AI 534 Machine Learning HW4 (15 pts)

Instructions:

1. This homework is about using word2vec as features to train (averaged) perceptron (and optionally, other ML algorithms such as SVM) for sentiment classification, the same task we did in HW2. But instead of sparse features in HW2, we will use dense features from word embeddings here. The point of this homework is more about getting familiar with elementary deep learning.
2. Reuse HW2 data, which includes training, dev, and semi-blind test files.
3. Download word embeddings from: `https://classes.engr.oregonstate.edu/eecs/fall2023/ai534-400/unit4/hw4/embs_train.kv`
4. You should submit a single `.zip` file containing `hw4.pdf`, `test.txt.predicted`, and all your code. Again, LaTeX'ing is recommended but not required. Make sure your `test.txt.predicted` has the same format as `train.txt` and `dev.txt`.

1 Word Embeddings (5 pts)

In natural language processing (NLP), a word embedding is a continuous representation of a word. The embedding is used in text analysis. Typically, the representation is a real-valued vector that encodes the meaning of the word in such a way that words that are closer in the vector space are expected to be similar in meaning. For example, word2vec[1] is one of the most widely used word embeddings model. In this section, we will use the word embeddings pretrained on part of google news dataset (about 100 billion words) as features, and each word is mapped to a 300-dimentional vector . We extracted the embeddings of those words apearing in `train.txt` and save them in the file `embs_train.kv`.

1.1 Load and Query (0 pts)

We use the Python package `gensim` to load and query `embs_train.kv`. This package is already installed on ENGR servers (`ssh access.engr.oregonstate.edu`), but if you'd like to use your own Python, you might need to install it by `pip3 install gensim`.

```
>>> from gensim.models import KeyedVectors
>>> wv = KeyedVectors.load('embs_train.kv')
>>> wv
<gensim.models.keyedvectors.KeyedVectors object at 0x7fa9a5b3e320>
```

The word embeddings are loaded into `wv`, which is a `gensim KeyedVectors` object. Now, we can query the embedding of each word in sentence `the man bit the dog` .

```
>>> wv['big']
array([ 0.11132812,  0.10595703, -0.07373047,  0.18847656,  0.07666016,
       -0.3828125 , -0.0625    , -0.07470703,  0.05957031,  0.22167969,
        ...,
       -0.17675781, -0.08984375, -0.09667969, -0.11669922, -0.09082031,
       -0.02490234, -0.00509644, -0.07226562,  0.03735352, -0.15625   ],
      dtype=float32)
>>> wv['dog']
```

[1] `https://code.google.com/archive/p/word2vec/`

```
array([ 5.12695312e-02, -2.23388672e-02, -1.72851562e-01,  1.61132812e-01,
       -8.44726562e-02,  5.73730469e-02,  5.85937500e-02, -8.25195312e-02,
       ...,
       -2.75390625e-01,  2.61718750e-01,  2.46093750e-01, -4.71191406e-02,
        6.25000000e-02,  4.16015625e-01, -3.55468750e-01,  2.22656250e-01],
      dtype=float32)
```

1.2 Vector Similarity (2.5 pts)

We can query the most similar words for a given word based on "cosine similarity".

```
>>> wv.most_similar('dog', topn=10)
[('dogs', 0.8680489659309387), ('puppy', 0.8106428384780884),
('cat', 0.7609457969665527), ('canines', 0.7221246361732483),
('pet', 0.7164785861968994), ('collie', 0.6714409589767456),
('puppies', 0.6637065410614014), ('pug', 0.6611860990524292),
('terrier', 0.6599656343460083), ('poodle', 0.6549598574638367)]
>>> wv.most_similar('man', topn=10)
[('woman', 0.5792575478553772), ('boy', 0.49911412596702576),
('teenager', 0.4928569197654724), ('girl', 0.45247867703437805),
('toddler', 0.39077460765838623), ('thief', 0.3815022110939026),
('men', 0.3768376111984253), ('guy', 0.35656818747520447),
('someone', 0.3522713780403137), ('soldier', 0.34559640288352966)]
```

Q: Can you find the top-10 similar words to `wonderful` and `awful`? Do your results make sense? (1 pt)
Q: Also come up with 3 other queries and show your results. Do they make sense? (1.5 pts)

1.3 Word Analogy (2.5 pts)

The word vectors capture many linguistic regularities, for example vector operations

`vector('bigger') - vector('big') + vector('good')`

results in a vector that is very close to `vector('better')`, and

`vector('king') - vector('man') + vector('woman')`

is close to `vector('queen')`, which can be demonstrated as following.

```
>>> wv.most_similar(positive=['bigger', 'good'], negative=['big'], topn=5)
[('better', 0.7805920839309692), ('stronger', 0.607010543346405), ('worse', 0.56091910600662),
 ('decent', 0.5300450921058655), ('excellent', 0.5028262734413147)]
>>> wv.most_similar(positive=['king', 'woman'], negative=['man'], topn=5)
[('queen', 0.7118193507194519), ('monarch', 0.6189674139022827), ('princess', 0.590243101119),
 ('kings', 0.5236844420433044), ('queens', 0.5181134343147278)]
```

Q: Find top 10 words closest to the following two queries. Do your results make sense? (1 pt)

`sister - woman + man` and `harder - hard + fast`.

