CS 4650/7650 Fall 2020: Homework 3

September 22, 2021

Instructions

1. This homework has two parts: questions 1–3 are theory questions, and Q4 is a programming assignment with some written components within a Jupyter Notebook.

   We will be using Gradescope to collect your assignments. Please read the following instructions for submitting to Gradescope carefully!

   (a) Each subproblem must be submitted on a separate page. When submitting to Gradescope (under HW3 Writing), make sure to mark which page(s) correspond to each problem or subproblem. For instance, Q2 has four subproblems, so the solution to each must start on a new page.

   (b) For the coding problem (Q3), please upload 'hw3_skeleton_char.py' to HW3 Code on Gradescope.

   You will also need to attach a pdf export of 'word_embedding.ipynb', including outputs, to your writeup, as well as copying outputs from other iPython notebooks into your write-up for Q3b and Q3c.

   (c) Note: This is a large class and Gradescope's assignment segmentation features are essential. Failure to follow these instructions may result in parts of your assignment not being graded. We will not entertain regrading requests for failure to follow instructions.

2. LaTeX solutions are strongly encouraged (a solution template is available on the class website), but scanned handwritten copies are also acceptable. Hard copies are not accepted.

3. We generally encourage collaboration with other students. You may discuss the questions and potential directions for solving them with another student. However, you need to write your own solutions and code separately, and not as a group activity. Please list the students you collaborated with on the submission site.

Questions

1. Consider the term-document matrix for four words in three documents shown in Table 1. The whole document set has $N = 30$ documents, and for each of the four words, the document frequency df_t is shown in Table 2.

term-document	Doc1	Doc2	Doc3
car	21	7	23
insurance	0	20	5
auto	2	35	29
best	10	30	0

Table 1: Term-document Matrix

	df
car	12
insurance	8
auto	16
best	10

Table 2: Document Frequency

   (a) **Compute** the *tf-idf* weights according to the definitions in Jurafsky and Martin's Textbook (equations 6.12-14) for each of the words *car*, *auto*, *insurance* and *best* in Doc1, Doc2, and Doc3. [5 pts]

   (b) Use the *tf-idf* weight you got in (a) to **represent** each document with a vector, and **calculate** the cosine similarities between these three documents. [3 pts]

   (c) Suppose we train the word vectors of four words "stocks", "currency", "can", "will" using *word2vec* and with *tf-idf*. **Which** one of the following two word pairs, ("stocks", "currency") and ("can", "will"), may have a high *tf-idf* cosine similarity but a low *word2vec* cosine similarity? And **which** one may have a low *tf-idf* cosine similarity but a high *word2vec* cosine similarity? **Explain** why. [2 pts]

2. Sobchak Industries, a dog-grooming supplies production company, is planning to train a word embedding model which will help power its support chatbot. To this end, it intends to run a **skipgram with negative sampling** model over a corpus of ten million words containing 80,000 unique word types. Larry, the chief designer of the system, set the hyperparameters at 6 training epochs, 200 hidden dimensions, and a window size of 5.

   Donny, a junior research engineer, is worried about the **out-of-vocabulary problem**. He tells Larry that if they add a **subword** component to their embeddings, the model will be better able to approximate vectors for new words it encounters during the chat sessions.

   (a) Donny's suggestion is the following: each non-space character in the corpus (there are 52 letters and 10 other characters) gets its own target embedding as a 'word part' (so, e.g. the character *a* has a separate embedding from the word *a*). During training, each pass over a target word w_t is augmented by an identical and independent pass over each of the characters it contains, predicting the same context words as for w_t (so no context characters are considered). During downstream inference (chatbot), a new word's embedding is initialized as the average of its characters' 'word part' embeddings.

   The average character length of a word in the corpus is 6.2. Answer the following: what is the percentage increase of training Donny's suggested model in terms of **space**? In terms of **time**? If we reduce the hidden dimensions parameter to 100 and only train 4 epochs, and want to change the **window size** so that the time costs are closest to the originally planned model, what window size should we choose? [2+2+3 pts]

   Note: You may ignore window effects of document boundaries; assume these are negligible.

