SI 630: Homework 2

Yashaswini Joshi (Unique Name : yjoshi) Kaggle Name: Yashaswini Joshi/Seabiscuit

TASK 1:

Problem 1. Modify function negativeSampleTable to create the negative sampling table.

Ans:

```
def negativeSampleTable(train_data, uniqueWords, wordcounts, exp_power=0.75):
    #global wordcounts
    #... stores the normalizing denominator (count of all tokens, each count raised to exp_power)
    max_exp_count = 0


    print ("Generating exponentiated count vectors")
    #... (TASK) for each uniqueWord, compute the frequency of that word to the power of exp_power
    #... store results in exp_count_array.
    exp_count_array = [math.pow(wordcounts[t], exp_power) for t in uniqueWords]
    max_exp_count = sum(exp_count_array)



    print ("Generating distribution")

    #... (TASK) compute the normalized probabilities of each term.
    #... using exp_count_array, normalize each value by the total value max_exp_count so that
    #... they all add up to 1. Store this corresponding array in prob_dist
  # prob_dist = exp_count_array / max_exp_count
    prob_dist = list(map(lambda x: float(x / max_exp_count), exp_count_array))





    print ("Filling up sampling table")
    #... (TASK) create a dict of size table_size where each key is a sequential number and its value is a
one-hot index
    #... the number of sequential keys containing the same one-hot index should be proportional to
its prob_dist value
    #... multiplied by table_size. This table should be stored in cumulative_dict.
    #... we do this for much faster lookup later on when sampling from this table.

    #cumulative_dict = #... fill in
    table_size = 1e7
    word_freqs = [int(p * table_size) for p in prob_dist]
```

```
    cumulative_dict = {}

    j = 0
    for ind, freq in enumerate(word_freqs):
        i = 0
        while i < freq:
            cumulative_dict[j] = ind
            i += 1
            j += 1
    return cumulative_dict
```

Problem 2. Modify function performDescent() to implement gradient descent.

Ans:

```
def performDescent(num_samples, learning_rate, center_token, context_words,W1,W2,negative_indices):
    # sequence chars was generated from the mapped sequence in the core code
    nll_new = 0
    chunks = [negative_indices[x:x+2] for x in range(0, len(negative_indices), 2)]
    for k in range(0, len(context_word_ids)):
        neg_ll_total = 0
        context_index = context_word_ids[k]
        h = np.array(W1[center_token])
        W2_p = np.array(W2[context_index])

        #  Updating W2 for the postive context sample
        s = sigmoid(np.dot(W2[context_index], h))
        W2[context_index] = W2_p - (learning_rate * ((s - 1) * h))

        tot_p_neg = 0

        # iterating over the negative samples for the given context word
        for neg in chunks[k]:
            #  Updating W prime for the two negtive samples
            W2_p_neg = np.array(W2[neg])
            s_neg = sigmoid(np.dot(W2[neg], h))
            W2[neg] = W2_p_neg - (learning_rate * ((s_neg - 0) * h))

            tot_p_neg += (sigmoid(np.dot(W2_p_neg, h) - 0) * W2_p_neg)

            # Negative LL for both the negative samples
            nsig = sigmoid(np.negative(np.dot(W2[neg], h)))
            neg_ll_total += np.log(nsig)

        #  Updating W1 for the center token
```

```
        s2_pos = sigmoid(np.dot(W2_p, h))
        pos_vj = (s2_pos - 1)* W2_p
        total_vj = (pos_vj + tot_p_neg)
        W1[center_token] = h - (learning_rate * total_vj)

        #  calculating the negtive LL for the postive context token
        pos = (np.negative(np.log(sigmoid(np.dot(W2[context_index], h)))))
        nll_new += pos - neg_ll_total


    return [nll_new]
```

