CSC8101 Assignment: Building a search-term to document index for Wikipedia using Spark.

This document describes your first piece of CSC8101 coursework, using the Spark data analytics platform.

The Task

Given a large subset of the articles which make up the English language version of Wikipedia, you are required to produce a search-term index. This index is essentially a long list of all the terms (words) which appear in any of the provided articles, where each word "points" to a list of the documents in which that word is most important. For example, see below:

> Aardvark : 14, 42057, 78, ...
> Abacus : 4718, 10, 3, 915, ...
> ...
> Zygote : 104, 98103, 63, ...

Pre-requisites

You should have completed the "Managing your AWS VM" introduction before you attempt this assignment. In particular - you should be familiar with how to use `ssh` to access your virtual machine, and `scp` to send and retrieve files to/from it.

You should also be familiar with the process of running spark jobs in either Python or Java. Instructions on how to execute both Python and Java based spark jobs on your OpenStack VMs are below:

Java

A base Java project has been provided for you here.

You should first use `scp` to copy your entire project folder over to your home folder on your vm. Your project folder is the folder which contains the file **pom.xml**.

The `scp` command should look similar to the below:

```
scp -i <path-to>/csc8101-2015-16-student.pem -rp <local-project-
directory> ubuntu@<myDNS-name>:<remote-file-path>
```

Once you have done this, you should `ssh` into your VM, move into the project folder you just copied over and execute the following command:

```
mvn clean package
```

This should produce a large amount of output. If you see a success message at the end then your project has been compiled and packaged into a `.jar` file without issue. Your project folder should now contain a folder called `/target/`. Move into it and execute the following final command, replacing `WordCount` with the name of your new class:

```
spark-submit --class uk.ac.ncl.csc8101.Wordcount csc8101-spark-tfidf-0.0.1.jar --
master local[2] --driver-memory 8g ~/Data/wikipedia ./output
```

This will execute your Spark job. If correct, after some time you will see that another folder has been created inside the `/target/` folder, named `/output/`. This will contain text files with names like `part-00000, part-0001, ...`. These text files are the serialized output of your progam. Your search-term index.

Python

The process for executing spark jobs written in Python is similar, though without the step of compiling. You can use the following command:

```
spark-submit --master local[2] wordcount.py
```

Note: You may find it easier to hard code things like file paths into your python code, rather than accepting command line arguments as the java version does.

The Data

The data you are working with has already been placed on your VM for you. It is stored in an amazon S3 bucket. You can copy it to the directory **~/Data/wikipedia** with the following command:

```
wget -P ~/Data https://s3-eu-west-1.amazonaws.com/csc8101-spark-assignment-
data/wikipedia
```

The original wikipedia data takes the form of a very large xml file. This has been pre-sanitized using the following steps:

1. Extracting article text from xml
2. Removing special syntax from article bodies
3. Removing most punctuation from article bodies
4. Reducing all text to *lower*-case

Although this has been done for you, we list the steps as working with Big Data often involves significant data pre-processing work such as this. Other possible steps might include removing those words most common in English (a, am, and, as, at ...), often known as **stop words**, or using a dictionary to replace words with their common root (mists, misty, misted => mist).

In **~/Data/wikipedia** each article occupies a single line; for example:

> this is an article about lorem ipsum dolore eu fugiat nulla pariatur excepteur sint occaecat
> this is another article about pseudolatin filler text often used as a placeholder in design or copyrighting

> work
> this is a third article about aardvarks medium sized burrowing nocturnal mammals native to africa

In case you need them for testing purposes there are other datasets available:

- Wikinews Articles (~30MB plaintext)
- Small Wikipedia dump (~200MB plaintext)
- Medium Wikipedia dump (~800MB plaintext)

Assessing the importance of a word to a document in a given corpus using TFIDF

In order to create a search-term index, which lists those documents in a corpus to which a given term is most important, you must first have some way of measuring that importance.

In this assignment you should use a well known measure known as *term frequency - inverse document frequency* (**TFIDF**). The TFIDF value for a term *a* in a document *b* increases with the number of times that *a* appears in *b*, but is negatively weighted by the frequency with which the term appears accross the whole corpus of documents. This helps to offset the importance of common words.

