Homework 3

CIS 545: Big Data Analytics

Spring 2018

Homework 3: The Cloud, Matrices, and Arrays

Due March 19, 2018 by 10pm

For this assignment, we will focus on *array* and *matrix* data. An array is a multidimensional set of values of the same type, and we'll often encode matrices within arrays. Arrays are going to be incredibly useful for multidimensional data, for images, for matrices, for representing documents, and for machine learning training data. (We'll see the last aspect in a future assignment.)

As part of this assignment, you'll also build upon HW2 and execute a computation on Amazon Elastic MapReduce or the Google Cloud.

Step 1. Set up Spark on the Cloud

1.1 Amazon Cloud (EMR)

If you'd like to use the Amazon cloud, please follow the instructions to set up your own Amazon AWS Educate account and server.

1.2 Google Cloud (currently requires a cis.upenn.edu account)

The first part of your homework will involve connecting to Jupyter on Google DataProc on the Cloud. Google DataProc includes Apache Spark. *This option currently only works for .cis.upenn.edu accounts.*

1. First, sign up for a $50 Google Cloud credit: Student Coupon Retrieval Link
   - You will be asked for a name and email address, which needs to end with **cis.upenn.edu** or **upenn.edu**. A confirmation email will be sent to you with a coupon code.
   - The coupon will be valid through 1/10/2019. You can only request ONE code.
2. Next, visit https://console.cloud.google.com/education to redeem your credits. *Note that this will be applied to the Google account to which you are logged in. Be sure you're logged in using the account you want to use for the Google Cloud.*
3. Now we need to do some basic setup:
   a. In the "Manage Resources" page, create a new project. Call it something unique such as **cis545-*{your-initials}*-1**. If you get a notification that the name is taken, update the -1 to a higher number. We'll call this the *project-id*.
   b. In the "Enable Billing" page, set your Google Education credits as the billing source.
   c. Enable the APIs and select your project.
   d. Download the and install the Google Cloud SDK for your machine.
   As instructed, run `./google-cloud-sdk/install.sh`

Run `./google-cloud-sdk/bin/gcloud init`

4. At this stage you should be "ready to go" with respect to setting up a Google cloud account. From here you need to do two main things:
   a. To store your data, create a bucket with a unique name, which we'll call *bucket-name*.
   b. To actually run a computation, create a cluster (see Step 5).
5. To create a cluster, you can run following commands using your Mac OS Terminal or Windows Command Prompt, and the Google Cloud SDK

   a.
```
gcloud dataproc clusters create cluster-name \
   --project project-id \
   --bucket bucket-name \
   --initialization-actions \
      gs://dataproc-initialization-actions/jupyter/jupyter.sh
```

6. Then, connect your browser to the Jupyter notebook running on your cluster's master node.
   a. Create an SSH tunnel to your cluster's master node from port 10000 on your localhost machine by:
```
gcloud compute ssh "cluster-name-m" \
 --project project-id \
 --zone=cluster-zone \
 -- -D 10000 -N
```
   (don't forget to add the "-m" at the end of the cluster name!)
   b. Configure your browser to use the proxy when connecting to your cluster:
```
<browser executable path> \
   "http://<cluster-name>-m:8123" \
   --proxy-server="socks5://localhost:10000" \
   --host-resolver-rules="MAP * 0.0.0.0 , EXCLUDE localhost" \    --user-data-dir=/tmp/
```
   Note that:
   - For mac, *<browser executable path>* is `/Applications/Google\ Chrome.app/Contents/MacOS/Google\ Chrome`
   - For Linux, it is `/usr/bin/google-chrome`
   - For Windows, it is `C:\Program Files (x86)\Google\Chrome\Application\chrome.exe`

   So far, you can use the Jupyter notebook on Google cloud just as you do in Docker.
7. **Note that you should remember that "the meter is ticking" when you are running your cluster (especially), downloading / uploading data, and keeping data in the bucket.** When you take a long break, you should delete your cluster. When you are done with this assignment you should delete your bucket.

Step 2. Use Spark on the (Amazon/Google) Cloud

Let's start by downloading the data files for this assignment, both into your Docker instance and into JupyterHub.

