Assignment 4
Individual Assignment

MET CS 755 - Cloud Computing
kNN classifier for the 20 newsgroups data set

1 Description

In this assignment you will implement a k-nearest neighbors classifier (kNN classifier) in multiple steps. KNN is an algorithm that can find the top-k nearest objects to a specific query object. In this assignment we want to use kNN to classify text documents. Given a specific search text like "how many goals Vancouver score last year?" the algorithm can search in the document corpus and find the top K similar documents.

We will create out of a text corpus a TF-IDF (Term Frequency - Inverse Document Frequency) Matrix for the top 20k words of the corpus. The TF-IDF matrix will be used to compute similarity distances between the given query text and each of the documents in the corpus.

2 Data Set

You will use the famous 20 newsgroups data set. The 20 newsgroups dataset is an old dataset about 20 different news groups. Newsgroup is an on-line information system similar to discussion forums. People have used the group discussions to discuss about different topics. You can read more about the data set here `http://qwone.com/~jason/20Newsgroups/`

Each document have a unique document ID and a specific newsgroup ID for example:

- **docID** is "20_newsgroups/comp.graphics/37261" then the
- **newsgroupID** for the above example will be "/comp.graphics/"

We have pre-processed the data so that each line of a large text corpus is a single document.

3 Obtaining the Dataset

You can find the data set here:

`https://s3.amazonaws.com/metcs755/20_news_same_line.txt`

And in AWS S3 URL form:

s3://metcs755/20_news_same_line.txt
Each line is a single document in a pseudo XML format.

4 Assignment Tasks

4.1 Task 1 : Generate the Top 20K dictionary (5 points)

Get the top 20,000 words in a local array and sort them based on the frequency of words. The top 20K most frequent words in the corpus is our dictionary and we want to go over each document and check if its words appear in the Top 20K words.

At the end, produce an RDD that includes the docID as key and a numpy array for the position of each words in the top 20K dictionary.

(docID, [dictionaryPos1, dictionaryPos2, dictionaryPos3...])

In the following pyspark code complete the missing implementation parts. The missing parts are marked using "???", 3 question marks.

```
import re
import numpy as np

# First load up all of the 19997 documents in the corpus
corpus = sc.textFile ("s3://metcs755/20_news_same_line.txt")

# Each entry in validLines will be a line from the text file
validLines = corpus.filter(lambda x : 'id' in x)

# Now, we transform it into a set of (docID, text) pairs
keyAndText = validLines.map(
  lambda x : (x[x.index('id="') + 4 : x.index('" url=')],
    x[x.index('">') + 2:]))

# Now, we split the text in each (docID, text) pair into a list of words
# After this step, we have a data set with
# (docID, ["word1", "word2", "word3", ...])
# We use a regular expression here to make
# sure that the program does not break down on some of the documents

regex = re.compile('[^a-zA-Z]')

keyAndListOfWords = keyAndText.map( \
  lambda x : (str(x[0]), regex.sub(' ', x[1]).lower().split()))

# Now get the top 20,000 words...
# first change (docID, ["word1", "word2", "word3", ...])
# to ("word1", 1) ("word2", 1)...
allWords = ???

# Now, count all of the words, giving us ("word1", 1433), ("word2", 3423423),
    etc.
allCounts = allWords.???

# Get the top 20,000 words in a local array in a sorted format based on
    frequency
topWords = allCounts.top (???)

# We'll create a RDD that has a set of (word, dictNum) pairs
# start by creating an RDD that has the number 0 through 20000
# 20000 is the number of words that will be in our dictionary
twentyK = sc.parallelize(range(20000))

# Now, we transform (0), (1), (2), ... to ("mostcommonword", 1)
```

```
# ("nextmostcommon", 2), ...
# the number will be the spot in the dictionary used to tell us
# where the word is located
dictionary = twentyK.map (lambda x : (topWords[x][0], x))

# Next step, we get a RDD that has, for each
# (docID, ["word1", "word2", "word3", ...]),
# ("word1", docID), ("word2", docId), ...
allWords = keyAndListOfWords.flatMap(lambda x: ((j, x[0]) for j in x[1]))

# Now join and link them, to get a set of ("word1", (dictionaryPos, docID)) pairs
allDictionaryWords = ???.join (???)

# Now, we drop the actual word itself to get a set of (docID, dictionaryPos) pairs
justDocAndPos = allDictionaryWords.map (lambda x: (x[1][1], x[1][0]))

# Now get a set of (docID, [dictionaryPos1, dictionaryPos2, dictionaryPos3 ...]) pairs
allDictionaryWordsInEachDoc = justDocAndPos.???
```

4.2 Task 2 - Create the TF-IDF Array (10 Points)

After having the top 20K words we want to create a large array that its columns are the word list in order and rows are documents.

