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.

---

grade points: .15

(**deadline**: Wed Dec 1, 23:59)

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

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.

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

grade points: .20

(**deadline**: Wed Dec 1, 23:59)

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.

grade points: .15

(**deadline**: Wed Dec 8, 23:59)

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

Given a large number of tweets stored in a semi-structured `json` format, calculate the most frequent hashtags among these tweets.

Not all tweets contain hashtags, but some tweets contain multiple hashtags (perhaps including duplicates in the same tweet). You should count all hashtags from all tweets (not only the first hashtag in a tweet). You may or may not include duplicate hashtags in the count, as you like. Print in the output **the most frequent 20 hashtags**, and how many times each hashtag appears in tweets. Sort this output descendingly by the hashtag count:

```
hashtag1    count1
hashtag2    count2
hashtag3    count3
...
```

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

Input

Assume that the input is big data. Take the first 10 minutes of multilingual, worldwide tweets from Jan 1, 2020 (one minute of tweets per archive), from the files in the HDFS folder:

```
/data/doina/Twitter-Archive.org/2020-01/01/00/0*.json.bz2
```

You can read all archives in one go with the same function as for a single file, `spark.read.json(...)`, and this accepts regular expressions like the one just above with `*`.

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

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

df = spark.read.json("/data/...")
df.printSchema()
```

You can get the hashtags from the tweets in at least two ways: by splitting into words the `text` column, or by enumerating the `entities.hashtags` column of the same schema (these were already split by Twitter). The results may be slightly different.

Note: You must practice the new library, Spark SQL (do not switch to RDDs in mid-program). See the built-in functions over DataFrames, `pyspark.sql.functions`, to have again some useful non-SQL functionality. `explode(col)` is like `flatMap`, and there are also good new ways to write something like `reduceByKey`.

Output

Note: the terminal cannot print (`print, show`) just any character encoding, and you'll have multi-lingual text. Printing should work in the terminal if the strings are re-encoded, `.encode('utf-8')`.

Can anyone read the non-English hashtags below?

```
Dispatch  150
स्वर्ण_युग    150
BTSxRockinEve    136
Master    135
방탄소년단   125
BTS    113
SEVENTEEN     100
RockinEve     98
THE8   95
eyezmag   94
BTSxNYRE  69
오늘의방탄   68
前澤お年玉  66
BTSatTimeSquare  61
ななにー    52
JIMIN  44
NCT    43
HappyNewYear  42
happynewyear2020    40
Hello2020     37
```

Compute the most also-bought games product on Amazon (challenge identifier: **GAMES**)

grade points: .15

(**deadline**: Wed Dec 8, 23:59)

Mine a dataset of Amazon products from the category Video Games, crawled from the public `amazon.com` pages. Most records for a product P contain a list of up to 100 other products that were frequently *also_bought* with P, by the same customer.

Find that product which was most frequently also_bought with the other products. In other words, this is the product most often found in also_bought lists.

Input

Take an archived `json` dataset of video-games products on sale:

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

See the first product in the file with:

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

The `asin` string in the schema is the unique identifier for a product. Here's a typical record:

```
{
  'asin': '0078764343',
  'description': 'Brand new sealed!',
  'price': 37.98,
  'imUrl': 'http://ecx.images-amazon.com/images/I/513h6dPbwLL._SY300_.jpg',
  'related': {
    'also_bought':
      ['B000TI836G', 'B003Q53VZC', 'B00EFFW0HC', 'B003VWGBC0', ...],
    'bought_together':
      ['B002I098JE'],
    'buy_after_viewing':
      ['B0050SY5BM', 'B000TI836G', 'B0037LTTRO', 'B002I098JE']
  },
  'salesRank': {'Video Games': 28655},
  'categories': [['Video Games', 'Xbox 360', 'Games']]
}
```

Output

Print in the output the complete record for that product P which was most frequently also_bought with the others (if that record exists in the dataset). Try to pretty-print this record in a readable fashion, so one can understand what that product is.

Is it still on sale at Amazon now? (Look it up online.)

A distributed sort (challenge identifier: DSORT)

grade points: .15

(**deadline**: Wed Dec 8, 23:59)

Sometimes, you don't need to modify the data; you instead need to *reorganise* it, and save it back to disk in a different format than it originally was.

Take many integers from a single, large input file. Sort these integers, then write all the integers back to disk in (now) 10 files instead of just one, each file with a roughly equal number of integers (so, balanced in size). Each file has to be internally sorted, and there should be a clear order in which one has to read the files, so that the whole data is also globally sorted.

The advantage of writing to multiple files is that the data can be written (and later read) in parallel.

Input

Read the plain-text file `"/data/doina/integers.txt"` in the input. This file stores some lines of plain text, each with approximately $N = 50000$ comma-separated integers. (The choice of how many lines are there and how many integers per lines was arbitrary. Your code should be able to deal with the data quite generally.)

A line looks like:

```
31362573,13613439,89612293,32154322,75678333,10756594,...
```

Find in the Spark API a suitable method to read this type of input.

Assume that one line in the input is always *small data*, but the entire input (if one adds more lines) could be *big data*. All integers are between 0 and 10^8, somewhat uniformly distributed.

Output

There should be 10 files written back to disk, somewhere in your own HDFS `/user/snumber` space.

Inside each output file, you can write integers in any way that's convenient: for example one per line.

In your submission, **state your HDFS path containing the files**, in a comment. This could be something like `/user/snumber/folderwith10files`. It can belong to either one of the two students submitting. This is important, because the teacher won't be able to execute this submission (because she has no writing rights to your own HDFS space, and the path is hardcoded in your code), so she'll just read your code and also read the files that you outputted yourselves.

Hints: You can use any Spark library. You could save an RDD to disk using `saveAsTextFile()`, which takes a complete path to a *folder* where Spark will save the data (note: the folder must not exist already, or you'll get an error). Quite logically, one RDD partition will be saved into one file! There is a built-in method for sorting an RDD, `sortBy`, which you can make use of. There are also many, many other solutions.

grade points: .20

(**deadline**: Thu Dec 16, 23:59)

Changing Web pages (challenge identifier: **WEB**)

Given two Web crawls with (roughly) the same set of Web pages, calculate **the difference in text size for each page between crawls**. The text size is already listed in the crawls as a data field. The difference should only be computed for pages which:

- appear in both crawls, and
- have a text size greater than 0 in at least one crawl (so, had some plain text).

This difference (text size in the last crawl minus text size in the first crawl) can be either positive or negative.

Save the results (URL and difference in text size) to HDFS (see **Output** below). Make it clear in the code under **which path** you saved.

Add a comment with the exact `spark-submit` **command and flags** you used.

Technical hints (which hold for all programming from here onwards)

Run code responsibly. When using big data, you must code well in Spark, with awareness of what is appropriate to do on big data and what not. You must run your code responsibly, limiting the resources you use (distributed cores and memory), and not overloading the driver machine (see Lecture 4).

Debugging your code. The interactive (`pyspark`) (in "local" mode by default) remains the easiest way to debug, exactly like before. I advise to write the code and test it on **a small fraction of the input data**, using `pyspark` in local mode. Read only the smallest file from the input folder(s), and write the code until it works in `pyspark`. Then, test the same code with `spark-submit` in local and/or distributed mode, while still reading only small data. Finally, if that worked OK, then read in the bigger data and run `spark-submit` in distributed mode, now choosing your maximum number of executors (and memory size per executor), to match the size of your input data. That should do it.

Allocating resources to a distributed run. This is an important part of coding for big data: choosing how many executors, cores per executor, and memory per executor, would minimally be needed. Look at the size of the input data first. Then, look at the default values (`spark-submit --help`) for the memory per executor and driver (1GB). You can increase it up to 8 GB (`--executor-memory`). The max. number of cores per executor is 4, and you should limit the number of executors (`--conf spark.dynamicAllocation.maxExecutors`; never go above 20 except with good reasons).

Caching. The RDD/DataFrame methods `persist()` or `cache()` "persist the contents of the DataFrame across operations after the first time it is computed". Only if your program runs *multiple actions* on the same RDD/DataFrame, cache it, so actions reuse that computation. The size of this cached data must be far lower than the size of the (distributed) memory allocated to your program. You may or may not need this for WEB, or the project.

Input

Take the weekly Web crawls at the HDFS path `/data/doina/WebInsight/` (folders starting in `2020-`). One weekly crawl is 2-3 GB compressed. Take only the first (2020-07-13) and the last (2020-09-14).

A Web page is a .json object with many features: the URL, IP address, text size, and many others. The URL is a unique identifier. Check a page with:

