WordVectors

November 13, 2023

0.1 Word Vectors

This assignment is comprised of two parts:

1. **Theory**: Solve two exercises about logistic regression and softmax regression. Prove that using sums of word vectors as phrase embeddings is problematic.
2. **Implementation**: You will experiment with sparse and dense vector representations of words.
3. **Analysis**: You will compare the vector representations of meaning with GPT models in terms of their ability to understand words, their relationships, and biases.

Word Vectors are often used as a fundamental component for downstream NLP tasks, e.g. question answering, text generation, translation, etc., so it is important to build some intuitions as to their strengths and weaknesses. Here, you will explore two types of word vectors:

- those derived from *co-occurrence matrices*, and
- those derived via *word2vec*.

Note on Terminology: The terms "word vectors" and "word embeddings" are often used interchangeably. The term "embedding" refers to the fact that we are encoding aspects of a word's meaning in a lower dimensional space. As Wikipedia states, "*conceptually it involves a mathematical embedding from a space with one dimension per word to a continuous vector space with a much lower dimension*".

Before getting started with the implementation, install the gensim library:

```
conda install gensim
```

To be able to download the large word2vec embeddings file, you may need to run Jupyter Notebook with the following command line option: `jupyter notebook --NotebookApp.iopub_msg_rate_limit=1.0e10`

0.2 Write Your Name Here: Claire Ardern

1 Submission Instructions

1. Click the Save button at the top of the Jupyter Notebook.
2. Please make sure to have entered your name above.
3. Select Cell -> All Output -> Clear. This will clear all the outputs from all cells (but will keep the content of ll cells).
4. Select Cell -> Run All. This will run all the cells in order, and will take several minutes.

5. Once you've rerun everything, select File -> Download as -> PDF via LaTeX and download a .pdf version showing the code and the output of all cells, and save it in the same folder that contains the notebook file .ipynb.
6. Look at the PDF file and make sure all your solutions are there, displayed correctly. The PDF is the only thing we will see when grading!
7. Submit **both** your PDF and notebook on Canvas.
8. Verify your Canvas submission contains the correct files by downloading them after posting them on Canvas.

2 Theory

2.1 Theory (4111 & 5111): Properties of cosine similarity

1. Prove that doubling the length of a vector $\mathbf{u}$ does not change its cosine similarity with any other vector $\mathbf{v}$, i.e. prove that $cos(2\mathbf{u}, \mathbf{v}) = cos(\mathbf{u}, \mathbf{v})$.
2. Could the cosine similarity be negative when using $tf.idf$ vector representations? Explain your answer.
3. Could the cosine similarity be negative when using prediction-based, dense vector representations? Explain your answer.

Answer 1:

$$cos(2\mathbf{u}, \mathbf{v}) = \frac{\mathbf{2u^T v}}{||2\mathbf{u}||||\mathbf{v}||}$$

$$= \frac{2 \times \mathbf{u^T v}}{2 \times ||\mathbf{u}||||\mathbf{v}||}$$

$$= \frac{2}{2} \times \frac{\mathbf{u^T v}}{||\mathbf{u}||||\mathbf{v}||}$$

$$= 1 \times \frac{\mathbf{u^T v}}{||\mathbf{u}||||\mathbf{v}||} = \frac{\mathbf{u^T v}}{||\mathbf{u}||||\mathbf{v}||}$$

$$\frac{\mathbf{u^T v}}{||\mathbf{u}||||\mathbf{v}||} = cos(\mathbf{u}, \mathbf{v})$$

So,

$$cos(2\mathbf{u}, \mathbf{v}) = cos(\mathbf{u}, \mathbf{v})$$

Answer 2: The cosine similarity cannot be negative when using $tf.idf$ vector representations. Since the two attribute vectors $\mathbf{u}$ and $\mathbf{v}$ are term frequency vectors of the documents and term frequencies cannot be negative, the cosine similarity of two documents will range from 0 to 1 (quadrant one).

Answer 3: The cosine similarity can be negative when using prediction-based, dense vector representations. This is because when we are using dense embeddings, we can vectors in very direction from the origin, including negative values. In other words, our vectors can now exist in all four quadrants rather than just quadrant one which includes only positive values. When we are using negative values to calculate cosine similatiry, it is possible to get a negative score as the possible scores can now range -1 to 1.

2.2 Theory (5111): On the fitting power of Logistic Regression

Consider a training set that contains the 4 training examples shown in the table below. Each training example $\mathbf{x}$ has 2 features x_1 and x_2 and a label $y \in \{0, 1\}$.

$\mathbf{x}$	x_1	x_2	y
$\mathbf{x}^{(1)}$	0	0	0
$\mathbf{x}^{(2)}$	0	1	1
$\mathbf{x}^{(3)}$	1	0	1
$\mathbf{x}^{(4)}$	1	1	0

Prove that no binary logistic regression model can perfectly classify this dataset.

Hint: Prove that there cannot be a vector of parameters $\mathbf{w} = [w_1, w_2]$ *and bias* b *such that* $P(t = 1|\mathbf{x}; \mathbf{w}, b) \geq 0.5$ *for all examples* $\mathbf{x}$ *that are positive, and* $P(t = 1|\mathbf{x}; \mathbf{w}, b) < 0.5$ *for all examples* $\mathbf{x}$ *that are negative.*

A logistic regression model for this dataset would use the linear equation $z = w_1x_1 + w_2x_2 + b$ to create the decision boundary which classifies each data point into its respective class. The parameters $\mathbf{w} = [w_1, w_2]$ and bias b are what determine the positioning of this decision boundary in a 2-dimensional space. Usually, given an acceptable dataset, we can adjust the location and slope of the line to correctly classify a majority of the data points in a dataset. However, as seen in the figure below, there is no possible linear boundary that can correctly classify all of the data points in this dataset. A few examples of attempted decision boundaries are shown in gray but when looking at the positioning of the data points in the 2-dimensional space, it becomes clear that there is no linear boundary that will perfectly classify this dataset.

Mathematically, we can prove this by plugging our data points into the linear equation $z = w_1x_1 + w_2x_2 + b$ which the logistic regression model would use for this dataset. Assuming that we will follow this equation with the sigmoid function as is usual for logistic regression, we will use the value $z \geq 0.5$ for all examples $\mathbf{x}$ that are positive, and $z < 0.5$ for all examples $\mathbf{x}$ that are negative.

$$x^{(1)} : z = w_1(0) + w_2(0) + b < 0.5$$

$$x^{(2)} : z = w_1(0) + w_2(1) + b \geq 0.5$$

$$x^{(3)} : z = w_1(1) + w_2(0) + b \geq 0.5$$

$$x^{(4)} : z = w_1(1) + w_2(1) + b < 0.5$$

Or, to simplify this proof by representing classes with 1 for all examples $\mathbf{x}$ that are positive, and 0 for all examples $\mathbf{x}$ that are negative, we create the following equations:

$$x^{(1)} : z = w_1(0) + w_2(0) + b = 0$$

$$x^{(2)} : z = w_1(0) + w_2(1) + b = 1$$

$$x^{(3)} : z = w_1(1) + w_2(0) + b = 1$$

$$x^{(4)} : z = w_1(1) + w_2(1) + b = 0$$

We can then simplify each equation as follows:

$$x^{(1)} : z = b = 0$$

$$x^{(2)} : z = w_2 + b = 1$$

$$x^{(3)} : z = w_1 + b = 1$$

$$x^{(4)} : z = w_1 + w_2 + b = 0$$

By looking at the simplified equation for {x}^{(1)}, we can see that the bias b must be equal to zero. Then, with the value of 0 for bias, following to the equations for {x}^{(2)} and {x}^{(3)}, we can see the parameters $\mathbf{w} = [w_1, w_2]$ must both be equal to 1. However, this contradicts the simplified equation for {x}^{(4)}. So, we can see that it is not possible to perfectly classify this dataset.

