EECS595: Natural Language Processing
Homework 2, Fall 2023

Student Name: Rishikesh A. Ksheersagar — uniqname: RISHIKSH

Assigned date: 09/18/2023 - Due date: 10/02/2023

Submission Guidelines

1. Please insert your student information in line 60 of this LaTeX file;
2. Please insert your answers between each pair of `\begin{solution}` and `\end{solution}`;
3. For programming problems (explicitly marked as `Programming`, please complete the python programs as instructed.
4. Zip the files and submit to Canvas. Checklist: `hw2.pdf`, `wordvector.py`, `rnnpos.py`, and `model.torch`.

Problem 1: Count-based Word Vector

Most word vector models start from the following idea: "You shall know a word by the company it keeps" [1]. Many word vector implementations are driven by the idea that similar words, *i.e.*, (near) synonyms, will be used in similar contexts. As a result, similar words will often be spoken or written along with a shared subset of words, *i.e.*, contexts. By examining these contexts, we can try to develop embeddings for our words. With this intuition in mind, many "old school" approaches to constructing word vectors relied on word counts.

For your convenience, a jupyter notebook `hw2.ipynb` and a skeleton python program `wordvector.py` is provided for you. You may use it as needed.

1. (Programming) Please implement `get_vocabulary(corpus)` in `wordvector.py` to obtain the vocabulary and its size in the corpus.

2. (Programming) Please implement `get_co_occurrence_matrix(corpus, window_size)` in `wordvector.py`. It constructs a co-occurrence matrix for a certain window-size n, considering words n before and n after the word in the center of the window.

3. (Programming) Please implement `reduce_dimension(M, k)` in `wordvector.py`. It performs dimensionality reduction on the matrix to produce k-dimensional embeddings. Use it to take the top k components and produce a new matrix of k-dimensional embeddings.

4. (Programming) Please implement `cosine_similarity(vec1, vec2)` in `wordvector.py` to obtain the cosine similarity of two NumPy arrays. Find the top 5 words with the highest cosine similarity between their 2D vectors.

5. (Written) Run `plot_embeddings` in `hw2.ipynb` to plot a set of 2D vectors in 2D space. Name some clusters of words in 2-dimensional embedding space that you expected. Name some words that don't cluster together that you might think should have. What is the cosine similarity of these words?

Solution:

Answer for Q4: There were several word pairs with cosine similarity = 1. Here are some of them:
spinal cord
already-brewed canned
flattened spheres
sunlight full
stalks coriander

Answer for Q5: machine-technology, neurobiology-philosophy-cognition, ricotta-marinating-coconut, pizza-food are some of the clusters which were expected.
Some examples of words which don't cluster together but I feel should have are :
1. conversation-dialogue (cosine similarity = 0.9588)
2. coffee-beverage (cosine similarity = 0.9413)

Problem 2: Prediction-based Word Vector

More recently prediction-based word vectors, such as word2vec and GloVe, have demonstrated better performance. Prediction-based word vectors are generated by training a model to "predict" a word based on the surrounding context words or vice versa. In this problem, we will explore the embeddings produced by GloVe. You will load the GloVe vectors as instructed in `hw2.ipynb` and look into the properties of these vectors.

For your convenience, a jupyter notebook `hw2.ipynb` and a skeleton python program `wordvector.py` is provided for you. You may use them as needed.

1. (Written) Polysemes and homonyms are words that have more than one meaning. Find a word with at least two different meanings such that the top 10 most similar words (according to cosine similarity) contain related words from both meanings. For example, "leaves" has both "go_away" and "a_structure_of_a_plant" meaning in the top 10, and "scoop" has both "handed_waffle_cone" and "lowdown". You will probably need to try several polysemous or homonymic words before you find one. Please state the word you discover, copy the output, and describe the multiple meanings that occur at the top. Why do you think many of the polysemous or homonymic words you tried didn't work (*i.e.*, the most similar words only contain one of the meanings of the words)?

Solution:

Word: "open"
Output:
[('opening', 0.6419216394424438),
('round', 0.636039137840271),
('opens', 0.6174360513687134),
('opened', 0.5827721953392029),
('next', 0.5660090446472168),
('through', 0.5608602166175842),
('tournament', 0.5605131387710571),
('closed', 0.5601108074188232),
('tennis', 0.5498055815696716),
('set', 0.5496442914009094)]
Open can be a verb and a noun. Open (verb) basically means to open something, opening and opened have similar meanings.
Open (noun) is like US Open, French Open which are well-known Tennis tournaments which have multiple rounds in them.
We get to see all such multiple meanings of Open in "open"'s top 10 most similar words.

