CMSC 25025 / STAT 37601

Machine Learning and Large Scale Data Analysis

Assignment 4*

Out: Thursday, April 27, 2017
Due: Tuesday, May 9, 2017 at 1:30 pm.

1. *Demystifying word2vec* (20 points)

   In this problem, we aim to understand the inner workings of word2vec, one of the most popular word embedding methods.

   The word2vec class of methods consist of two different approaches to learn word embeddings. Here we will focus on what's called the skip-gram method. See the word2vec Wikipedia page for more information.

   The specifics of the skip-gram method depend on the notion of a *context*. Given a sequence of words $w_1, w_2, \ldots, w_T$, the contexts of the t-th word w_t are the words in a symmetric window around w_t. For example, with a context window of size two, the contexts of the word "embeddings" in the sentence

   $$\text{I } \underbrace{\text{love word}} \text{ embeddings } \underbrace{\text{so much}}.$$

   are *love, word, so,* and *much*. The word-context pairs (w, c) associated with the word "embeddings" are therefore (embeddings, love), (embeddings, word), (embeddings, so), and (embeddings, much).

   Under this definition of context, both words and contexts are drawn from the same vocabulary V. Let D denote the set of observed word-context pairs in a given corpus. Let $\#(w, c)$ denote the number of times the pair (w, c) appears in D, and let $\#(w) = \sum_{c' \in V} \#(w, c')$ denote the number of times w appears in D. Similarly, $\#(c)$ is the number of times the context c occurs in D.

   Consider a word-context pair (w, c) and let $\mathcal{O}$ denote a random variable whose value is one if the pair (w, c) was observed in the corpus, and zero otherwise. In the skip-gram approach, this distribution is modeled as:

   $$\mathbb{P}(\mathcal{O} = 1 \mid w, c) = \sigma(v_w^T v_c) = \frac{1}{1 + e^{-v_w^T v_c}}$$

   where σ denotes the logistic function, and $v_w, v_c \in \mathbb{R}^d$ are d-dimensional word embeddings for the word w and the context word c, respectively.

*Compiled 2017/04/27 at 19:52:50

The skip-gram method aims to learn embedding vectors for all words and contexts so as to maximize $\mathbb{P}(\mathcal{O} = 1 \mid w, c)$ for pairs (w, c) found in the corpus, while maximizing $\mathbb{P}(\mathcal{O} = 0 \mid w, c)$ for randomly sampled negative examples, assuming that randomly sampling a context word for a given word is likely to result in an unobserved (w, c) pair. It has been shown empirically that this approach gives rise to similar word embeddings for words that share similar contexts (e.g. republican and democrat).

Explicitly, for a single observation (w, c), skip-gram with negative sampling maximizes the following objective:

$$\log\left(\sigma(v_w^T v_c)\right) + k\, \mathbb{E}_{v_{c_n}}\left[\log\left(\sigma(-v_w^T v_{c_n})\right)\right]$$

where k is the number of "negative" examples and c_n is the (negatively) sampled context, drawn according to the empirical distribution $\mathbb{P}(c) = \frac{\#(c)}{|D|}$. The global objective is therefore:

$$\ell = \sum_{w \in V} \sum_{c \in V} \#(w, c)\left(\log\left(\sigma(v_w^T v_c)\right) + k\, \mathbb{E}_{c_n}\left[\log\left(\sigma(-v_w^T v_{c_n})\right)\right]\right) \tag{1}$$

(a) Starting from Equation (1), show that for an arbitrary $\widehat{c}$,

$$\ell = \sum_{w \in V} \sum_{c \in V} \#(w, c) \log\left(\sigma(v_w^T v_c)\right) + \sum_{w \in V} k \cdot \#(w) \cdot \left(\frac{\#(\widehat{c})}{|D|} \log\left(\sigma(-v_w^T v_{\widehat{c}})\right) + \sum_{c_n \in V - \{\widehat{c}\}} \frac{\#(c_n)}{|D|} \log\left(\sigma(-v_w^T v_{c_n})\right) \right)$$

and conclude that the local objective for a specific (w, c) pair is

$$\ell(w, c) = \#(w, c) \log\left(\sigma(v_w^T v_c)\right) + k \cdot \#(w) \frac{\#(c)}{|D|} \log\left(\sigma(-v_w^T v_c)\right). \tag{2}$$