Problem 3. Modify function loadData to remove stop-words from the source data input.
Ans:

```
    def loadData(filename):
    global uniqueWords, wordcodes, wordcounts
    override = False
    if override:
        #... for debugging purposes, reloading input file and tokenizing is quite slow
        #...  >> simply reload the completed objects. Instantaneous.
        fullrec = pickle.load(open("w2v_fullrec.p","rb"))
        wordcodes = pickle.load( open("w2v_wordcodes.p","rb"))
        uniqueWords= pickle.load(open("w2v_uniqueWords.p","rb"))
        wordcounts = pickle.load(open("w2v_wordcounts.p","rb"))
        return fullrec


    # ... load in first 15,000 rows of unlabeled data file.  You can load in
    # more if you want later (and should do this for the final homework)
    handle = open(filename, "r", encoding="utf8")
    fullconts = handle.read().split("\n")
    fullconts = fullconts[1:15000]  # (TASK) Use all the data for the final submission
    #... apply simple tokenization (whitespace and lowercase)
    fullconts = [" ".join(fullconts).lower()]
    print ("Generating token stream...")
    #... (TASK) populate fullrec as one-dimension array of all tokens in the order they appear.
    #... ignore stopwords in this process
    #... for simplicity, you may use nltk.word_tokenize() to split fullconts.
    #... keep track of the frequency counts of tokens in origcounts.
    word_tokens = nltk.word_tokenize((fullconts[0]))
    stop_words = set(stopwords.words('english'))
    min_count = 50

    fullrec = [w for w in word_tokens if not w in stop_words]
```

```
    origcounts = Counter(fullrec)
    print ("Performing minimum thresholding..")
    #... (TASK) populate array fullrec_filtered to include terms as-is that appeared at least min_count times
    #... replace other terms with <UNK> token.
    #... update frequency count of each token in dict wordcounts where: wordcounts[token] = freq(token)
    fullrec_filtered = [w if origcounts[w] >= min_count else "<UNK>" for w in fullrec]


    #... after filling in fullrec_filtered, replace the original fullrec with this one.
    fullrec = fullrec_filtered
    wordcounts = Counter(fullrec)
    print ("Producing one-hot indicies")
    #... (TASK) sort the unique tokens into array uniqueWords
    #... produce their one-hot indices in dict wordcodes where wordcodes[token] = onehot_index(token)
    #... replace all word tokens in fullrec with their corresponding one-hot indices.
    uniqueWords = set(fullrec)  # ... fill in
    for i, word in enumerate(uniqueWords):
            wordcodes[word] = i
    fullrec = list(map(lambda x: wordcodes[x], fullrec))
    handle.close()
    pickle.dump(fullrec, open("w2v_fullrec.p","wb+"))
    pickle.dump(wordcodes, open("w2v_wordcodes.p","wb+"))
    pickle.dump(uniqueWords, open("w2v_uniqueWords.p","wb+"))
    pickle.dump(dict(wordcounts), open("w2v_wordcounts.p","wb+"))
    return fullrec
```

Problem 4. Modify function loadData to convert all words with less than min count occurrences into tokens. Modify function trainer to avoid cases where is the input token.

