AI539 Natural Language Processing with Deep Learning – Homework 1

Word Vectors: Distributed Representations of Words

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.

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

Listing 1: Code to load the AG News Dataset

The `dataset` structure returned looks like this

```
DatasetDict({
    train: Dataset({
        features: ['text', 'label'],
        num_rows: 120000
    })
    test: Dataset({
        features: ['text', 'label'],
        num_rows: 7600
    })
}
```

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.

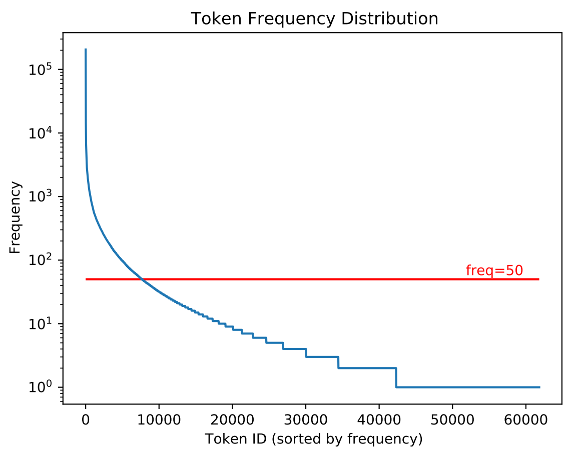
```
> vocab.tokenize("The quick, brown fox jumped over the lazy dog.")
["the", "quick", "brown", "fox", "jumped", "over", "the", "lazy","dog"]
```

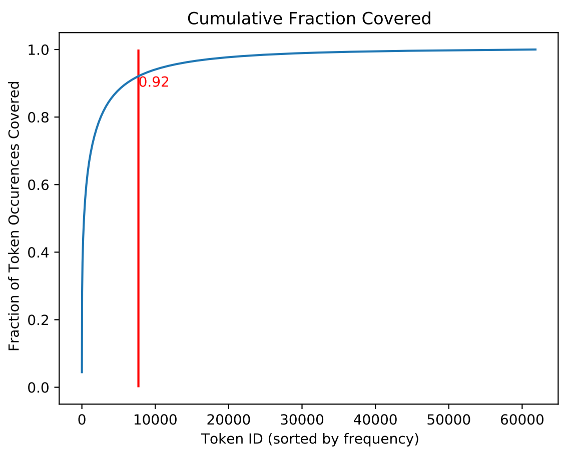
Basing a system on these tokens is fine, but it is a design decision. For instance – conjugations such as `jumped`, `jump`, and `jumps` would all be different tokens despite having common root meanings, as would any misspellings (`jupmed`). Alternative tokenizations include sub-word tokenization (`jumped` → `[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 lemmatization, 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.

▶ TASK 1.1 WRITE-UP For tokenization, every given text is changed to lower case, and the numbers or symbols are removed from the lower text (only lower alphabets remain). For example, the sentence "The blue dog jumped, but not high" would be "the blue dog jumped but not high" after lowering and cleaning. Then, based on each space, words are split as each token. Thus, the final result of tokenization from the example above is ["the", "blue", "dog", "jumped", "but", "not", "high"].

