came can cannot care careful carefully carried carry carrying case caught certain certainly chance change changed chief child children choose clear close come comes coming conversation copy cost could count country course cross cry curious cut dare date daughter day days dead deal dear death decided deep did different direction distance do does doing done door doubt down dressed drew drink drive drop during duty each ears easily easy eat ebook edge eight either else empty end english enough enter entirely escape even evening ever every everything exactly eyes ...

If you're interested

You could (hypothetically) implement your favourite variant of *tf-idf* over this basic code. No need to submit such code, it's only something to think about. It is implemented already in the `pyspark.mllib` package.

grade points: .20 (deadline: Wed Dec 1, 23:59)

K-nearest neighbours regression in Spark (challenge identifier: **KNN**)

Say you have a (large) file with many two-dimensional data points of the form *x,y,value*. For each data point, x and y are two features of that point (for example, two-dimensional geographic coordinates, or the height and width of an object). *value* is the true value of an important aspect of that data point, for example a weather measurement such as the annual amount of rain at those coordinates, or the weight of the object.

You also get **one new data point** for which you know x and y, but you must find out the most likely *value*. The algorithm you should implement is simple: the *value* of the new data point is the mean value among the values of the k **closest points** in terms of Euclidean distance.

Input

The dataset with known values is stored as a file in plain text (`.csv`) at the HDFS location `/data/doina/xyvalue.csv`, one point per line without any spaces (spaces are added here for clarity):

```
258.2, 227.3, 463
113.2,  39.9, 356
203.9,  82.3, 735
149.2, 215.5, 735
 70.3, 264.3, 390
```

Assume this single file could be big data. First, find a suitable Spark method to read a *single text file*. It's very easy to parse this file in the code.

You can fix k to a small value, say $k = 100$.

Assume the new data point is given to you from the beginning (so you can hardcode it for simplicity), say:

```
point = (100, 100)
```

Output

Output a single number: the *value* of the new data point that was given.

If you're interested

Could you (hypothetically) compute the values of p new data points at the same time, where p is much larger than 1, $p \gg 1$? The big question is whether the program still looks efficient. No need to implement this, but you could add a comment stating the time complexity of your solution for 1 point, and then for p points. This time complexity means roughly how many computations you make in total, so could be written as an (asymptotic) function of the dataset size (call it N), k (with k much smaller than N, $k \ll N$), and p (with p taking any value). Something like $\mathcal{O}(N \cdot k + p)$.

The answer may explain to you why the `pyspark.ml` library only provides an approximate method for the nearest-neighbour algorithm.