Ans:

```
def trainer(curW1 = None, curW2=None):
    global uniqueWords, wordcodes, fullsequence, vocab_size, hidden_size,np_randcounter, randcounter
    vocab_size = len(uniqueWords)       #... unique characters
    hidden_size = 100                   #... number of hidden neurons
    context_window = [-2,-1,1,2]        #... specifies which context indices are output. Indices relative to target word. Don't include index 0 itself.
    nll_results = []                    #... keep array of negative log-likelihood after every 1000 iterations
    iteration_number = []

    #... determine how much of the full sequence we can use while still accommodating the context window
```

```
start_point = int(math.fabs(min(context_window)))
end_point = len(fullsequence)-(max(max(context_window),0))
mapped_sequence = fullsequence



#... initialize the weight matrices. W1 is from input->hidden and W2 is from hidden->output.
if curW1==None:
	np_randcounter += 1
	W1 = np.random.uniform(-.5, .5, size=(vocab_size, hidden_size))
	W2 = np.random.uniform(-.5, .5, size=(vocab_size, hidden_size))
else:
	#... initialized from pre-loaded file
	W1 = curW1
	W2 = curW2



#... set the training parameters
epochs = 5
num_samples = 2
learning_rate = 0.05
nll = 0
iternum = 0




#... Begin actual training
for j in range(0,epochs):
	print ("Epoch: ", j)
	prevmark = 0

	#... For each epoch, redo the whole sequence...
	for i in range(start_point,end_point):

		if (float(i) / len(mapped_sequence)) >= (prevmark + 0.1):
			print("Progress: ", round(prevmark + 0.1, 1))
			prevmark += 0.1
		if iternum % 10000 == 0:
			print("Negative likelihood: ", nll)
			print("Iteration Number: ", iternum)
			nll_results.append(nll)
			iteration_number.append(iternum)
```

```
nll = 0

#... (TASK) determine which token is our current input. Remember that we're looping through mapped_sequence
if wordcodes["<UNK>"] == mapped_sequence[i]:
    continue
center_token = mapped_sequence[i] #... fill in
#... (TASK) don't allow the center_token to be <UNK>. move to next iteration if you found <UNK>.




iternum += 1
#... now propagate to each of the context outputs
#for k in range(0, len(context_window)):

    #... (TASK) Use context_window to find one-hot index of the current context token.
    #context_index = #... fill in
mapped_context = [mapped_sequence[i + ctx] for ctx in context_window]
negative_indices = []
for q in mapped_context:
    negative_indices += generateSamples(q, num_samples)


    #... construct some negative samples
    #negative_indices = generateSamples(context_index, num_samples)

    #... (TASK) You have your context token and your negative samples.
    #... Perform gradient descent on both weight matrices.
    #... Also keep track of the negative log-likelihood in variable nll.

[nll_new] = performDescent(num_samples, learning_rate, center_token, mapped_context, W1, W2,negative_indices)
nll += nll_new


for i in range(len(nll_results)):
    stri += str(nll_results[i]) + "\t" + str(iteration_number[i]) + "\n"
file_handle.write(stri)
```

```
return [W1,W2]
```

TASK 3:

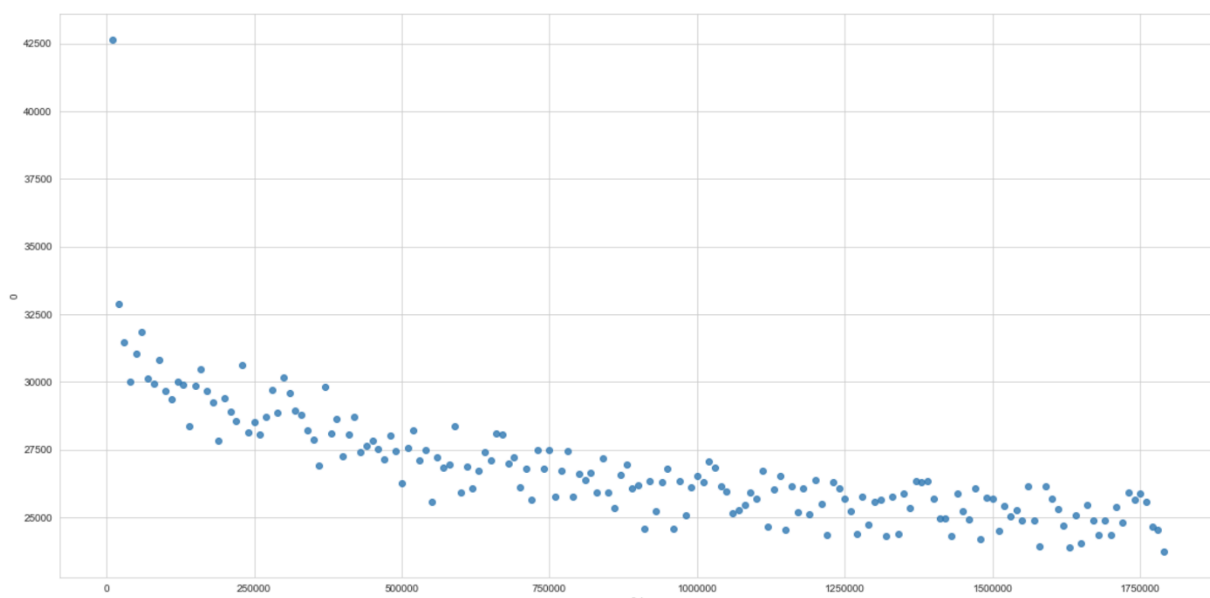
Problem 5. Load the model (vectors) you saved in Task 2 by commenting out the train vectors() line and uncommenting the load model() line. Modify the file names in load model() according to what you named them in Task 2.