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 ith 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:

```
["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:

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

> vocab.idxToText([0, 9, 1, 2, 6, 5, 0, 7, 8])
["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 `word2indx` 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 rational for setting it.

▶ TASK 1.3 WRITE-UP For obtaining proper frequency threshold of token frequency distribution, cumulative fraction is needed to be calculated by the cumulative number of frequencies from each token. To do this, each token's frequency is sorted in non-increasing order and stacked from the largest frequency to the least frequency to find the exact 92% proportion of the given corpus. As shown in Figure 1, when the cumulative fraction is 92%, the frequency threshold is 36 (the rest of them would be "UNK" tokens), meaning that in this assignment, the words over 36 frequency will be used.
The reason why it is rational is that there is a memory issue if all the corpus is used, and the choice of 92% cumulative fraction is based on the long-tail distribution, which can cover a large portion of the total word occurrences in a corpus by keeping just the most frequent words and discarding the rare ones.

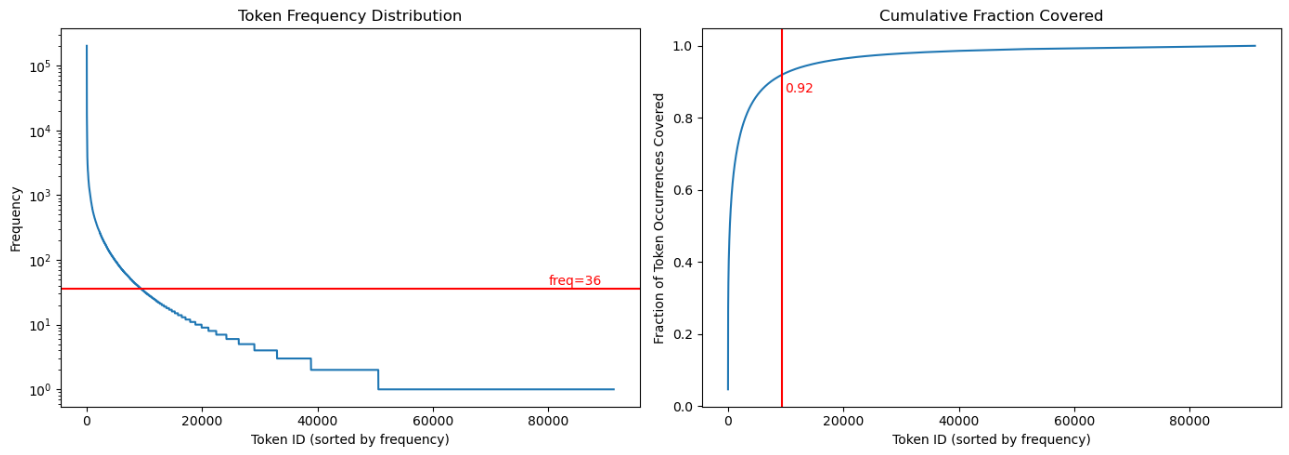


Figure 1: Frequency Distribution and Cumulative Fraction Covered

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:

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

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 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?

▶ TASK 2.1 WRITE-UP

- What are the minimum and maximum values of PMI (Eq. 1)?

  the minimum value of PMI is based on the probability of when two tokens w_i and w_j rarely or never co-occur, meaning that numerator has very low value ($p(w_i, w_j)$ is close to 0). So, the minimum value would be $-\infty$. On the other hand, the maximum value of PMI is based on the probability of when two tokens always appear together, implying that the probability of co-occurrence can be equal to the probability of when word token w_i or w_j appears ($p(w_i, w_j) = p(w_i) = p(w_j)$). Thus, The maximum value would be $log(\frac{1}{p(w_i)})$ if $p(w_i)$ is very small.

- If two tokens have a positive PMI, what does that imply about their relationship?

  A positive PMI means $log(p(w_i, w_j)) - log(p(w_i)p(w_j)) > 0$. This can be simplified as $p(w_i, w_j) > p(w_i)p(w_j)$, implying that the two words co-occur more often than expected by chance (semantically and contextually related).

- If they have a PMI of zero?

  $PMI(w_i, w_j) = 0$ means $log(p(w_i, w_j)) = log(p(w_i)p(w_j))$. This can be also simplified as $p(w_i, w_j) = p(w_i)p(w_j)$, implying that knowing one word tells you nothing about the other word (they are independent).

- What about if the PMI is negative?

  $PMI(w_i, w_j) = 0$ means $log(p(w_i, w_j)) < log(p(w_i)p(w_j))$. This can be also simplified as $p(w_i, w_j) < p(w_i)p(w_j)$, implying that the two words co-occur less often than expected (they may avoid each other in usage).

- Based on your answers, what is an intuition for the use of PPMI?

  By using PPMI, we can expect the relationship between the two words, emphasizing positive, meaningful associations and ignoring negative or weak ones.