The first step is to create the term frequency $TF(x, w)$ vector for each document

$$TF(x, w) = \frac{x[w]}{\sum_{w'} x[w']}$$

Inverse Document Frequency is:

$$IDF(w) = log\left(\frac{size\ of\ corpuse}{\sum_{"x\ in\ corpus"}\ 1\ if\ (x[w]\geq 1),0\ otherwise}\right)$$

$$TF - IDF(w) = TF(x, w) \times IDF(w)$$

For more description about TF-IDF see the Wikipedia page:
`https://en.wikipedia.org/wiki/Tf-idf`

Here is an example code that you can complete the missing parts - marked with **"???"**.

```
# Now, extract the newsgrouID, so that on input we have a set of
# (docID, [dictionaryPos1, dictionaryPos2, dictionaryPos3...]) pairs,
# but on output we
# have a set of ((docID, newsgroupID) [dictionaryPos1, dictionaryPos2, dictionaryPos3...]) pairs
# The newsgroupID is the name of the newsgroup extracted from
```

```
# the docID... for example
# if the docID is "20_newsgroups/comp.graphics/37261" then
# the newsgroupID will be "s/comp.graphics/"

regex = re.compile('/.*?/')

allDictionaryWordsInEachDocWithNewsgroup = allDictionaryWordsInEachDoc.map (
    lambda x: ((x[0], regex.search(x[0]).group (0)), x[1]))

# The following function gets a list of dictionaryPos values,
# and then creates a TF vector
# corresponding to those values... for example,
# if we get [3, 4, 1, 1, 2] we would in the
# end have [0, 2/5, 1/5, 1/5, 1/5] because 0 appears zero times,
# 1 appears twice, 2 appears once, etc.

def buildArray (listOfIndices):
        returnVal = np.zeros (20000)
        for index in listOfIndices:
                returnVal[index] = returnVal[index] + 1
        mysum = np.sum (returnVal)
        returnVal = np.divide (returnVal, mysum)
        return returnVal

# The following line this gets us a set of
# ((docID, newsgroupID) [dictionaryPos1, dictionaryPos2, dictionaryPos3...])
    pairs
# and converts the dictionary positions to a bag-of-words numpy array...

allDocsAsNumpyArrays = allDictionaryWordsInEachDocWithNewsgroup.map (lambda x:
     (x[0], buildArray (x[1])))

# Now, create a version of allDocsAsNumpyArrays where, in the array,
# every entry is either zero or one.
# A zero means that the word does not occur,
# and a one means that it does.
zeroOrOne = allDocsAsNumpyArrays.map (???)

# Now, add up all of those arrays into a single array, where the
# i^th entry tells us how many
# individual documents the i^th word in the dictionary appeared in
dfArray = zeroOrOne.reduce (lambda x1, x2:("", np.add (x1[1], x2[1])))[1]

# Create an array of 20,000 entries, each entry with the value 19997.0 (number
     of docs)
multiplier = np.full (20000, 19997.0)

# Get the version of dfArray where the i^th entry is the inverse-document
    frequency for the
# i^th word in the corpus
idfArray = np.log (np.divide (???, ???))

# Finally, convert all of the tf vectors in allDocsAsNumpyArrays to tf * idf
    vectors
allDocsAsNumpyArrays = allDocsAsNumpyArrays.map (lambda x: (x[0], np.multiply
    (x[1], idfArray)))
```

4.3 Task 3 - Implement the getPrediction function (5 Points)

Finally, you should implement the function getPrediction (textInput, k). This function will predict the membership of this text string in one of the 20 different newsgroups.

You should use the cosine similarity to calculate the distances.

$$Cos(x_1, x_2) = \sum_{i=1}^{d}(x_1 \times x_2)$$

Here is an example code that you can complete the missing parts - marked with "????".

```
# and finally, we have a function that returns the prediction for the label of
    a string, using a kNN algorithm
def getPrediction (textInput, k):
    # Create an RDD out of the textIput
    myDoc = sc.parallelize (('', textInput))
    # Flat map the text to (word, 1) pair for each word in the doc
    wordsInThatDoc = myDoc.flatMap (lambda x : ((j, 1) for j in regex.sub(' ',
    x).lower().split()))
    #
    # This will give us a set of (word, (dictionaryPos, 1)) pairs
    allDictionaryWordsInThatDoc = dictionary.join (wordsInThatDoc).map (lambda
    x: (x[1][1], x[1][0])).groupByKey ()
    # Get tf array for the input string
    myArray = buildArray (allDictionaryWordsInThatDoc.top (1)[0][1])
    #
    # Get the tf * idf array for the input string
    myArray = np.multiply (???, ???)
    #
    # Get the distance from the input text string to all database documents,
    using cosine similarity
    distances = allDocsAsNumpyArrays.map ( ??? )
    #
    # get the top k distances
    topK = distances.top (k, lambda x : x[1])
    #
    # and transform the top k distances into a set of (newsgroupID, 1) pairs
    newsgroupsRepresented = sc.parallelize (topK).map (lambda x : (x[0], 1))
    #
    # now, for each newsgroupID, get the count of the number of times this
    newsgroup appeared in the top k
    numTimes = newsgroupsRepresented.aggregateByKey (0,
                            lambda x1, x2: x1 + x2, lambda x1, x2: x1 + x2)
     # Return the top 1 of them.
     return numTimes.top (1, lambda x: x[1])
```