Step 2.1. Download the Repo into Your Local Docker Container

Go to your operating system's Terminal or Command Prompt. Go to **~/jupyter**. Run the command:

```
git clone https://bitbucket.org/pennbigdataanalytics/hw3.git
```

You'll be using this for later parts of the homework. And shortly be replicating this to the Google Cloud.

Step 2.2. Download the Repo into Your Cloud Jupyter/JupterHub Server

Step 2.2.1 Amazon Setup

If you're using the Amazon Cloud setup, go to Jupyter on Amazon in your browser (if you set up your own server then it is as in the end of this document), open a new Terminal there (click the "New" dropdown in the top right and select "Terminal") and run the following to clone the HW3 repository:

```
git clone https://bitbucket.org/pennbigdataanalytics/hw3.git
```

Now in JupyterHub, go into **hw3** and create a new Python 3 Notebook called **Spark.** Type in the following:

```
from emrspark import *
from pyspark.sql.types import *
import pyspark.sql.functions as F

conf.set("fs.s3n.awsAccessKeyId","**")
conf.set("fs.s3n.awsSecretAccessKey","**")

spark = SparkSession.builder.config(conf=conf).appName('Graph HW3').getOrCreate()
```

Where, for the AWS access key and secret key above, **replace the "**"s with the values from your AWS setup**. *(Please don't share your credentials in a public place, or someone will mine bitcoins on your dollar!)*

Step 2.2.2 Google Cloud Setup

Now open your connection to Jupyter running on Google Cloud - enter the URL into your browser:

```
http://<cluster-name>-m:8123
```

where *cluster-name* is set as above, and don't forget the -m. You should get a "blank" copy of Jupyter. Use the Jupyter Upload button to upload **Spark.ipynb** from where you cloned **hw3** above.

Then, create a Spark notebook (naturally) called **Spark**. The spark session will already be initialized. Type in the following:

```
from pyspark.sql.types import *
import pyspark.sql.functions as F
```

Step 2.3. Transitive Closure over the Stack Overflow Network

In the previous assignment, you had built several primitives, including **breadth-first search**, to compute over the Stack Overflow social graph. Some of your computations were limited by the amount of computational resources available. We will now try to explore the broader graph.

Step 2.3.1. Loading the Data

For Amazon Cloud Users

The three source files (`sx-stackoverflow-a2q.txt, sx-stackoverflow-c2q.txt, sx-stackoverflow-c2a.txt`) from Homework 2 are on Amazon S3.

As with Homework 2 Step 2.3, you will combine them into a unified **graph_sdf**. You can use the syntax:

```
my_sdf = spark.read.format("com.databricks.spark.csv").option("delimiter", ' ') \
  .load("s3n://upenn-bigdataanalytics/data/myfile.txt")
```

to load the space-delimited file *myfile.txt* into a Spark DataFrame with columns named _c0, _c1, etc. (Downloading these files may take a while, so don't worry). See if you can further add a function call (e.g., to **selectExpr()**) to rename the first two columns to **from_node** and **to_node**, to convert these to integers, and to drop the third column. Union everything together (and remove duplicates) to create **graph_sdf** with all data. Note that **.union()** in Spark is an alias for **.unionAll()**.

Hints: For early testing of your solutions to Step 2, you should probably just load one of these files (eg sx-stackoverflow-a2q.txt) into graph_sdf to validate your solutions. Later go back, add the contents of the other files to graph_sdf, and re-run your solutions. Make sure you rerun your code with the full graph_sdf before submitting.

For Google Cloud Users

You should upload the three source files (`sx-stackoverflow-a2q.txt, sx-stackoverflow-c2q.txt, sx-stackoverflow-c2a.txt`) from Homework 2 into your bucket in the cloud.

You can use either following ways:

1. From the browser, go to your project home in Google Cloud, click the Cloud Storage under Resources, click your bucket, enter the notebooks directory and upload your data by the browser.
2. Use gsutil command of google cloud sdk, run command line to copy your file to your bucket,

`gsutil cp *.txt gs://my-bucket/notebooks/`

As with Homework 2 Step 2.3, you will combine them into a unified **graph_sdf**.

