Katherine Olson

STA 141C Homework #2

Code can be found here: https://katherineolson.github.io/141C_HW2_code.html

Problem 1

This problem uses the PageRank and Hubs & Authorities scores for the Wikipedia hyperlink network (https: //snap.stanford.edu/data/enwiki-2013.html). It uses two files from this website and reads "enwiki-2013.txt" into a sparse CSR matrix.

1. Problem:

   Implement the power method for computing the top singular value and the corresponding right singular vector for a given CSR matrix (don't use eigs or svds). Input of this function is a sparse CSR matrix and number of iterations (an integer). Output of this function is the leading right singular vector (array) and singular value (float). You need to use the fact that the leading right singular vector of A is the same with the leading eigenvector of A^TA, and the power method can be used for computing the eigenvector (not singular vector).

   Code snippets:

   The main function sends the file to be read, loops through each iterations and calls the power method, times it, computes the quality, and does the same for the svds function in Scipy.

   Here is the function that reads the data into a sparse matrix:

```
def readIntoSparse(filePath):
    """
    Input: File path to the data
    Output: A sparse matrix of the data
    """
    # Read in the file:
    links = pd.read_table(filePath, sep = " ",
        dtype = np.int32, skiprows = 4, engine = "c",
        names = ("from", "to"))
```

```
    n = len(links["from"])
    vals = np.ones(n) # The values in the matrix
    n2 = max( max(links["from"]), max(links["to"])) + 1 # How big to make it

    # Put into the matrix:
    wikiSparse = csr_matrix((vals, (links["from"], links["to"])), shape = (n2, n2))

    return wikiSparse
```

Here is the power method function:

```
def power_method(wikiSparse, iterations):
    """
    Input: A sparse CSR matrix and the number of iterations
    Output: The leading right singular vector (array) and singular value (float) using
    the power method
    """
    # Initialize the random vector:
    eigVec = csr_matrix(np.random.rand(wikiSparse.shape[0])).T

    # Compute an approximation to the eigenvalue and eigenvector for each iteration:
    for i in xrange(iterations + 1):
        eigVec = wikiSparse.dot(eigVec)# wikiSparse times initial vector
        eigVec = eigVec / scipy.sparse.linalg.norm(eigVec)# Get the eigenvector

    # Get the corresponding eigenvalue:
    eigVal1 = eigVec.T.dot(wikiSparse)
    eigVal2 = wikiSparse.T.dot(eigVec)
    eigVal = eigVal1.dot(eigVal2)

    return eigVec, eigVal ## (top_right_singular_vector and top_singular_value)
```

2. The results:

Power Method		
Iterations	**Quality**	**Run Time (sec)**
1	2.620709	21.55
3	3.401727	29.24
5	3.824876	40.66
10	3.918720	59.89
20	3.864739	103.2

Scipy's svds() Function	
Quality	**Run Time (sec)**
61.674726	106.46

From this table, it can be seen that the quality of the solution is much higher for the svds Method. The larger quality is larger because that vector is better aligned to an eigenvector.

3. The leading right singular vector corresponds to the "authority" score of a web page. List the names of the top-5 authoritative pages and their scores for this Wikipedia hyperlink network

Website	Authority Score
List of Sovereign States	0.2464
Geographic Names Information System	0.1784
Political Divisions of the United States	0.1710
Federal Information Processing Standard	0.1521
Race and Ethnicity in the United States Census	0.1503

Code:

```
# Get the vector:
_, _, svdVec = scipy.sparse.linalg.svds(wikiSparse, k = 1)
svdVecD = svdVec

# Find top 5 scores:
print "Top 5 scores: ", np.sort(abs(svdVecD))[0, -6:-1]
ids = np.argsort(abs(svdVecD))[0, -6:-1]
print "The 5 node IDs:", ids
scores = pd.read_csv("enwiki-2013-names.csv", error_bad_lines=False)
print "In order from 5th place to 1st: "
for oneID in ids:
        print scores[scores["node_id"] == oneID]["name"]
```

Problem 2: PPMI

This problem uses the Quora question pairs data. Where each row of data has two questions and a label representing whether or not those questions are asking the same thing.

1. Problem:

   Write a function in python to compute the PPMI matrix given a list of sentences. The PPMI matrix is defined by the following:

   - #(w, c): number of times two words w, c appear in the same sentence within distance L. Here we set L = 3
   - #(w): number of times a word w appeared in the dataset
   - |D|: total number of pairs in the dataset
   - n: number of distinct words in the dataset
   - The PPMI value of two words w, c is defined by
     $$PPMI(w, c) = max(0,\ log((\#(w, c)|D|) \div (\#w\ \#c)\ ))$$
   - The PPMI matrix is a n-by-n matrix, each element is the PPMI value between two distinct words ($M_{w,c}$ = PPMI(w, c))
   - The PPMI matrix is stored in CSR sparse format

   Code snippets:

My main function reads in the data, grabs all the sentences, gets the PPMI matrix, saves it to a file, and prints out the information in question three.

