SI 630: Homework 2 – Word Embeddings

Due: Wednesday, March 10, 11:59pm

1 Introduction

How do we represent word meaning so that we can analyze it, compare different words' meanings, and use these representations in NLP tasks? One way to learn word meaning is to find regularities in how a word is used. Two words that appear in very similar contexts probably mean similar things. One way you could capture these contexts is to simply count which words appeared nearby. If we had a vocabulary of V words, we would end up with each word being represented as a vector of length $|V|$[1] where for a word w_i, each dimension j in w_i's vector, $w_{i,j}$ refers to how many times w_j appeared in a context where w_i was used.

The simple counting model we described actually words pretty well as a baseline! However, it has two major drawbacks. First, if we have a lot of text and a big vocabulary, our word vector representations become very expensive to compute and store. A 1,000 words that all co-occur with some frequency would take a matrix of size $|V|^2$, which has a million elements! Even though not all words will co-occur in practice, when we have hundreds of thousands of words, the matrix can become infeasible to compute. Second, this count-based representation has a lot of redundancy in it. If "ocean" and "sea" appear in similar contexts, we probably don't need the co-occurrence counts for all $|V|$ words to tell us they are synonyms. In mathematics terms, we're trying to find a lower-rank matrix that doesn't need all $|V|$ dimensions.

Word embeddings solve both of these problems by trying to encode the kinds of contexts a word appears in as a low-dimensional vector. There are many (many) solutions for how to find lower-dimensional representations, with some of the earliest and successful ones being based on the Singular Value Decomposition (SVD); one you may have heard of is Latent Semantic Analysis. In Homework 2, you'll learn about a relatively recent technique, `word2vec`, that outperforms prior approaches for a wide variety of NLP tasks and is *very* widely used. This homework will build on your experience with stochastic gradient descent (SGD) and log-likelihood (LL) from Homework 1. You'll (1) implement a basic version of word2vec that will learn word representations and then (2) try using those representations in intrinsic tasks that measure word similarity and an extrinsic task for sentiment analysis.

For this homework, we've provided skeleton code in Python 3 that you can use to finish the implementation of `word2vec` and comments within to help hint at how to turn some of the math into python code. You'll want to start early on this homework so you can familiarize yourself with the code and implement each part.

[1] You'll often just see the number of words in a vocabulary abbreviated as V in blog posts and papers. This notation is shorthand; typically V is somehow related to vocabulary so you can use your judgment on how to interpret whether it's referring to the set of words or referring to the total number of words.

2 Notes

We've made the implementation easy to follow and avoided some of the useful-to-opaque optimizations that can make the code *much* faster.[2] As a result, training your model may take some time. We estimate that on a regular laptop, it might take 30-45 minutes to finish training a single epoch of your model. That said, you can still quickly run the model for $\sim$10K steps in a few minutes and check whether it's working. A good way to check is to see what words are most similar to some high frequency words, e.g., "january" or "good." If the model is working, similar-meaning words should have similar vector representations, which will be reflected in the most similar word lists. We have included this as an automated test which will print out the most similar words.

The skeleton code also includes methods for writing word2vec data in a common format readable by the Gensim library. This means you can save your model and load the data with any other common libraries that work with word2vec. Once you're able to run your model for $\sim$100K iterations (or more), we recommend saving a copy of its vectors and loading them in a notebook to test. We've included an exploratory notebook.

On a final note, this is the most challenging homework in the class. Much of your time will be spent on Task 1, which is just implementing word2vec. It's a hard but incredibly rewarding homework and the process of doing the homework will help turn you into a world-class information and data scientist!