Then, we will try to do the parsing into fields inline with the read. You can use a command such as:

```
my_sdf = spark.read.format("com.databricks.spark.csv").option("delimiter", ' ') \
  .load("gs://my-bucket/notebooks/myfile.txt")
```

to load the space-delimited file *myfile.txt* into a Spark DataFrame with columns named _c0, _c1, etc. (Downloading these files may take a while, so don't worry). See if you can further add a function call (e.g., to **selectExpr()**) to rename the first two columns to **from_node** and **to_node**, to convert these to integers, and to drop the third column. Union everything together (and remove duplicates) to create **graph_sdf** with all data. Note that **.union()** in Spark is an alias for **.unionAll()**.

Hints: For early testing of your solutions to Step 2, you should probably just load one of these files (eg sx-stackoverflow-a2q.txt) into graph_sdf to validate your solutions. Later go back, add the contents of the other files to graph_sdf, and re-run your solutions. Make sure you rerun your code with the full graph_sdf before submitting.

Step 2.3.2. Transitive Closure of the Graph

Now we would like to do the following: given a set of nodes, compute the set of **all nodes reachable** from these nodes. This can be obtained via a type of **transitive closure computation**.

Define a function **transitive_closure(graph_sdf, origins_sdf, depth)** that returns a Spark DataFrame.
The result should be the set of all nodes from the input **graph_sdf** that are reachable via graph edges from the set of **origins_sdf**, in at most **depth** iterations (hops). Both **origins_sdf** and the returned result should be **DataFrames** with a single attribute called **node**.

You should treat the edges in the **graph_sdf** as *directed* edges! You should iterate until you have either hit the maximum depth or the set of newly discovered (frontier) nodes is empty.

Hints: this resembles your BFS algorithm from HW2, but you should take advantage of the opportunities to optimize. Both the graph and the various node sets can easily have duplicates; you should make heavy use of duplicate removal (and repartition and cache) since you are only computing the set of reachable nodes. Also, to quickly check whether a DataFrame is empty, you can use something similar to the following:

```
def sdf_is_empty(sdf):
    try:
        sdf.take(1)
        return False
    except:
        return True
```

Data Check for Step 2.3. Let's now create some Cells:

1. Compute a Spark DataFrame called **nodes_sdf** with all node IDs in the graph strictly less than 8.
2. Compute a Spark DataFrame called **reachable_sdf** with the results of **transitive_closure(graph_sdf, nodes_sdf, 3)**.
3. Create and execute the Markdown Cell titled "## Step 2.3 Results" and hit **Shift-Enter**
4. Add and run two code Cells that call **count()** and then **show()**, respectively, on **reachable_sdf**.
5. Go to the File menu in Jupyter, choose **Download as** > **Notebook (ipynb)**. Download as **Spark.ipynb** and put this into your **~/Jupyter/hw3** directory.
6. If you are using the shared server, please go to the JupyterHub Control Panel and shut down your JupyterHub server, as per Step 1.3.

If you are using the shared JupyterHub server, please be courteous to your fellow students. The JupyterHub server is a shared resource. Whenever you are done with this part of the assignment, or wish to take a break, please shut down your personal Jupyter Notebook instance, as per the instructions.

Step 3. PageRank with Matrices

[Note: if you can't read the equations below due to a bug in GDoc, click on the link to the lower right that says: Document doesn't display correctly? See the original Google Doc. Alternatively, if you right-click on the broken image and choose "Open Image in New Tab" you'll see it.]