2.3 Theory (5111): Binary vs. Multiclass Logistic Regression

Show that binary Logistic Regression is a special case of multiclass Logistic (Softmax) Regression. That is to say, if $\mathbf{w}_1$ and $\mathbf{w}_2$ are the parameter vectors of a Softmax Regression model for the case of two classes, then there exists a parameter vector $\mathbf{w}$ for binary Logistic Regression that results in the same classification as the Softmax Regression model. It goes without saying that the proof should work for any $\mathbf{w}_1$ and $\mathbf{w}_2$.

Hint: Find $\mathbf{w}$ *as a function of* $\mathbf{w}_1$ *and* $\mathbf{w}_2$*.*

When using multiclass Logistic (Softmax) Regression, we compute our class probabilities based on the following equation:

$$P(y_k) = softmax(z_i) = \frac{e^{z_i}}{\sum_{k=0}^{K} e^{z_k}}$$

Where $z = \mathbf{w_k^T x}$ and k can be any positive integer ≥ 2. In the case that $k = 2$, which would be the case of binary Logistic Regression, our softmax equation would appear as follows:

$$P(y_k) = softmax(z_i) = \frac{e^{z_i}}{\sum_{k=0}^{1} e^{z_k}}$$

So, for the two classes that are represented in the binary case, 0 and 1, out softmax outputs would be as follows:

$$P(y = 1) = softmax(z_1) = \frac{e^{\mathbf{w_1^T x}}}{\sum_{k=0}^{1} e^{\mathbf{w_k^T x}}}$$

$$P(y = 0) = softmax(z_0) = \frac{e^{\mathbf{w_0^T x}}}{\sum_{k=0}^{1} e^{\mathbf{w_k^T x}}}$$

We can see that these equations are overparameterized. So, if we expand the denominator, we can subtract w_1 from each of the two parameters and simplify the equation, giving us the following result:

$$P(y = 1) = softmax(z_1) = \frac{e^{\mathbf{0^T x}}}{e^{\mathbf{0^T x}} + e^{(\mathbf{w_0 - w_1})^\mathbf{T x}}}$$

$$P(y = 0) = softmax(z_0) = \frac{e^{(\mathbf{w_0 - w_1})^\mathbf{T x}}}{e^{\mathbf{0^T x}} + e^{(\mathbf{w_0 - w_1})^\mathbf{T x}}}$$

Now, we simplify:

$$P(y = 1) = softmax(z_1) = \frac{1}{1 + e^{(\mathbf{w_0 - w_1})^\mathbf{T x}}}$$

$$P(y = 0) = softmax(z_0) = \frac{e^{(\mathbf{w_0 - w_1})^\mathbf{T x}}}{1 + e^{(\mathbf{w_0 - w_1})^\mathbf{T x}}} = 1 - \left(\frac{1}{1 + e^{(\mathbf{w_0 - w_1})^\mathbf{T x}}}\right)$$

We are left with softmax equations that are equal to the probability equations used in binary logistic regression.

2.4 Theory (Bonus points): Phrase embeddings

Given a phrase consisting of a sequence of M words, $phrase = [word_1, word_2, ..., word_M]$, and given that we have already trained word embeddings $E(word)$ for all the words $word \in V$ in the vocabulary, a simple way of creating an embedding for the phrase is by summing up the embeddings of its words:

$$E(phrase) = \sum_{m=1}^{M} E(word_m) \tag{1}$$

Considering an entire movie review to be a very long phrase, we could then train a binary logistic regression model with parameters $\mathbf{w}$ and b for sentiment classification. In that case, the larger

the logit score $z(phrase) = \mathbf{w}^T E(phrase) + b$, the higher the probability the model assigns to the positive sentiment for this *phrase*. Prove that in this approach, irrespective of the model parameters, the inequalities below cannot both hold:

$$z(good) > z(not\ good) \tag{2}$$
$$z(bad) < z(not\ bad) \tag{3}$$

2.4.1 YOUR SOLUTION goes here.

2.5 Theory (Bonus points): Time and memory complexity

1. Describe an **efficient** procedure (pseudocode) for computing the $tf.idf$ vectors for all the words in a vocabulary V, given a set of documents D that contain a total of N word occurrences, and a context window of size C. Compute its time and memory complexity, as a function of the size of V, D, N, and C.
2. What are the time and memory complexity of the skip-gram word2vec model described in class for learning dense word embeddings? Assume the vocabulary is V, the corpus is a sequence of words of length N, the context window contains C words, and that for every context word we sample K negative words. Assume a gradient descent update is made for each center (target) word, and that the algorithm runs E passes over the entire corpus.

2.5.1 YOUR SOLUTION goes here.

3 Implementation

[1]:
```
# All required import statements are here.
from collections import defaultdict, Counter
import math
import operator
import gzip

import numpy as np
import random
import scipy as sp
from sklearn.decomposition import TruncatedSVD

import pprint
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [10, 5]

np.random.seed(0)
random.seed(0)
```

[2]:
```
#conda install gensim
```

3.1 Part 1: Count-Based Word Vectors

Most word vector models start from the following idea:

You shall know a word by the company it keeps (Firth, J. R. 1957:11)

Many word vector implementations are driven by the idea that similar words, i.e., (near) synonyms, will be used in similar contexts. As a result, similar words will often be spoken or written along with a shared subset of words, i.e., contexts. By examining these contexts, we can try to develop embeddings for our words. With this intuition in mind, many "old school" approaches to constructing word vectors relied on word counts.

This part explores distributional similiarity in a dataset of 10,000 Wikipedia articles (4.4M words), building high-dimensional, sparse representations for words from the distinct contexts they appear in. These representations allow for analysis of the most similar words to a given query.

[3]:
```
window = 4
vocabSize = 10000
```

3.1.1 Question 1.1: Load corpus and create document frequency dictionary

Load the data from the Wikipedia file. Each line contains a Wikipedia document. After running this code, `wiki_data` should contain a list of all lowercased tokens in the corpus that contain only letters, whereas `dfs` should be a dictionary that maps each unique token to the number of Wikipedia documents in which the token appears (i.e. its document frequency).

[4]:
```
filename = "../data/wiki.10K.txt"

dfs = defaultdict(int)
Ndocs = 0
wiki_data = []
with open(filename, 'r', encoding = "utf-8") as fwiki:
    for line in fwiki:
        tokens = [t for t in line.lower().split() if t.isalpha()]
        # YOUR CODE HERE
        Ndocs += 1
        token_tracker = []

        if Ndocs % 1000 == 0:
            print('Processed %d documents' % Ndocs)

        for t in tokens:

            if t not in dfs and t not in token_tracker:
                dfs[t] = 1
            if t in dfs and t not in token_tracker:
                dfs[t] += 1

            token_tracker.append(t)

        #tokens_add = [x for x in token_tracker if x not in wiki_data]
        wiki_data.extend(token_tracker)
```

```
# Should be 10k documents
print('Total number of documents:', Ndocs)
```

```
Processed 1000 documents
Processed 2000 documents
Processed 3000 documents
Processed 4000 documents
Processed 5000 documents
Processed 6000 documents
Processed 7000 documents
Processed 8000 documents
Processed 9000 documents
Processed 10000 documents
Total number of documents: 10000
```

[5]:
```
# Let's print 20 tokens with the largest document frequency.
top = sorted(dfs.items(), key = lambda item: item[1], reverse = True)
print(top[:20])
```

```
[('the', 9579), ('in', 9275), ('a', 9161), ('of', 8773), ('is', 8222), ('and',
8107), ('was', 6894), ('to', 6803), ('as', 5854), ('by', 5836), ('on', 5822),
('for', 5795), ('with', 5420), ('at', 5316), ('from', 5290), ('it', 5080),
('an', 4967), ('that', 4015), ('also', 3881), ('which', 3716)]
```

[6]:
```
# We'll only create word representation for the most frequent K words
def create_vocab(data):
    word_representations = {}
    vocab = Counter()
    for i, word in enumerate(data):
        vocab[word] += 1

    topK = [k for k,v in vocab.most_common(vocabSize)]
    for k in topK:
        word_representations[k] = defaultdict(float)
    return word_representations
```