To clean the sentences, I reuse the preprocess() function from homework one.

Here is my PPMI function, a description of it can be seen in question two.

```
def compute_ppmi( sentences ):
	"""
	Input: A list of Quora questions
	Output: A sparse PPMI matrix with a score for every word pair
	"""
	wordCount, pairCount = Counter(), Counter()
	mapNumber = {} # Maps each word to a number - "word" : number
	mappingNum = 0 # Start the mapping at zero

	# Go through each question:
	for question in sentences:
		try: # Split and preprocess the question
			wordsString = preprocess(question).split()
		except: # Catches questions with "nan"
			continue

		# Go through each word in the sentence:
		for word in wordsString:
			try: # If the word is already mapped to a number
				mapNumber[word]
			except: # If not in map, give it a number value
				mapNumber[word] = mappingNum
				mappingNum += 1

			wordCount[mapNumber[word]] += 1 # Increment word count

		# Go through the index of each word:
		for index in xrange(len(wordsString)):
			# Go through each index's next three words:
			for distance in xrange(index + 1, index + 4):
				try: # Look at the three words after that index if not at end:
```

```
                    # Add pair to the count as well as the swapping of that pair:
                    pairCount[(mapNumber[wordsString[index]],
                        mapNumber[wordsString[distance]])] += 1
                    pairCount[(mapNumber[wordsString[distance]],
                        mapNumber[wordsString[index]])] += 1
                except: # If at end of sentence, move along
                    pass

    # Compute PPMI matrix:
    D = float(len(pairCount))
    n = len(wordCount)

    w1 = np.array([wordCount[w] for w, _ in pairCount])
    w2 = np.array([wordCount[w] for _, w in pairCount])
    counts = np.array(pairCount.values())
    ppmi = np.log((counts * D) / (w1 * w2))

    ppmiMatrix = csr_matrix((ppmi, (tuple([w for w, _ in pairCount]), tuple([w for _, w in
        pairCount]))), shape = (n, n)) # Make the matrix

    return ppmiMatrix
```

2. Briefly describe your algorithm for forming the PPMI matrix. What is the time complexity of your algorithm?

   My algorithm goes through each questions, preprocess and splits it, maps each word to a number if that word does not yet have a mapping, and increments the word counter. Then while still looping through each question, it loops over the index of each word. For each index it updates the pair counter with the word from that index and the pairs it makes with the next three words (if they exist). Then outside of the questions loop, the algorithm computes D, n, an array of word counts for the left word in each pair, an array for the right word, an array of all the count values for each pair, and then plug them into the ppmi formula to get an array of ppmi values. Those ppmi values are then passed into a sparse matrix as well as two tuples of each left and right word and n to specify the matrix size.

   So, all of my looping in the worst case:

   - I go through each question O(q) and do commands that are constant time.

- I go through each word twice within that loop O(2w) and also do commands that are constant time (finding / adding to a dictionary is very fast - hash table)
- Go through the pair four times O(4D)
- Total: O(2wq) + O(4D) = O(wq) + O(D) where q = number of questions, w = number of words, and D = number of pairs within L = 3

3. After running the script on training.csv: (Takes just under 2 minutes)

Shape of PPMI Matrix	(97504, 97504)
Number of Nonzero Elements	4918035
Frobenius Norm	10689.7166989

So there were 97,504 total words used in the questions of training.csv. Saving this matrix in a sparse matrix saves saving over 9 billion zeros.

Problem 3: Word2Vec

1. Problem:

After getting the PPMI matrix, compute the top-k eigenvectors $V_k \in \mathbf{R}^{n\times k}$ and eigenvalues

$\Sigma_k \in \mathbf{R}^{k\times k}$ (diagonal matrix). Set k = 100. Form the word embedding matrix $F = V_k\Sigma_k$ where each row is a k-dimensional embedding feature vector for a word. Now we use these features to classify Quora question pairs. For each sentence q, compute the feature vector for the sentence by averaging all the word embeddings features:

$$\text{feature vector for question } q := x_q := (1 / |q|) \sum_{w:w\in q} f_w,$$

where f_w is the word embedding for word w, q is the set of words in the sentence, and |q| is number of words in sentence q. The cosine similarity of two sentences can be computed by

$$\text{cosine similarity between } q_1, q_2 = (x^T_{q1}\, x_{q2}) / (||x_{q1}||\ ||x_{q2}||\ )$$

We can predict the label for a question pairs (q_1, q_2) by

$$\text{sign(cosine similarity}(q_1, q_2) - \text{thr}),$$

where thr is a positive real number for thresholding.