For this part of the assignment, we can go back to our Jupyter instance on Docker, since we'll be doing matrix operations in Numpy/Scipy. In your **local Jupyter instance** (localhost:8888) create a new Python 3 notebook called **PageRank. [You can also do this on the Google Cloud; just be careful not to spend too many of your credits since we'll likely need them for the future.]**

Recall that PageRank can be modeled using matrix operations as follows. Let M be a *weight transfer matrix* in which:

$$M[i,j] = \frac{1}{n_j}$$ if page *i* is pointed to by page *j* and page *j* has n_j outgoing links

$$= 0$$ otherwise

And define a *dampening factor* $\alpha = 0.85$ and a corresponding $\beta = 1 - \alpha$. Initialize the PageRank vector

$$PR^{(0)} = |1,1,1,\ldots|^T$$

(i.e., a matrix with m rows by 1 column, filled with ones). Then we can compute the PageRank PR for each iteration as:

$$PR^{(i)} = \alpha \cdot M \cdot PR^{(i-1)} + \beta \cdot |1,1,1,\ldots|^T$$

Step 3.1. Download a Web Graph

Recall from Homework 2 that you were given a Jupyter Notebook called **Retrieve.ipynb**, which downloaded a series of zipped text files from Stanford SNAP and decompressed them. Adapt that code to download:
https://snap.stanford.edu/data/web-NotreDame.txt.gz

which is a reasonably sized Web crawl done by Notre Dame University, and to extract it into web-NotreDame.txt. Run the program to acquire your Web graph.

Step 3.2. Load the Notre Dame Web Graph into a Matrix

Next, write Python code to take the data from **web-NotreDame.txt**, read and parse the rows in a Pandas DataFrame (not a Spark DataFrame!). Restrict the node IDs to values less than 10,000. Create a **weight transfer matrix M** corresponding to the Web graph, with edges whose weights are scaled as per the PageRank definition of a weight transfer matrix. This will form an input into your PageRank algorithm. Note that the dataset already includes node IDs that go from 0 .. m, so you can directly use the node IDs as indices in your matrix. You should not use for loops, and instead, use the DataFrame and array functions that Pandas and NumPy provide as they are much more efficient.

*Hints: If you use **read_csv**, you may need to look at the **sep** and **skiprows** options. Also take a look at the raw data and make sure you know how many rows don't contain data, and how the items are separated. When building M, you may need to build some "auxiliary" data structures to speed up performance, e.g., to quickly look up weights associated with node edges. Note that lookup in an array is typically faster than lookup in a DataFrame. Finally, you might want to use the **apply** function for Pandas DataFrames or Numpy Matrices as they are orders of magnitude faster than trying to iterate through every row. However, this is not a requirement -- just make sure you aren't using for loops!*

Step 3.3. Compute Matrix-Based PageRank

Implement a function **pagerank(M, alpha, num_iter)** that, when given a square *m* x *m* transition matrix *M* from Step 3.2, initializes the PageRank vector to *m* 1's, sets α= **alpha**, sets β appropriately givenα, and iterates **num_iter** times. Return an *m*-element **vector** that consists of the final PageRank scores.

Output for Step 3. Let's now create some Cells:
1. Create a Cell in Markdown with contents "## Step 3.3" and hit **Shift-Enter**
2. Create a Cell with the output of the 2D array slice **M[10:30,10:30]**
3. Call **pagerank(M, 0.85, 15)** with the data from your Notre Dame crawl.
4. Output a DataFrame with the schema **(id, pagerank)** containing the **original IDs and PageRanks of the 10 nodes with highest PageRank**, in descending order.

Step 4. Images

For the next step, you will do some simple image manipulation. Create a new Jupyter Notebook called **Images**. Your task is to write a function **convert_to_grayscale(image, crop_left, crop_top, crop_right, crop_bottom, contrast_scale)** that, given a color image array, returns a new copy of the image that (1) has been converted to grayscale, (2) increases the *contrast* of the image as specified by the **contrast_scale** below, (3) crops the image by the specified **crop_left**, **crop_top**, **crop_right**, and **crop_bottom** margins (in pixels). Perform the steps in the following order:

1. **Cropping:** you should crop the image appropriately according to the **crop_left, crop_top, crop_right,** and **crop_bottom** parameters (which you can safely assume will be non-negative).

