XCS224N: NLP with Deep Learning

Assignment 1:
Exploring Word Embeddings

Group Office Hours

Raul Salles de Padua

Stanford | Center for Professional Development

Agenda for today's OHs

1. **Quick recap on Word Embeddings**
2. **Overview on Assignment 1**

Agenda for today's OHs

1. **Quick recap on Word Embeddings**
2. **Overview on Assignment 1**

Co-occurrence Matrix

A co-occurrence matrix counts how often things co-occur in some environment.

Given some word w_i occurring in the document, we consider the context window surrounding w_i . Supposing our fixed window size is n, then this is the n preceding and n subsequent words in that document, i.e. words $w_{i-n} \ldots w_{i-1}$ and $w_{i+1} \ldots w_{i+n}$. We build a co-occurrence matrix M, which is a symmetric word-by-word matrix in which M_{ij} is the number of times w_j appears inside w_i 's window.

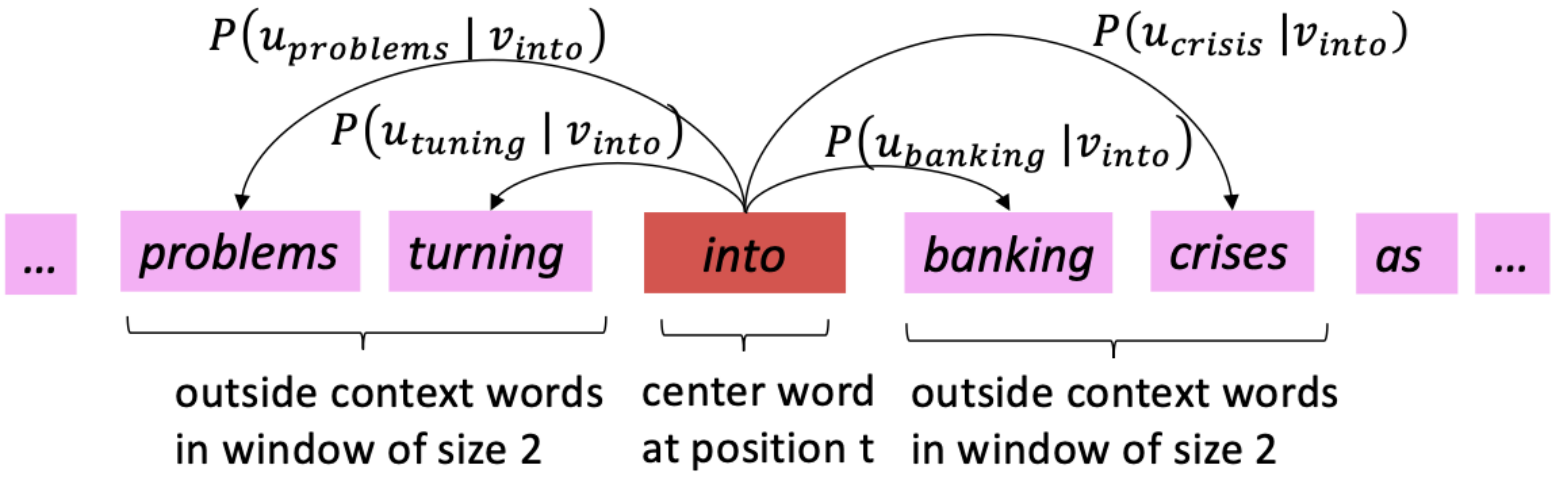
Document 1: "all that glitters is not gold"
Document 2: "all is well that ends well"

$\mathbf{M} =$

	START	all	that	glitters	is	not	gold	well	ends	END
START	0	2	0	0	0	0	0	0	0	0
all	2	0	1	0	1	0	0	0	0	0
that	0	1	0	1	0	0	0	1	1	0
glitters	0	0	1	0	1	0	0	0	0	0
is	0	1	0	1	0	1	0	1	0	0
not	0	0	0	0	1	0	1	0	0	0
gold	0	0	0	0	0	1	0	0	0	1
well	0	0	1	0	1	0	0	0	1	1
ends	0	0	1	0	0	0	0	1	0	0
END	0	0	0	0	0	0	1	1	0	0

Word2Vec Overview

- Example windows and process for computing $P(w_{t+j} \mid w_t)$
- $P(u_{problems} \mid v_{into})$ short for $P(problems \mid into\ ; u_{problems}, v_{into}, \theta)$



Word2Vec Overview

Word2vec: prediction function

② Exponentiation makes anything positive

① Dot product compares similarity of *o* and *c*.
$u^T v = u.v = \sum_{i=1}^{n} u_i v_i$
Larger dot product = larger probability

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

③ Normalize over entire vocabulary to give probability distribution

- This is an example of the **softmax function** $\mathbb{R}^n \to (0,1)^n$ ← Open region