Q: Also come up with 3 other queries and show your results. Do they make sense? (1.5 pts)

2 Better k-NN and Perceptron using Embeddings (~~5.5~~ 6.5 pts)

2.1 Reimplement k-NN with Sentence Embedding (~~2.5~~ 3 pts)

We continue to treat each sentence as a collection of individual words, referred to as a "bag of words." To represent the sentence, we calculate the average of the embeddings of its constituent words. For example, the sentence embedding of `the man hit the dog` is computed as below.

```
>>> (wv['the'] + wv['man'] + wv['bit'] + wv['the'] + wv['dog'])/5
array([ 1.51464850e-01,  9.12353545e-02, -4.32617180e-02,  8.23242217e-02,
       -2.34497078e-02, -2.36328132e-02,  3.10546868e-02, -8.69018584e-02,
        ...,
       -1.01428226e-01,  4.88281250e-02,  1.11816404e-02, -4.30419929e-02,
        3.66699211e-02,  4.65820320e-02, -7.85644501e-02,  1.79199222e-02],
      dtype=float32)
```

Note that `wv` might not have embeddings for some words in `dev.txt` and `test.txt`; just ignore them. No need to prune one-count words (although that would be better). Also note 1-NN could be wrong (as in HW1).

1. For the first sentence in the training set (`+`), find a different sentence in the training set that is closest to it in terms of sentence embedding. Does it make sense in terms of meaning and label? (0.5 pts)
2. For the second sentence in the training set (`-`), find a different sentence in the training set that is closest to it in terms of sentence embedding. Does it make sense in terms of meaning and label? (0.5 pts)
3. Report the error rate of k-NN classifier on dev for $k = 1, 3, ...99$ using sentence embedding. You can reuse your code from HW1 or use `sklearn`. (~~1.5 pts~~ 1 pt)
4. Report the error rate of k-NN classifier on dev for $k = 1, 3, ...99$ using one-hot vectors from HW2. You can reuse your code from HWs 1-2 and/or use `sklearn`. (1 pt). (should be around 40%).

2.2 Reimplement Perceptron with Sentence Embedding (3 pts)

Now that each sentence has an embedding, you can reimplement perceptron and average perceptron using your new features.

1. For basic perceptron, show the training logs for 10 epochs (should be around ~~26~~ 33%). Compare your error rate with the one from HW2. (0.5 pts)
2. For averaged perceptron, show the training logs for 10 epochs (should be around 23%). Compare your error rate with the one from HW2. (0.5 pts)
3. Do you need to use smart averaging here? (0.5 pts)
4. For averaged perceptron after pruning one-count words, show the training logs for 10 epochs. Compare your error rate with the one from HW2. (0.5 pts)
5. For the above setting, give at least two examples on dev where using features of word2vec is correct but using one-hot representation is wrong, and explain why. (1 pt)
6. ~~For averaged perceptron after pruning one or two-count words, show the training logs for 10 epochs. Compare your accuracy with the one from HW2. (0.5 pts)~~

2.3 Summarize the error rates in a 2x2 table: {k-NN, perceptron} × {one-hot, embedding} (0.5 pts).

3 Try some other learning algorithms with sklearn (1.5 pts)

Now please try **one** other machine learning algorithm of your choice (either covered or not covered in our course) using `sklearn` (so that you don't need to implement them), such as k-NN, SVM, logistic regression, neural networks, decision trees, XGBoost, etc. To use `sklearn`, you will need to convert an `svector` object to a numpy vector. Note that this training might take a very long time, so you should prune one-count (and possibly two-count) words first. After pruning, the training might still take a long time. If you find it too slow, you can further prune more low-count words, and/or use a subset (e.g., 5,000) covered in the training set.

1. Which algorithm did you try? What adaptations did you make to your code to make it work with that algorithm? (0.5 pts)
2. What's the dev error rate(s) and running time? (0.5 pts)
3. What did you learn in terms of the comparison between averaged perceptron and these other (presumably more popular and well-known) learning algorithms? (0.5 pts)

4 Deployment (~~3~~ 2.5 pts)

Collect your best model and predict on `test.txt` to `test.txt.predicted`. As in HW1, part of your grade depends on your error rate on the semi-blind test set (~~2.75~~ 2.25 pts).

Q: what's your best error rate on dev, and which algorithm and setting achieved it? (0.25 pts)

5 Debriefing (required):

1. Approximately how many hours did you spend on this assignment?
2. Would you rate it as easy, moderate, or difficult?
3. Did you work on it mostly alone, or mostly with other people?
4. How deeply do you feel you understand the material it covers (0%–100%)?
5. Any other comments?