The above function on the given 20 newsgroup data set should return :

```
getPrediction ("God Jesus Allah", 30)
>>> [('/alt.atheism/', 12)]

getPrediction ("How many goals Vancouver score last year?", 30)
>>> [('/rec.sport.hockey/', 23)]
```

4.4 Task 4 - Using Linear Regression as Classifier (For Advanced Students)

The above kNN classifier implementation works fine but the 20K matrix multiplication is an expensive operation, and in the case that I have much more documents like changing the data set to Wikipedia Dataset then I will have a very expensive matrix multiplication.

Go ahead and reduce the dimensions from 20K down to 1K dimension and use the linear regression as classifier. The classic approach for reducing the dimensions is PCA but you can use a simple sampling approach and sample from a Normal (0, 1) distribution. Then you need to train a linear regression model to use it as classifier.

Drawback: simple regression only allows two classes, like it can predict for a given text query if the the text is about religion or not.

Let X be the data matrix, y the outcome vector (-1 or 1 if the doc is a specific category like religion). Each row in X is a doc, column TF-IDF value for a given word. You would need to compute the $Gram\ Matrix\ = XX^T$ For prediction you would need to use $(Gram\ Matrix)^-1X^Ty$ where y is the outcome vector.

5 Important Considerations

5.1 Machines to Use

One thing to be aware of is that you can choose virtually any configuration for your EMR cluster - you can choose different numbers of machines, and different configurations of those machines. And each is going to cost you differently!

Pricing information is available at: `http://aws.amazon.com/elasticmapreduce/pricing/`

Since this is real money, it makes sense to develop your code and run your jobs locally, on your laptop, using the small data set. Once things are working, you'll then move to Amazon EMR. We are going to ask you to run your Spark jobs over the "real" data using five **c3.2xlarge machines** as workers. This provides 8 cores per machine (40 cores total) so it is quite a bit of horsepower.

As you can see on EC2 Price list , this costs around 50 cents per hour. That is not much, but **IT WILL ADD UP QUICKLY IF YOU FORGET TO SHUT OFF YOUR MACHINES**. Be very careful, and stop your machine as soon as you are done working. You can always come back and start your machine or create a new one easily when you begin your work again. Another thing to be aware of is that Amazon charges you when you move data around. To avoid such charges, do everything in the N. Virginia region. That's where data is, and that's where you should put your data and machines.

- You should document your code very well and as much as possible.
- You code should be compilable on a unix-based operating system like Linux or MacOS.

5.2 Academic Misconduct Regarding Programming

In a programming class like our class, there is sometimes a very fine line between "cheating" and acceptable and beneficial interaction between peers. Thus, it is very important that you fully understand what is and what is not allowed in terms of collaboration with your classmates. We want to be 100% precise, so that there can be no confusion.

The rule on collaboration and communication with your classmates is very simple: you cannot transmit or receive code from or to anyone in the class in any way—visually (by showing someone your code), electronically (by emailing, posting, or otherwise sending someone your code), verbally (by reading code to someone) or in any other way we have not yet imagined. Any other collaboration is acceptable.

The rule on collaboration and communication with people who are not your classmates (or your TAs or instructor) is also very simple: it is not allowed in any way, period. This disallows (for example) posting any questions of any nature to programming forums such as **StackOverflow**. As far as going to the web and using Google, we will apply the **"two line rule"**. Go to any web page you like and do any search that you like. But you cannot take more than two lines of code from an external resource and actually include it in your assignment in any form. Note that changing variable names or otherwise transforming or obfuscating code you found on the web does not render the "two line rule" inapplicable. It is still a violation to obtain more than two lines of code from an external resource and turn it in, whatever you do to those two lines after you first obtain them.

Furthermore, you should cite your sources. Add a comment to your code that includes the URL(s) that you consulted when constructing your solution. This turns out to be very helpful when you're looking at something you wrote a while ago and you need to remind yourself what you were thinking.

5.3 Turnin

Create a single document that has results for all three tasks. For each task, copy and paste the result that your last Spark job wrote to Amazon S3. Also for each task, for each Spark job you ran, include a screen shot of the Spark History.

Please zip up all of your code and your document (use .gz or .zip only, please!), or else attach each piece of code as well as your document to your submission individually.