```
hdfs dfs -text /data/doina/WebInsight/2020-07-13/1M.2020-07-13-aa.gz | head -1
```

Find the text size of a page (in bytes) in the data:

```
{ "fetch": {..., "textSize": 406, ...} ...}
```

Output

Compute the text difference. Save the entire results to disk, *sorted* from lowest (the biggest decrease) to highest (the biggest increase) in page size. Make it very clear in the code **which path** you saved to.

Coalesce the partitions before saving (1-10 partitions will suffice here), so you get few files that are reasonably large (instead of many tiny files).

The files should be `.json` uncompressed, with only *two fields* (URL and difference), such as:

```
{"url":"https://overheardintheoffice.com/pages/contributors", "sizediff":-14689}
{"url":"https://everything-everywhere.com/best-travel-blogs/", "sizediff":17167}
```

grade points: .25

(**deadline**: Thu Dec 16, 23:59)

Performance analysis (benchmark) on the cluster (challenge identifier: BENCH)

This is a meta-challenge. It has less to do with programming, and more to do with understanding how the Spark framework works when you execute a program distributedly.

First, choose one of your existing Spark programs. For example, take HASHTAGS, or the `tweet selection` code from Lecture 4. If it's HASHTAGS, you have only read small data so far, so modify the program to save all the results (all the hashtag counts, for example) to the HDFS, instead of printing anything.

Then, measure runtimes. Design a benchmark for the runtime of your program. You have 3 variables of interest: (1) how much **data** you put in (in MB, or number of files if they're roughly balanced), (2) how much **compute** you use (the number of executors), and (3) the **runtime** of the program. Your benchmark should fix either (1) or (2), vary the other, and measure (3).

To fix the maximum number of executors (see Lecture 4) set `spark.dynamicAllocation.maxExecutors` to a fixed number (say, 2, 5, or 10). To measure the runtime, you can use `time spark-submit ...` as usual, or get it from the dashboard, `http://ctit048.ewi.utwente.nl:8088/cluster`. The advantage of the dashboard is that you could also get the runtime *per execution stage* in your program! (but this is optional).

You can decide how many measurements to make for your plot (the bare minimum expected is 5). You can decide also which data size(s) or number of executors to consider, in which increments, and where to start and stop the increments. You must definitely reach big data sizes, so the multi-GB domain. Note that your choices have to make sense: think what data sizes match the number of executors chosen.

The resulting line or curve may tell you what speed-up you can expect with a distributed execution. You should see one or more distinct line slopes, and (if you measure with a fine increment) also some bumps in the lines.

Output

The output should be a **graphical plot showing two variables** out of the three possible:

- either the **size of the input data** that your program reads, specifying what units it is measured in (MB/GB compressed, or number of files of some size),
- or the **number of executors**, or any other clear way of quantifying compute resources,
- and definitely the **runtime of the program**.

Mark the measurement points clearly. Fit a line through the points if possible.

To submit

Submit both your code and your plot the same way as for the other challenges. The code should still contain all the different data paths that you read from (some or all can be commented out). Submit the plot as a separate (second) file with the same file name as the code file, but different extension (.pdf, .png, or .jpg). Use only typical image formats. The plot doesn't have to be very polished, but it should be labelled and readable.

(Note: I will also see your runs on the dashboard.)