2. **Converting to grayscale**: you should *average* (compute the mean) of the (R, G, B) values associated with each pixel. For example, an RGB pixel (100, 108, 104) would become the single grayscale value 104. This has the effect of reducing the dimension of the image array from 3D (color image) to 2D (grayscale image).
3. **Increasing contrast**: you should first compute the **median** grayscale pixel value across the entire image. For every pixel with value **below** the median, scale its grayscale value down by the percentage specified in **contrast_scale** (which should be a real value from 0 to 1). For every pixel with value **above** the median, scale its grayscale value up by the percentage specified in **contrast_scale**. A **contrast_scale** of 0 corresponds to not changing the grayscale pixel values at all. Do not let grayscale values fall outside of the range [0, 255]; you may need to threshold them.

To process images, you will use the **ndimage** package in **scipy**. We will use **pyplot** to render the images in Jupyter. You can load and plot an image like this:

```
from scipy import ndimage
import matplotlib.pyplot as plt
import numpy as np
real_pandas = ndimage.imread('panda-mania-12.jpg')
plt.imshow(real_pandas)
```

where real_pandas will be a 3D array. To plot a grayscale image you will need to call:

```
plt.imshow(gray_img, cmap=plt.cm.gray)
```

Output for Step 4. Let's now create some Cells:
1. Create a Cell in Markdown with contents "## Step 4" and hit **Shift-Enter**
2. Load into an array **data** the file **'data-behind-everything.jpg'**.[1]
3. Call **convert_to_grayscale(data, 10, 50, 10, 50, 0.3)** and return the results in **gray_data**.
4. Output the contents of **gray_data** in the next cell.
5. Call **plt.imshow** and render the image in grayscale in the next cell.

Step 5. Advanced: Document Vectors

Our task here is to support the computation of **similarity between pairs of documents** -- where we will treat a keyword search as a (very short) document. This is the basis of both *keyword search*, and also *document clustering*. (We'll see a lot more about clustering in a few weeks.)

Step 5.1. Brief Overview: Document Vectors

The basic idea is to first model each document as a **(multi-)set of words**, ignoring any information about sentence structure. Each word will receive a **weight** and each document will be represented as a **vector** of weights (one for each word). We will compare document vectors to measure how similar they are.

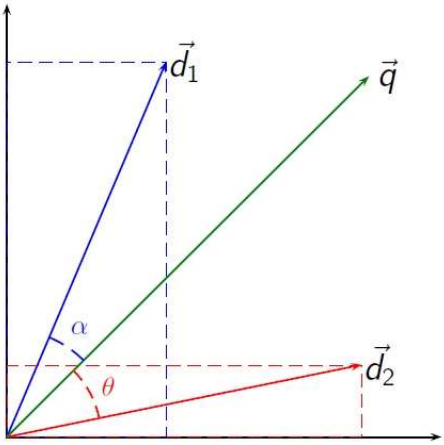
Step 5.1.1. Basic Term Frequencies

Intuitively, we can start by assuming a document is more "about" a particular subject (word) if it mentions the word more times. This is the **term frequency (tf)**.

Example. *Suppose we have two documents:*
- *doc1 = "the quick brown fox jumped over the lazy dog"*
- *doc2 = "the fox doggedly jumped up the hill"*

*Now for document d_j, we **parse** the document into words, and create a vector with one element for each potential word w_i. We can put a **term frequency $tf_{i,j}$** in $w_{i,j}$, capturing how many times word w_i appears in document d_j. We can do this as follows:*

1. *Assign an index position to each word, potentially as we encounter the word: {the: 0, quick: 1: brown: 2, fox: 3, jumped: 4, over: 5, lazy: 6, dog: 7, doggedly: 8, up: 9, hill: 10}.*
2. *Create a **vector** for each document, with a count for the number of occurrences of each word:*
doc1: [2, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0]



doc2: [2, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1]

Of course, rather than having separate vectors for each document, we can have a single 2-dimensional array with one row per document and one column per word. (Over time we will find that the array is **sparse** -- most documents have 0's in most word positions. We won't exploit that here, but in a real search system we would.)