2. (Written) Analyze synonyms and antonyms. Follow the instructions in `hw2.ipynb` and find three words (w_1, w_2, w_3) where w_1 and w_2 are synonyms and w_1 and w_3 are antonyms, but Cosine Distance (w_1, w_3) < Cosine Distance (w_1, w_2). Please state the words you discover, copy the output, and give a possible explanation on why this counter-intuitive result may have happened.

Solution:

w_1: *close*
w_2: *near*
w_3: *far*
0.6234283 0.576759
Similarity of antonym pair CLOSE and FAR: 0.6234282851219177
Similarity of synonym pair CLOSE and NEAR: 0.5767589807510376

Here, *close* and *near* are synonyms, whereas *close* and *far* are antonyms. Yet, we can clearly see that *close* and *far* have a higher cosine similarity that *close* and *near*. This could be happening due to the limited data used for getting the embeddings. For instance, if our corpus has more sentences like *"Why do close friends live far away?"* etc., we might get such results. Moreover, the word "near" is usually used only for indicating temporal or spatial closeness - for example, *"His place is near to mine."* or *"The exams are near."*, but the word "close" has a lot wider usage and is also polysemous. Due to these reasons, I feel we are observing such a behavior.

3. (Programming) Exploring Bias in Word Vectors. Word vectors can sometimes inadvertently capture societal biases present in the text data on which they were trained. Using the GloVe vectors, identify at least one example where the word vectors might be capturing a gender, race,

or other bias. Follow the instructions in `hw2.ipynb` to quantify the degree of bias between pairs of words. Feel free to use any functions you just wrote and/or external packages. One method could be using cosine distances or cosine similarities of two words as an indicator of bias. You will need to report the pair of words you found, the degrees of bias, and a short explanation of its meaning.

Solution: I checked a few instances to identify gender biases in the data with the help of cosine similarities between the word vectors for *man* and *woman* against various professions. The first instance was checking cosine similarity between *woman-nurse* = 0.5646 whereas cosine similarity between *man-nurse* = 0.3576.
Based on this result it seems that if the word *nurse* occurs, the model has a higher chance of believing that the subject involved is a female.
Similarly, I checked for cosine similarity between *man-engineer* = 0.3912 whereas for *woman-engineer* = 0.2936.
This is another clear indicator of a gender bias where the model has a higher chance of tagging the subject involved with engineer as a male.

4. (Programming) Finding analogies with word vectors. Word vectors have been shown to sometimes exhibit the ability to solve analogies. Let m, g, w, and x denote the word vectors for man, grandfather, woman, and the answer, respectively. Please implement `estimate_analogy(m, g, w)` in `wordvector.py`, using only vectors m, g, w, and the vector arithmetic operators $+$ and $-$ in your answer. What is the expression in which we are maximizing cosine similarity with x?

5. (Written) Reflect on Problem 1 and 2, what are some of the limitations of these counting- and prediction-based word vectors? You may find some clues in [2].

Solution:

1. Counting-Based Methods:

Dimensionality and Sparsity: Counting-based methods often produce high-dimensional and sparse vectors, similar to the initial stage of static word representations mentioned. This high dimensionality can make them computationally expensive and memory-intensive.
Limited Contextual Information: Like static embeddings, counting-based methods may struggle with polysemy and capturing fine-grained contextual information. They treat all occurrences of a word as equivalent, which makes it challenging to distinguish between different meanings.
Data Dependency: They require large corpora of text data, which may not be available for all languages or domains.
Fixed Vocabulary: These models have a fixed vocabulary and may not handle out-of-vocabulary words effectively.

2. Prediction-Based Methods:

Lack of Interpretability: Similar to the static word embeddings, prediction-based methods may lack interpretability. It can be difficult to understand why a particular vector

represents a word in a specific way.
Polysemy Challenge: These methods typically assign a single vector to each word, making it challenging to handle polysemy effectively. If a word has multiple meanings, the same vector may not adequately capture all of them.
Short Context: These models have a limited context window, limiting their ability to capture long-range dependencies.
Training Data Bias: Quality and biases in the training data can impact the quality of embeddings.
Language Dependency: Many prediction-based models are language-specific and may not generalize well to other languages.