(b) Define $x = v_w^T v_c$ and substitute x into Equation (2). Take the derivative $\frac{\partial \ell}{\partial x}$, set it equal to zero, and *show all steps* needed to get the following equivalence:

$$\frac{\partial \ell}{\partial x} = 0 \tag{3}$$

$$\Rightarrow\ e^{2x} - \left(\frac{\#(w, c)}{k \cdot \#(w) \cdot \frac{\#(c)}{|D|}} - 1 \right) e^x - \frac{\#(w, c)}{k \cdot \#(w) \cdot \frac{\#(c)}{|D|}} = 0. \tag{4}$$

(c) Define $y = e^x$, substitute y into Equation (4), and solve for the roots of the resulting quadratic equation. Finally, by substituting y with e^x and x with $v_w^T v_c$, show that one of the roots occurs at

$$v_w^T v_c = \log\left(\frac{\#(w, c) \cdot |D|}{\#(w) \cdot \#(c)}\right) - \log k.$$

LSDA

2. *Word Embedding Experiments* (40 points)

The following experiments should be done with the Wikipedia corpus here:

```
/project/cmsc25025/wikipedia/wiki-text.txt
```

While it is not necessary, you may find it helpful to split this text into chunks to use with Spark, e.g.:

```
wiki_file =
  open('/project/cmsc25025/wikipedia/wiki-text.txt','r').readlines()[0]
num_chunks = 10 #make it multiple of number of your cores
chunks, chunk_size = len(wiki_file), len(wiki_file)/num_chunks
wiki_chunks =
  [wiki_file[i:i+chunk_size] for i in range(0, chunks, chunk_size)]
wiki_data = spark.sparkContext.parallelize(wiki_chunks)
```

The number of unique words in the Wikipedia corpus is too large for our purposes. Before proceeding, you should come up with a smaller vocabulary V that you will use for the remainder of the word embedding experiments. You can filter all of the unique words in the Wikipedia corpus in a number of ways. Here are some examples:

- Remove words that appear less than n times (e.g. try $n = 500$). You may use any existing python packages to compute word counts, for example `nltk.FreqDist` or `collections.Counter`. To speed things up, you might try computing word counts in parallel with Spark.
- Remove all words that appear in the `stopwords` list of `nltk` package.

You should aim to have roughly $15,000$ words in your vocabulary. In what follows, you may simply ignore words that do not appear in your final filtered vocabulary V.

PMI Embeddings

Following the results in (1), let $M \in \mathbb{R}^{|V|\times|V|}$ denote a matrix such that the (i,j)-th entry is the dot product between the word embedding vectors for the i-th and j-th words:

$$M_{ij} = v_{w_i}^T v_{w_j} = \log\left(\frac{\#(w_i, w_j)\cdot|D|}{\#(w_i)\cdot\#(w_j)}\right) - \log k.$$

The expression $\log\left(\frac{\#(w_i,w_j)\cdot|D|}{\#(w_i)\cdot\#(w_j)}\right)$ is known as the pointwise mutual information (PMI) of (w_i, w_j).

If we stack the word embedding vectors as rows of a matrix $W \in \mathbb{R}^{|V|\times d}$, then we have that $M = WW^T$. In (1), we have therefore shown (after sweeping some assumptions under the rug) that skip-gram with negative sampling is equivalent to factorizing the symmetric matrix M whose entries consist of the pointwise mutual information shifted by $-\log k$. In the following experiments, we will work with $k = 1$.

Note: the matrix M below will be quite large. For this part of the problem set, you might need to request an RCC node with more memory. 15GB should suffice.

(a) Using the Wikipedia data with a symmetric context window size of 5, compute the PMI matrix $M \in \mathbb{R}^{|V| \times |V|}$ whose (i, j)-th entry is the PMI of (w_i, w_j). To avoid degeneracy, add 1 to the cooccurence statistics $\#(w_i, w_j)$ so that

$$M_{ij} = \log \left(\frac{(\#(w_i, w_j) + 1) \cdot |D|}{\#(w_i) \cdot \#(w_j)} \right)$$

where D is the set of all word-context pairs observed in the Wikipedia corpus. Note that $\#(w_i, w_j)$ is simply the number of times w_i and w_j cooccur within 5 words of each other across the entire corpus.