[7]:
```
# Word representation for a word = its unigram distributional context (the␣
 ↪unigrams that show
# up in a window before and after its occurence)
def count_unigram_context(data, word_representations):
    for i, word in enumerate(data):
        if word not in word_representations:
            continue
        start = i - window if i - window > 0 else 0
        end = i + window + 1 if i + window + 1 < len(data) else len(data)
        for j in range(start, end):
            if i != j:
```

```
                word_representations[word][data[j]] += 1
```

[8]:
```
# Normalize a word representation vector that its L2 norm is 1.
# We do this so that the cosine similarity reduces to a simple dot product
def normalize(word_representations):
    for word in word_representations:
        total = 0
        for key in word_representations[word]:
            total += word_representations[word][key] *␣
 ↪word_representations[word][key]

        total = math.sqrt(total)
        for key in word_representations[word]:
            word_representations[word][key] /= total
```

[9]:
```
def dictionary_dot_product(dict1, dict2):
    dot=0
    for key in dict1:
        if key in dict2:
            dot += dict1[key] * dict2[key]
    return dot
```

[10]:
```
def find_sim(word_representations, query):
    if query not in word_representations:
        print("'%s' is not in vocabulary" % query)
        return None

    scores = {}
    for word in word_representations:
        cosine = dictionary_dot_product(word_representations[query],␣
 ↪word_representations[word])
        scores[word] = cosine
    return scores
```

[11]:
```
# Find the K words with highest cosine similarity to a query in a set of␣
 ↪word_representations
def find_nearest_neighbors(word_representations, query, K):
    scores  =find_sim(word_representations, query)
    if scores != None:
        sorted_x = sorted(scores.items(), key = operator.itemgetter(1),␣
 ↪reverse=True)
        for idx, (k, v) in enumerate(sorted_x[:K]):
            print("%s\t%s\t%.5f" % (idx,k,v))
```

[12]:
```
word_representations = create_vocab(wiki_data)
count_unigram_context(wiki_data, word_representations)
normalize(word_representations)
```

[13]:
```
find_nearest_neighbors(word_representations, "musician", 10)
```

```
0       musician        1.00000
1       writer  0.91968
2       journalist      0.89769
3       photographer    0.89593
4       pianist 0.89561
5       businessman     0.89480
6       poet    0.88196
7       politician      0.88074
8       entrepreneur    0.87987
9       composer        0.87770
```

3.1.2 Question 1.2: Implement Tf.Idf Representation

Q1: Fill out a function `tfidf` below. This function takes as input a dict of word_representations and for each context word in `word_representations[word]` replaces its *count* value with its tf-idf score. Use $\log(count + 1)$ for tf and $\log \frac{N}{df}$ for idf. This function should modify `word_representations` in place.

[14]:
```
def tfidf(word_representations):
    for word in word_representations:
        for key in word_representations[word]:
            # YOUR CODE HERE

            count = word_representations[word][key]
            tf = math.log(count + 1)
            idf = math.log(count / Ndocs)

            word_representations[word][key] = tf * idf
```

[15]:
```
tf_idf_word_representations = create_vocab(wiki_data)
count_unigram_context(wiki_data, tf_idf_word_representations)
tfidf(tf_idf_word_representations)
normalize(tf_idf_word_representations)
```

3.1.3 Question 1.3: Compare Count Representations with Tf.Idf Representations

How does the tf.idf representation change the the nearest neighbors? Use `find_nearest_neighbors` on some of the words below.

[16]:
```
query = "musician" # "musician" student" "education" "bacteria" "beer" "brook"␣
 ↪"greedy" "carbon" "prisoner" "river" "mountain" "germany" "child" "computer"␣
 ↪"actor" "science"
find_nearest_neighbors(word_representations, query, 10)
print()
find_nearest_neighbors(tf_idf_word_representations, query, 10)
```

```
0       musician        1.00000
1       writer  0.91968
2       journalist      0.89769
3       photographer    0.89593
4       pianist 0.89561
5       businessman     0.89480
6       poet    0.88196
7       politician      0.88074
8       entrepreneur    0.87987
9       composer        0.87770

0       musician        1.00000
1       songwriter      0.44070
2       pianist 0.42982
3       singer  0.41835
4       actor   0.41393
5       producer        0.40596
6       composer        0.40422
7       performer       0.39974
8       guitarist       0.39887
9       jazz    0.39546
```

It seems that the tf.idf representation changes the nearest neighbors in respect to their cosine similarity scores. The scores seen using the td.idf representation are much lower – basically half of their previous scores. Despite these lower scores, it seems that the words are more similar when using the tf.idf representation. Previously, the similar words that were returned were similar to some degree as all of the words listed were also professions (journalist, politician, etc.). However, when we use the tf.idf representation, the words that returned were more similar (songwriter, singer, composer, etc.) by remaining within the realm of the arts (we are matching with musician). In other words, the tf.idf representation resulted in neighbors with a higher degree of similarity.

3.2 Part 2: Prediction-Based Word Vectors

As discussed in class, more recently prediction-based word vectors have come into fashion, e.g. word2vec. Here, we shall explore the embeddings produced by word2vec. Please revisit the class notes and lecture slides for more details on the word2vec algorithm. If you're feeling adventurous, challenge yourself and try reading the original paper.

Then run the following cells to load the word2vec vectors into memory. **Note**: This might take several minutes.

[17]:
```
def load_word2vec():
    """ Load Word2Vec Vectors
        Return:
            wv_from_bin: All 3 million embeddings, each lengh 300
    """
    import gensim.downloader as api
    wv_from_bin = api.load("word2vec-google-news-300")
    vocab = list(wv_from_bin.key_to_index.keys())
```

```
    print("Loaded vocab size %i" % len(vocab))
    return wv_from_bin
```

[18]:
```
# -----------------------------------
# Run Cell to Load Word Vectors
# Note: This may take several minutes
# -----------------------------------
wv_from_bin = load_word2vec()
```

```
Loaded vocab size 3000000
```

Note: If you are receiving out of memory issues on your local machine, try closing other applications to free more memory on your device. You may want to try restarting your machine so that you can free up extra memory. Then immediately run the jupyter notebook and see if you can load the word vectors properly.

jupyter notebook –NotebookApp.iopub_msg_rate_limit=1.0e10

3.2.1 Question 2.1: Compare Word2Vec Embeddings with Co-occurrence Embeddings

Let's use the word2vec embeddings to find the most similar words, usign the same targets as in part 1 above. Compare the quality of the top 10 words using word2vec with the top 10 most similar words from part 1 above. Which method is better?

[19]:
```
wv_from_bin.most_similar("musician") # "musician" student" beer" education"␣
↪"bacteria" "brook" "greedy" "carbon" "prisoner" "river" "mountain" "germany"␣
↪"child" "computer" "actor" "science"
```

[19]:
```
[('singer_songwriter', 0.7869659662246704),
 ('jazz_musician', 0.7756308317184448),
 ('songwriter', 0.7425625920295715),
 ('singer', 0.7350084781646729),
 ('guitarist', 0.7226887345314026),
 ('musicians', 0.7069464325904846),
 ('jazz_saxophonist', 0.7034097909927368),
 ('pianist', 0.7009111642837524),
 ('Musician', 0.6974103450775146),
 ('musican', 0.6972053647041321)]
```

To determine which method is better may depend on the desired results. It seems that the Word2Vec embeddings resulted in a good performance as far as finding similar words such as songwriter, singer, guitarist, etc. However, many of the "similar" words that were returned were simply other forms of the same word – we searched for words similar to 'musician' and some returned words were 'Musician,' 'musican,' and 'jazz_musician.' It would be incorrect to say that these words are not similar, so we cannot disagree with these results. However, we are essentially just looking at the same word, not a similar word. This happens because of the dense embedding that is being used here. Contextual information that is stored in the word embeddings is causing the words 'musician'

and 'Musician' to be represented as different words, although, ignoring the capitalization, they are not.

In my opinion, I would prefer the results from part 1. This is because I am able to see more similar words rather than several versions of the same word I am trying to match.

3.2.2 Reducing dimensionality of Word2Vec Word Embeddings

1. Put the 3 million word2vec vectors into a matrix M
2. Run reduce_to_k_dim (your Truncated SVD function) to reduce the vectors from 300-dimensional to 2-dimensional.