Problem 3: Recurrent Neural Network

If you are new to deep learning coding, please start early!

In this programming assignment, you will implement a part-of-speech (POS) tagger using a Recurrent Neural Network (RNN). For this assignment, the goal is to get you familiar with deep learning programming in PyTorch. **Please use the Python 3.8 and PyTorch 1.9 versions for consistency**. You are free to use any variations of RNN (*e.g.*, LSTM, etc.) to implement your POS tagger. You are also free to use any help you find online. **However, you cannot directly copy the code from online resources; you need to write your own code.**

Dataset

You are provided with 3 data files, which can be downloaded from the assignment directory on Canvas. **Please do not share the data**, as this is part of the Penn Treebank, which is licensed by Linguistic Data Consortium.

The format of the data is straightforward. Each word is followed by a tag. Punctuation marks are considered as words. **You will need to write code to pre-process the 3 data files.**

1. `wsj1-18.training`. This file contains the training data where each word is annotated with a POS tag. You will train your models using this data.
2. `wsj19-21.testing`. This file contains the testing data. You will apply the trained model to predict POS tags for each word.
3. `wsj19-21.truth`. This file contains ground-truth POS tags for the testing data. You will compare the results returned by your model with the ground-truth tags to evaluate model performance. You will need to report the accuracy of your model.

Note that in the testing data: `wsj19-21.testing`, infrequent words have already been replaced by the special unknown token `UNKA`. During training, you should map words occurring less than 3 times in the training data to the special token `UNKA`.

Hyper-parameter Tuning

In deep learning research, we typically experiment with several *hyper-parameters* when building models to ensure we achieve the best performance. **You must experiment with at least 3 different hyper-parameters**. For example, consider varying the following:

- Different word embeddings (dimension of embedding, pre-trained vs. learned)
- Size of the hidden layer (*i.e.*, vector dimensions in the hidden layer)
- Nonlinear activation functions (*e.g.*, ReLU vs. tanh)
- Types of optimizers (*e.g.*, Adam vs. SGD)
- Training mini-batch size
- Optimizer learning rate
- Number of training epochs
- Dropout rate

Be sure to record how the testing accuracy changes with these hyper-parameters and note the best accuracy achieved, as you will need to report your experimental results. Read more about documentation requirements in the last section of this problem.

Helpful Hints and Resources

A few notes and resources about RNN implementation:

- **New to PyTorch?** Some tutorials are great resources for beginners. See PyTorch's official guide and Prof. Justin Johnson's tutorials. The latter is based in Google Colab, but the code will look mostly the same.
- **Sentence representation and word vectors.** A sentence is represented as a sequence of numbers where each number is an index to a word in the vocabulary. You can use lookup to get the pre-trained embedding for each word. Alternatively, you can use one-hot vectors and compute embedded representations using matrix multiplication, but this may be slower. Use of a pre-trained model is encouraged.
- **GloVe pre-trained word embedding.** The GloVe pre-trained word embedding in Problem 2 can be downloaded from here, and comes in various vector sizes. For information on how to use GloVe embeddings in your code, please refer to this article.
- **Dealing with sentences with varied lengths.** Sentences have different lengths. You will have to use a fixed length. You can get the maximum length based on your training sentences. Once a length is determined, you will need to do padding if the input sentence is shorter than the length. To do the padding, essentially you just add "0" (index to a designated symbol for padding) to the end of your input sentence.
- **Data pre-processing.** You will need to spend some time on data pre-processing. A very nice tutorial on data pre-processing (*e.g.*, padding, mask, etc.) can be found on towards data science.
- **Saving and loading models.** Examples for storing and loading PyTorch models can be found by following PyTorch's official guide.

Grading and Code Expectations

Your code must be able to do the following:

1. Train the POS tagger and save it;
2. Load the model file for testing;
3. Apply the model to test data (given both the `wsj19-21.testing` and `wsj19-21.truth` files, and output the accuracy of the model on the test data).

You will find a skeleton code file `rnnpos.py` that imports some libraries you may need, handles command-line parameters, saving the trained model, and loading the model for testing. You need to provide code that trains and evaluates the model in the areas marked between "Your code starts here" and "Your code ends here"; some example lines are provided in these spaces, but they can be modified or removed. Some example code is provided, but you may add any other functions you need, and edit the `RNNTagger` model as needed. You should not need to modify the names or parameters of the `train`, `test`, or `main`, or the argument parsing code at the bottom.

