

Word Vectors: Distributed Representations of Words

Name: Ju Hyun Kim (934488882)

Overview and Objectives. This assignment will cover the basics of processing text (simple tokenization and vocabulary building) and distributional representations of words. We will learn our own word vectors from a training corpus and examine what relationships have been learned in our small-scale problem. Finally, we will take a look at word2vec [1] word vectors learned on large-scale data and identify some implicit biases captured in training.

How to Do This Assignment. The assignment walks you through completing the provided skeleton code and analyzing some of the results. Anything requiring you to do something is marked as a "Task" and has associated points listed with it. You are expected to turn in both your code and a write-up answering any task that requested written responses. Submit a zip file containing your completed skeleton code and a PDF of your write-up to Canvas.

To ensure you have the right packages, run `pip install -r requirements.txt` in the skeleton code folder.

Advice. Start early. Students will need to become familiar with `numpy` and `matplotlib` for this and future assignments.

1 Getting Used to Handling Language Data [10pt]

We will be using the AG News Benchmark [2] dataset as our data source for this homework. AG News is a collection of over a million news articles scraped from over 2000 news sources. The AG News Benchmark [2] is a 120,000 article subset of these covering four topics (World / Sports / Business / Technology) and is used as a benchmark for text classification. The benchmark retains only the article titles and overviews. The skeleton code file `build_freq_vectors.py` already loads this dataset based on the HuggingFace dataset package API (`pip install datasets`). It will download the dataset the first time it is run.

```
1 from datasets import load_dataset
2 ...
3 dataset = load_dataset("ag_news")
```

Listing 1: Code to load the AG News Dataset

The dataset structure returned looks like this

```
1 DatasetDict({
2   train: Dataset({
3     features: ['text', 'label'],
4     num_rows: 120000
5   })
6   test: Dataset({
7     features: ['text', 'label'],
8     num_rows: 7600
9   })
10 })
```

Listing 2: AG News Dataset structure

and contains article text and label for 120,000 training instances and 7600 test instances.

1.1 Building a Token Vocabulary

Tokenization. It is often easier to consider text as a sequence of discrete tokens. The process of converting a string to such a sequence is known as tokenization. Given a sentence like "The quick, brown fox jumped over the lazy dog", one possible tokenization is to ignore punctuation and capitalization then break on white-space as below.

```
1 > vocab.tokenize("The quick, brown fox jumped over the lazy dog.")
2 ["the", "quick", "brown", "fox", "jumped", "over", "the", "lazy", "dog"]
```

Basing a system on these tokens is fine, but it is a design decision. For instance – conjugations such as jumped, jump, and jumps would all be different tokens despite having common root meanings, as would any misspellings (jumpe). Alternative tokenizations include sub-word tokenization (jumped → [jump, ed]) or pre-processing with *lemmitization* to reduce words to their roots (jumped → [jump], are → [be]). Lemmitization has its own drawbacks – possibly merging different word-senses, potentially resulting in poor specificity (i.e. can't tell jump from jumping).

► **TASK 1.1 [2pt]** Implement the `tokenize` function in `Vocabulary.py` that processes a string into an array of strings corresponding to tokens. You are free to implement any tokenization schema that eliminates punctuation. If you want to try out lemmitization, the `nlTK` package may be helpful. In your writeup for this question, include a description of what you implemented and include an example input-output pair.

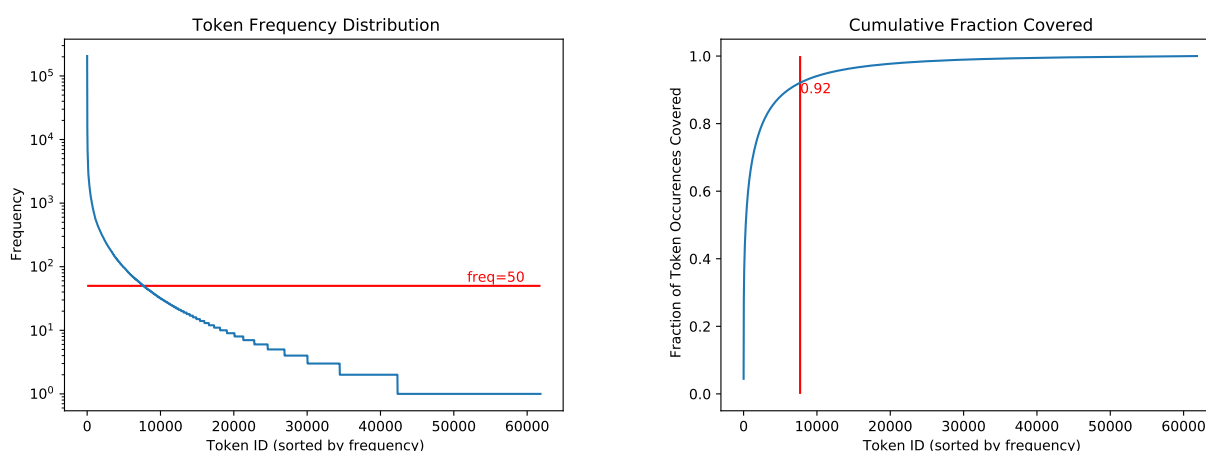
► **Answer 1.1**

- **Punctuation Removal:** The function strips punctuation from the text, ensuring that words are separated purely based on whitespace and are not attached to any punctuation marks.
- **Lowercasing:** All characters in the text are converted to lowercase. This standardizes the tokens so that words at the beginning of a sentence or in all caps are treated the same as those in lowercase.
- **Splitting:** Splits the text into tokens

Input: "The quick, brown fox jumped over the lazy dog."

