WordVectors

April 4, 2024

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 LLM 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:

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 Properties of cosine similarity (20p)

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.

2.1.1 YOUR ANSWER goes here.

It is important that math is formatted appropriately, e.g. x_i looks good, xi looks bad.

2.2 [5111] On the fitting power of Logistic Regression (20p)

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.*

2.2.1 YOUR SOLUTION goes here.

2.3 [5111]: Binary vs. Multiclass Logistic Regression (20p)

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, for any inptu feature vector $\mathbf{x}$. 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* $\mathbf{w}_2$*.*

2.3.1 YOUR SOLUTION goes here.

2.4 [Bonus]: Phrase embeddings (20p)

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 [Bonus] Time and memory complexity (20p)

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

[ ]:

```
# 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)
```

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.

[ ]:

```
window = 4
vocabSize = 10000
```

3.1.1 Question 1.1: Load corpus and create document frequency dictionary (20p)

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).

[ ]:

```
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






print('Total number of documents:', Ndocs)
```

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

[ ]:
```
# 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
```

[ ]:
```
# 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
```

[ ]:
```
# 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
```

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

```
[ ]: 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
```

```
[ ]: # 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))
```

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

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

3.1.2 Question 1.2: Implement Tf.Idf Representation (10p)

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.

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

```

```
[ ]: tfidf_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 (10p)

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

[ ]:

```
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)
```

3.1.4 YOUR ANSWER goes here.

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.

[ ]:

```
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
```

[ ]:

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

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 (10p)

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?

[ ]:

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

3.2.2 YOUR ANSWER goes here.

3.2.3 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.

[ ]:

```
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
```

```
[ ]: 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.vocab.keys())
         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
```

3.2.4 Question 2.2: Word2Vec Plot Analysis (10p)

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

[ ]:
```
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?

[ ]:
```
# -----------------------------------------------------------------
# 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)
```

3.2.5 YOUR ANSWER goes here.

3.2.6 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:

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

3.2.7 Question 2.3: Polysemous Words [code + written] (10p)

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**.

[ ]:

```
# ------------------
# Write your polysemous word exploration code here.

wv_from_bin.most_similar("")

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

3.2.8 YOUR ANSWER goes here.

3.2.9 Question 2.4: Synonyms & Antonyms (10p)

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.

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

w1 = ""
w2 = ""
w3 = ""
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))

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

3.2.10 YOUR ANSWER goes here.

3.2.11 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**.

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

3.2.12 Question 2.5: Finding Analogies (10p)

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.

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

```
pprint.pprint(wv_from_bin.most_similar(positive=[], negative=[]))

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

3.2.13 YOUR ANSWER goes here.

3.2.14 Question 2.6: Guided Analysis of Bias in Word Vectors (10p)

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?

[ ]:

```
# 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']))
```

3.2.15 YOUR ANSWER goes here.

3.2.16 Question 2.7: Independent Analysis of Bias in Word Vectors (10p)

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.

[ ]:

```
# ------------------
# Write your bias exploration code here.

pprint.pprint(wv_from_bin.most_similar(positive=[], negative=[]))
print()
pprint.pprint(wv_from_bin.most_similar(positive=[,], negative=[]))

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

3.2.17 YOUR ANSWER goes here.

3.2.18 Question 2.8: Thinking About Bias (10p)

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

3.2.19 YOUR ANSWER goes here.

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 LLM model you use (GPT-3.5, GPT-4, Mixtral, Llama2).

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

### Chat completion API setup

- Edit accordingly for your choice of LLM (GPT-3.5, GPT-4, Mixtral, Llama2).
- Follow the instructions posted for Assignment 3 if you want to use Mixtral and Llama2 off campus and the HPC educational cluster.

[ ]:

```
import os
from openai 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())
client = 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 = client.chat.completions.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 (10p)

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.

3.4.1 YOUR ANALYSIS goes here.

```
[ ]: # YOUR CODE HERE
```

3.5 Part 3.4: Synonyms & Antonyms (10p)

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.

3.5.1 YOUR ANALYSIS goes here.

```
[ ]: # YOUR CODE HERE
```

3.6 Part 3.5: Finding Analogies (10p)

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

3.6.1 YOUR ANALYSIS goes here.

```
[ ]: # YOUR CODE HERE
```

3.7 Part 3.6: Guided Analysis of Bias in Word Vectors (10p)

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?

3.7.1 YOUR ANALYSIS goes here.

```
[ ]: # YOUR CODE HERE
```

3.8 Part 3.7: Independent Analysis of Bias in Word Vectors (10p)

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.

3.8.1 YOUR ANALYSIS goes here.

```
[ ]: # YOUR CODE HERE
```

3.9 Implementation: Bonus points

Anything extra goes here. For example:

1. (10p) How does changing the window size (smaller, larger) change the word to word similarities?
2. (10p) 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.