Ans:

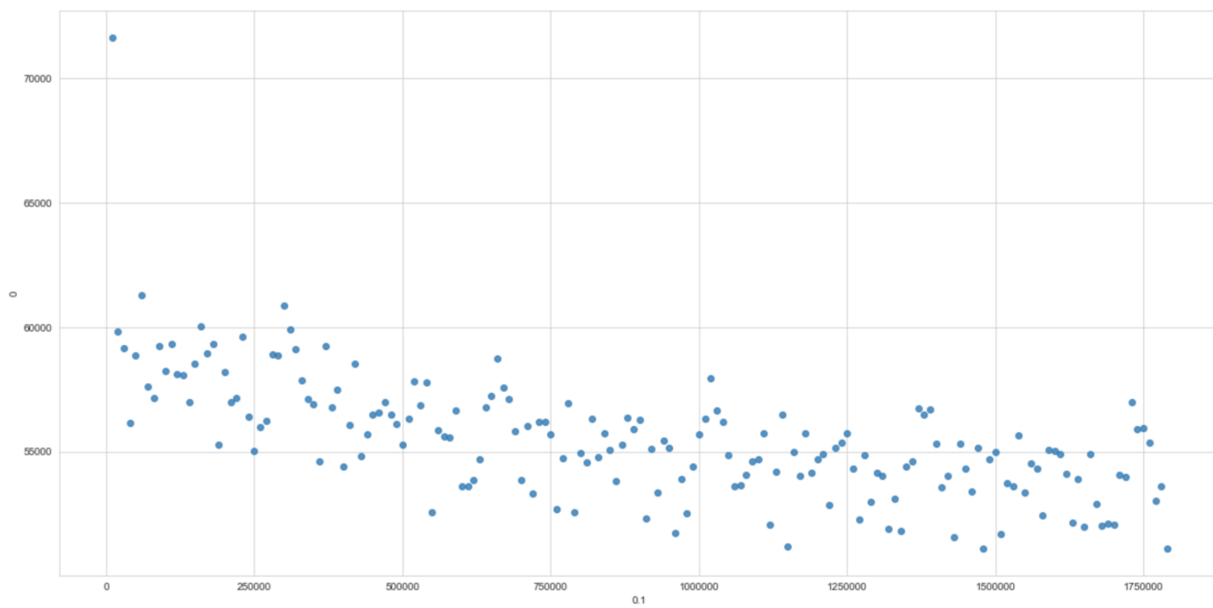
```
word_embeddings = []
proj_embeddings = []
def train_vectors(preload=False):
    global word_embeddings, proj_embeddings
    if preload:
        [curW1, curW2] = load_model()
    else:
        curW1 = None
        curW2 = None
    [word_embeddings, proj_embeddings] = trainer(curW1,curW2)
    save_model(word_embeddings, proj_embeddings)
```

Graph:

Plot 1: Negative Log Likelihood (±2 context) vs. iterations (250,000 per tick)



Plot 2: Negative Log Likelihood (±4 context) vs. iterations (250,000 per tick)



Based on the above plots of negative log-likelihoods of the 2 context windows vs. iterations, ±2 context window graph started around the point above 42,500 negative log-likelihoods and ±4 context window graph started around the point above 70,000 negative log-likelihoods, but declined over time. Definitively, the ±2 context window performed better in terms of reducing negative loglikelihood, with the rapid decrease in log-likelihood significantly larger for the ±2 context window. This may indicate that the predictive power of context words is related to absolute distance from the target word, since the absolute maximum distance in the ±2 context window is only 2 compared to the other 2 model's 4 words distance.

The graph of each context window also has large areas where the negative log likelihood hits 0, probably indicating that the program hit an unknown word token at that iteration, meaning that no negative log-likelihood was calculated at the time, defaulting to 0. Changing the context window from ±2 words to +4 or -4 also changed the time in which the program recorded the 0, as seen in the combined graph.