Output: ["the", "quick", "brown", "fox", "jumped", "over", "the", "lazy", "dog"]

Vocabulary. Estimates suggest there are millions of words in active use in English; however, we are unlikely to encounter the vast, vast majority of them in our limited dataset. For efficiency, it is common practice to consider only frequent tokens in the training corpus – common heuristics include taking the most frequent k tokens, taking only tokens that occur at least n times, or taking as many words as necessary to cover $X\%$ of the total token occurrences.



The above plots show the effect of cutting off our vocabulary by selecting only tokens with greater than 50 occurrences. The frequency distribution plot (left) has a log-scale for the y-axis and demonstrates the *long-tail* property of language. A few tokens are used very frequently but very many tokens are used rarely. The right plot shows the cumulative fraction of word occurrences covered. That is to say, if the vocabulary is cut off at the i th word on the x-axis, the curve shows what fraction of the total number of tokens appearing in the training set (including all instances) would have a corresponding word in the vocabulary. The line marked in red corresponds to a `freq=50` cutoff and captures 92% of the tokens in the dataset. As is common practice, we consider the other 8% of tokens (and any new ones we see at test time) to correspond to a special UNK token in our vocabulary.

Finally, it is often easier to deal with sequences of tokens as indexes into our vocabulary rather than as raw text (We'll see this in the next section and in later homeworks). That is, if we have the following vocabulary:

```
1 ["the", "brown", "fox", "dish", "spoon", "over", "jumped", "lazy", "dog", "UNK"]
```

Then the index-represented tokenization of "The quick, brown fox jumped over the lazy dog" would be:

```

1 > vocab.textToIdx(vocab.tokenize("The quick brown fox jumped over the lazy dog."))
2 [0, 9, 1, 2, 6, 5, 0, 7, 8]
3
4 > vocab.idxToText([0, 9, 1, 2, 6, 5, 0, 7, 8])
5 ["The", "UNK", "brown", "fox", "jumped", "over", "the", "lazy", "dog"]

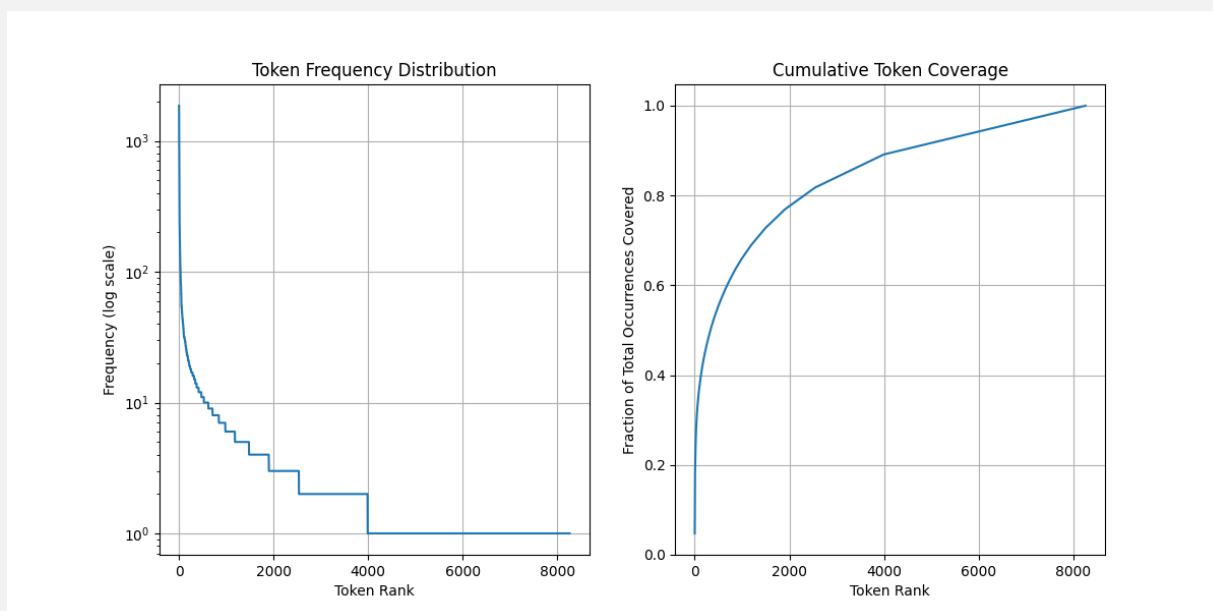
```

Notice that the vocabulary did not include the word quick and so this token was mapped to UNK. To make these operations simple, most implementations build two indexes – word2idx and idx2word – which map between the two representations via a simple lookup.

► **TASK 1.2 [3pt]** Implement the `build_vocab` function in `Vocabulary.py` which constructs a finite vocabulary from a string containing all the text from the training corpus. This includes implementing some heuristic for thresholding the number of words and building the `word2idx` and `idx2word` indexes.

► **TASK 1.3 [5pt]** Implement the `make_vocab_charts` function in `Vocabulary.py` to produce Token Frequency Distribution and Cumulative Fraction Covered charts like those above for your tokenizer and vocabulary cutoff heuristic. We recommend `matplotlib` for this. Afterwards, running `build_freq_vectors.py` will generate these plots. In your write-up for this question, include these plots and briefly describe the cutoff heuristic you implemented and explain your rationale for setting it.