(b) Now we will factorize the matrix M to obtain word embeddings. Take the k-SVD of M with $k = 50$ to obtain

$$M = U\Sigma V^T = \underbrace{U\Sigma^{\frac{1}{2}}}_{W} \underbrace{\Sigma^{\frac{1}{2}} V^T}_{C},$$

where $U \in \mathbb{R}^{|V| \times 50}$, $\Sigma \in \mathbb{R}^{50 \times 50}$, and $V^T \in \mathbb{R}^{50 \times |V|}$. You may use the `scipy` package to compute the k-SVD:

```
U, s, V =
  scipy.sparse.linalg.svds(scipy.sparse.csr(M), k=50)
```

(c) Let the rows of $W = U\Sigma^{\frac{1}{2}}$ be the learned PMI word embeddings. You will use these embeddings in several tasks below. We recommend that you serialize these embeddings to disk for later use (e.g., by using the `cPickle` package).

GloVe

In addition to word2vec, GloVe is another popular word embedding method.

(a) Clone the GloVe repository from github:

```
> git clone https://github.com/stanfordnlp/GloVe
```

(b) Inside the resulting GloVe directory, copy the `demo.sh` shell script to a new file called `wiki.sh`

```
> cp demo.sh wiki.sh
```

(c) Change the variables below in `wiki.sh` to match the following:

```
CORPUS=/project/cmsc25025/wikipedia/wiki-text.txt
VOCAB_FILE=wiki-vocab.txt
COOCCURRENCE_FILE=wiki-cooccurrence.bin
COOCCURRENCE_SHUF_FILE=wiki-cooccurrence.shuf.bin
SAVE_FILE=wiki-vectors
VOCAB_MIN_COUNT=50
WINDOW_SIZE=10
```

(d) Run the final `wiki.sh` script to obtain word embedding vectors using the GloVe method. The embeddings will be stored in the file named `wiki-vectors.txt`. We will call these your "local GloVe embeddings."

(e) Download the pretrained word vectors from the Wikipedia Gigaword 5 collection:

`http://nlp.stanford.edu/data/glove.6B.zip`

Use the 50-dimensional vectors in `glove.6B.50d.txt` for the experiments below. Just like the local embeddings file, the first token of each line is the word, and the remaining tokens are the components of the word embedding. We will call these your "pretrained GloVe embeddings."

Experiments

Conduct the following experiments with all of your embeddings: the PMI embeddings, the local GloVe embeddings, and the downloaded pretrained GloVe embeddings. Comment on the differences.

(a) To get started, here is an example of how to read your local GloVe embeddings in python:

```
vocab_file   = 'wiki-vocab.txt'
vectors_file = 'wiki-vectors.txt'

with open(vocab_file, 'r') as f:
    words = [x.rstrip().split(' ')[0] for x in f.readlines()]
with open(vectors_file, 'r') as f:
    vectors = {}
    for line in f:
        vals = line.rstrip().split(' ')
        vectors[vals[0]] = [float(x) for x in vals[1:]]

vocab  = {w: idx for idx, w in enumerate(words)}
ivocab = {idx: w for idx, w in enumerate(words)}
```

Then, for example, to obtain the word embedding for the word `umbrella`, you could run the following:

```
vectors['umbrella']
```

(b) For each of the following words, find the 5 closest words in the embedding space: `physics`, `republican`, `einstein`, `algebra`, `fish`. Report and comment on your results and the differences between the three sets of embeddings.

(c) A surprising consequence of some word embedding methods is the resulting linear substructure. This structure can be used to solve analogies like

`france : paris :: england : ?`

You might find that the pretrained embeddings work best here because they were fit on a larger corpus. Solve this analogy with every set of embeddings by computing the nearest embedding vector to v where v is

$$v = v_{\text{paris}} - v_{\text{france}} + v_{\text{england}}$$

(d) Define 10 analogies `X:Y=Z:W`. Post your analogies to Piazza in the following format (all lower case):

```
republican:democrat=conservative:?
dog:cat=puppy:?
...
```

Try out the embeddings to complete all of the posted analogies. Compare the different results with the different embeddings.

(e) Use the t-SNE dimensionality reduction technique to visualize only the *pretrained GloVe embeddings* in two dimensions. Using the code below in a Jupyter notebook, you can interact with the plot by zooming in and out. Explore the plot until you find a clustering of words that makes sense. Turn off the interactive plot by hitting the "power button" in the top right so that the graders can see the final product. Comment on the final area you landed on.