3 Data

For data, we'll be using a sample of cleaned Wikipedia biographies that's been shrunk down to make it manageable. This is pretty fun data to use since it lets us use word vectors to probe for knowledge (e.g., what's similar to chemistry?). If you're very ambitious, we've include the full cleaned biographies which will be slow. Feel free to see how the model works and whether you can get through a single epoch! We've provided four files for you to use:

1. wiki-bios.med.txt – **Train your `word2vec` model on this data**
2. wiki-bios.DEBUG.txt – The first 100 biographies. Your model won't learn much from this but you can use the file to quickly test and debug your code without having to wait for the tokenization to finish.
3. wiki-bios.HUGE.txt – A much larger dataset of biographies. You should only use this if you want to test scalability or see how much you can optimize your method.
4. word_pairs_to_estimate_similarity.test.csv – This is the data for the intrinsic evaluation on word similarity; you'll upload your predictions to Kaggle for this.
5. synonyms.txt – You'll use this to extend word2vec in Part 4.

[2]You'll also find a *lot* of wrong implementations of word2vec online, if you go looking. Those implementations will be much slower and produce worse vectors. Beware!

4 Task 1: Word2vec

In Task 1, you'll implement parts of `word2vec` in various stages. Word2vec itself is a complex piece of software and you won't be implementing all the features in this homework. In particular, you will implement:

1. Skip-gram negative sampling (you might see this as SGNS)
2. Rare word removal
3. Frequent word subsampling

You'll spend the majority of your time on Part 1 of that list which involves writing the gradient descent part. You'll start by getting the core part of the algorithm up without parts 2 and 3 and running with gradient descent and using negative sampling to generate output data that is incorrect. Then, you'll work on ways to speed up the efficiency and quality by removing overly common words and removing rare words.

Parameters and notation The vocabulary size is V , and the hidden layer size is k. The hidden layer size k is a hyperparameter that will determine the size of our embeddings. The units on these adjacent layers are fully connected. The input is a one-hot encoded vector **x**, which means for a given input context word, only one out of V units, $\{x_1, \ldots, x_V\}$, will be 1, and all other units are 0. The output layer consists of a number of *context words* which are also V-dimensional one-hot encodings of a number of words before and after the input word in the sequence. So if your input word was word w in a sequence of text and you have a context window[3] ± 2, this means you will have four V-dimensional one-hot outputs in your output layer, each encoding words $w_{-2}, w_{-1}, w_{+1}, w_{+2}$ respectively. Unlike the input-hidden layer weights, the hidden-output layer weights are shared: the weight matrix that connects the hidden layer to output word w_j will be the same one that connects to output word w_k for all context words.

The weights between the input layer and the hidden layer can be represented by a $V \times k$ matrix W and the weights between the hidden layer and each of the output contexts similarly represented as C with the same dimensions. Each row of W is the k-dimension embedded representation v_I of the associated word w_I of the input layer—these rows are effectively the word embeddings we want to produce with word2vec. Let input word w_I have one-hot encoding **x** and **h** be the output produced at the hidden layer. Then, we have:

$$\mathbf{h} = W^T \mathbf{x} = v_I \tag{1}$$

Similarly, v_I acts as an input to the second weight matrix C to produce the output neurons which will be the same for *all* context words in the context window. That is, each output word vector is:

$$\mathbf{u} = C\mathbf{h} \tag{2}$$

[3]Typically, when describing a window around a word, we use negative indices to refer to words *before* the target, so a ± 2 window around index i starts at $i - 2$ and ends at $i + 2$ but excludes index i.

and for a specific word w_j, we have the corresponding embedding in C as $v_j^{'}$ and the corresponding neuron in the output layer gets u_j as its input where:

$$u_j = v_j^{'T}\mathbf{h} \tag{3}$$

Note that in both of these cases, multiplying the one-hot vector for a word w_i by the corresponding matrix is the same thing has simply selecting the row of the matrix corresponding to the embedding for w_i. If it helps to think about this visually, think about the case for the inputs to the network: the one-hot embedding represents which word is the center word, with all other words not being present. As a result, their inputs are zero and never contribute to the activation of the hidden layer (only the center word does!), so we don't need to even do the multiplication. In practice, we typically never represent these one-hot vectors for word2vec as it's much more efficient to simply select the appropriate row.