► **Answer 1.3**



I tried with the high value frequency, but in such cases, the quantity of data decreased, making it difficult to discern relevance among the data. With too few words remaining, the characteristics of the corpus were not well represented, and there was a risk that models or analyses would not reflect general language use accurately. Therefore, after a more careful approach, I decided to set the vocabulary to include only those words that appear more than 5 times in the corpus. By doing this, I ensured both vocabulary diversity and sufficient data, while also preventing excessively rare words from introducing unnecessary noise into the model.

2 Frequency-Based Word Vectors – PPMI [20pts]

As we discussed in class, the distributional hypothesis suggests a word's meaning can be derived from how it is used with other words in natural language. In order to follow this philosophy, we must first define some notion of the relationship between words in text. One option is to use raw co-occurrence statistics, but as we noted in class this results in large, sparse vectors and frequent words can dominate similarity calculations.

2.1 Pointwise Mutual Information (PMI) Matrix

First used in NLP in 1990 [3], pointwise mutual information (PMI) has been widely applied in word similarity tasks. Let $p(w_i, w_j)$ be the probability of tokens w_i and w_j co-occurring in any context and let $p(w_i)$ be the probability of observing token w_i in any context. We can then define the pointwise mutual information between tokens w_i and w_j as:

$$\text{PMI}(w_i, w_j) = \log \frac{p(w_i, w_j)}{p(w_i)p(w_j)} \quad (1)$$

Commonly, practitioners will clip PMI values less than zero. The resulting function is referred to as Positive Pointwise Mutual Information (PPMI) and can be written as $\text{PPMI}(w_i, w_j) = \max(0, \text{PMI}(w_i, w_j))$.

► **TASK 2.1 [5pt]** What are the minimum and maximum values of PMI (Eq. 1)? If two tokens have a positive PMI, what does that imply about their relationship? If they have a PMI of zero? What about if the PMI is negative? Based on your answers, what is an intuition for the use of PPMI?

► **Answer 2.1**

The minimum PMI value is negative infinity and for the maximum value of PMI is determined by the lowest possible probability of a word occurring by itself, which in practice is never zero and also it does not reach to positive infinity such as below.

$$-\infty \leq \text{pmi}(x; y) \leq \min \{-\log p(x), -\log p(y)\}.$$

If two tokens have a positive PMI, it indicates a stronger than expected co-occurrence, suggesting a meaningful relationship between the tokens. With PMI of zero, it mean no particular association. Negative PMI will implies less frequent co-occurrence than by chance, suggesting no relationship or a pattern of avoidance. Using PPMI over PMI emphasizes positive associations and reduces noise from negative values. Therefore it leads to more informative data representation for the NLP tasks.

Estimating PPMI. Computing an estimate for PPMI from a dataset is straight-forward given co-occurrence statistics. Consider a dataset of N text contexts. A context could be a whole document, paragraph, sentence, or even small windows in text (e.g. "The dog jumped over the moon." \rightarrow "The dog jumped", "dog jumped over", "jumped over the", "over the moon"). The scale of these contexts determine our notion of relatedness. Often word vector learning algorithms will use a small window of text.

Let C be a matrix of co-occurrence counts such that the ij th element of C denoted C_{ij} is the number of times both w_i and w_j occur in a context. Note that the diagonals of this matrix (C_{ii}) count the number of contexts in which each token occurs. We can then compute

$$\text{PPMI}(w_i, w_j) = \max \left(0, \log \frac{C_{ij}N}{C_{ii}C_{jj}} \right) \quad (2)$$

We can store these PPMI values in a matrix P such that $P_{ij} = \text{PPMI}(w_i, w_j)$.

► **TASK 2.2 [2pt]** Implement the `compute_cooccurrence_matrix` function in `build_freq_vectors.py` which takes a list of article overviews and a vocabulary and produces a co-occurrence matrix C . It is up to the student to define what a context is. Note that looping in Python is quite slow such that unoptimized versions of this function can take quite a long time. Feel free to save the result while you are developing to reduce time in future runs (see `numpy.save/numpy.load`). In your writeup for this task, describe how you defined your context.

► **Answer 2.2**

To construct a co-occurrence matrix for analyzing word relationships within a text corpus, begin by initializing a zero matrix with dimensions $n \times n$, where n represents the number of words in the vocabulary. This setup facilitates the tracking of how frequently pairs of words occur near each other.

Each document in the dataset is processed by first tokenizing the text, then employing a sliding window (with a window size of 4) to count how frequently each vocabulary word appears near every other word. Updates to the matrix are made symmetrically to both $C[i][j]$ and $C[j][i]$ to reflect the bidirectional nature of word relationships.

Given the computational demands associated with large datasets, particularly to prevent memory overflow, the dataset's scope is deliberately limited. This is accomplished by constructing the vocabulary from just the first 1000 documents (`vocab = Vocabulary(dataset_text[: 1000])`), thereby making the process manageable while still capturing significant lexical patterns.