   (b) Maude, another engineer, points out that language isn't built simply on characters which carry meaning. She proposes an *affix-based* method: collect the 1,000 most common prefixes (of any length) in the vocabulary, and the 1,000 most common suffixes in the vocabulary, and during the pass over the corpus, any word ending in one of the suffixes and/or starting with one of the prefixes is *averaged* with the affix embedding(s) and the cross-entropy loss is used to update all components of the embedding. This time, the change is applied to <u>both target and context embeddings</u>. **List all** statistics which would help calculate the added space and time complexities of this variant (an example is: "number of word types in corpus with suffix xor prefix"). Assume each word can only be associated with one prefix, one suffix, or one of each. [4 pts]

   (c) **Why** is an exact calculation of the added complexities in Maude's formulation impossible, even given the entire training corpus? **Can** changing the training regime to continuous bag-of-words (CBOW) remove this uncertainty? **Explain**. [3 pts]

   (d) The skipgram embeddings can be evaluated using *extrinsic* methods such as measuring the performance of the chat bot or other downstream tasks, such as sequence labeling. Another method of evaluation is using *intrinsic* methods, which test whether the representations cohere with our intuitions about word meaning. **Describe** at least two intrinsic evaluation methods that could be used to assess the quality of the embeddings. [1 pt]

3. In this assignment, we will be exploring word embeddings and language modeling. Start by downloading this zip file: `https://www.cc.gatech.edu/classes/AY2022/cs4650_fall/programming/h3_lm.zip`.

   For all of these notebooks, you will need to export the PDF outputs and concatenate them to your writing portion. The coding portion of this homework counts for 40 points, with 9 bonus points available.

   (a) **word_embeddings.ipynb [10 points]**: We'll start by looking at word2vec, a technique to generate word vectors. You will **not** be training your own word embeddings: instead you would be using pre-trained word embeddings from GenSim. This programming assignment is designed to provide you a better understanding of the vector space the word embeddings lie in. Specifically, you will be looking at similarities, semantics, analogies, biases and visualization. Download and complete the notebook, following the instructions provided therein. Attach your notebook here to respond to 3a.

   (b) **ngram.ipynb [20 points + 7 bonus]**:

      i. Complete 'hw3_skeleton_char.py.' Detailed instructions can be found in the 'ngram.ipynb' file in hw3.zip. You should also use test cases in 'ngram.ipynb' to get development results for parts (iii) and (iv) of this subproblem. Submit 'hw3_skeleton_char.py' to HW3 Code on Gradescope for **[15 points]**.

         As bonus, you can also complete 'hw3_skeleton_word.py,' along with the corresponding cells for word-based LMs in 'ngram.ipynb' **[Bonus: 5pt]**

      ii. Observe the generation results of your character-level n-gram language models ($n \geq 1$). The paragraphs which character-level n-gram language models generate all start with *F*. Did you get such results? **Explain** what is going on. **[3 pts]**

      iii. **[This question is a bonus, if you chose to do the word-level ngram model] Compare** the generation results of character-level and word-level n-gram language models. Which do you think is better? **Compare** the perplexity of 'shakespeare_sonnets.txt' when using character-level and word-level n-gram language models. **Explain** what you found. **[Bonus: 1 pt]**

      iv. When you compute perplexity, you can play with different sets of hyper-parameters in both character-level and word-level n-gram language models. You can tune n, k and λ. Please **report** here the best results and the corresponding hyper-parameters in development sets. For character-level n-gram language models, the development set is 'shakespeare_sonnets.txt' **[2 pts]**.
         [The following points are bonus if you chose to complete the word-level implementation] For word-level n-gram language models, the development sets are 'shakespeare_sonnets.txt' and 'val_e.txt' **[Bonus: 1 pt]**.

   (c) **rnn.ipynb [10 points + 2 Bonus]**:

      i. For RNN language models, you should complete the forward method of the RNN class in rnn.ipynb. You need to:

A. Figure out the forward pass and tune hyperparameters, **[5 pts]**
B. **Copy** a paragraph generated by your model here, **[2 pts]**
C. **Report** hyperparameters and perplexity on the development set 'shakespeare_sonnets.txt' here, **[2 pts]** and
D. **Compare** the results of character-level RNN language model and character-level n-gram language model here. **[1 pts]**
E. Imagine you designed an RNN trained on word-level tokens from a corpus similar to Gensim's. **How** would the RNN's hidden state for a word x be similar to Gensim's word vectors? **How** would it be different? **[Bonus: 2pts]**

Export 'word_embedding.ipynb,' 'ngram.ipynb', and 'rnn.ipynb' with output to a PDF and **attach all** the PDFs to your writeup. Your writeup should be uploaded to HW3 Writing.