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-occurence counts such that the ijth 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

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

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

▶ 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.

▶ 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* $\log(0)$.

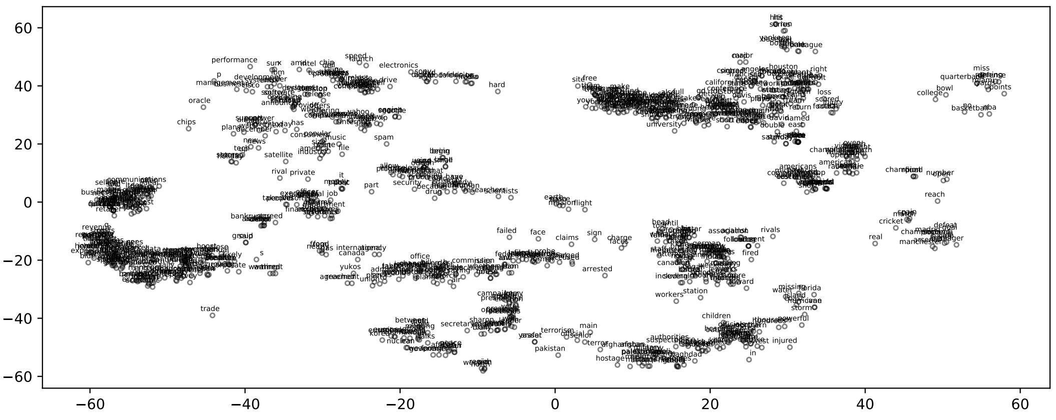
▶ TASK 2.2 WRITE-UP K-window method is used to find the context of the words in a sentence. In the code, the value of K is 4, and based on the window size K, the words that have a constant distance, which is $-4 \leq K \leq 4,\ K \neq 0$, between a chosen center token during each for-loop are counted in the co-occurrence matrix.

Our First Word Vector. A single row $P_{i:}$ 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.

▶ TASK 2.4 WRITE-UP As shown in Figure 2, the entire TSNE plot represents the contextual relationship among words based on the PPMI. There are three different regions marked by red circles, which are related to economics, international politics, and sports. Figure 3 shows economic words such as dollars, economic, production, and costs. These words are strongly related to economic concepts. Figure 4 displays international politics since there are many different countries such as India, China, and Canada, as well as political terms such as election, leaders, and secretary. Figure 5 presents sports-related words such as Madrid, soccer, and championship. Thus, the given examples above are representative of economics, international politics, and sports.

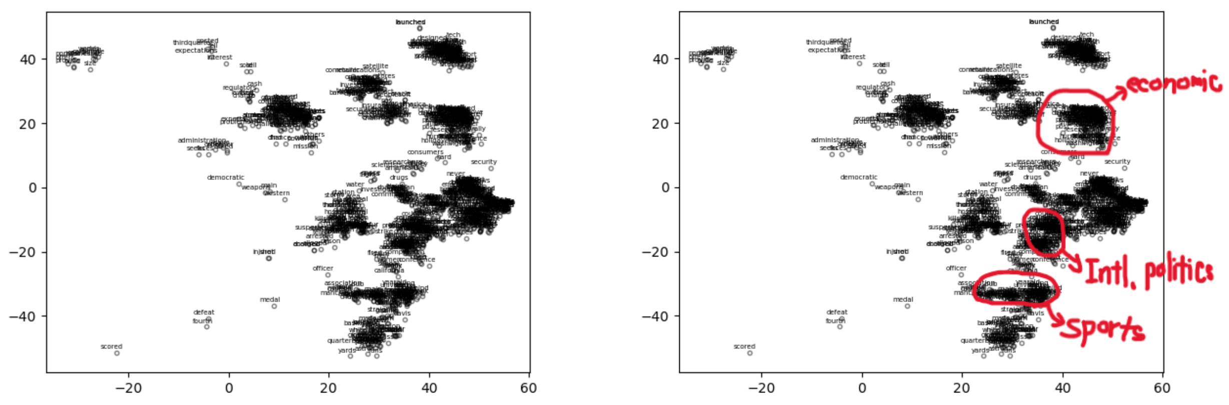


Figure 2: TSNE plot as a whole

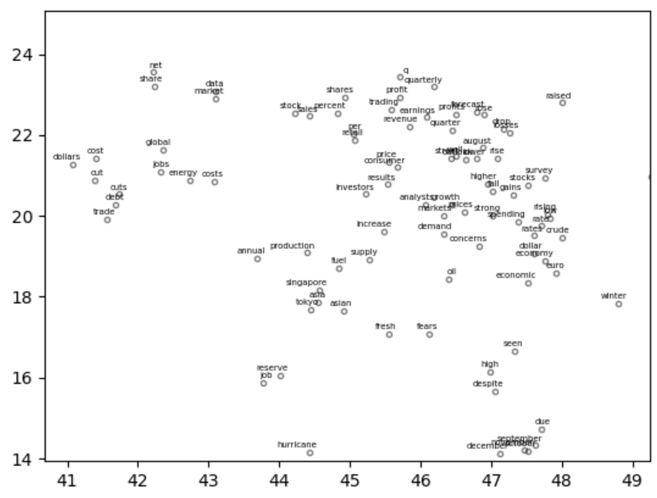


Figure 3: TSNE: Economics

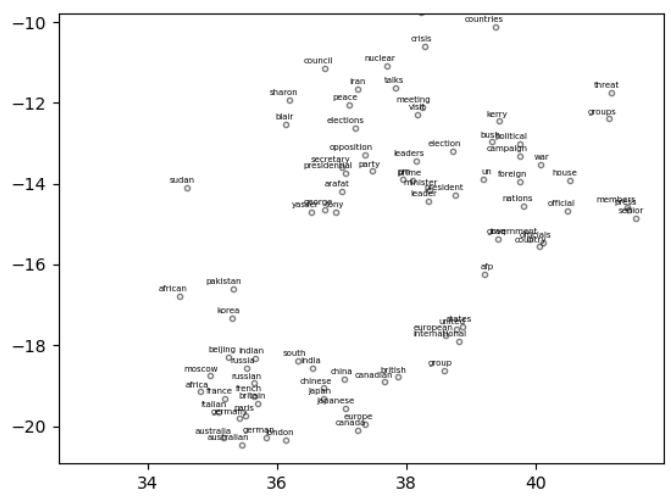


Figure 4: TSNE: International politics

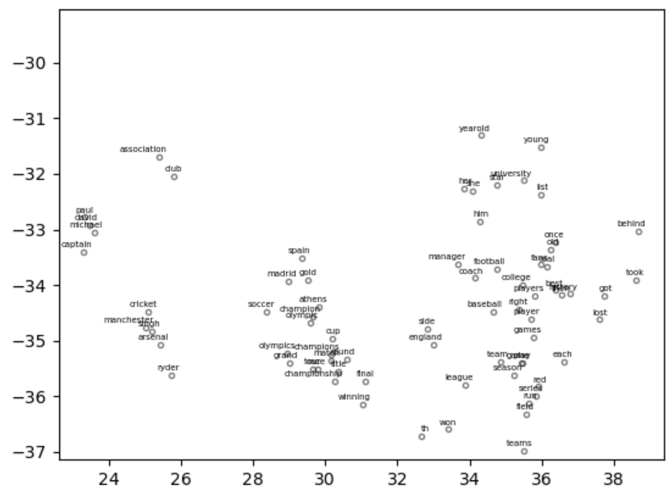


Figure 5: TSNE: Sports

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,

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

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

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

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:

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

▶ 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.

▶ TASK 3.1 WRITE-UP To obtain the gradient of the objective J_B, the error term can be simplified as e_{ij} for convenience. So, the simplified equation would be:

$$J_B = \sum_{(i,j)\in B} f(C_{ij}) \cdot e_{ij}^2$$

Then, the equation above can be used to derive the gradient. The gradient with respect to w_i is:

$$\nabla_{w_i} J_B = \frac{\partial J_B}{\partial w_i} = f(C_{ij}) \cdot 2 \cdot e_{ij} \cdot \tilde{\mathbf{w}}_j$$

The gradient with respect to $\tilde{w}_j$ is:

$$\nabla_{\tilde{w}_j} J_B = \frac{\partial J_B}{\partial \tilde{w}_j} = f(C_{ij}) \cdot 2 \cdot e_{ij} \cdot \mathbf{w}_i$$

The gradient with respect to b_i is:

$$\nabla_{b_i} J_B = \frac{\partial J_B}{\partial b_i} = f(C_{ij}) \cdot 2 \cdot e_{ij}$$

The gradient with respect to $\tilde{b}_j$ is:

$$\nabla_{\tilde{b}_j} J_B = \frac{\partial J_B}{\partial \tilde{b}_j} = f(C_{ij}) \cdot 2 \cdot e_{ij}$$

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.

```
i = idx[shuf_idx, 0] #vector of indices of word vectors in the batch
j = idx[shuf_idx, 1] #vector of indices of context vectors in the batch

# Get word and context vectors for pairs in the batch
w_batch = wordvecs[i, :]
c_batch = contextvecs[j, :]

# Compute f(C_i,j) for i,j pairs in batch (Bx1 dimensional vector)
fval = (np.minimum(1, C[i, j]/100)**0.75)[:,np.newaxis]

# Compute error term as (w_i^T \tilde{w}_j + b_i + \tilde{b}_i - log(C_ij)) for each i,j
    pair in the batch. (Bx1 dimensional vector)
error = (np.sum(np.multiply(w_batch, c_batch), axis=1)[:,np.newaxis] + wordbiases[i] +
    contextbiases[j] - np.log(C[i,j])[:,np.newaxis])

# Compute the overall objective loss
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.

```
########################################################################
# Task 3.2
########################################################################
# REMOVE THIS ONCE YOU IMPLEMENT THIS SECTION
raise UnimplementedFunctionError("You have not yet implemented the gradient computation.")

# write expressions using numpy to implement the gradients you derive in 3.1.
wordvecs_grad = np.zeros( (bSize,d) )
wordbiases_grad = np.zeros( (bSize,1) )
contextvecs_grad = np.zeros( (bSize,d) )
contextbiases_grad = np.zeros( (bSize,1) )
########################################################################
```

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

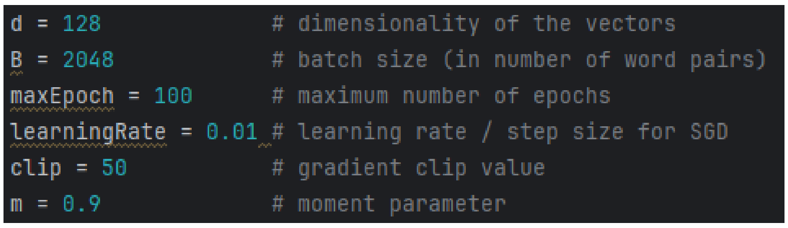
Once the gradients have been implemented, running `build_glove_vectors.py` should build a vocabulary, compute a co-occurence matrix, and then start optimizing the GloVe objective via SGD with momentum. The code passes over the data five times (5 epochs) and will print the average objective value every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

```
2020-12-27 18:15     12635115 non-zero entries in the count matrix
2020-12-27 18:15     Epoch 1 / 5: learning rate = 0.1
2020-12-27 18:15     Iter 100 / 12338: avg. loss over last 100 batches = 1.5028382185809
2020-12-27 18:15     Iter 200 / 12338: avg. loss over last 100 batches = 0.247632015612976
2020-12-27 18:15     Iter 300 / 12338: avg. loss over last 100 batches = 0.153665193206001
2020-12-27 18:15     Iter 400 / 12338: avg. loss over last 100 batches = 0.122809739341802
2020-12-27 18:15     Iter 500 / 12338: avg. loss over last 100 batches = 0.106307783161738
2020-12-27 18:15     Iter 600 / 12338: avg. loss over last 100 batches = 0.099031105070185
2020-12-27 18:15     Iter 700 / 12338: avg. loss over last 100 batches = 0.096116501282613
2020-12-27 18:15     Iter 800 / 12338: avg. loss over last 100 batches = 0.091317471566911
```

The hyperpameters should be set to reasonable values for most vocabularies and the loss should converge to something around 0.03. Your values may differ if you made different choices about tokenization and context earlier on. If you find your loss is increasing or oscillating, you can try reducing the learning rate, but it may be a sign that something is wrong in your gradient computation.

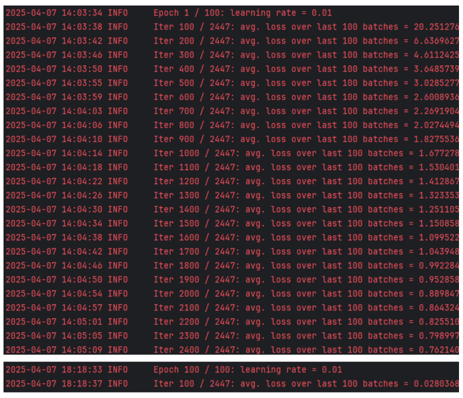
▶ TASK 3.3 [5pts] Run `build_glove_vectors.py` to learn GloVe vectors and visualize them with TSNE! In your write-up, describe how the loss behaved during training (how stably it decreased, what it converged to, etc). Also include the TSNE plot. If everything has been done correctly, you should observe similar clustering behavior as in Task 2.3.

▶ TASK 3.3 WRITE-UP The hyperparameter setting is in Figure 6. As shown in Figure 7, train losses stably decrease from 20.251 to 0.028 until convergence during training. After finishing, the whole Glove TSNE plot is in Figure 8. As can be seen in Figure 9, economical words such as investors, increase, and economic are plotted. In Figure 10, there are some international political words such as India, Pakistan, and international. Figure 11 presents some sport words such as England, soccer, and Arsenal.



```
d = 128                  # dimensionality of the vectors
B = 2048                 # batch size (in number of word pairs)
maxEpoch = 100           # maximum number of epochs
learningRate = 0.01      # learning rate / step size for SGD
clip = 50                # gradient clip value
m = 0.9                  # moment parameter
```

Figure 6: Hyperparameters



```
2025-04-07 14:03:34 INFO     Epoch 1 / 100: learning rate = 0.01
2025-04-07 14:03:38 INFO     Iter 100 / 2447: avg. loss over last 100 batches = 20.251276
2025-04-07 14:03:42 INFO     Iter 200 / 2447: avg. loss over last 100 batches = 6.6369627
2025-04-07 14:03:46 INFO     Iter 300 / 2447: avg. loss over last 100 batches = 4.6112425
2025-04-07 14:03:50 INFO     Iter 400 / 2447: avg. loss over last 100 batches = 3.6485739
2025-04-07 14:03:55 INFO     Iter 500 / 2447: avg. loss over last 100 batches = 3.0285277
2025-04-07 14:03:59 INFO     Iter 600 / 2447: avg. loss over last 100 batches = 2.6008936
2025-04-07 14:04:03 INFO     Iter 700 / 2447: avg. loss over last 100 batches = 2.2691904
2025-04-07 14:04:06 INFO     Iter 800 / 2447: avg. loss over last 100 batches = 2.0274494
2025-04-07 14:04:10 INFO     Iter 900 / 2447: avg. loss over last 100 batches = 1.8275536
2025-04-07 14:04:14 INFO     Iter 1000 / 2447: avg. loss over last 100 batches = 1.677278
2025-04-07 14:04:18 INFO     Iter 1100 / 2447: avg. loss over last 100 batches = 1.530401
2025-04-07 14:04:22 INFO     Iter 1200 / 2447: avg. loss over last 100 batches = 1.412867
2025-04-07 14:04:26 INFO     Iter 1300 / 2447: avg. loss over last 100 batches = 1.323353
2025-04-07 14:04:30 INFO     Iter 1400 / 2447: avg. loss over last 100 batches = 1.251105
2025-04-07 14:04:34 INFO     Iter 1500 / 2447: avg. loss over last 100 batches = 1.150858
2025-04-07 14:04:38 INFO     Iter 1600 / 2447: avg. loss over last 100 batches = 1.099522
2025-04-07 14:04:42 INFO     Iter 1700 / 2447: avg. loss over last 100 batches = 1.043948
2025-04-07 14:04:46 INFO     Iter 1800 / 2447: avg. loss over last 100 batches = 0.992284
2025-04-07 14:04:50 INFO     Iter 1900 / 2447: avg. loss over last 100 batches = 0.952858
2025-04-07 14:04:54 INFO     Iter 2000 / 2447: avg. loss over last 100 batches = 0.889847
2025-04-07 14:04:57 INFO     Iter 2100 / 2447: avg. loss over last 100 batches = 0.864324
2025-04-07 14:05:01 INFO     Iter 2200 / 2447: avg. loss over last 100 batches = 0.825510
2025-04-07 14:05:05 INFO     Iter 2300 / 2447: avg. loss over last 100 batches = 0.798997
2025-04-07 14:05:09 INFO     Iter 2400 / 2447: avg. loss over last 100 batches = 0.762140

2025-04-07 18:18:33 INFO     Epoch 100 / 100: learning rate = 0.01
2025-04-07 18:18:37 INFO     Iter 100 / 2447: avg. loss over last 100 batches = 0.0280368
```