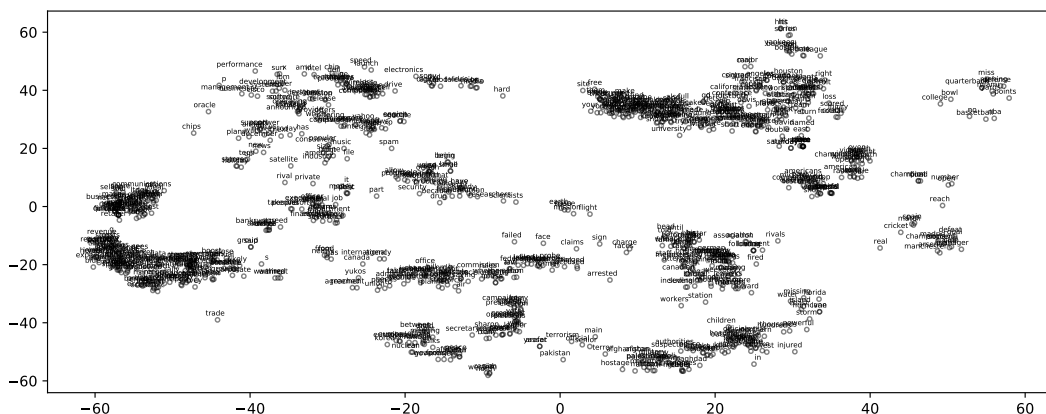
► **TASK 2.3 [3pt]** Implement the `compute_ppmi_matrix` function in `build_freq_vectors.py` which calls `compute_cooccurrence_matrix` and then computes a PPMI matrix given a list of article overviews and a vocabulary. *Hint: Add a small constant to C to avoid problems with $\log(0)$.*

Our First Word Vector. A single row $P_{i\cdot}$ of this PPMI matrix corresponds to a vector of PPMI values between token w_i and all other tokens. This is our first distributed word vector; however, it is quite large and mostly sparse such that distances between words vectors in this high of a dimension are unlikely to be very meaningful.

2.2 Dimensionality Reduction and Visualization

Dimensionality Reduction with Truncated SVD. To reduce the dimensionality of our word vectors, we apply Truncated Singular Value Decomposition. Truncated SVD of a matrix A finds a k -rank matrix D such that $\|D - A\|_F$ is minimized. Specifically, D has the form $D = U\Sigma V^T$ where Σ is a diagonal matrix of the k largest singular values. Following [4], we take our final word vectors as $[U\Sigma^{1/2}, V\Sigma^{1/2}]$ where $[\cdot, \cdot]$ denotes concatenation. This is already implemented in the `dim_reduce` function in `build_freq_vectors.py`.

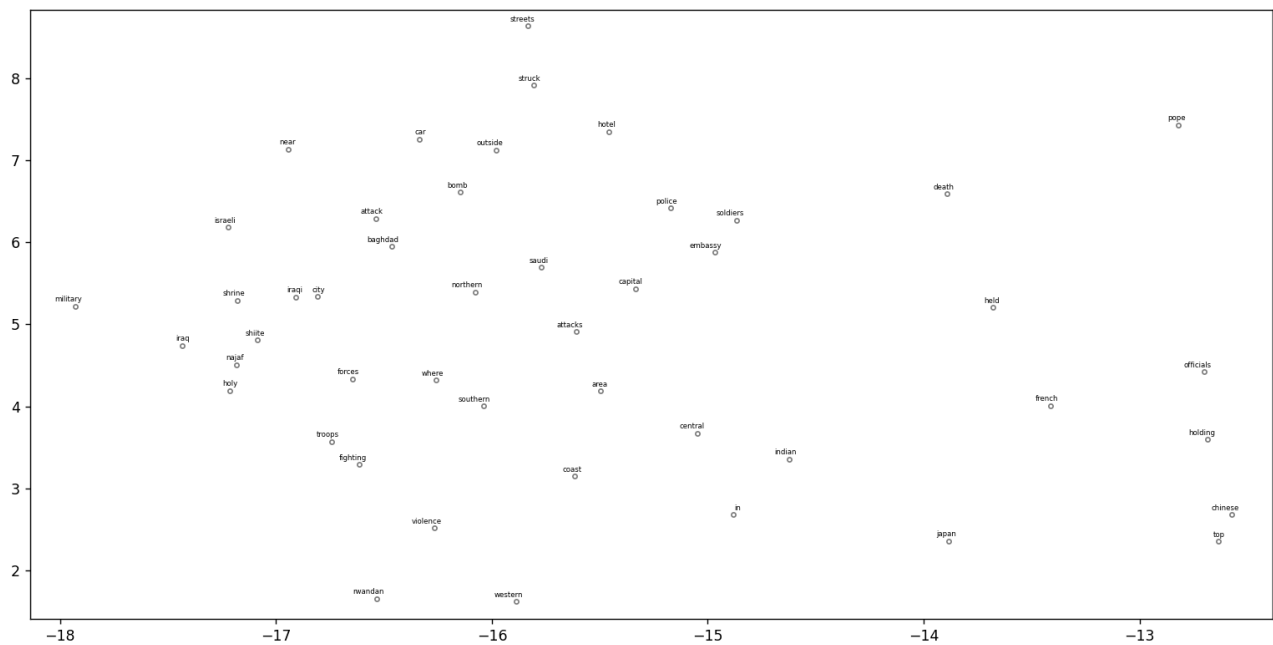
Visualizing Word Vectors. To visualize our word vectors in 2D, the code applies T-SNE [5]. T-SNE is a non-linear dimensionality reduction method commonly used to display word vectors and other high-dimensional data. The `plot_word_vectors_tsne` function in `build_freq_vectors.py` is already set up to produce this visualization. An example visualization from our run is below.