Here, we construct a method that performs dimensionality reduction on the matrix to produce k-dimensional embeddings. We use SVD to take the top k components and produce a new matrix of k-dimensional embeddings.

Note: All of numpy, scipy, and scikit-learn (`sklearn`) provide *some* implementation of SVD, but only scipy and sklearn provide an implementation of Truncated SVD, and only sklearn provides an efficient randomized algorithm for calculating large-scale Truncated SVD. So please use sklearn.decomposition.TruncatedSVD.

[20]:
```
def reduce_to_k_dim(M, k = 2):
    """ Reduce a co-occurence count matrix of dimensionality (num_corpus_words,␣
↪num_corpus_words)
        to a matrix of dimensionality (num_corpus_words, k) using the following␣
↪SVD function from Scikit-Learn:
            - http://scikit-learn.org/stable/modules/generated/sklearn.
↪decomposition.TruncatedSVD.html

        Params:
            M (numpy matrix of shape (number of corpus words, number of corpus␣
↪words)): co-occurence matrix of word counts
            k (int): embedding size of each word after dimension reduction
        Return:
            M_reduced (numpy matrix of shape (number of corpus words, k)):␣
↪matrix of k-dimensioal word embeddings.
                    In terms of the SVD from math class, this actually returns␣
↪U * S
    """
    n_iters = 10     # Use this parameter in your call to `TruncatedSVD`
    M_reduced = None
    print("Running Truncated SVD over %i words..." % (M.shape[0]))

    # ------------------
    svd = TruncatedSVD(n_components = k, n_iter = n_iters, random_state=42)
    svd.fit(M)
    M_reduced = M @ svd.components_.T

    # ------------------
```

```
    print("Done.")

    return M_reduced
```

[21]:
```
def get_matrix_of_vectors(wv_from_bin, required_words = []):
    """ Put the word2vec vectors into a matrix M.
        Param:
            wv_from_bin: KeyedVectors object; the 3 million word2vec vectors␣
 ↪loaded from file
        Return:
            M: numpy matrix shape (num words, 300) containing the vectors
            word2Ind: dictionary mapping each word to its row number in M
    """
    import random
    words = list(wv_from_bin.index_to_key)
    print("Shuffling words ...")
    random.shuffle(words)
    words = words[:10000]
    print("Putting %i words into word2Ind and matrix M..." % len(words))
    word2Ind = {}
    M = []
    curInd = 0
    for w in words:
        try:
            M.append(wv_from_bin.word_vec(w))
            word2Ind[w] = curInd
            curInd += 1
        except KeyError:
            continue
    for w in required_words:
        try:
            M.append(wv_from_bin.word_vec(w))
            word2Ind[w] = curInd
            curInd += 1
        except KeyError:
            continue
    M = np.stack(M)
    print("Done.")
    return M, word2Ind
```

3.2.3 Question 2.2: Word2Vec Plot Analysis

Here we write a function to plot a set of 2D vectors in 2D space. For graphs, we will use Matplotlib (`plt`).

[22]:
```
def plot_embeddings(M_reduced, word2Ind, words):
    """ Plot in a scatterplot the embeddings of the words specified in the list␣
 ↪"words".
        NOTE: do not plot all the words listed in M_reduced / word2Ind.
        Include a label next to each point.

        Params:
            M_reduced (numpy matrix of shape (number of unique words in the␣
 ↪corpus , k)): matrix of k-dimensional word embeddings
            word2Ind (dict): dictionary that maps word to indices for matrix M
            words (list of strings): words whose embeddings we want to visualize
    """

    # ------------------
    xvals = []
    yvals = []
    for word in words:
        embed2D = M_reduced[word2Ind[word]]
        xvals.append(embed2D[0])
        yvals.append(embed2D[1])

    fig, ax = plt.subplots()
    ax.scatter(xvals, yvals)

    for i, word in enumerate(words):
        ax.annotate(word, (xvals[i], yvals[i]))
    # ------------------
```

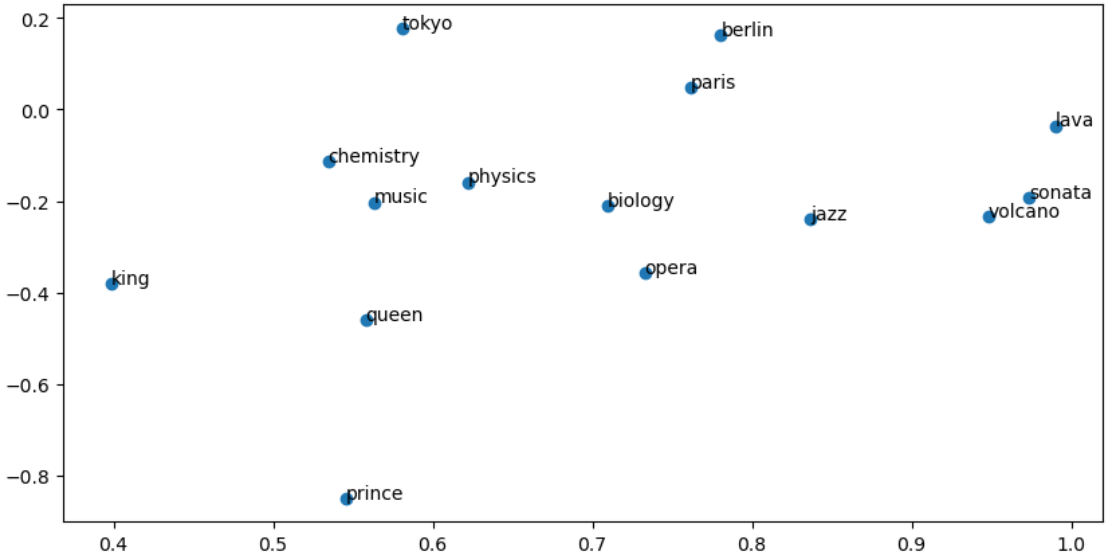
Run the cell below to plot the 2D word2vec embeddings for ['music', 'jazz', 'opera', 'paris', 'berlin', 'tokyo', 'queen', 'king', 'prince', 'volcano', 'chemistry', 'biology', 'physics', 'lava', 'sonata'].

What clusters together in 2-dimensional embedding space? What doesn't cluster together that you might think should have?

[23]:
```
# -----------------------------------------------------------------
# Run this code to Reduce 300-Dimensinal Word Embeddings to k Dimensions
# Note: This may take several minutes
# -----------------------------------------------------------------
words = ['music', 'jazz', 'opera', 'paris', 'berlin', 'tokyo', 'queen', 'king',␣
 ↪'prince', 'volcano', 'chemistry', 'biology', 'physics', 'lava', 'sonata']
M, word2Ind = get_matrix_of_vectors(wv_from_bin, required_words = words)
M_reduced = reduce_to_k_dim(M, k = 2)

plot_embeddings(M_reduced, word2Ind, words)
```

```
Shuffling words …
Putting 10000 words into word2Ind and matrix M…
```

```
Done.
Running Truncated SVD over 10015 words…

C:\Users\user\AppData\Local\Temp\ipykernel_6036\3672454343.py:20:
DeprecationWarning: Call to deprecated `word_vec` (Use get_vector instead).
  M.append(wv_from_bin.word_vec(w))
C:\Users\user\AppData\Local\Temp\ipykernel_6036\3672454343.py:27:
DeprecationWarning: Call to deprecated `word_vec` (Use get_vector instead).
  M.append(wv_from_bin.word_vec(w))

Done.
```



It seems that words are being clustered based on category. For example, geographical locations tokyo, berlin, and paris are clustered while subjects chemistry, music, physics, and biology are in another cluster. Another example could be positions of royalty such as king, queen, and prince, which are also clustered together in the 2-dimensional space.

It is surprising to see that music did not cluster closely with opera, jazz, and sonata. It seems that opera and jazz, which are genres of music, remained the closest together while sonata, and music (which are, of course, in the music category) did not cluster together. Following the observed pattern of clustering based on category leads me to believe that these words should have clustered together.