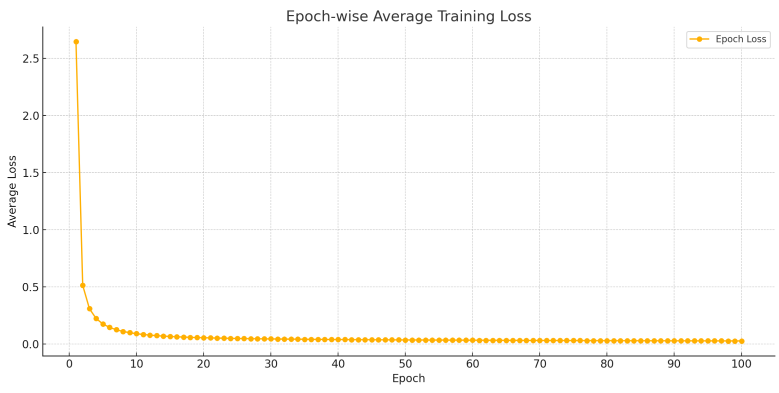


Figure 7: Epoch-wise average training loss across 100 epochs

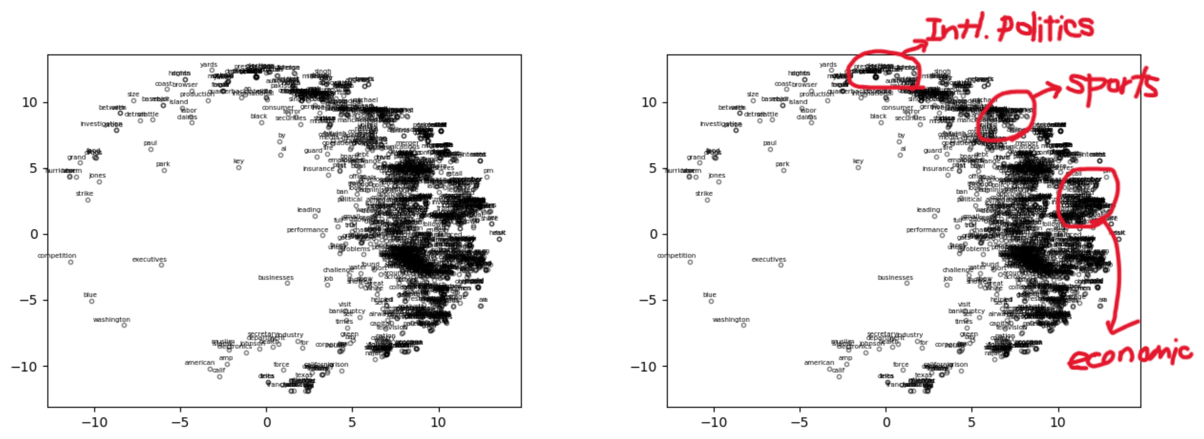


Figure 8: Glove TSNE plot as whole

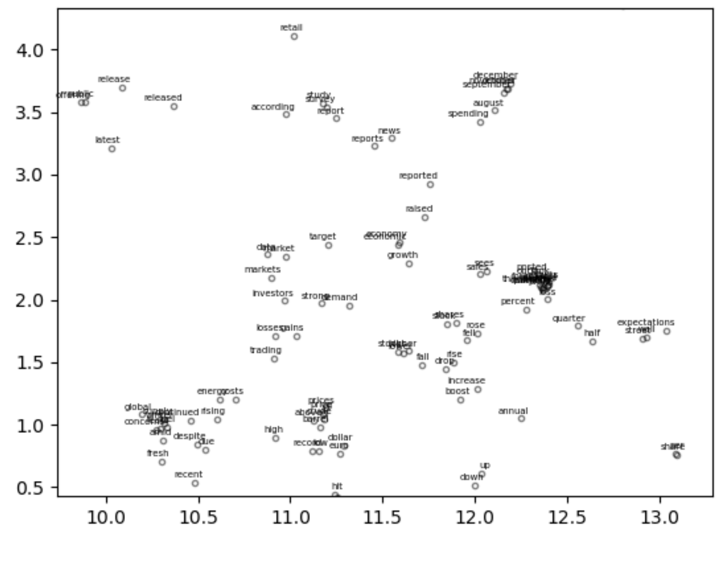


Figure 9: Glove TSNE: Economics

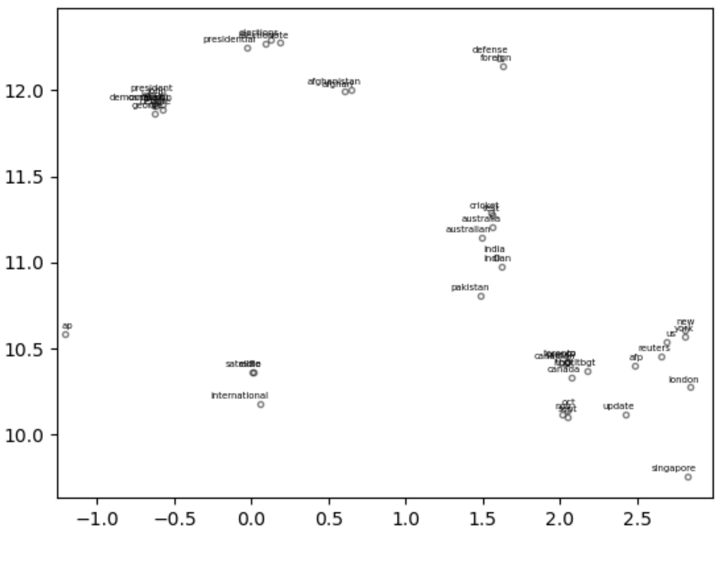


Figure 10: Glove TSNE: International Politics

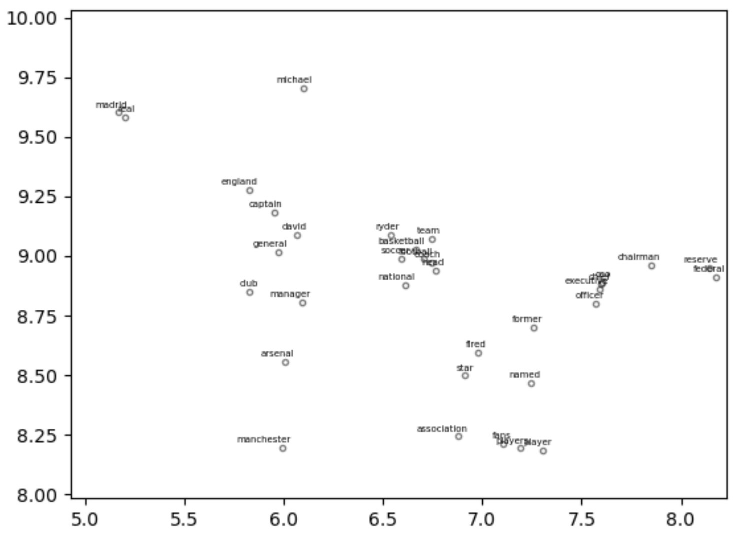


Figure 11: Glove TSNE: Sports

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:

```
import gensim.downloader
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.

```
>>> def analogy(a, b, c):
        print(a+" : "+b+" :: "+c+" : ?")
        print([(w, round(c,3)) for w,c in w2v.most_similar(positive=[c,b], negative=[a])])

>>> analogy('man', 'king', 'woman')
    man : king :: woman : ?
    [('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.