It is pretty messy at this resolution but zooming in on the clusters shows the semantics we've learned. For instance, the small middle-right cluster corresponds to soccer and includes words like "*arsenal, manchester, madrid, soccer, england, side, club, manager, goal, defeat, spain, cup, match, ...*". If you make different choices about tokenization, vocabulary, or the scale of your contexts than we did – your generated plot will be different.

► **TASK 2.4 [10pt]** It has all led up to this! Run `build_freq_vectors.py` and examine the visualized word vectors. You should have semantically meaningful clusters in your plot. Zoom in and identify three such clusters. In your write-up, include an image of each cluster and what category you think it is representing. Also include the plot as a whole.

► Answer 2.4



3 Learning-Based Word Vectors – GloVe [25pts]

As discussed in lecture, an alternative approach is to assume word co-occurrence can be modelled purely as an interaction between word vectors. This approach is taken by popular algorithms like word2vec[1] and GloVe [6]. In this section, we'll implement the GloVe algorithm in our setting.

The GloVe algorithm poses the problem as a weighted log bilinear regression problem wherein interactions between word vectors should be predictive of the log of their co-occurrence frequency. As before, let C be a matrix of co-occurrence counts. The GloVe objective can be written as a sum of weighted squared error terms for each word-pair in a vocabulary,

$$J = \sum_{i,j \in V} \underbrace{f(C_{ij})}_{\text{weight}} \underbrace{(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log C_{ij})^2}_{\text{error term}} \quad (3)$$

where each word i is associated with word vector w_i , context vector \tilde{w}_i , and word/context biases b_i and \tilde{b}_i . Unlike word2vec which we studied in class, this objective does not sum over a corpus, but rather over pairs of words in the vocabulary. The $f(C_{ij})$ term is a weighting to avoid frequent co-occurrences from dominating the objective and is defined as

$$f(X_{ij}) = \min(1, C_{ij}/100)^{0.75} \quad (4)$$

To minimize this objective, `build_glove_vectors.py` implements a simple stochastic gradient descent with momentum optimization algorithm; however, it is missing the expressions for the gradients. Given a batch of word-pairs $B = \{(i_m, j_m)\}_{m=1}^M$, we can write the per-batch objective as:

$$J_B = \sum_{(i_m, j_m) \in B} f(C_{i_m j_m})(w_{i_m}^T \tilde{w}_{j_m} + b_{i_m} + \tilde{b}_{j_m} - \log C_{i_m j_m})^2 \quad (5)$$

► **TASK 3.1 [10pts]** Derive the gradient of the objective J_B in Eq. 5 with respect to the model parameters w_i , \tilde{w}_j , b_i , and \tilde{b}_j . That is, write the expression for $\nabla_{w_i} J$, $\nabla_{\tilde{w}_j} J$, $\nabla_{b_i} J$, and $\nabla_{\tilde{b}_j} J$. Note that parameters corresponding to words not in B will have zero gradient.

► **Answer 3.1**

$$\begin{aligned}\nabla_{w_i} J &= f(C_{ij}) \cdot 2 \cdot \left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log C_{ij} \right) \cdot \tilde{w}_j \\ \nabla_{\tilde{w}_j} J &= f(C_{ij}) \cdot 2 \cdot \left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log C_{ij} \right) \cdot w_i \\ \nabla_{b_i} J &= f(C_{ij}) \cdot 2 \cdot \left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log C_{ij} \right) \\ \nabla_{\tilde{b}_j} J &= f(C_{ij}) \cdot 2 \cdot \left(w_i^T \tilde{w}_j + b_i + \tilde{b}_j - \log C_{ij} \right)\end{aligned}$$

The training code in `build_glove_vectors.py` samples a batch of word interactions for each iteration – ignoring zero values in C as $f(0) = 0$. This corresponds to a set of indices $\{i_b, j_b\}_{b=1}^M$ denoted in the code as vectors i and j . A snippet from `build_glove_vectors.py` below shows how the f value and error term are computed as matrix operations for all elements in the batch simultaneously. This is significantly faster than iterating over every instance. Note that both `fval` and `error` are $|B| \times 1$ vectors with one value for each element in the batch.