The formula for calculating TFIDF is as follows:

$$TFIDF(t, D) = TF(t, d) \cdot IDF(t, D)$$

Where D is a document corpus, t is a term and d a document. The function $TF(t, d)$ is simply the frequency of a term t in a given document d. The function $IDF(t, D)$ is given below:

$$IDF(t, D) = log \frac{|D| + 1}{DF(t, D) + 1}$$

The $DF(t, D)$ function returns the number of documents in the corpus D which contain the term t.

Assignment Steps

The steps we expect you to undertake in order to complete this coursework. Note that there exist some built in functions to calculate TFIDF in the `spark.mllib` package, as seen here. You **must not** use these built in functions, including `org.apache.spark.mllib.feature.HashingTF` and `org.apache.spark.mllib.feature.IDF`.

Generally, when undertaking this assignment, you should consider how many passes your code is making over the data. A core principle of big data is that you should attempt to achieve something in as few passes over the data as possible.

Also note that Spark provides many built in methods for counting, grouping and so on, which encapsulate common uses of the more fundamental operations (map, reduce, fold etc...). Whilst we do not mind if you use these build-in methods, this should not be at the expense of program performance. Make sure that you understand what each built-in method is actually doing before you choose it. If you are unsure: ask a demonstrator, or perhaps try having a look at the source code for the methods in question.

Step 1: Read in documents (10%)

Efficiently read the wikipedia articles from the provided file, producing an RDD of arrays of strings. Each array of strings is an element of the the RDD, and represents a document. Each string in an array represents a word.

Step 2: Calculate TF scores for each document (15%)

Map over the documents RDD to produce another RDD, each element of which contains all of the distinct terms appearing in the corresponding document, associated to the frequency with which they appear.

Hint: Remember from step 1 that entire collections can be individual elements of an RDD.

Note: The identity of a document can be thought of as its index in the documents RDD. The transformation described above should produce a new RDD with the same size and order, therefore preserving document identity.

Step 3: Count the number of documents each term appears in (15%)

Using the RDD that was created in the previous step, you should count the number of documents in D which include a given term t at least once.

Hint: Logically, this operation may require both map and reduce operations. Make sure to familiarise yourselves with working with Key/Value Pairs in spark (*Learning Spark* Chapter 4).

Hint: You will need access to this Map of document counts => terms in the next step. Remember however that RDDs are distributed. You must use `collect` or `collectAsMap` to bring the whole RDD together in the *driver* program.

Step 4: Calculate TFIDF for each term (30%)

You should once again map over the RDD produced in Step 2. Remember that each element in this RDD represents a document, and contains a collection of the disinct terms in the document, along with their associated frequencies. Calculate the TFIDF score for each of these distinct terms, according to the formula above.

Hint: You should end up with an RDD of Maps, whose strings are the distinct terms in a document, and whose values are those terms' TFIDF score.

Note: You should have calculated all the information you need to compute TFIDF in the previous 3 steps.

Step 5: Calculate a search term index (30%)

Using the RDD produced in the previous step, you should produce an RDD of pairs (or Tuple2's). Each pair should contain a string (the term) and a list of document ids, sorted into descending order by the TFIDF score which that term held for each document in the list. Finally, output the resulting datastructure as a text file.

Note: Remember that document ids correspond to element indices in the document RDD (as in step 2).

Hint: This task may be mostly achieved using a map and reduceByKey operation. This is broadly similar to how you may have performed Step 3, unless you used a built-in *count* function, in which case this is broadly similar to how that built-in function is implemented. Spark is open-source software ...

Hint: You may apply a sorting operation to each list of documents using a final map operation.

Deliverables

If you have written your assignment using the provided Java project stub, then you should submit the contents of your `src` directory, zipped, to ness. This directory is **within** your main project directory, alongside the **pom.xml** file.

If you have written your assignment in python, then you should submit ensure that your code is within a single **tfidf.py** file, extracting relevant parts from your Jupyter notebooks if you have used those during development. This tfidf.py file should then be zipped and submitted.

Regardless of the language in which you wrote your coursework, you should submit part of the first output file. Use the following command to create a text file for submission to ness:

```
head -n 100 output/part-r-0000 > result.txt
```