Now we make a simplifying assumption: the occurrence of each word in the document is **independent** of any other words. Then if there are *t* terms, we can plot each document's vector in a *t*-dimensional space. Similarly to the creation of document vectors, we can convert a keyword query into a *t*-dimensional **query vector** by creating a vector with the number of occurrences of each word. Finally, we can compute a **distance** between document and query vectors by looking at how closely their vectors match -- by computing the **cosine of the angle between the vectors**. See the figure to the right (from here) for an example using two dimensions. In this example, q is our query vector and d1 and d2 are the two document vectors. The cosine similarity between one of the document vectors and q would tell us the cosine of the angle between them, which we interpret as how similar the vectors are. The cosine would have value 1 if the vectors exactly match; and it would have value 0 if the vectors were orthogonal.

The cosine similarity between vector d_j and q is equal to their dot product divided by the product of their Euclidean norms:

$$\frac{d_j \cdot q}{\|d_j\| \, \|q\|}$$

***Example**. Given the previous document vectors, suppose we are given the query "quick fox."*

The query vector would be: [0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0] and the cosine similarity for the documents would thus be:

doc	score
0	0.426401
1	0.235702

Step 5.1.2. Stopwords and Stemming

The above formulation captures the gist of the techniques. However, we often decide that some words are of so little value that we should ignore them: these are **stopwords**, and we just ignore any word that appears in our stopword list. (An example above is the word "the.") Stopwords need to be ignored in both the query and each document. Stopwords are typically provided to us by experts in linguistics.

A second issue is that many languages, including English, have many different conjugations and forms of word: e.g., "run -> running -> ran." We would like to normalize each word to its core "stem." Stemming for English is very complex because of the many different word variations. We thus need to use specialized algorithms -- like the heuristic **Porter's**

Stemming Algorithm -- or a more linguistically based technique called lemmatization. Applying stemming or lemmatization to words in the document and the query will "regularize" them. In the example above, "doggedly" would be stemmed back to "dog" (which might or might not be helpful in this case!).

All told, an improved version of the technique from Step 5.1.1 parses each word (from the document or query), drops every stop word, and creates a vector of the stemmed / lemmatized words.

Step 5.1.3. Inverse Document Frequency

Still the techniques we've described so far have a weakness: they count all (non-stopword) words as being of equal value. We would like to give **rare words** more importance than ones found in all documents. To measure a word's importance, we will develop a metric called **inverse document frequency (idf)**. In its simplest form, a word's idf is a ratio between the total number of documents, and how many documents include a word. (Note that the idf is independent of a given document -- it is a measure of the word's popularity across the full set of documents, sometimes called a **corpus** and represented by the letter **D**.) Typically, we don't directly use the ratio, however -- instead we use its base-10 logarithm. Thus, we define:

$$idf_{i,D} = log(\frac{|D|}{|d_j \in D s.t. w_{i,j} > 0|})$$

Example. *For illustrative simplicity, let's temporarily ignore stemming and stopwords. Going back to the example in Step 5.1.1, we can compute an IDF vector for our words and the corpus:*

$$IDF = [0, 0.3, 0.3, 0, 0, 0.3, 0.3, 0.3, 0.3, 0.3, 0.3]$$

Note that 0.3 is the log of 2 / 1 and 0 is the log of 2/2.

Now, in our above cosine similarity measure, we will use the tf * idf term (not just the tf term) for each word w_i, in both the document and query vectors. The cosine similarity between these vectors will be our measure of relevance or similarity.

Example. *Again without stemming and stopwords, we can match the query "quick fox" and the documents. Here the word "quick" has 0.3 IDF, and "fox" has 0 IDF (since it appears in both documents). We will get:*

doc	*score*
0	*0.447214*
1	*0.000000*

Step 5.2. Initializing Parser Code

Now you've seen the basics. Let's get started by setting up some tools you'll find to be helpful in eliminating "grunge" work. The **Natural Language Toolkit (nltk)** does a few useful things for you, including parsing sentences with punctuation, as well as stemming.

For this part of the homework, you'll want to start with the **Advanced-Starter** Jupyter notebook we've given you. Open it and rename it to **Advanced**. Run the first Cell and follow the instructions in the code comments: download the **punkt** package, which will parse text documents and respect punctuation.