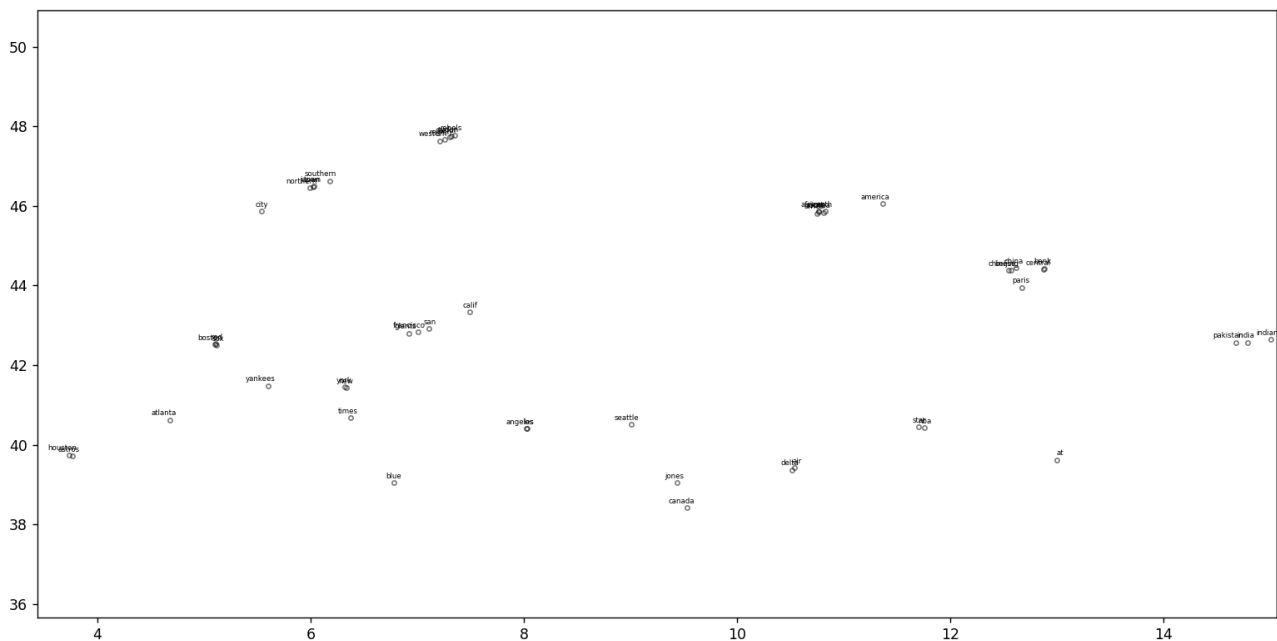
```
1 i = idx[shuf_idx, 0] #vector of indices of word vectors in the batch
2 j = idx[shuf_idx, 1] #vector of indices of context vectors in the batch
3
4 # Get word and context vectors for pairs in the batch
5 w_batch = wordvecs[i, :]
6 c_batch = contextvecs[j, :]
7
8 # Compute f(C_i,j) for i,j pairs in batch (Bx1 dimensional vector)
9 fval = (np.minimum(1, C[i, j]/100)**0.75)[: ,np.newaxis]
10
11 # Compute error term as (w_i^T \tilde{w}_j + b_i + \tilde{b}_j - log(C_ij)) for each i,j
    pair in the batch. (Bx1 dimensional vector)
12 error = (np.sum(np.multiply(w_batch, c_batch), axis=1)[: ,np.newaxis] + wordbiases[i] +
    contextbiases[j] - np.log(C[i, j])[: ,np.newaxis])
13
14 # Compute the overall objective loss
15 loss += np.sum(fval*np.square(error))
```

Just below this, there is a stub section for implementing your derived gradients for elements of the batch. That is to say that the row `wordvecs_grad[b, :]` should contain $\nabla_{w_{i_b}} J$ and likewise `contextvecs_grad[b, :]` $\leftarrow \nabla_{\tilde{w}_{j_b}} J_B$. The `np.zeros` lines show the expected dimensions of the gradients corresponding to this API.

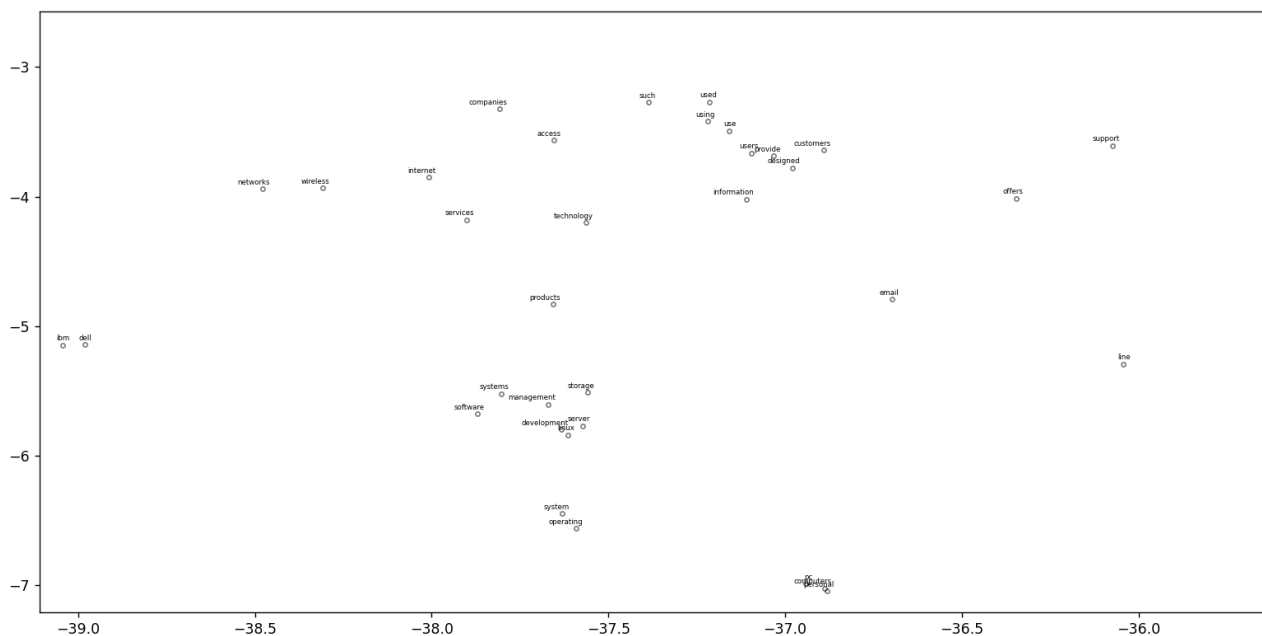
```
1 #####
2 # Task 3.2
3 #####
4 # REMOVE THIS ONCE YOU IMPLEMENT THIS SECTION
5 raise UnimplementedFunctionError("You have not yet implemented the gradient computation.")
6
7 # write expressions using numpy to implement the gradients you derive in 3.1.
8 wordvecs_grad = np.zeros( (bSize,d) )
9 wordbiases_grad = np.zeros( (bSize,1) )
10 contextvecs_grad = np.zeros( (bSize,d) )
11 contextbiases_grad = np.zeros( (bSize,1) )
12 #####
```