<u>Code Snippets:</u>

I import the preprocess function from problem 2 and a few other necessary libraries.

The main function starts by reading in the data specified in the command line and then grabbing a list of all question ones and another of question twos. Main then reads in the sparse matrix from problem 2 by loading in the .npz file it was saved in. Main then gets a list of all cosine similarity values by calling the following function cosSimList():

```
def cosSimList(ppmiMatrix, questions1, questions2):
    """
    Input: The PPMI matrix from problem 2, a list of all question1s, and a list of all
    question2s.
    Output: A list of all of the cosine similarity values between questions.
    """
    # Get the matrix F = word embedding matrix:
    eigVals, eigVecs = linalg.eigs(ppmiMatrix, k = 100)
    eigVals = np.diag(eigVals)
    F = eigVecs.dot(eigVals)

    # Get word mapping for training that matches the PPMI matrix:
    training = pd.read_csv("training.csv", header = None, names = ["qid", "qid2",
                        "question1", "question2", "duplicate"])
    wordNumMap = wordMapping(training["question1"].tolist() +
                        training["question2"].tolist())

    # Get the cosine similarity values:
    cosSim = [getCosSim(computeFeatureVec(q1, F, wordNumMap),
        computeFeatureVec(q2, F, wordNumMap)) for q1, q2 in zip(questions1,
                                                                questions2)]

    return cosSim
```

Within this function, many other functions are called. One of these is the wordMapping() function. This function repeats the mapping of each word to a number that was done in problem 2 with training.csv so the PPMI matrix can be properly interpreted. Another function that is called is the getCosSim() function. This function is passed parameters from the computeFeatureVec() function:

```
def computeFeatureVec(question, F, wordMapNum):
    """
    Input: A string containing a question, the F matrix, and the dictionary with word
    mapping.
    Output: The feature vector for that question.
    """
    words = preprocess(str(question)).split()

    q = len(words)

    try: # Plug into formula for the feature vector
        featVec = sum([F[wordMapNum[word], ] for word in words]) / float(q)
    except: # In the cases where finding the number mapped to that word fails
        featVec = np.nan

    return featVec
```

The results for the feature vector of two question pairs are passed to following function to get the cosine similarity for that function:

```
def getCosSim(featVec1, featVec2):
    """
    Input: Two feature vectors for a question pair.
    Output: The cosine similarity for that pair.
    """
    # Catch nans:
    if type(featVec1) == float: return 0.0
    if type(featVec2) == float: return 0.0

    try: # Plug into the formula:
        cosSim = featVec1.T.dot(featVec2) / (np.linalg.norm(featVec1) *
            np.linalg.norm(featVec2))
        cosSim = cosSim.real
    except: # Catches cases where question was ".":
        cosSim = 0.0

    return cosSim
```

Now that the main function has all of the cosine similarity values for each pair, the accuracy can be computed:

```
def getAccuracy(cosSim, label, thrAmmount):
	"""
	Input: A list of the cosine similarity values, a list of the correct labels, and either a
	single threshold or the string "ALL" representing the values from 0.8 to 1.0 with
	increments of  0.02.
	Output: A DataFrame containing each of the specified thresholds and the
	corresponding accuracy at predicting
	whether or not the questions have the same meaning.
	"""
	# Then try all thresholds:
	if thrAmmount == "ALL":
		thrList = np.arange(0.8, 1.02, 0.02)
	else: # Just try the threshold passed in through the command line:
		thrList = [float(thrAmmount)]

	results = pd.DataFrame(columns = ["Threshold", "Accuracy"])

	# For each threshold:
	for thr in thrList:
		est = np.sign(np.array(cosSim) - thr) # Get the estimate
		correct = sum(est == label) + sum(label + est == -1) # Get the num correct
		classRate = float(correct) / label.shape[0] # Classification rate
		results = results.append(pd.DataFrame.from_dict([{"Threshold" : thr,
			"Accuracy" : classRate}]))

	return results
```

2. For training.csv:

Threshold	**Accuracy**
0.8	0.576390
0.82	0.586298
0.84	0.595371
0.86	0.602654
0.88	0.609029
0.9	0.614239
0.92	0.617063
0.94	0.617484
0.96	0.614885
0.98	0.609855
1.0	0.630832

   So, the threshold with the best result is 1.0.
3. Running this threshold on validation gives:

Threshold	**Accuracy**
1.0	0.630356

4. Discuss findings:
   For the training data, as the threshold increases, for the most part, so does the accuracy. The accuracy peaks at a threshold of 1.0 with an accuracy of about 63.1%. This same threshold gives an accuracy of about 63.0% on the validation data. So the training set did a good job on the testing set. This means there was not too much overfitting. But, an accuracy of 63% is not very high. So, this is either a very tricky problem, or there are better methods to be found.