For grading, your code must be able to handle the following commands, and utilize the arguments passed in, as they may change.

To train your model, we will run the following command, which should save the trained model as `model.torch`:

```
$ python3 rnnpos.py --train --model_file=model.torch --training_file=wsj1-18.training
```

To test your model, we will run the following command, which will load the pre-trained model from `model.torch`:

```
$ python3 rnnpos.py --data_file=wsj19-21.testing --label_file=wsj19-21.truth --model_file=model.torch
```

This command should output: `The accuracy is` ⟨*accuracy*⟩%, where ⟨*accuracy*⟩ is the testing accuracy of the model rounded to one decimal place, *e.g.*, "93.7". This command must be able to run without having to first run the training code, *e.g.*, by only providing the pre-trained model file.

(Programming) Submission of RNN Models

Your accuracy on the test data should be above 87% for full credit.

Please submit the following files:

- Your Python program implementing the RNN for POS tagging: `rnnpos.py`
- The trained model file for your **best** hyper-parameter configuration: `model.torch`

(Written) Submission of Documentation

Besides submitting your code and model as described above, you need to submit a report in this LaTeX file. This document should report the best testing accuracy of your RNN model and its corre-

sponding hyper-parameter configuration[1]. You should list the hyper-parameters you experimented with, and provide the results from each configuration you tried.

Solution: *Please write your report below.*

Model Architecture

I have built an LSTM model which has one LSTM layer and one Linear layer. The activation function used in the model is Tanh.

Note: It is mentioned on several websites that using tanh activation is the best for LSTMs. ReLU and others perform sub-optimally.

The model architecture details are as follows-
RNNTagger(
(embedding): Embedding(15573, 200)
(lstm): LSTM(200, 128, batch_first=True)
(fc): Linear(in_features=128, out_features=45, bias=True)
)
Trainable Params: 3,289,365

Best Model Configuration (Hyperparams) -

1. embedding_dim = 200
2. hidden_dim = 128
3. batch_size = 64
4. learning_rate = 0.001
5. num_epochs = 5
6. optimizer = Adam

train accuracy = 94.99%
test accuracy = 92.74%

Model Performance

Below is a table listing the performances of several other model configurations -

Sr.No.	batch_size	learning_rate	num_epochs	Optimizer	hidden layer size	train accuracy	test accuracy
1	64	0.01	5	Adam	128	95.88	92.5
2	32	0.01	5	Adam	128	95.31	92.44
3	128	0.01	10	Adam	128	97.25	92.29
4	32	0.01	10	Adam	128	95.76	92.26
5**	64	0.001	5	Adam	128	94.99	92.74**
6	64	0.001	10	Adam	128	97.00	92.60
7	64	0.01	5	Adam	256	96.04	92.49
8	64	0.001	5	Adamax	128	92.62	91.48
9	64	0.01	10	Adamax	128	97.2	92.62

A few notes on Model Architecture and Performance:

1. Increasing the batch_size sped up the training process by some time.
2. The accuracy scores received during training with SGD optimizer were below par and hence not mentioned in this report.
3. Increasing the epochs led to minor overfitting in many cases.

[1]For grading, we will verify that your reported accuracy matches what your code outputs when we run testing with the submitted model file.

4. The best performing model has is a good fit.
5. Increasing the hidden_size leads to an increase in the training time but also had some overfitting. This could be solved using some dropouts.
6. The accuracy can be improved further by using larger word embeddings and improving the model architecture - like using a Bidirectional LSTM.

A few notes on functioning and solution -

1. I have saved the text word_to_index and tags word_to_index dictionaries separately. I am submitting them along with other files in a zip folder. This is necessary if anyone tries to get the test accuracy without re-training the model on their systems.
2. I have trained the model on paddings as well but **NOTE**: The accuracy scores are calculated by ignoring the padded sequences. For calculation of the accuracy scores, I am only using the original labels from the truth file and comparing it with the unpadded predictions.

References

[1] Firth, John R. (1957). A synopsis of linguistic theory, 1930-1955. Studies in linguistic analysis, Basil Blackwell.

[2] Wang, Y., Hou, Y., Che, W. et al. (2020). From Static to Dynamic Word Representations: a Survey. nt. J. Mach. Learn. and Cyber. 11, 1611–1630. https://doi.org/10.1007