Now run the remaining Cells until you get to the bottom, where you can write your own code. At this point, you should have several available data structures useful for writing your code:

1. A dictionary called **docs** from document name to document content. Docs has some content pulled from Wikipedia.
2. A set, **stopwords**, containing (shockingly!) stopwords.
3. A variable **MAX_WORDS** that is set to the maximum number of words we will consider (after which we'll ignore all subsequent words). We can use this to initialize 2D arrays in numpy.
4. A variable **stemmer**, pointing to an object which you will be able to use to stem words.

Step 5.3. Build Document Vectors

As a first step, you will need to simultaneously build a **lexicon** (a list of known words, which you will want to map to document vector indices) and a list of document vectors. You may also want to build an **inverse lexicon** that maps from position back to word.

Create a 2D array representing the document vectors. This should have a number of rows equal to the size of the corpus, and a number of columns equal to **MAX_WORDS**.

Iterate through the various documents in **docs**. Call a function **doc_vector(content, vector, lexicon, inverse_lexicon, stopwords, word_count)** that will calculate the term frequency (tf) for the contents of each document. **Word_count** is the number of unique words seen so far. **Lexicon** and **inverse_lexicon** are also the current state of the respective dictionaries. **Vector** should be a blank (read: zeroed) vector that your function will overwrite to contain the document vector for the given **content**.

Define **doc_vector** as above. You can use **nltk.word_tokenize()** to parse the documents into streams of words. Make sure you account for variations in capitalization etc. Remove any "words" that don't have letters (these are clearly not words! You can call the function **has_letter** to test this) as well as stop words. Call **stemmer.stem()** on a word to get its stem. For each stem, if necessary extend the lexicon (and inverse lexicon), unless you have already seen **MAX_WORDS** (if you have seen this, you need to simply ignore the keyword). Set the term frequency in the appropriate element in the document vector. Return the updated **word_count** after you have added any new terms to the lexicon.

Now compute a single vector **idf** representing, for each word, its idf within the corpus.

Step 5.4. Create Query Vectors

As we described previously, standard search simply takes a keyword query and treats it much as a document. Write a function **create_query_vector(query)** that takes a keyword query string, appropriately calls **doc_vector** to create a query vector, and returns the query vector.

Step 5.5. Produce Ranked Results

Write a function **search(vectors, idf, query, num_results)** that, when given a 2D array of document vectors, a query vector, and a number **num_results**:

1. Creates a Pandas DataFrame with schema (docid, docname, score) for the results of matching the query against each document.
2. Uses Numpy multiplication (* between arrays), dot product (@ or **np.dotproduct()**) and other operations (+, **np.linalg.norm()** for vector norms, etc) to compute a cosine similarity score for every document. Add each document ID and score to the DataFrame.
3. Sorts the DataFrame by descending score.
4. Returns the first **num_results** results.

Output for Step 5. Let's now create some Cells:

1. Create a Cell in Markdown with contents "## Step 5" and hit **Shift-Enter**

2. Call **search()** and **create_query_vector()** to return the top 10 results to "Apple Steve jobs" and output the DataFrame in a Cell.
3. Call **search()** and **create_query_vector()** to return the top 5 results to "Trump Putin" and output the DataFrame in a Cell.
4. Call **search()** and **create_query_vector()** to return the top 5 results to "Google Cloud" and output the DataFrame in a Cell.

6.0 Submitting Homework 3

Please sanity-check that your Jupyter notebooks contain both code and corresponding data. Retrieve from your Docker container the following notebook files and zip them into **hw3.zip**, much as you did for HW0 and HW1. The notebooks should be:

- `Spark.ipynb`
- `PageRank.ipynb`
- `Images.ipynb`
- `Advanced.ipynb` (if applicable)

Next, go to the submission site, and if necessary click on the Google icon and log in using your Google@SEAS or GMail account. At this point the system should know you are in the appropriate course. Select CIS 545 **Homework 3** and upload `hw3.zip` from your Jupyter folder, typically found under `/Users/`*{myid}*.

If you check on the submission site after a few minutes, you should see whether your submission passed validation. You may resubmit as necessary.

---

[1] Image from http://beyondrealtime.blogspot.com/2014/02/the-red-pill-redux.html