In the full version of word2vec that does not include the optimizations you will implement, our task is predict which context word w_c was present given an input word w_I by estimating the probabilities across the whole vocabulary using the softmax function:

$$P(w_c = w_c^{*}|w_I) = y_c = \frac{\exp(u_c)}{\sum_{i=1}^{V}\exp(u_i)} \tag{4}$$

This original log-likelihood function is then to maximize the probability that the context words (in this case, $w_{-2}, \ldots, w_{+2}$) were all guessed correctly given the input word w_I. **Note that you are not implementing this function!**

Showing this function raises two important questions (1) why is it still be described and (2) why aren't you implementing it? First, the equation represents an *ideal* case of what the model should be doing: given some positive value to predict for *one* of the outputs (w_c), everything else should be close to zero. This objective is similar to the likelihood you implemented for Logistic Regression: given some input, the weights need to be moved to push the predictions closer to 0 or closer to 1. However, think about how many weights you'd need to update to minimize this particular log-likelihood? For each positive prediction, you'd need to update $|V| - 1$ other vectors to make their predictions closer to 0. That strategy which uses the softmax results a huge computational overhead—despite being the most conceptually sound. The success of word2vec is, in part, due to coming up with a smart way to achieve nearly the same result *without* having to apply the softmax. Therefore, to answer the second question, now that you know what the goal is, you'll be implementing a far more efficient method known as **negative sampling** that will approximate creating a model that minimizes this equation!

If you read the original `word2vec` paper, you might find some of the notation hard to follow. Thankfully, several papers have tried to unpack the paper in a more accessible format. If you want another description of how the algorithm works, try reading Goldberg and Levy [2014][4] or Rong [2014][5] for more explanation. There are also plenty of good blog tutorials for how `word2vec` works and you're welcome to consult those[6] as well as some online demos that show how things work.[7]

[4]`https://arxiv.org/pdf/1402.3722.pdf`

[5]`https://arxiv.org/pdf/1411.2738.pdf`

[6]E.g., `http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/`

[7]`https://ronxin.github.io/wevi/`

4.1 Getting Started

Before we can even start training, we'll need to determine the vocabulary of the input text and then convert the text into a sequence of IDs that reflect which input neuron corresponds to which word. Word2vec typically treats all text as one long sequence, which ignores sentences boundaries, document boundaries, or otherwise-useful markers of discourse. We too will follow suit. In the code, you'll see general instructions on which steps are needed to (1) create a mapping of word to ID and (2) processing the input sequence of tokens and covert it to a sequence of IDs that we can use for training. This sequence of IDs is what we'll use to create our training data. As a part of this process, we'll also keep track of all the token frequencies in our vocabulary.

■ **Problem 1.** Modify function `load_data` to read in the text data and fill in the `id_to_word`, `word_to_id`, and `full_token_sequence_as_ids` fields. You can safely skip the rare word removal and subsampling for now.

■ **Problem 2.** Modify the `init_weights` function to initialize the W and C matrices based on the size of the vocabulary $|V|$ and the size of the embeddings. Unlike in logistic regression where we initialized our β vector be zeros, here, we'll initialize the weights to have small non-zero values centered on zero and sampled from (`-init_range`, `init_range`).[8]

During training (later), you'll iterate through all token IDs in the sequence. At each time step, the current token ID will become the *center token*. You'll use the `window_size` parameter to decide how many nearby tokens should be included as positive training examples.

4.2 Negative sampling

Once you have the tokens in place, the next step is get the next piece ready to actually train the model. Just like you did for Logistic Regression in Homework 1, we'll learn the weight matrices W and C iteratively by making a prediction and then using the difference between the model's output and ground truth to update the model's parameters. Normally, in SGD using the softmax, you would update the parameters (embeddings) for all the words after seeing each training instance. However, consider how many parameters we have to adjust: for one prediction, we would need to change $|V|N$ weights—this is expensive to do! Mikolov *et al.* proposed a slightly different update rule to speed things up. Instead of updating all the weights, we update only a small percentage by updating the weights for the predictions of the words in context and then performing *negative sampling* to choose a few words at random as negative examples of words in the context (i.e., words that shouldn't be predicted to be in the context) and updating the weights for these negative predictions.

Say we have the input word "fox" and observed context word "quick". When training the network on the word pair ("fox", "quick"), recall that the "label" or "correct output" of the network is a one-hot vector. That is, for the output neuron corresponding to "quick", we get an output of 1 and for all of the other thousands of output neurons an output of 0.

[8] Why initialize this way? Consider what would happen if our initial matrices were all zero and we had to compute the inner product of the word and context vectors. The value would always be zero and the model would never be able to learn anything!

With negative sampling, we are instead going to randomly select just a small number of negative words (lets say 5) to update the weights for. (In this context, a negative word is one for which we want the network to output a 0 for). We will also still update the weights for our positive word (which is the word quick in our current example).

The paper says that selecting 5-20 words works well for smaller datasets, and you can get away with only 2-5 words for large datasets. In this assignment, you will update with 2 negative words per context word. This means that if your context window selects four words, you will randomly sample 8 words as *negative examples* of context words. We recommend keeping the negative sampling rate at 2, but you're welcome to try changing this and seeings its effect (we recommend doing this *after* you've completed the main assignment).

4.2.1 Selecting Negative Samples

The "negative samples" (that is, the 8 output words that we'll train to output 0) are chosen using a unigram distribution raised to the $\frac{3}{4}$ power: Each word is given a weight equal to its frequency (word count) raised to the $\frac{3}{4}$ power. The probability for a selecting a word is just its weight divided by the sum of weights for all words. The decision to raise the frequency to the $\frac{3}{4}$ power is fairly empirical and this function was reported in their paper to outperform other ways of biasing the negative sampling towards infrequent words.

Computing this function for each sample is expensive, so one important implementation efficiency is to create a table so that we can quickly sample words. We've provided some notes in the code and your job will be to fill in a table that can be efficiently sampled.[9]

■ **Problem 3.** Modify function `generate_negative_sampling_table` to create the negative sampling table.

4.2.2 Gradient Descent with Negative Sampling

Because we are limiting the updates to only a few samples in each iteration, the error function E (negative likelihood) for a single context word (positive sample) with K negative samples given by set $\mathcal{W}_{neg}$ is given as:

$$E = -\log \sigma(v_{c^*}'^{T}\mathbf{h}) - \sum_{w_{neg} \in \mathcal{W}_{neg}} \log \sigma(-v_{neg}'^{T}\mathbf{h}) \tag{5}$$

where $\sigma(x)$ is the sigmoid function, $v_{c^*}'^{T}$ is the embedding of the positive context word w_{c^*} from matrix C and the $v_{neg}'^{T}$ are the embeddings of each negative context word in the same matrix C. Then, similar to what we saw earlier with neural networks, we need to apply backpropagation to reduce this error. To compute the gradient descent for the error from the output layer to the hidden layer, we do the following update for all context words (positive and negative samples, that is, for all $w_j \in \{w_{c^*}\} \cup \mathcal{W}_{neg}$):

$$v_j'^{(new)} = v_j'^{(old)} - \eta \left(\sigma(v_j'^{T}\mathbf{h}) - t_j \right) \mathbf{h} \tag{6}$$

[9]Hint: In the slides, we showed how to sample from a multinomial (e.g., a dice with different weights per side) by turning it into a distribution that can be sampled by choosing a random number in [0,1]. You'll be doing something similar here.

where $t_j = 1$ if the term is a positive context word, $t_j = 0$ if the term is a negative sampled word and $v_j^{'(old)}$ is the embedding of w_j in C *before* performing the update. Similarly, the update for the input-hidden matrix of weights will apply on a sum of updates as follows:

$$v_I^{(new)} = v_I^{(old)} - \eta \sum_{w_j} \left(\sigma(v_j^{'(old)T} \mathbf{h}) - t_j \right) v_j^{'(old)} \tag{7}$$

where w_j represents the current positive context word and its negative samples. Recall that **h** here is the embedding of v_I from matrix W. **Hint:** In your code, if a word w_a has one-hot index given as $\texttt{wordcodes}(w_a)$, then v_a is simply $W[\texttt{wordcodes}(w_a)]$ and similarly $v_a^{'}$ is simply $C[\texttt{wordcodes}(w_a)]$.

The default learning rate is set to $\eta = 0.05$ but you can experiment around with other values to see how it affects your results. Your final submission should use the default rate. For more details on the expressions and/or the derivations used here, consult Rong's paper [Rong, 2014], especially Equations 59 and 61.

In your implementation we recommend using these default parameter values:

- $k = 50$
- $\eta = 0.05$
- window ± 2
- min_count $= 5$
- epochs $= 2$

■ **Problem 4.** Modify the `train` function to complete the required training loops. Your code should complete the specified number of epoch and steps. (We've also included code for early stopping which can be left as-is). For each step, you code should examine the center token and for each context token in the context window, sample negative instances. Your code will call `predict_and_backprop` to make predictions for each context and negative-sample word.

■ **Problem 5.** Modify the function `predict_and_backprop` to implement gradient descent. Note that above this function is the line: `@jit(nopython=True)`. This can speed up the gradient descent by up to 3x but requires that your implementation uses a subset of data types and shouldn't make function calls to other parts of your code (except `sigmoid()` because it has also been jit-compiled). You don't have to use `@jit` and if this complicates your code too much, you can remove it.

4.3 Implement stop-word and rare-word removal

Using all the unique words in your source corpus is often not necessary, especially when considering words that convey very little semantic meaning like "the", "of", "we". As a preprocessing step, it can be helpful to remove any instance of these so-called "stop words".

Note that when you remove stop words, you should keep track of their position so that the context doesn't include words outside of the window. This means that a sentence with "my big *cats* of the kind that..." if you have a context window of ± 2, then you would only have "my" and "big" as context words (since "of" and "the" get removed) and not include "kind."

4.3.1 Minimum frequency threshold.

In addition to removing words that are so frequent that they have little semantic value for comparison purposes, it is also often a good idea to remove words that are so *infrequent* that they are likely very unusual words or words that don't occur often enough to get sufficient training during SGD. While the minimum frequency can vary depending on your source corpus and requirements, we will set `min_count = 5` as the default in this assignment.

Instead of just removing words that had less than `min_count` occurrences, we will replace these all with a unique token <UNK>. In the training phase, you will skip over any input word that is <UNK> but you will still keep these as possible context words.

■ **Problem 6.** Modify function `load_data` to convert all words with less than `min_count` occurrences into <UNK> tokens. Modify function `trainer` to avoid cases where <UNK> is the input token.

4.3.2 Frequent word subsampling

Words appear with varying frequencies: some words like "the" are very common, whereas others are quite rare. In the current setup, most of our positive training examples will be for predicting very common words as context words. These examples don't add much to learning since they appear in many contexts. The word2vec library offers an alternative to ensure that contexts are more likely to have meaningful words. When creating the sequence of words for training (i.e., what goes in `full_token_sequence_as_ids`), the software will randomly drop words based on their frequency so that more common words are less likely to be included in the sequence. This subsampling effectively increases the context window too—because the context window is defined with respect to `full_token_sequence_as_ids` (not the original text), dropping a nearby common words means the context gets expanded to include the next-nearest word that was not dropped.

To determine whether a token in `full_token_sequence_as_ids` should be subsampled, the word2vec software uses this equation to compute the probability $p_k(w_i)$ of a token for word w_i being kept in for training:

$$p_k(w_i) = \left(\sqrt{\frac{p(w_i)}{0.001}} + 1\right) \cdot \frac{0.001}{p(w_i)} \tag{8}$$

where $p(w_i)$ is the probability of the word appearing in the corpus initially. Using this probability, each occurrence of w_i in the sequence is randomly decided to be kept or removed based on $p_k(w_i)$.

■ **Problem 7.** Modify function `load_data` to compute the probability $p_k(w_i)$ of being kept during subsampling for each word w_i.

■ **Problem 8.** Modify function `load_data` so that after the initial `full_token_sequence_as_ids` is constructed, tokens are subsampled (i.e., removed) according to their probability of being kept $p_k(w_i)$.

5 Task 2: Save Your Outputs

Once you've finished running the code for a few iterations, save your vector outputs. The rest of the homework will use these vectors so you don't have even re-run the learning code. Task 2 is here just so that you have an explicit reminder to save your vectors. We've provided a function to do this for you and a command line flag `--output_file` that will save it to the specified file.

6 Task 3: Word Similarities

Once you've learned the word2vec embeddings from how a word is used in context new we can use them! How can we tell whether what it's learned is useful? As a part of training, we put in place code that shows the nearest neighbors, which is often a good indication of whether words that we think are similar end up getting similar representations. However, it's often better to get a more quantitative estimate of similarity. In Task 3, we'll begin evaluating the model by hand by looking at which words are most similar another word based on their vectors.

Here, we'll compare words using the *cosine similarity* between their vectors. Cosine similarity measures the angle between two vectors and in our case, words that have similar vectors end up having similar (or at least related) meanings.

■ **Problem 9.** Load the model (vectors) you saved in Task 2 by using the Jupyter notebook provided (or code that does something similar) that uses the Gensim package to read the vectors. Gensim has a number of useful utilities for working with pretrained vectors.

■ **Problem 10.** Pick 10 target words and compute the most similar for each using Gensim's function. Qualitatively looking at the most similar words for each target word, do these predicted word seem to be semantically similar to the target word? Describe what you see in 2-3 sentences. **Hint:** For maximum effect, try picking words across a range of frequencies (common, occasional, rare words).

■ **Problem 11.** Given the analogy function, find five interesting word analies with your word2vec model. For example, when representing each word by word vectors, we can generate the following equation, king - man + woman = queen. In other word, you can understand the equation as queen - woman = king - man, which mean the vectors similarity between queen and women is equal to king and man. What kinds of other analogies can you find? (**NOTE:** Any analogies shown in the class recording cannot be used for this problem.) What approaches worked and what approaches didn't? Write 2-3 seconds in a cell in the notebook.

Intrinsic Evaluation: Word Similarity Now, we'll formally test your model's similarity judgments using a standard benchmark. Typically, in NLP we use word similarity benchmarks where human raters have judged how similar two words are on a scale, e.g., from 0 to 9. Words that are similar, e.g., "cat" and "kitten," receive high scores, where as words that are dissimilar, e.g., "cat" and "algorithm," receive low scores, even if they are topically related. We'll use a subset of the data for which we have good estimates of similarity. We've already reduced it down so that we're only testing on the pairs that are in your model's vocabulary. Your scores may differ a bit since you're using a small amount of data, but ideally they should be reasonably correlated with these judgments.

■ **Problem 12.** 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. Upload these predictions to the Kaggle `https://www.kaggle.com/t/af94815ee8184b1eb3a3aadcb6881dcf` task associated with intrinsic similarity. Your file should have the header "id,sim" where `id` is the pair's ID from the `intrinsic-test.csv` file and `sim` your cosine similarity score for the pair of words with that ID.

We don't expect high performance on this task since (1) the corpus you're using is small, (2) you're not required to train for lots of iterations, and (3) you're not required to optimize much. However, you're welcome to try doing any of these to improve your score! On canvas, we will post what is the expected range of similar scores so you can gauge whether your model is in the ballpark. Since everyone is using the same random seed, it's expected that most models will have very similar scores unless people make other modifications to the code.

7 Task 4: Extending word2vec

As a software library, word2vec offers a powerful and extensible approach to learning word meaning. Many follow-up approaches have extended this core approach with aspects like (1) added more information on the context with additional parameters or (2) modifying the task so that the model learns multiple things. In Task 4 you'll try one easy extension: Using external knowledge!

Even though word2vec learns word meaning from scratch, we still have quite a few resources around that can tell us about words. One of those resources which we'll talk much more about in Week 12 (Semantics) is WordNet, which encodes word meanings in a large knowledge base. In particular, WordNet contains information on which word meanings are synonymous. For example, "couch" and "sofa" have meanings that are synonymous. In Task 4, we've provided you with a set of synonyms (`synonyms.txt`) that you'll use during training to encourage word2vec to learn similar vectors for words with synonymous meanings.

How will we make use of this extra knowledge of which woulds should have similar vectors? There have been many approaches to modifying word2vec, some which are in your weekly readings for the word vector week. However, we'll take a simple approach: during training, if you encounter a token that has one or more synonyms, replace that token with a token sampled from among the synonymous tokens (which includes that token itself). For example, if "state" and "province" are synonyms, when you encounter the token "state" during training, you would randomly swap that token for one sampled from the set ("state", "province"). Sometimes, this would keep the same token, but other times, you force the model to use the synonym—which requires that synonym's embedding to predict the context for the original token. Given more epochs, you may even predict the same context for each of the synonyms. One of the advantages of this approach is that if a synonyms word shows up much less frequently (e.g., "storm" vs. "tempest"), the random swapping may increase the frequency of the rare word and let you learn a similar vector for both.

■ **Problem 13.** Update the `main` function to take in an optional file with a list of synonyms. Update the `train` function (as you see fit) so that if synonyms are provided, during training tokens with synonyms are recognized and randomly replaced with a token from the set of synonyms (which includes the original token too!)

■ **Problem 14.** Train the synonym-aware model for the same number of epochs as you used to solve Task 3 and save this model to file.

As you might notice, the `synonyms.txt` has synonyms that only make sense in some contexts! Many words have multiple meanings and not all of these meanings are equally common. More over a word can have two parts of speech (e.g., be a noun and a verb), which word2vec is unaware of when modeling meaning. As a result, the word vectors you learn are effectively trying to represent *all* the meanings for a word in a *single* vector—a tough challenge! The synonyms we've provided are a initial effort of identifying common synonyms, yet even these may shift the word vectors in unintended ways. In the next problem, you'll assess whether your changes have improved the quality of the models.

■ **Problem 15.** Load both the original (not-synonym-aware) model and your new model into a notebook and examine the nearest neighbors of some of the same words. For some of the words in the `synonyms.txt` file, which vector space learns word vectors that have more reasonable nearest neighbors? Does the new model produce better vectors, in your opinion? Show at least five examples of nearest neighbors that you think help make your case and write at least two sentences describing why you think one model is better than the other.

8 Optional Tasks

`Word2vec` has spawned many different extensions and variants. For anyone who wants to dig into the model more, we've included a few optional tasks here. **Before attempting any of these tasks, please finish the rest of the homework and then save your code in a separate file so if anything goes wrong, you can still get full credit for your work.** These optional tasks are intended entirely for educational purposes and no extra credit will be awarded for doing them.

8.1 Optional Task 1: Modeling Multi-word Expressions

In your implementation `word2vec` simply iterates over each token one at a time. However, words can sometimes be a part of phrases whose meaning isn't conveyed by the words individually. For example "White House" is a specific concept, which in NLP is an example of what's called a **multiword expression**.[10] In our particular data, there are *lots* of multi-word expressions. As biographies a lot of people are born in the United States, which ends up being modeled as "united" and "states"—not ideal! We'll give you two ideas.

In Option 1 of Optional Task 1, we've provided a list of common multi-word expressions in our data on Canvas (`common-mwes.txt`). Update your program to read these in and during the

[10] `https://en.wikipedia.org/wiki/Multiword_expression`

`load_data` function, use them to group multi-world expressions into a single token. You're free to use whatever way you want, recognizing that not all instances of a multi-word expression are actually a single token, e.g., "We were united states the leader." This option is actually fair easy and a fun way to get multi-word expressions to show up in the analogies too, which leads to lots of fun around people analogies.

Option 2 is a bit more challenging. Mikolov *et al.* describe a way to automatically find these phrases as a preprocessing step to `word2vec` so that they get their own word vectors. In this option, you will implement their phrase detection as described in the "Learning Phrases" section of Mikolov et al. [2013].[11]

8.2 Optional Task 2: Better UNKing

Your current code treats all low frequency words the same by replacing them with an `<UNK>` token. However, many of these words could be collapsed into specific types of unknown tokens based on their prefixes (e.g., "anti" or "pre") or suffixes (e.g., "ly" or "ness") or even the fact that they are numbers or all capital letters! Knowing something about the context in which words occur can still potentially improve your vectors. In Optional Task 2, try modifying the code that replaces a token with `<UNK>` with something of your own creation.

8.3 Optional Task 3: Some performance tricks for Word2Vec

Word2vec and deep learning in general has many performance tricks you can try to improve how the model learns in both speed and quality. For Optional Task 3, you can try two tricks:

- **Dropout**: One useful and deceptively-simple trick is known as *dropout*. The idea is that during training, you randomly set some of the inputs to zero. This forces the model to not rely on any one specific neuron in making its predictions. There are many good theoretical reasons for doing this [e.g., Baldi and Sadowski, 2013]. To try this trick out, during training (and only then!), when making a prediction, randomly choose a small percentage (10%) of the total dimensions (e.g., 5 of the 50 dimensions of your embeddings) and set these to zero before computing anything for predictions.

- **Learning Rate Decay**: The current model uses the same learning rate for all steps. Yet, as we learn the vectors are hopefully getting better to approximate the task. As a result, we might want to make a *smaller* change the vectors as time goes on to keep them close to the values that are producing good results. This idea is formalized in a trick known as *learning rate decay* where as training continues, you gradually lower the learning rate in hopes that the model converges better to a local minima. There are many (many) approaches to this trick, but as an initial idea, try setting a lower bound on the learning rate (which could be zero!) and linearly decrease the learning rate with each step. You might even do this after the first epoch. If you want to get fancier, you can try to only start decreasing the learning rate when the change in log-likelihood becomes smaller, which signals that the models is converging, but could still potentially be fine-tuned a bit more.

[11] `http://arxiv.org/pdf/1310.4546.pdf`

9 Hints

1. Start early; this homework will take time to debug and you'll need time to wait for the model to train for Task 1 before moving on to Tasks 2-4 which use its output or modify the core code.
2. Run on a small amount of data at first to get things debugged.
3. The total time for `load_data` on the reference implementation loading the normal training data is under 20 seconds.
4. Average time per epoch on our tests was $\sim$ one hour. If you are experiencing times much greater than than, it's likely due to a performance bug somewhere. We recommend checking for extra loops somewhere.
5. If your main computer is a tablet, please consider using Great Lakes

10 Submission

Please upload the following to Canvas by the deadline:

1. your code for `word2vec`
2. your jupyter notebook for Parts 3 and 4
3. a PDF copy of your notebook (which as the written answers) that also includes what your username is on Kaggle.

Please upload your code and response-document separately. We reserve the right to run any code you submit; **code that does not run or produces substantially different outputs will receive a zero**.

In addition, you should upload your model's predictions for Tasks 3 to the KaggleInClass site.

11 Academic Honesty

Unless otherwise specified in an assignment all submitted work must be your own, original work. Any excerpts, statements, or phrases from the work of others must be clearly identified as a quotation, and a proper citation provided. Any violation of the University's policies on Academic and Professional Integrity may result in serious penalties, which might range from failing an assignment, to failing a course, to being expelled from the program. Violations of academic and professional integrity will be reported to Student Affairs. Consequences impacting assignment or course grades are determined by the faculty instructor; additional sanctions may be imposed.

Copying code from any existing word2vec implementation is considered grounds for violation of Academic Integrity and will receive a zero.

References

Pierre Baldi and Peter J Sadowski. Understanding dropout. *Advances in neural information processing systems*, 26: 2814–2822, 2013.

Yoav Goldberg and Omer Levy. word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722*, 2014.

Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

Xin Rong. word2vec parameter learning explained. *arXiv preprint arXiv:1411.2738*, 2014.