3.2.4 Cosine Similarity

Now that we have word vectors, we need a way to quantify the similarity between individual words, according to these vectors. One such metric is cosine-similarity. We will be using this to find words that are "close" and "far" from one another.

We can think of n-dimensional vectors as points in n-dimensional space. If we take this perspective L1 and L2 Distances help quantify the amount of space "we must travel" to get between these two

points. Another approach is to examine the angle between two vectors. From trigonometry we know that:

Instead of computing the actual angle, we can leave the similarity in terms of $similarity = cos(\Theta)$. Formally the Cosine Similarity s between two vectors p and q is defined as:

$$s = \frac{p \cdot q}{||p||||q||}, \text{ where } s \in [-1, 1]$$

3.2.5 Question 2.3: Polysemous Words [code + written]

Find a polysemous word (for example, "leaves" or "scoop") such that the top-10 most similar words (according to cosine similarity) contains related words from *both* meanings. For example, "leaves" has both "vanishes" and "stalks" in the top 10, and "scoop" has both "handed_waffle_cone" and "lowdown". You will probably need to try several polysemous words before you find one. Please state the polysemous word you discover and the multiple meanings that occur in the top 10. Why do you think many of the polysemous words you tried didn't work?

Note: You should use the `wv_from_bin.most_similar(word)` function to get the top 10 similar words. This function ranks all other words in the vocabulary with respect to their cosine similarity to the given word. For further assistance please check the **GenSim documentation**.

[24]:
```
# ------------------
# Write your polysemous word exploration code here.

wv_from_bin.most_similar("head")

# ------------------
# Didn't work: sound (noise vs. solid/reliable), bright (luminous vs␣
 ↪intelligent), bank (river side vs finanical)
# Basically worked: toast (bread vs cheers)
```

[24]:
```
[('heads', 0.6393267512321472),
 ('Head', 0.5965096354484558),
 ('director', 0.5187345743179321),
 ('assistant', 0.5164069533348083),
 ('deputy', 0.4884401857852936),
 ('chief', 0.4807194173336029),
 ('chair', 0.47800156474113464),
 ('Youssef_Kanjo', 0.462150514125824),
 ('arm', 0.4410949945449829),
 ('vice_president', 0.43873125314712524)]
```

It is likely that many of the polysemous words that I tried did not work because only one word meaning was learned. This is likely due to the context that the word is most often seen or used in. For example, if the word "sound" is used most often in reference to noise, it makes sense for that word meaning to be recognized and learned. If "sound" is rarely used in reference to a stable or good condition, this word meaning will not be recognized but will likely be treated as outlying cases instead. To offer another explanation, since we are clustering words based on their similarity

(by category), we will likely see it tied more strongly to one category that represents its most common use. Since we are specifically searching for words that can have multiple "categories," it makes sense for there to be trouble with this task unless the word is commonly used in both contexts/categories.

3.2.6 Question 2.4: Synonyms & Antonyms

When considering Cosine Similarity, it's often more convenient to think of Cosine Distance, which is simply 1 - Cosine Similarity.

Find three words (w1,w2,w3) where w1 and w2 are synonyms and w1 and w3 are antonyms, but Cosine Distance(w1,w3) < Cosine Distance(w1,w2). For example, w1="happy" is closer to w3="sad" than to w2="cheerful".

Once you have found your example, please give a possible explanation for why this counter-intuitive result may have happened.

You should use the the `wv_from_bin.distance(w1, w2)` function here in order to compute the cosine distance between two words. Please see the **GenSim documentation** for further assistance.

[25]:
```
# ------------------
# Write your synonym & antonym exploration code here.

w1 = "small"
w2 = "miniature"
w3 = "big"
w1_w2_dist = wv_from_bin.distance(w1, w2)
w1_w3_dist = wv_from_bin.distance(w1, w3)

print("Synonyms {}, {} have cosine distance: {}".format(w1, w2, w1_w2_dist))
print("Antonyms {}, {} have cosine distance: {}".format(w1, w3, w1_w3_dist))

# ------------------
```

```
Synonyms small, miniature have cosine distance: 0.6674894094467163
Antonyms small, big have cosine distance: 0.5041321516036987
```

In the case where w1 = "small," w2 = "miniature," and w3 = "big," the cosine distance between the antonyms "small" and "big" is smaller than the cosine distance between the synonyms "small" and "miniature." This could have possibly occured due to the fact that "small" and "big" are both vague, weak words to describe size, while miniature is a much more detailed, strong word. For example, an item that is described as miniature is likely drastically smaller than an item that is described as small – the word "miniature" is stronger when used in the same context. So, as far as the weight that these words hold as adjectives in context, it makes sense for "small" and "big" to be more similar (have a smaller distance) than "small" and "miniature."

3.2.7 Solving Analogies with Word Vectors

Word2Vec vectors have been shown to *sometimes* exhibit the ability to solve analogies.

As an example, for the analogy "man : king :: woman : x", what is x?

In the cell below, we show you how to use word vectors to find x. The `most_similar` function finds words that are most similar to the words in the `positive` list and most dissimilar from the words in the `negative` list. The answer to the analogy will be the word ranked most similar (largest numerical value).

Note: Further Documentation on the `most_similar` function can be found within the **GenSim documentation**.

[26]:
```
# Run this cell to answer the analogy -- man : king :: woman : x
pprint.pprint(wv_from_bin.most_similar(positive=['woman', 'king'],␣
 ↪negative=['man']))
```

```
[('queen', 0.7118193507194519),
 ('monarch', 0.6189674735069275),
 ('princess', 0.5902431011199951),
 ('crown_prince', 0.5499460697174072),
 ('prince', 0.5377321243286133),
 ('kings', 0.5236844420433044),
 ('Queen_Consort', 0.5235945582389832),
 ('queens', 0.518113374710083),
 ('sultan', 0.5098593831062317),
 ('monarchy', 0.5087411999702454)]
```

3.2.8 Question 2.5: Finding Analogies

Find 5 examples of analogies that holds according to these vectors (i.e. the intended word is ranked top). In your solution please state the full analogy in the form x:y :: a:b. If you believe the analogy is complicated, explain why the analogy holds in one or two sentences.

Note: You may have to try many analogies to find ones that work!

Document also 5 examples of analogies that do not hold according to the learned word vectors.

[33]:
```
# ------------------
# Write your analogy exploration code here.

# WORKS! France : Paris :: Poland : Warsaw
pprint.pprint(wv_from_bin.most_similar(positive=['Poland', 'Paris'],␣
 ↪negative=['France']))

# WORKS! boy : son :: girl : daughter
#pprint.pprint(wv_from_bin.most_similar(positive=['girl', 'son'],␣
 ↪negative=['boy']))

# WORKS! bad : worst :: big : biggest
#pprint.pprint(wv_from_bin.most_similar(positive=['big', 'worst'],␣
 ↪negative=['bad']))

# WORKS! do : did :: go : went
```

```
#pprint.pprint(wv_from_bin.most_similar(positive=['go', 'did'],␣
 ↪negative=['do']))

# WORKS: dog : puppy :: cat : kitten
#pprint.pprint(wv_from_bin.most_similar(positive=['cat', 'puppy'],␣
 ↪negative=['dog']))

# Doesn't work! glove : hand :: sock : foot --> top word is "Hand"
#pprint.pprint(wv_from_bin.most_similar(positive=['sock', 'hand'],␣
 ↪negative=['glove']))

# Doesn't work! winter : cocoa :: summer : lemonade/iced tea --> top word is␣
 ↪'cocoa_beans'
#pprint.pprint(wv_from_bin.most_similar(positive=['summer', 'cocoa'],␣
 ↪negative=['winter']))

# Doesn't work! good : heaven :: bad : hell --> top word is 'heavens'
#pprint.pprint(wv_from_bin.most_similar(positive=['bad', 'heaven'],␣
 ↪negative=['good']))

# Doesn't work! car : road :: plane : sky --> top word is "taxiway"
#pprint.pprint(wv_from_bin.most_similar(positive=['plane', 'road'],␣
 ↪negative=['car']))

# Doesn't work! shovel : dirt :: pickaxe : stone --> top word is "dusty"
#pprint.pprint(wv_from_bin.most_similar(positive=['pickaxe', 'dirt'],␣
 ↪negative=['shovel']))


# ------------------
```

```
[('Warsaw', 0.7734049558639526),
 ('Prague', 0.6898577809333801),
 ('Budapest', 0.6714177131652832),
 ('Krakow', 0.6266399025917053),
 ('Poznań', 0.599148690700531),
 ('Kraków', 0.5872530341148376),
 ('Bratislava', 0.5808363556861877),
 ('Moscow', 0.5766443610191345),
 ('Polish', 0.5763624310493469),
 ('Lodz', 0.5718404054641724)]
```

Analogies That Work:

1. France : Paris :: Poland : Warsaw
2. boy : son :: girl : daughter

3. bad : worst :: big : biggest
4. do : did :: go : went
5. dog : puppy :: cat : kitten

Analogies That Do Not Work:

1. glove : hand :: sock : foot
2. winter : cocoa :: summer : lemonade/iced tea
3. good : heaven :: bad : hell
4. car : road :: plane : sky
5. shovel : dirt :: pickaxe : stone

The second analogy listed that did not work when using word vectors to solve is intended to be a comparison of popular seasonal drinks. In this case, we are drawing a comparison between the drink hot cocoa (or hot chocolate) that is very popular in winter and the drink lemonade (or iced tea could also work) that is very popular in the summer. It could be that this analogy does not hold as it could be a local or cultural trend that isn't represented in the data.

3.2.9 Question 2.6: Guided Analysis of Bias in Word Vectors

It's important to be cognizant of the biases (gender, race, sexual orientation etc.) implicit to our word embeddings.

Run the cell below, to examine (a) which terms are most similar to "woman" and "boss" and most dissimilar to "man", and (b) which terms are most similar to "man" and "boss" and most dissimilar to "woman". What do you find in the top 10?

[28]:
```
# Run this cell
# Here `positive` indicates the list of words to be similar to and `negative`␣
 ↪indicates the list of words to be
# most dissimilar from.
pprint.pprint(wv_from_bin.most_similar(positive=['woman', 'boss'],␣
 ↪negative=['man']))
print()
pprint.pprint(wv_from_bin.most_similar(positive=['man', 'boss'],␣
 ↪negative=['woman']))
```

```
[('bosses', 0.5522644519805908),
 ('manageress', 0.49151360988616943),
 ('exec', 0.459408164024353),
 ('Manageress', 0.4559843838214874),
 ('receptionist', 0.4474116563796997),
 ('Jane_Danson', 0.44480544328689575),
 ('Fiz_Jennie_McAlpine', 0.44275766611099243),
 ('Coronation_Street_actress', 0.44275572896003723),
 ('supremo', 0.4409853219985962),
```

```
 ('coworker', 0.43986251950263977)]

[('supremo', 0.6097398400306702),
 ('MOTHERWELL_boss', 0.5489562153816223),
 ('CARETAKER_boss', 0.5375303030014038),
 ('Bully_Wee_boss', 0.5333974361419678),
 ('YEOVIL_Town_boss', 0.5321705341339111),
 ('head_honcho', 0.5281980037689209),
 ('manager_Stan_Ternent', 0.525971531867981),
 ('Viv_Busby', 0.5256164073944092),
 ('striker_Gabby_Agbonlahor', 0.5250812768936157),
 ('BARNSLEY_boss', 0.5238943696022034)]
```

It seems that the top 10 most similar words differ between the two cases ("woman" and "boss" vs "man" and "boss"). There is only one word that is found in both lists: "supremo," though it is considered to be more similar to man than woman. In addition, there are more feminine terms in the list associated with woman such as "manageress" and "Manageress" while there are more word pairs with "boss" such as "CARETAKER_boss" and "MOTHERWELL_boss" that are shown in the list associated with man. There are also less powerfull terms such as "receptionist" and "coworker" are shown in the list associated with woman while the powerful term "head_honcho" shows up in the list associated with man.

It does seem that there is a bias here that tends to relate more powerful roles with men over women. This is likely due to the data that makes up the vocabulary – the powerful words and powerful roles are seen in contexts with men more often than women.

3.2.10 Question 2.7: Independent Analysis of Bias in Word Vectors

Use the `most_similar` function to find at least 2 other cases where some bias is exhibited by the vectors. Please briefly explain the type of bias that you discover.

[70]:
```
# ------------------
# Write your bias exploration code here.

print("First Bias: weath\n")
# Interesting that poor students are associated with suspensions and expulsions␣
 ↪but rich students aren't
pprint.pprint(wv_from_bin.most_similar(positive=['rich', 'student'],␣
 ↪negative=['poor']))
print()
pprint.pprint(wv_from_bin.most_similar(positive=['poor', 'student'],␣
 ↪negative=['rich']))
print()

print("Second Bias: age\n")
# Seems to be an age bias. Many more intelligence words associated with young.␣
 ↪Not many with old.
pprint.pprint(wv_from_bin.most_similar(positive=['young', 'intelligent'],␣
 ↪negative=['old']))
```

```
print()
pprint.pprint(wv_from_bin.most_similar(positive=['old', 'intelligent'],␣
 ↪negative=['young']))

# ------------------
```

```
First Bias: weath

[('students', 0.44873565435409546),
 ('faculty', 0.4361094832420349),
 ('teacher', 0.41817253828048706),
 ('undergraduate', 0.4101096987724304),
 ('professors', 0.4046473801136017),
 ('Student', 0.40239205956459045),
 ('gradate', 0.4003406763076782),
 ('precalculus', 0.3952144682407379),
 ('PBAU', 0.3928479254245758),
 ('campus', 0.3917272984981537)]

[('students', 0.5209745764732361),
 ('stu_dent', 0.5054574608802795),
 ('Student', 0.48852047324180603),
 ('school', 0.46412113308906555),
 ('suspensions_expulsions', 0.42189735174179077),
 ('teacher', 0.42129772901535034),
 ('Shanno_Khan', 0.42003384232521057),
 ('university', 0.4132259786128998),
 ('Uprep', 0.409466415643692),
 ('academic', 0.4093310236930847)]

Second Bias: age

[('smart', 0.5389206409454346),
 ('talented', 0.48834770917892456),
 ('perceptive', 0.46465036273002625),
 ('creative_problem_solvers', 0.4645535945892334),
 ('intellegent', 0.46018487215042114),
 ('articulate', 0.4553522765636444),
 ('educated', 0.45132336020469666),
 ('entrepreneurially_minded', 0.449795663356781),
 ('wellrounded', 0.4494137763977051),
 ('wheelchair_TAO', 0.4480825960636139)]

[('Telkonet_SmartEnergy_TSE', 0.3959064483642578),
 ('wheelchair_TAO', 0.39093920588493347),
 ('Kamran_Saleen', 0.36364445090293884),
 ('charismatic_manipulator', 0.3493396043777466),
```

```
('intuitive', 0.3485572040081024),
('Inc._Nasdaq_ADEP', 0.3485078513622284),
('antireflective_AR_coating', 0.34809496998786926),
('SVC_AIM_SAND', 0.345797061920166),
('testable_theory', 0.3299047648906708),
('Sandvine_TSX', 0.32943034172058105)]
```

First Bias: Weath It seems that there is some bias exhibited in association with wealth. As seen above, when looking for words similar to "rich" and "student" and dissimilar from "poor," the results list many words that are obviously associated with education such as "undergraduate" and "precalculus." Similarly, when looking for words similar to "poor" and "student" and dissimilar from "rich," the results list words that are obviously associated with education such as "university" and "academic." The results between these two cases are very similar, which would hopefully be the case as wealth should not affect education. However, there is a major outliar between the two lists that does indicate a bias based on wealth. When finding similar words with "student" and "poor," the term "suspensions_expulsions" which is a negative term associated with education is shown. This word does not appear when finding similar words with "student" and "rich." Obviously, a student getting suspended or expelled should have nothing to do with their economic status. So, why does this term only show as similar when comparing with "poor" and not "rich?" There is a bias here.

Second Bias: Age It seems that there is also some bias exhibited in association with age. As seen above, when looking for words similar to "young" and "intelligent" and dissimilar from "old," the results list 9 positive words that are generally associated with intelligence (there is one outliar, though it is seen in both similarity lists). On the other hand, when looking for words similar to "old" and "intelligent" and dissimilar from "young," the results list only 1 positive word that is generally associated with intelligence. The rest of the words listed seem to have little to no correlation with intelligence or old age. Why are young people 9 times as likely to be associated with positive words regarding intelligence? This shows a dramatic bias.

3.2.11 Question 2.8: Thinking About Bias

What might be the cause of these biases in the word vectors?

A possible cause of these biases in the word embeddings is the context in which the words are used within the data (corpus). As the vocabularity and word similarities are created using the documents within the corpus, it is important to recognize that the contents of the documents within the corpus will influence the learned word meanings. If there is bias in the word embeddings, that likely means that these biases are learned from the corpus. For example, in the experiment above we can see that powerful words and powerful roles are not associated as strongly with women as with men. This means that, within the corpus, men are more commonly seen in contexts that allow powerful words or powerful roles. Tbe same can be said for the bias that is recognized with weath and age.

3.3 Part 3: Using the chat completion API

Use the chat completion API to answer the questions 2.3 to 2.7, using the same examples. Always use a temperature of 0. Specify which GPT model you use.

Simply showign results is vastly insufficient. Most important is the analysis that you include at the top for each question.

Chat completion API setup

[72]:
```
import os
import openai
import tiktoken

from dotenv import load_dotenv, find_dotenv

# Read the local .env file, containing the Open AI secret key.
_ = load_dotenv(find_dotenv())
openai.api_key = os.environ['OPENAI_API_KEY']

# Let's use the `deeplearning.ai` approach and define a function for this use␣
 ↪pattern.
# Set `temperature = 0` to do greedy decoding => deterministic output.
def get_completion_from_messages(messages, model = "gpt-3.5-turbo", temperature␣
 ↪= 0, max_tokens = 500):
    response = openai.ChatCompletion.create(model = model,
                                            messages = messages,
                                            temperature = temperature, # the␣
 ↪degree of randomness of the model's output.
                                            max_tokens = max_tokens) # the␣
 ↪maximum number of tokens the model can output.
    return response.choices[0].message["content"]
```

3.4 Part 3.3: Polysemous Words

1. For the ploysemous word that you found at 2.3 above, ask the GPT model to return the 10 most similar words. Do the same for the polysemous words that you tried at 2.3 above for which the word embeedings approach did not work. For each ploysemous word (at least 5) that you tried, compare the GPT output with the output from 2.3.
2. For each polysemous word that you tried, further instruct GPT to partition the 10 most similar words so that words that have the same meaning are put in the same cluster. Evaluate its output.

[77]:
```
# YOUR CODE HERE

system_message = f"""
    You will be provided with a polysemous word. Return only the 10 most␣
 ↪similar words. \
    Partition the 10 most similar words so that words that have the same␣
 ↪meaning are put in the same cluster.

    """
```

```
user_message = "head"

messages =  [
{'role':'system', 'content': system_message},
{'role':'user', 'content': user_message},]

response = get_completion_from_messages(messages)

print(response)
```

```
Cluster 1: Body Part
1. face
2. neck
3. hair
4. eyes
5. mouth

Cluster 2: Leader/Chief
1. chief
2. leader
3. boss
4. captain
5. director
```

[78]:
```
user_message = "toast"

messages =  [
{'role':'system', 'content': system_message},
{'role':'user', 'content': user_message},]

response = get_completion_from_messages(messages)

print(response)
```

```
Cluster 1: Toast (noun) - a slice of bread that has been browned by exposure to
radiant heat

1. Bread
2. Bagel
3. Croissant
4. Muffin
5. English muffin

Cluster 2: Toast (verb) - to make a toast or propose a toast to someone or
something

1. Celebrate
```

```
2. Commemorate
3. Honor
4. Salute
5. Propose a toast
```

[79]:
```
user_message = "bright"

messages =  [
{'role':'system', 'content': system_message},
{'role':'user', 'content': user_message},]

response = get_completion_from_messages(messages)

print(response)
```

```
Cluster 1: Light
1. Light
2. Radiant
3. Shining
4. Luminous
5. Glowing

Cluster 2: Intelligent
1. Intelligent
2. Smart
3. Clever
4. Brilliant
5. Sharp
```

[80]:
```
user_message = "sound"

messages =  [
{'role':'system', 'content': system_message},
{'role':'user', 'content': user_message},]

response = get_completion_from_messages(messages)

print(response)
```

```
Cluster 1: Sound (as in noise)
1. Noise
2. Acoustic
3. Volume
4. Pitch
5. Tone
6. Frequency
7. Resonance
8. Echo
```

```
9. Amplify
10. Reverberation

Cluster 2: Sound (as in healthy or in good condition)
1. Healthy
2. Well
3. Fit
4. Robust
5. Vigorous
6. Strong
7. Sturdy
8. Soundness
9. Wholesome
10. Intact
```

[81]:
```
user_message = "bank"

messages =  [
{'role':'system', 'content': system_message},
{'role':'user', 'content': user_message},]

response = get_completion_from_messages(messages)

print(response)
```

```
Cluster 1: Financial Institution
1. branch
2. institution
3. lender
4. finance
5. credit
6. loan
7. deposit
8. account
9. transaction
10. mortgage

Cluster 2: River Bank
1. shore
2. edge
3. embankment
4. riverside
5. waterfront
6. quay
7. levee
8. dam
9. jetty
10. wharf
```

It seems that the GPT model did a much better job of understanding the polysemous words than word embeddings. Even in the cases that the word embeddings could not recognize the two meanings of the polysemous words at all (only one word meaning was recognized), the GPT model performed flawlessly. In the shown example case that the word embeddings allowed the identification of both word meanings of a polysemous word (part 2.3), there was an obvious bias towards one word meaning. Only one word was shown that indicated that both word meanings were recognized. This is not the case for the GPT model which shows drastically better results as seen above.

3.5 Part 3.4: Synonyms & Antonyms

For the same words w_1, w_2, and w_3 found at 2.4 above, obtain answers from GPT by asking: 1. "Of the two words, w_2 and w_3, which one is more similar to w_1?" 2. "Of the two words, w_2 and w_3, which one is closer in meaning to w_1?" 3. "Of the two words, w_2 and w_3, which one is more related to w_1?"

Compare to what you obtained at 2.4.

[86]:
```
# YOUR CODE HERE

system_message = f"""
    You will be provided with three words in the format (w1, w2, w3). For this␣
 ↪word set, return the following: \
    1. Of the two words, w2 and w3, which one is more similar to w1? \
    2. Of the two words, w2 and w3, which one is closer in meaning to w1? \
    3. Of the two words, w2 and w3, which one is more related to w1?

    """

user_message = "(small, miniature, big)"

messages =  [
{'role':'system', 'content': system_message},
{'role':'user', 'content': user_message},]

response = get_completion_from_messages(messages)

print(response)
```

```
1. Of the two words, miniature and big, miniature is more similar to small.
2. Of the two words, miniature and big, miniature is closer in meaning to small.
3. Of the two words, miniature and big, miniature is more related to small.
```

The GPT results for this task differ from the results that are seen in part 2.4. In part 2.4, the cosine distance between "big" and "small" is smaller than the cosine distance between "small" and "miniature," which indicates that "big" and "small" have a higher degree of similarity. This is not the case when we use the GPT model to determine similarity.

The reasons behind the results in section 2.4 have been discussed, but in comparison, it seems GPT is focusing much more on the actual word meanings rather than the contextual similarity

between the words (though the two could technically be used interchangeably). To put it simply, GPT is basing the answer off of actual word meaning rather than the similarity in contexts when these words are used. For example, it would be more likley to see the sentences "the box is small" and the "box is big" rather than the sentence "the box is miniature." This could be why the word embeddings find "big" and "small" to be more similar. However, GPT understands that "small" and "miniature" are both being used to decribe something tiny in size, while "big" is describing the opposite characteristic.

3.6 Part 3.5: Finding Analogies

Use the GPT API to answer the same 5 analogical questions that you covered at 2.5 above, and compare.

[102]:
```
# YOUR CODE HERE

system_message = f"""
    You will be provided with three words that create an analogy in the format␣
 ↪(w1, w2, w3). For this word set, \
    complete the analogy by finding w4. Respond in the following format: "w1 :␣
 ↪w2 :: w3 : w4"
    For example, given the wordset (man, king, woman), the correct response␣
 ↪would be: "woman : queen :: man : king"

    """

user_message = "(France, Paris, Poland)"

messages =  [
{'role':'system', 'content': system_message},
{'role':'user', 'content': user_message},]

response = get_completion_from_messages(messages)

print(response)
```

```
France : Paris :: Poland : Warsaw
```

[92]:
```
user_message = "(boy, son, girl)"

messages =  [
{'role':'system', 'content': system_message},
{'role':'user', 'content': user_message},]

response = get_completion_from_messages(messages)

print(response)
```

```
boy : son :: girl : daughter
```

[93]:
```
user_message = "(bad, worst, big)"

messages =  [
{'role':'system', 'content': system_message},
{'role':'user', 'content': user_message},]

response = get_completion_from_messages(messages)

print(response)
```

```
big : biggest :: bad : worst
```

[94]:
```
user_message = "(do, did, go)"

messages =  [
{'role':'system', 'content': system_message},
{'role':'user', 'content': user_message},]

response = get_completion_from_messages(messages)

print(response)
```

```
do : did :: go : went
```

[95]:
```
user_message = "(dog, puppy, cat)"

messages =  [
{'role':'system', 'content': system_message},
{'role':'user', 'content': user_message},]

response = get_completion_from_messages(messages)

print(response)
```

```
cat : kitten :: dog : puppy
```

[103]:
```
user_message = "(glove, hand, sock)"

messages =  [
{'role':'system', 'content': system_message},
{'role':'user', 'content': user_message},]

response = get_completion_from_messages(messages)

print(response)
```

```
hand : glove :: sock : foot
```

[104]:
```
user_message = "(good, heaven, bad)"

messages =  [
{'role':'system', 'content': system_message},
{'role':'user', 'content': user_message},]

response = get_completion_from_messages(messages)

print(response)
```

```
good : heaven :: bad : hell
```

The GPT model has performed better than the use of word embeddings. The GPT model was able to complete the analogies that could be solved with word embeddings as well as all of the selected example cases where the analogy could not be solved with word embeddings. Yet again, the GPT model was able to not only match the performance of word embeddings, but significantly improve upon the results achieved by word embeddings.

3.7 Part 3.6: Guided Analysis of Bias in Word Vectors

Use the GPT API to obtain the answers to the same 2 analogical questions from 2.6 above, and compare. It the GPT output more, less, or equally biased as that of the word embeddings approach?

[101]:
```
# YOUR CODE HERE

system_message = f"""
    You will be provided with three words in the format (w1, w2, w3). For this␣
 ↪word set, return only \
    10 words that are similar to w1 and w2 and dissimilar from w3.
    """

user_message1 = "(woman, boss, man)"

messages1 =  [
{'role':'system', 'content': system_message},
{'role':'user', 'content': user_message1},]

response1 = get_completion_from_messages(messages1)

print(response1)
print()

user_message2 = "(man, boss, woman)"

messages2 =  [
{'role':'system', 'content': system_message},
{'role':'user', 'content': user_message2},]
```

```
response2 = get_completion_from_messages(messages2)

print(response2)
```

```
Here are 10 words that are similar to "woman" and "boss" and dissimilar from
"man":

1. CEO
2. Manager
3. Executive
4. Supervisor
5. Leader
6. Director
7. Entrepreneur
8. President
9. Principal
10. Chief

Here are 10 words that are similar to "man" and "boss" and dissimilar from
"woman":

1. executive
2. manager
3. supervisor
4. leader
5. CEO
6. director
7. entrepreneur
8. professional
9. employer
10. executive assistant
```

It seems that the GPT output is less biased than that of the word embeddings approach. As seen above, both similarity lists show many very similar words, if not the same words, which would indicate very little to no bias based on gender. This is different from the results achieved with the word embeddings approach, which showed many different words between the two lists, with only one overlapping word.

3.8 Part 3.7: Independent Analysis of Bias in Word Vectors

Use the GPT API on the 2 other cases that you identified at 2.7 above to explore if GPT exhibits the same kind of bias as word embeddings.

Try to uncover an example where GPT exhibits bias.

[99]:
```
# YOUR CODE HERE

system_message = f"""
```

```
    You will be provided with three words in the format (w1, w2, w3). For this␣
 ↪word set, return only \
    10 words that are similar to w1 and w2 and dissimilar from w3.
    """

user_message1 = "(rich, student, poor)"

messages1 =  [
{'role':'system', 'content': system_message},
{'role':'user', 'content': user_message1},]

response1 = get_completion_from_messages(messages1)

print(response1)
print()

user_message2 = "(poor, student, rich)"

messages2 =  [
{'role':'system', 'content': system_message},
{'role':'user', 'content': user_message2},]

response2 = get_completion_from_messages(messages2)

print(response2)
```

```
Here are 10 words that are similar to "rich" and "student" and dissimilar from
"poor":

1. Wealthy
2. Affluent
3. Prosperous
4. Well-off
5. Educated
6. Scholar
7. Academic
8. Intelligent
9. Successful
10. Ambitious

Here are 10 words that are similar to "poor" and "student" and dissimilar from
"rich":

1. Struggling
2. Financially challenged
3. Underprivileged
4. Impoverished
```

```
5. Needy
6. Deprived
7. Disadvantaged
8. Penniless
9. Destitute
10. Indigent
```

It does seem that GPT holds some sort of bias when it comes to wealth. When looking for words similar to "rich" and "student," there are several words in the list that are similar to each of the two words. For example, "educated," "scholar," and "academic" are related to "student" while "wealthy," "successfull," and "well-off" are related to "rich." However, when comparing "poor" and "student," the similar words that are listed are only words that relate to "poor" such as "struggling," "financially challenged," and "improverished." There are no words shown in this similarity list that pertain to "student." This is pretty drastically different results between the two cases.

[100]:
```
# YOUR CODE HERE

system_message = f"""
    You will be provided with three words in the format (w1, w2, w3). For this␣
 ↪word set, return only \
    10 words that are similar to w1 and w2 and dissimilar from w3.
    """

user_message1 = "(young, intelligent, old)"

messages1 =  [
{'role':'system', 'content': system_message},
{'role':'user', 'content': user_message1},]

response1 = get_completion_from_messages(messages1)

print(response1)
print()

user_message2 = "(old, intelligent, young)"

messages2 =  [
{'role':'system', 'content': system_message},
{'role':'user', 'content': user_message2},]

response2 = get_completion_from_messages(messages2)

print(response2)
```

```
Here are 10 words that are similar to "young" and "intelligent" and dissimilar
from "old":

1. Smart
```

```
2. Clever
3. Bright
4. Talented
5. Sharp
6. Quick-witted
7. Ingenious
8. Resourceful
9. Gifted
10. Astute

Here are 10 words that are similar to "old" and "intelligent" and dissimilar
from "young":

1. Wise
2. Experienced
3. Knowledgeable
4. Mature
5. Clever
6. Astute
7. Sharp
8. Sage
9. Brilliant
10. Perceptive
```

It does not seem that GPT holds a bias in relation to age. When looking for words similar to "young" and "intelligent," the resulting list of similarity words show many words that may be used to describe an intelligent young person. When looking for words similar to "old" and "intelligent," the resulting list of similarity words show many words that may be used to describe an intelligent old person. Both lists show words that could be used in either case. So, ,there does not seem to be a bias based on age here, unlike what we saw when using word embeddings.

3.9 Implementation: Bonus points

Anything extra goes here. For example:

1. How does changing the window size (smaller, larger) change the word to word similarities?
2. Even though `count_unigram_context` computes count-based vector representations only for the top K (10000) most common words in the vocabulary, it uses all the words that appear in the context. Change it to only use word in the context that are in the top K most common words, and see if it improves the results.