► **TASK 3.2 [10pts]** Implement the gradient computation for a batch in the corresponding Task 3.2 section of `build_glove_vectors.py`.

Once the gradients have been implemented, running `build_glove_vectors.py` should build a vocabulary, compute a co-occurrence matrix, and then start optimizing the GloVe objective via SGD with momentum. The code passes over the data five times (5 epochs) and will print the average objective value every 100 batches as shown below.



cluster about Location: seattle, canada, atlanta, at, etc.



cluster about Technology: software, system, server, etc.

4 Exploring Learned Biases in word2vec Vectors [10pts]

As shown in [4, 7], the two approaches we just applied – truncated SVD of a PPMI matrix and optimizing the GloVe objective – are optimize the same objective theoretically. This is also true for the widely-used word2vec algorithm. In this part, we examine relationships learned by word2vec vectors when trained on very large corpuses.

Many packages provide implementations of word2vec. As loading the vectors can be slow, we recommend you use an interactive python session for this part of the assignment. To load them, try:

```
1 import gensim.downloader
2 w2v = gensim.downloader.load('word2vec-google-news-300')
```

4.1 Exploring Word Vector Analogies

The structure of word vectors can sometimes solve simple analogies like *man is to king as woman is to queen*, or *man:king::woman:queen* in a compact notation. We can query `word2vec` with this analogy *man:king::woman:?* using the built-in `most-similar` command. We've written a wrapper around this command for ease-of-use.

```
1 >>> def analogy(a, b, c):
2     print(a+" : "+b+" :: "+c+" : ?")
3     print([(w, round(c,3)) for w,c in w2v.most_similar(positive=[c,b], negative=[a])])
4
5 >>> analogy('man', 'king', 'woman')
6     man : king :: woman : ?
7     [('queen', 0.7118192911148071), ...]
```

which return a list of the 10 most similar words given this analogy along with their a score (cosine similarity).

► **TASK 4.1 [3pts]** Use the `most_similar` function to find three additional analogies that work. In your response, provide the analogies in the compact *a:b::c:?* form, the model's list of outputs, and why you consider this output to satisfy the analogy.

► **Answer 4.1**

```
1 >>> red : apple :: green : ?
2     [('apples', 0.517), ('fruit', 0.464), ('orcharding', 0.461), ('Honeycrisp_apples',
3     , 0.456), ('Granny_Smiths', 0.452), ('apples_ripen', 0.447), ('potato', 0.443), ('
4     orchard', 0.442), ('mango', 0.44), ('apple_orchard', 0.438)]
5 >>> ladybug : insect :: dog : ?
6     [('dogs', 0.631), ('animal', 0.619), ('pit_bull', 0.566), ('cat', 0.541), ('
7     Rottweiler', 0.534), ('puppy', 0.532), ('canines', 0.525), ('pit_bulls', 0.517),
8     ('rodent', 0.514), ('pet', 0.513)]
9 >>> girl : boy :: man : ?
10    [('teenager', 0.579), ('woman', 0.507), ('youngster', 0.494), ('guy', 0.493), ('
11    teen_ager', 0.488), ('suspected_purse_snatcher', 0.485), ('robber', 0.483), ('Man
12    ', 0.481), ('lad', 0.475), ('horribly_horribly_deranged', 0.471)]
```

► **TASK 4.2 [3pts]** Use the `most_similar` function to find three analogies that did not work. In your response to this question, provide the analogies in the compact *a:b::c:?* form, the model's list of outputs, and why you consider this output to not satisfy the analogy.

► **Answer 4.2**

```
1 >>> apple : ios :: samsung : ?
2     [('3gs', 0.62), ('droid_x', 0.593), ('greenpoison', 0.593), ('dvd_burner', 0.589)
3     , ('Sony_ericsson', 0.58), ('unlock_iphone', 0.578), ('symbian', 0.578), ('CM7',
4     0.577), ('iphone_3gs', 0.575), ('nokia', 0.575)]
5 >>> pen : paper :: chalk : ?
6     [('chalking', 0.397), ('unprinted', 0.358), ('chalked', 0.358), ('kraft', 0.349),
7     ('Chalking', 0.348), ('Paper', 0.337), ('claylike', 0.336), ('They_unroll_toilet
8     ', 0.332), ('Telgi_stamp', 0.327), ('abir', 0.327)]
9 >>> shoe : foot :: hat : ?
10    [('feet_tall', 0.426), ('#/#-foot', 0.42), ('#/#-foot-tall', 0.418), ('hats',
11    0.398), ('feet', 0.389), ('Smokey_Bear_hat', 0.385), ('#/#-feet_tall', 0.381)]
```

4.2 Learned Bias

This sort of analysis can demonstrate some biased assumptions. Two examples regarding gender:

► `word2vec` reinforces stereotypes about gender roles in medicine – associating female doctors with careers in nursing or specializations involving women or children's health.

```

1
2 >>> analogy('man', 'doctor', 'woman')
3 man : doctor :: woman : ?
4 [('gynecologist', 0.709), ('nurse', 0.648), ('doctors', 0.647), ('physician', 0.644), ('
5   'pediatrician', 0.625), ('nurse_practitioner', 0.622), ('obstetrician', 0.607), ('
6   'ob_gyn', 0.599), ('midwife', 0.593), ('dermatologist', 0.574)]
7
8 >>> analogy('woman', 'doctor', 'man')
9 woman : doctor :: man : ?
10 [('physician', 0.646), ('doctors', 0.586), ('surgeon', 0.572), ('dentist', 0.552), ('
11   'cardiologist', 0.541), ('neurologist', 0.527), ('neurosurgeon', 0.525), ('urologist',
12   0.525), ('Doctor', 0.524), ('internist', 0.518)]

```

> word2vec reinforces stereotypes about gender roles in victimhood – associating men with aggression and criminality.

```

1
2 >>> analogy('man', 'victim', 'woman')
3 man : victim :: woman : ?
4 [('victims', 0.582), ('victm', 0.569), ('Victim', 0.566), ('vicitim', 0.542), ('girl',
5   0.533), ('complainant', 0.518), ('mother', 0.516), ('perpetrator', 0.506), ('she',
6   0.5), ('Craite', 0.498)]
7
8 >>> analogy('woman', 'victim', 'man')
9 woman : victim :: man : ?
10 [('suspect', 0.573), ('perpetrator', 0.553), ('victims', 0.53), ('assailant', 0.521), ('
11   'Victim', 0.519), ('vicitm', 0.501), ('boy', 0.488), ('robber', 0.484), ('vicitim',
12   0.468), ('supect', 0.463)]

```

► **TASK 4.3 [2pts]** Use the `most_similar` function to find two additional cases of bias based on gender, politics, religion, ethnicity, or nationality. In your response, provide the analogies in the compact `a:b::c:?` form, the model's list of outputs for both `a:b::c:?` and `c:b::a:?`, and why you consider this output to be biased based on the model's two responses.

► **Answer 4.3**

```

1
2 >>> man : engineer :: women : ?
3 [('engineering', 0.489), ('engineers', 0.486), ('electrical_engineer', 0.409), ('
4   'Engineer', 0.394), ('mechanical_engineer', 0.391), ('consultant', 0.373), ('
5   'Maurer_Stutz_Inc.', 0.363), ('Carollo_Engineers', 0.36), ('KLH_Engineers', 0.36),
6   ('TY_Lin_International', 0.358)]
7
8 >>> women : engineer :: man : ?
9 [('mechanical_engineer', 0.555), ('electrical_engineer', 0.546), ('electrician',
10   0.532), ('mechanic', 0.527), ('engineer', 0.519), ('technician', 0.495), ('
11   'salesman', 0.49), ('Engineer', 0.488), ('computer_programmer', 0.468), ('
12   'repairman', 0.467)]

```

For case `[man : engineer :: women : ?]`, I thought there would be a less engineering job or field, such as literature, art, etc. The value is slightly smaller than the opposite case, but I was glad that similar words came out and the ratio of female engineers has increased these days.

```

1 >>> ceo : asians :: worker : ?
2 [('migrant_worker', 0.508), ('workers', 0.49), ('laborers', 0.479), ('housemaid',
3   0.451), ('sweatshop_laborers', 0.444), ('laborer', 0.438), ('woman', 0.433), ('
4   'caucasian', 0.433), ('uneducated_unskilled', 0.429), ('laowai', 0.428)]
5
6 >>> worker : asians :: ceo : ?
7 [('brazilians', 0.455), ('asia', 0.454), ('sns_ap', 0.453), ('germans', 0.449), ('
8   'serbia', 0.447), ('cks', 0.445), ('mr', 0.444), ('nico', 0.444), ('
9   'african_americans', 0.441), ('denna', 0.439)]

```

► **TASK 4.4 [2pts]** Why might these biases exist in word2vec and what are some potential consequences that might result if word2vec were used in a live system?

► **Answer 4.4**

I think the reason for the bias in word2vec is that it also contains a lot of text information about stereotypes created by individuals such as social networks because data sources are brought from media such as the Internet. This seems to be a bias because several of these are harvested. I think that such biased information can cause negative effects such as reinforcement of stereotypes. In order to solve this problem, there should be rules on AI ethics, but I think it would be better to use unbiased data above all else.

References

- [1] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.
- [2] X. Zhang, J. J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," in *Neural Information Processing Systems (NeurIPS)*, 2015.
- [3] K. Church and P. Hanks, "Word association norms, mutual information, and lexicography," *Computational Linguistics*, vol. 16, no. 1, pp. 22–29, 1990.
- [4] O. Levy and Y. Goldberg, "Neural word embedding as implicit matrix factorization," in *Advances in neural information processing systems*, pp. 2177–2185, 2014.
- [5] L. v. d. Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.
- [6] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Empirical Methods in Natural Language Processing (EMNLP)*, 2014.
- [7] K. Kenyon-Dean, E. Newell, and J. C. K. Cheung, "Deconstructing word embedding algorithms," in *Empirical Methods in Natural Language Processing (EMNLP)*, pp. 8479–8484, 2020.