Northeastern University, Khoury College of Computer Science

CS 6120 — Assignment 5

Due: February 27, 2025 (100 points)

YOUR NAME + LDAP

We will make use of nearly 3 million arXiv papers that currently reside on http://arxiv.org. This set of papers has served the scientific community for over 30 years, and the subject matter is diverse, ranging from physics, math, computational biology, and yes, machine learning and artificial intelligence. The dataset `arxiv-titles-dataset.txt` that I provide is *not* shuffled presently; it is ordered according to topics, and it is formatted where each line in the text file is the title of a particular paper.

In class, we described several word embedding vector approaches, including:

- collaboratively filtered (SVD) embeddings
- the bag of words neural network
- the popular `word2vec` algorithm

Question 1: Pre-Processing

In many of the questions below, we will need the distribution of words, which can help us identify which words we want to create word embeddings for, and which words we want to ignore.

First, we will need to create the vocabulary of words that we can sample in our distribution. Since many of the scientific words are not used frequently, we can cut off those that do not appear in the corpus enough and those that appear too many times. (You decide what the thresholds are for these.)

Question 2: Matrix Factorization

One of the easiest ways to create dense representations of words is by using matrix factorization. In this question, we will need to create an adjacency matrix and then subsequently factorize it.

Q 2.1: Create an Adjacency Matrix

Create an adjacency matrix where entry (i, j) is the number of times word i co-occurs with word j in a single article title. To make computation feasible, use a window of maximum size 10, where you would increment .

Question 3: Bag of Words

In this section, we will be using a traditional neural network to predict the word context with a full cross-entropy classification cost function.

Q 3.1: Training Data Generation

The input vector can be context or content.

Q 3.2: Train Your Algorithm

Plot out your loss function

Q 3.3: Evaluate Your Neural Network

See how well you do.

Question 4: Word2Vec: Negative Sampling

In this section, we are going to implement the popular word2vec paper, which originally was written in C. There are two main components that make `word2vec` an effective optimization. These two components comprise the curricula for this week's homework assignment:

1. **Positive and Negative Sampling**
   - Skip-gram positive sampling from context.
   - Negative sampling from the word distribution
2. **Optimizing vectors in matrices V_i and V_o**
   - You will maximize similarity between v_i and v_o.
   - You will maximize the distance between v_i and each v_n

In *word2vec*, we optimize two sets of vectors, which we store in V_i and V_o. The **first set of vectors** is stored in the input matrix V_i, where v_i are the target vectors Each vector $\mathbf{v}_i$ is a *query* vector. For our particular dataset, when optimizing, we will be randomly sampling titles and any word in the title to obtain the vector $\mathbf{v}_i$. The **second set of vectors** is stored in the output matrix V_o. For every iteration, you will sample both a positive context vector $\mathbf{v}_o$ and multiple negative irrelevant vectors $\{\mathbf{v}_n\}$ from this matrix.

In order to optimize vectors in the matrices V_i and V_o, we will be pulling those vectors belonging to words that we say are similar closer to each other closer. Simultaneously, we will be pushing those vectors belonging to words that we say are dissimilar further apart. This process is called

noise contrastive estimation, because we are getting signal by contrasting our targets with noise.

Q 4.1: Positive and Negative Sampling

In the original paper, we obtain the *negative samples* from the distribution of words that does not contain any of the positively sampled words in a particular window. In practice, we can sample from the entire distribution of words, since the sampled words are unlikely to contain the positive samples. In this section, we will write two functions: `create_distribution` and `sample_distribution` to prepare us for training the `word2vec` algorithm.

- Write a function that builds the distribution of words
- Plot the distribution of words. This is called a Zipfian distribution.

Q 4.2: Gradient Derivation

In Mikolov's original paper in equation (4), the loss function for `word2vec` can be written as follows:

$$\mathcal{J}(\mathbf{v}_i, \mathbf{v}_o, \{\mathbf{v}_n\}) = \log \sigma(\mathbf{v}_i^T \mathbf{v}_o) + \sum_{\mathbf{v}_n \in P_n}^{k} \log \sigma(-\mathbf{v}_i^T \mathbf{v}_n) \tag{4.1}$$

where the target word is $\mathbf{v}_i$, a word in its context is is $\mathbf{v}_o$, and $\mathbf{v}_n$ are vectors that have been negatively sampled from the words that don't appear in the target word's context, i.e., $w_n \sim P_n(w)$. Derive the gradients with respect to $\boldsymbol{v}_i$ and $\boldsymbol{v}_o$, for any given i and o.

Write your solution to:

$$\begin{aligned} \frac{\partial \mathcal{J}}{\partial \mathbf{v}_i} &= ?? \\ \frac{\partial \mathcal{J}}{\partial \mathbf{v}_o} &= ?? \\ \frac{\partial \mathcal{J}}{\partial \mathbf{v}_n} &= ??, \quad \forall n \end{aligned} \tag{4.2}$$

Q 4.3: Gradient Implementation

Write a function that calculates the gradients of $\mathbf{v}_i$ and $\mathbf{v}_o$. Please use the following signature for our autograders.

```
def w2vgrads( vi, vo, Vns ):
  """
  This function implements the gradient for all vectors in
  input matrix Vi and output matrix Vo.

  Args:
```

```
    vi:  Vector of shape (d,), a sample in the input word
         vector matrix
    vo:  Vector of shape (d,), a positive sample in the output
         word vector matrix
    vns: Vector of shape (d, k), k negative samples in the
         output word vector matrix

  Returns:
    dvi, dvo, dVns: the gradients of J with respect to vi
                    and vo.
  """

  dvi, dvo, dVns = <YOUR-CODE-HERE>
  return dvi, dvo, dVns
```

Q 4.4: Noise Contrastive Estimation

Do gradient ascent (or descent, depending on your implementation) for several epochs. Play with your learning rate and see what works. You can track your objective function $\mathcal{J}$. To keep computations at a minimum while you iterate, one trick that you can do is to only calculate the objective function for the positive portion of $\mathcal{J}$.

Print out the ten nearest neighbors of the following five words and add it to your report: "neural", "machine", "dark", "string", and "black".
Save your word vectors to a numpy dictionary. There should be three files.

Submission Instructions