▶ 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.

▶ TASK 4.1 WRITE-UP

- korea : seoul :: japan : ?

  ⇒ [('tokyo', 0.59), ('nikkei', 0.478), ('gma', 0.463), ('japanese', 0.46), ...]

  The output 'tokyo' is reasonable since (korea, seoul) is the pair of (country, city), meaning that (japan, tokyo) is also in the same context.

- walking : walked :: swimming : ?

  ⇒ [('swam', 0.693), ('swim', 0.673), ('swimmers', 0.592), ('swum', 0.586), ...]

  The output 'swam' is right since the expected output should be the past tense of "swim".

- boy : girl :: man : ?

  ⇒ [('woman', 0.871), ('teenage_girl', 0.685), ('lady', 0.592), ('teenager', 0.584), ...]

  The output 'woman' is correct since the expected output should have opposite word of given one.

▶ TASK 4.2 WRITE-UP

- bird : fly :: fish : ?

  ⇒ [('longline_vessels', 0.481), ('fishing', 0.457), ('fished', 0.447), ('fishes', 0.431), ...]

  The expected output should be related to possible behavior such as 'swim', but 'longline_vessels' is hard to be answer of it.

- sun : day :: moon : ?

  ⇒ [('month', 0.466), ('week', 0.45), ('days', 0.412), ('year', 0.392), ('lunar', 0.387), ...]

  The expected output would be 'night', but it shows 'month', which is not reasonable.

- teacher : school :: doctor : ?

  ⇒ [('doctors', 0.608), ('clinic', 0.558), ('physician', 0.538), ('hospital', 0.517), ...]

  The expected output should be 'hospital', which is the location where doctor works, but it displays 'doctors', which is related to plural.

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.

```

>>> analogy('man', 'doctor', 'woman')
    man : doctor :: woman : ?
    [('gynecologist', 0.709), ('nurse', 0.648), ('doctors', 0.647), ('physician', 0.644), ('pediatrician', 0.625), ('nurse_practitioner', 0.622), ('obstetrician', 0.607), ('ob_gyn', 0.599), ('midwife', 0.593), ('dermatologist', 0.574)]

>>> analogy('woman', 'doctor', 'man')
    woman : doctor :: man : ?
    [('physician', 0.646), ('doctors', 0.586), ('surgeon', 0.572), ('dentist', 0.552), ('cardiologist', 0.541), ('neurologist', 0.527), ('neurosurgeon', 0.525), ('urologist', 0.525), ('Doctor', 0.524), ('internist', 0.518)]
```

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

```

>>> analogy('man', 'victim', 'woman')
    man : victim :: woman : ?
    [('victims', 0.582), ('vicitm', 0.569), ('Victim', 0.566), ('viciitm', 0.542), ('girl',
     0.533), ('complainant', 0.518), ('mother', 0.516), ('perpetrator', 0.506), ('she',
    0.5), ('Craite', 0.498)]


>>> analogy('woman', 'victim', 'man')
    woman : victim :: man : ?
    [('suspect', 0.573), ('perpetrator', 0.553), ('victims', 0.53), ('assailant', 0.521), (
    'Victim', 0.519), ('vicitm', 0.501), ('boy', 0.488), ('robber', 0.484), ('viciitim',
    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.

▶ 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?

▶ TASK 4.3 WRITE-UP

- man : computer_programmer :: woman : ?

  [('homemaker', 0.563), ('housewife', 0.511), ('graphic_designer', 0.505), ('schoolteacher', 0.498), ...]

  woman : computer_programmer :: man : ?

  [('mechanical_engineer', 0.572), ('programmer', 0.521), ('electrical_engineer', 0.519), ('carpenter', 0.505), ...]

  ⇒ The dataset implies that woman is related to 'homemaker', 'housewife' whereas man is related to 'mechanical_engineer', 'programmer', showing typical gender stereotype.

- american : handsome :: asian : ?

  [('plumpish', 0.465), ('scruffily_dressed', 0.462), ('dark_haired', 0.456), ...]

  asian : handsome :: american : ?

  [('ruggedly_handsome', 0.439), ('rakish', 0.428), ('impeccable_manners', 0.397), ('debonair', 0.396), ...]

  ⇒ The output shows american is more related to handsome, but asian is more related to 'plumpish' or style, which can be the race stereotype.

▶ TASK 4.4 WRITE-UP word2vec is trained on large-scale real-world text corpora such as news, wikipedia, and these resources often reflect existing social biases, such as gender stereotypes, racial bias, or religious prejudices. Since word2vec learns co-occurence statistics directly from the data, it also inevitably captures and reproduces those biases in the resulting word embeddings.

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.