Problem 6. Write a function get neighbors(word) so that it takes one argument, the target word, and computes the top 10 most similar words based on the cosine similarity.

Ans:

```
def get_neighbors(target_word):
    global word_embeddings, uniqueWords, wordcodes
    targets = [target_word]
    outputs = []
pred = {}

    for uniqueWord in uniqueWords:
        cos_similarity = abs(1 -
scipy.spatial.distance.cosine(word_embeddings[wordcodes[uniqueWord]],
                              word_embeddings[wordcodes[target_word]]))
        pred[uniqueWord] = cos_similarity

    return dict(Counter(pred).most_common(11))


def calculate_cosine_similarity(word1, word2):
    global word_embeddings, uniqueWords, wordcodes

    cos_similarity = abs(1 - scipy.spatial.distance.cosine(word_embeddings[wordcodes[word1]],
                             word_embeddings[wordcodes[word2]]))
    return cos_similarity
def output(filename):
  global word_embeddings, uniqueWords, wordcodes
  file = open(filename)
  output = open('intrinsic-output.csv', 'w')
  output.write('id,sim\n')
  for x in file.readlines()[1:]:
    words = x.rstrip('\n').split('\t')
    s1_idx = uniqueWords.index(words[1])
    s2_idx = uniqueWords.index(words[2])
    distance = cosine(word_embeddings[s1_idx], word_embeddings[s2_idx])
    word_similarity = 1 - distance
    output.write('{},{},{}\n'.format(words[0], word_similarity))
  intr.close()
  output.close()

def learn_morphology(train_data):
    global word_embeddings, proj_embeddings, uniqueWords, wordcodes
    s_suffix = []
    for d in train_data:
```

```
            s_suffix.append(word_embeddings[wordcodes[d[0]]]                                    -
word_embeddings[wordcodes[d[1]]]])
    return np.mean(s_suffix)


def knearest(target_word_vector, k, morphological_variation):
    global word_embeddings, uniqueWords, wordcodes
    pred = {}
    for uniqueWord in uniqueWords:
            cos_similarity                    =                    abs(1                    -
scipy.spatial.distance.cosine(word_embeddings[wordcodes[uniqueWord]],
                        target_word_vector))
            pred[uniqueWord] = cos_similarity

    k_nearest = dict(Counter(pred).most_common(k + 1))
    print(k_nearest)
    i = 0
    for key in k_nearest:
            i += 1
            if morphological_variation == key:
                    return i


def get_or_impute_vector(test_data, suffix_vector):
    global word_embeddings, uniqueWords, wordcodes
    precision_k = []
    for d in test_data:
            total_word_vector = word_embeddings[wordcodes[d[1]]] + suffix_vector
            precision_k.append(knearest(total_word_vector, 20, d[0]))
    return precision_k
```

Problem 7. Pick 10 target words and compute the most similar for each using your function. Record these in a file named prob7_output.txt

Ans: The target words I chose were ["good", "bad", "food", "apple","fresh","yummy","water","meal","look","amazing"]. I have attached the prob7_output.txt along with this file. From the txt file we can observe that most of the pair have good score. In general, for all 3 files, the prediction provided pretty similar words or synonyms compared to the target, with 1 or 2 deviants (like good-vince, or scary-sorority). Noticeably, the top 10 results for each context window are significantly different, with only 1 or 2 intersecting words. Another noticeable thing is that, the predictions using the ±2 context window have meanings closer to the words alone, while the + or – 4 context window has words that can be identified as specific words from the corpus.

Problem 8. Implement the analogy function and find five set of interesting word analogy with your word2vec model.

Ans: good,caffeine,0.41564134998980684

amazing,lovely,0.49535765400720555

water,steep,0.45720830183409034

yummy,delicious,0.554528291549804

water,pour,0.5231415132200762

These are the five I found interesting as they are actually relatable analogies are also it's proved by their scores.

TASK 4:

Problem 9. For each word pair in the intrinsic-test.csv file, create a new csv file containing their cosine similarity according to your model and the pair's instance ID.

Ans: intrinsic-output.csv is uploaded to the Kaggle https://www.kaggle.com/c/hw2-word2vec/ task.

Kaggle Reported Score: