SI 630: Homework 2 – Word Embeddings

Due: Friday, February 9, 11:59pm

Changes

Version 4:

1. Clarified notations and added explanations for gradient descent and negative log-likelihood equations in Section 4.1.2. Also added an implementation hint in the same section.

2. Updated Problem 8 in Section 6 to be more clear.

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 Decompositions (SVD); one you may have heard of is Latent Semantic Analysis. In Homework 2, you'll learn about a new recent technique, `word2vec`, that outperforms prior approaches for a wide variety of NLP tasks. This homework will build on your experience with

[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.

stochastic gradient descent (SGD) and softmax classification 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.

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. 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 your model on a high number of iterations (e.g., 10-15) to get the vectors to converge. That said, you can still quickly run the model for a few iterations (3-4) 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., "good" or "bad." If the model is working, similar-meaning words should have similar vector representations, which will be reflected in the most similar word lists.

The skeleton code also includes methods for reading and writing word2vec data in a common format. This means you can save your model and load the data with any other common libraries that work with word2vec.[2]

3 Data

For data, we'll be using movie reviews text! We've provided four files for you to use:

1. unlabeled-text.tsv – **Train your `word2vec` model on this data**
2. extrinsic-train.tsv – This is the training data for the *extrinsic evaluation* (sentiment analysis).
3. extrinsic-dev.tsv – This is the development data for the extrinsic evaluation, which you can use to get a sense of performance
4. intrinsic-test.tsv – This is the data for the intrinsic evaluation on word similarity; you'll upload your predictions to Kaggle for this.
5. extrinsic-test.tsv – This is the test data for the extrinsic evaluation; you'll upload your scores to a second Kaggle challenge for this.

[2]Gensim provides some easy functionality that you can use: `https://rare-technologies.com/word2vec-tutorial/`. Check out the parts on "Storing and loading models" and "Using the model" for how to load your data into their code.

4 Task 1: Word2vec

In Task 1, you'll implement `word2vec` in various stages. You'll start by getting the core part of the algorithm up 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. Finally, you'll change the implementation so that you have an adjustable window, e.g., instead of looking at ± 2 words around the target word, you'll look at either 4 after the target word or 4 words before the target word.[3]

Parameters and notation The vocabulary size is V , and the hidden layer size is N. The units on these adjacent layers are fully connected. The input is a one-hot encoded vector $\mathbf{x}$, 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 $C = [-2, -1, 1, 2]$, 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 $V \times N$ matrix W and the weights between the hidden layer and each of the output contexts similarly represented as W'. Each row of W is the N-dimension embedded representation v_I of the associated word w_I of the input layer. Let input word w_I have one-hot encoding $\mathbf{x}$ and $\mathbf{h}$ be the output produced at the hidden layer. Then, we have:

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

Similarly, v_I acts as an input to the second weight matrix W' 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:

$$\mathbf{u} = W' \mathbf{h}$$

and for a specific word w_j, we have the corresponding embedding in W' as v'_j and the corresponding neuron in the output layer gets u_j as its input where:

$$u_j = v'^T_j \mathbf{h}$$

Given that the true context word for a given context is w^*_c, our task is to find which word the model estimated to be most likely by applying softmax over all $\mathbf{u}$. In particular, the probability that word w_c is the true context word is given as:

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

[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. Thought exercise: Why do we exclude index i? (Hint: think about what happens to the prediction task if we include i)

The 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. In particular, we want to maximize:

$$\log P(w_c = w_c^* | w_I) \;\; \forall c \in C$$

It's important to note that in our formulation, the relative positions of the words in the context aren't explicitly modeled by the `word2vec` neural network. Since we use the W' matrix to predict all of the words in the context, we learn the same set of probabilities for the word before the input versus the word after (i.e., word order doesn't matter).

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]

In your implementation we recommend using these default parameter values:

- $N = 100$
- $\eta = 0.05$
- $C = [-2, -1, 1, 2]$
- min_count $= 50$
- epochs $= 5$

4.1 Negative sampling

Just like you did for Logistic Regression in Homework 1, we'll learn the weight matrices W and $W^{'}$ 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, you would update all the parameters 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.

[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/`

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 C has four words, you will randomly sample 8 negative 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).

4.1.1 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.[7]

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

4.1.2 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:

$$E = -\log \sigma(v_{c^*}^{'T}\mathbf{h}) - \sum_{w_{neg} \in \mathcal{W}_{neg}} \log \sigma(-v_{neg}^{'T}\mathbf{h}) \tag{1}$$

where $\sigma(x)$ is the sigmoid function, $v_{c^*}^{'T}$ is the embedding of the positive context word w_{c^*} from matrix W' and the $v_{neg}^{'T}$ are the embeddings of each negative context word in the same matrix W'. 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 $w_j \in \{w_{c^*}\} \cup \mathcal{W}_{neg}$):

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

[7] 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 $W^{'}$ *before* performing the update. Similarly, the update for the input-hidden matrix of weights will apply on a sum of updates as follows:

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

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 `wordcodes`(w_a), then v_a is simply W[`wordcodes`(w_a)] and similarly $v_a^{'}$ is simply $W^{'}$[`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.

■ **Problem 2.** Modify function `performDescent()` 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.

4.2 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".

4.2.1 Stop-word removal.

Use the NLTK library in Python to remove stop-words after loading in your data file. If your installation of NLTK does not currently have the set of English stop-words, use the download manager to install it:

```
import nltk
nltk.download()
```

Select the "Corpora" tab and install the corpus "stopwords".

■ **Problem 3.** Modify function `loadData` to remove stop-words from the source data input.

4.2.2 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 = 50` 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 4.** Modify function `loadData` 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 Implement adjustable context window

The choice of context window length and indices can have a substantial impact on the quality of the learned embeddings. Usually, the context window will comprise of a certain number of words before the target word and the same number of words after the target. So a context window of length 4 might look like: $[-2, -1, 1, 2]$. For example, consider an input word w_I="good" which appears in the sentence: "the visuals were **good** but sound effects were not." The context words for the above window would be w_{-2} = 'visuals', w_{-1} = 'were', w_{+1} = 'but', w_{+2} = 'sound'.

■ **Problem 5.** Train your model with this context window and the default parameters specified at the start. After every 10,000 iterations, store the cumulative negative log-likelihood (Equation 1), reset the cumulative value to 0 and repeat. Plot these results against iteration number.

Note: Whenever you train a model in the assignment, the program will output two binary files `saved_W1.data` and `saved_W2.data`. If you plan to re-run your program with altered parameters (as in the next problem), **make sure** to create a backup of these two files as you will need to load these in later and as you've probably realized by now, training from scratch is very slow.

■ **Problem 6.** Re-train your network using the context windows of $C = [-4, -3, -2, -1]$ (the four words preceding the input word) and again on the window $C = [1, 2, 3, 4]$ (the four words after the input word). Repeat the analysis of Problem 5 with these context windows. What do you observe?

5 Task 2: Word Morphology

Like many languages, English has regular morphology that lets us modify the root form of word to change its meaning in predictable ways. One of the most common is adding an 's' to make a noun plural—which doesn't always work since some words are irregular. We can potentially take advantage of this morphological regularity to create vectors for OOV words if we have a vector for the root form of the word.

A common example of the power of `word2vec` is doing analogical tasks using vector arithmetic: $king - man + woman \approx queen$ (if we apply those operations to the vectors). We can take advantage of this to *learn* how different morphological suffixes and affixes affect word meaning.

For example, if we know the meanings of "effort" and "effortless," we can subtract one from the other to learn the operator for "-less", i.e., $effortless - effort \approx -less$. Learning this for one pair of words will likely give us noisy results, but if we have many examples of a root-form and an application of a suffix or affix, we can learn how *on average* the word meaning changes. Then if we have a new OOV word that has a known suffix or affix and we know its root word, we can combine the two to create a *pseudo*-vector, rather than having the word be OOV.

Part 1 Choose **at least five kinds of morphological variations** that you can find in the data[8] and for each, construct lists of all word pairs of (1) the root form and (2) either the affix+root or root+suffix. You should have at least 20 examples for each variation.

Divide each variation's data into three lists: (1) train, which has 80% of the data, (2) dev, which has 10% of the data, and (3) test, which has the remaining 10% of the data. Be sure that no data is shared between train, test, and dev! (They should be distinct datasets).

Part 2 For part 2, you will write two functions that expand the abilities of your learned word2vec model:

- `learn_morphology` will take in each list of word pairs from your **training** data and learn a mapping from the suffix/affix to the average difference of the two words (e.g., like we did for effort and effortless above).
- `get_or_impute_vector` will take a word and either (1) return your word2vec vector if the word is in-vocabulary, (2) check if the OOV word is a construction of a known root word and suffix/affix and, if so, construct the pseudo vector for the OOV word or (3) return `None` if the word cannot be deconstructed into a recognized root + suffix/affix form. Your method should have a parameter `do_impute` that forces it impute a vector for a word, even it's already known.

Part 3 In part 3, you'll test out your `get_or_impute_vector` using the **test** data. Here, we're in luck since we already know the correct word vector!

1. Given a test pair (r_i, w_i) where r_i is the root word and w_i is a morphological variation (e.g., happy, happier)
2. Call `get_or_impute_vector` to *impute* the vector for w_i (be sure the set `do_impute=True`!)
3. Given the imputed vector for w_i, compute the k nearest neighbors to it using the cosine similarity, i.e., find the k words whose vectors have the highest cosine similarity.
4. Given the k nearest neighbors, see if the w_i is in the set. If so, mark the pair as generating a correct imputation (1); otherwise, mark it as incorrect (0). This evaluation is known as Precision@k.
5. For $k = 1 \ldots 20$, plot the average Precision@k for all the words.[9]

[8] though we encourage you to try adding as many variations as you can think of

[9] **Important implementation note:** Since you're doing a range of k values, it's often much more efficient to

6 Task 3: Word similarities and analogies

In Task 1, you built a program to learn word2vec embeddings for different context windows. How can we tell whether what it's learned is useful? In Task 3, we'll begin evaluating by introducing some simple metrics and comparing mode trained on a small corpus with another pre-trained model on a much (*much*) larger corpus. Here, we'll look at the most similar words to get a sense of how much our model has learned.

In particular, for a set of target words T, your model should predict the top 10 most similar words for each target word based on the cosine similarity measure. To evaluate how good these predictions are, we will use a pre-trained word2vec model that was trained on the 2014 data dump of the English Wikipedia as the gold standard. The evaluation will be based on the Precision@k for $k = 5$ (that is, the fraction of the top 5 predictions your model made for each target word that also appeared in the top 5 results from the pre-trained model). You shouldn't expect to get a very high overlap as the model you trained in the earlier section is a much simpler model and trained on far less iterations.

■ **Problem 7.** Modify function `prediction()` so that it takes one argument, the target word, and computes the top 10 most similar words based on absolute cosine similarity.

■ **Problem 8.** Separately preload the models trained in Problem 6 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 Problem 6.

Consider the set of target words T = ['good', 'bad', 'scary','funny']. For each word, use the `prediction()` function to get the top most similar words along with their cosine similarity scores. Create an output CSV file called `p8_output_1.txt` for the model trained on context window $C = [-4, -3, -2, -1]$ and a file called `p8_output_2.txt` for the model trained on window $C = [1, 2, 3, 4]$. Both should be line-separated. Each line should be of the following format: `target_word, similar_word, similar_score`.

■ **Problem 9.** Repeat the steps in Problem 8 but using the word vectors trained in Problem 5. This output file should be called `p9_output.txt`.

■ **Problem 10.** Qualitatively looking at the topmost 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.

7 Task 4: Intrinsic Evaluation: Word Similarity

Task 3 should have proven to you that your model has learned something and in Task 4, we'll formally test your model's similarity judgments using a standard benchmark. Typically, in NLP

compute the ordered list of k most-similar words for the largest k and then compute the Precision@k for smaller k by looking at a subregion in the list; e.g., if you already have a list of the 20 most similar items in order of similarity, you can compute Precision@2 by only checking the first two items (i.e., the two most similar).

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. In Task 4, we'll use a subset of the SimLex-999 benchmark[10]. We've already reduced it down so that we're only testing on the pairs that are in your model's vocabulary.

■ **Problem 11.** For each word pair in the `intrinsic-test.tsv` file, create a new file containing their cosine similarity according to your model and the pair's instance ID. Upload these predictions to the Kaggle task associated with intrinsic similarity.

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.

8 Task 5: Extrinsic Evaluation: Sentiment Analysis

Task 4 should have shown you that your model has learned similarities that at least somewhat agree with human judgments. However, the ultimate test of word vectors is really whether they're useful in practice in downstream tasks. In Task 5, you'll implement a *very basic* sentiment analysis system using logistic regression from sklearn.[11]

When you built your logistic regression classifier in Homework 1, each word in a document was a feature. One downside to this text representation is that it doesn't take into account that some words have similar meanings, which is especially problematic for out of vocabulary words; for example, if our training data only has "good" but not "awesome," if we saw the word "awesome" in test, we would have to remove it since it's out of vocabulary—even though it has a similar meaning.

One way to solve the problem of out of vocabulary words to create a text representation using the entire document from word vectors. This representation is actually easy to make: we simply compute the average word vector from all the words in our document.[12] This average vector becomes the input to logistic regression! So if you have word vectors with 50 dimensions, you have 50 features for logistic regression.

For Task 5, you'll use the n instances in extrinsic-train.tsv to construct your X matrix and y vector (with the ground truth labels), similar to how you did for homework 1. However, this time, instead of using a sparse matrix of $n \times |V|$ sparse matrix, you'll have a $n \times k$ matrix where k is the dimensions of your word vector. Each row in the matrix will be the average word vector for

[10] `https://www.cl.cam.ac.uk/~fh295/simlex.html`

[11] `http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html`

[12] In mathematical terms, we're performing an element-wise average. For example, if we have two vectors of length 2, $v_1 = [2, 6]$ and $v_2 = [1, 2]$, we average each dimension; in this example, the average vector is then $[1.5, 4]$.

the words in the tweet.[13] While this can seem like an overly simplistic representation of the text, in practice, the average word vector is fairly effective at capturing regularities in how we discuss things.

You'll then train the sklearn LogisticRegression classifier using X and y and then use the trained classifier to predict sentiment values for texts in extrinsic-test.tsv .[14]

■ **Problem 12.** Write a method that trains a logistic regression classifier on the average word vector of each tweet using the data in extrinsic-train.tsv

■ **Problem 13.** Write a method that uses your trained classifier to predict the sentiment score for the average word vectors in the extrinsic-test.tsv. Upload these predictions to the Kaggle shared task listed on Canvas.

9 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.

9.1 Optional Task 1: Taking Word Order Into Account

Nearly all implementations of `word2vec` train their model without taking word order into account, i.e., they train only $W^{'}$. In this optional task, you will extend the model to take word order into account by training separate $W^{'}$ matrices for each relative position to the target word. Just note that this task will increase the memory requirements substantially. Once you've retrained the model, try exploring the lists of most-similar words for the words you used in Task 3 and comparing the lists with the regular word2vec model. Did adding ordering change anything? Similarly, if you look at the most-similar words using the vector in each $W^{'}$, what are they capturing?

10 Optional Task 2: Phrase Detection

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

[13] Hint: you can construct your X matrix by using a regular python list of numpy arrays (i.e., your average word vectors) and then passing that `np.array`.

[14] For help in getting a basic sklearn classifier up and running see `https://machinelearningmastery.com/get-your-hands-dirty-with-scikit-learn-now/`, which shows how to train the classifier using an existing dataset. Note that for this task, you'll need to write the parts that generate the data, but the overall training and prediction code is the same.

multiword expression.[15] 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 optional task, you will implement their phrase detection as described in the "Learning Phrases" section of Mikolov et al. [2013].[16]

11 Optional Task 3: Word Weighting for Downstream Tasks

In Task 5, you used the average word vector for Logistic Regression. However, not all words are equally informative. For this optional task, instead of using the *average* vector, try re-weighting each word's vector so more useful words contribute more to the final representation. For example, try dropping stop words as a first step.

12 Submission

Please upload the following to Canvas by the deadline:

1. a PDF (preferred) or .docx with your responses and plots for the questions above
2. your code for `word2vec`

Code should be submitted as a `.py` file. 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 4 and 5 to their respective Kaggle leaderboards, which will be posted to Canvas.

13 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.

References

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

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

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

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.