Assuming the variable `W` contains the embedding vectors as rows, and `words` contains the word labels in the same order as the rows of `W`, the following code will produce an interactive plot of the first 1000 embedding vectors:

```
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
%matplotlib notebook

tsne = TSNE(n_components=2, random_state=0)
np.set_printoptions(suppress=True)
Y = tsne.fit_transform(W[:1000,:])

plt.scatter(Y[:, 0], Y[:, 1])
for label, x, y in zip(words, Y[:, 0], Y[:, 1]):
    plt.annotate(label, xy=(x, y), xytext=(0, 0),
        textcoords='offset points')
plt.show()
```

LSDA

3. *Song and Artist Embeddings Experiments* (40 points)

In this problem, we will use a collection of playlists and tags obtained from Yes.com and Last.fm. We will treat each playlist as a document, and each song (or artist) in the playlist as a word. By feeding this dataset to GloVe, we will be able to learn song (or artist) embeddings.

Song Embeddings

The file `/project/cmsc25025/playlist/song-playlists.txt` contains the playlist data. These data were obtained from Yes.com. Each line in this file is a playlist of songs. Each unique number corresponds to a song ID. The map from song IDs to the song name and artist is in `/project/cmsc25025/playlist/song_hash.txt`.

(a) Inside the resulting GloVe directory, copy the `demo.sh` shell script to a new file called `song.sh`

```
> cp demo.sh song.sh
```

(b) Change the following variables in `song.sh` to match the values below:

```
CORPUS=/project/cmsc25025/playlist/song-playlists.txt
VOCAB_FILE=song-vocab.txt
COOCCURRENCE_FILE=song-cooccurrence.bin
COOCCURRENCE_SHUF_FILE=song-cooccurrence.shuf.bin
SAVE_FILE=song-vectors
VOCAB_MIN_COUNT=0
WINDOW_SIZE=750
```

(c) Run the final `song.sh` script to obtain song embedding vectors using the GloVe method. The vectors will be saved in a file called `song-vectors.txt`. You can read these vectors using code similar to the example given in the word embeddings question.

Important: As you can see from the previous code, the first token in the vectors file is the "word." In our case, the words are song IDs. Do not mistake the first token as a coordinate of your embedding vector!

(d) Each song in this dataset is also associated with a corresponding set of tags obtained from Last.fm. This file contains the mapping from song ID to tag IDs:

`/project/cmsc25025/playlist/song-to-tags.txt`

On each line, the song ID is the first token, followed by a tab, followed by a comma-separated list of tag IDs for that song.

This file contains the mapping from tag ID to tag:

`/project/cmsc25025/playlist/tag_hash.txt`

Perform k-means on the song embeddings with $k = 5$. For each cluster, print the top 5 most commonly occurring tags and comment on the results.

(e) Similar to the word embeddings question, use the t-SNE dimensionality reduction technique to visualize the song embeddings. Find an area of the plot that makes sense, save the plot for the graders, and discuss the area you landed on.

Artist Embeddings

This file contains the same playlist data as the previous question, except the song IDs have been replaced with artist IDs:

```
/project/cmsc25025/playlist/artist-playlists.txt
```

This file contains the map from artist IDs to artists:

```
/project/cmsc25025/playlist/artist_hash.txt
```

(a) Inside the resulting GloVe directory, copy the `demo.sh` shell script to a new file called `artist.sh`

```
> cp demo.sh artist.sh
```

(b) Change the following variables in `artist.sh` accordingly:

```
CORPUS=/project/cmsc25025/playlist/artist-playlists.txt
VOCAB_FILE=artist-vocab.txt
COOCCURRENCE_FILE=artist-cooccurrence.bin
COOCCURRENCE_SHUF_FILE=artist-cooccurrence.shuf.bin
SAVE_FILE=artist-vectors
VOCAB_MIN_COUNT=0
WINDOW_SIZE=750
```

(c) Run the final `artist.sh` script to obtain artist embedding vectors using the GloVe method. The embeddings will be stored in the file named `artist-vectors.txt`.

(d) Find the 5 closest artist embedding vectors to the following artists: `'The Beatles'`, `'The Red Hot Chili Peppers'`, and `'Usher'`. Comment on the results.

(e) Similar to the word embeddings question, use the t-SNE dimensionality reduction technique to visualize the artist embeddings. Find an interesting area of the plot, save the plot for the graders, and discuss the area you landed on.

`chisubmit` instructions

Three `ASSIGNMENT_ID`s will be created in the `chisubmit` system: `a4` (Problem 1), `a4words` (Problem 2) and `a4songs` (Problem 3)

Please name your submissions

- `assn4_theory.pdf` for `a4`
- `assn4_prob2.ipynb` for `a4words`
- `assn4_prob3.ipynb` for `a4songs`.

If you work on LSDA problems with a partner, you need to register `a4words` and `a4songs` using the `chisubmit --partner` option. Please see the details at `http://chi.cs.uchicago.edu/chisubmit/students.html`.