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^{n} \exp(x_j)} = p_i$$

- The softmax function maps arbitrary values x_i to a probability distribution p_i
  - "max" because amplifies probability of largest x_i
  - "soft" because still assigns some probability to smaller x_i
  - Frequently used in Deep Learning

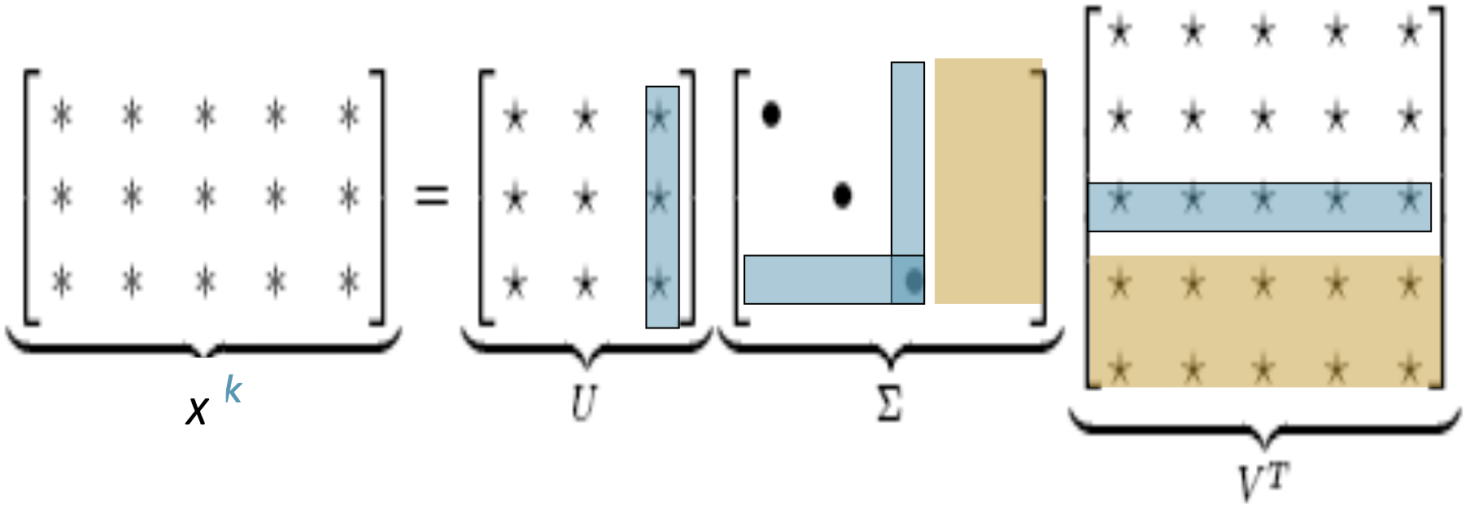
But sort of a weird name because it returns a distribution!

- Useful basic fact: $\frac{\partial \mathbf{x}^T \mathbf{a}}{\partial \mathbf{x}} = \frac{\partial \mathbf{a}^T \mathbf{x}}{\partial \mathbf{x}} = \mathbf{a}$

Dimensionality Reduction

Singular Value Decomposition of co-occurrence matrix *X*

Factorizes *X* into $U\Sigma V^T$, where *U* and *V* are orthonormal



Retain only *k* singular values, in order to generalize.
$\hat{X}$ is the best rank *k* approximation to *X* , in terms of least squares.
Classic linear algebra result. Expensive to compute for large matrices.

n rows corresponding to *n* words– based on word-word co-occurrence.

Select top k principal components.

SVD decomposes X to singular values in diagonal ∑
Matrix, and **our new shorter length-k word vectors in U_K .**

Agenda for today's OHs

1. Quick recap on Word Embeddings
2. **Overview on Assignment 1**

1. Computing Co-occurrence Embeddings

(a) [2 points]

Question: Implement the distinct words function in src/submission.py. You can do this with for loops, but it's more efficient to do it with Python list comprehensions.

```
def distinct_words(corpus):
    """ Determine a list of distinct words for the corpus.
        Params:
            corpus (list of list of strings): corpus of documents
        Return:
            corpus_words (list of strings): list of distinct words across the corpus, sorted (using python 'sorted' function)
            num_corpus_words (integer): number of distinct words across the corpus
    """
    corpus_words = []
    num_corpus_words = 0

    # ### START CODE HERE ###



    # ### END CODE HERE ###

    return corpus_words, num_corpus_words
```

Why this matters: Creates the vocabulary that will become our matrix dimensions

Objective: Extract and sort all unique words from the corpus

What it does:

- Takes a corpus (list of documents, where each document is a list of words)
- Returns all unique words sorted alphabetically + count of unique words

Overview:

- **Set comprehension:** nested iteration
- **Sorting**: ensures consistent ordering for matrix indexing later

1. Computing Co-occurrence Embeddings

(b) [5 points]

Question: Implement the compute co occurrence matrix function in src/submission.py. If you aren't familiar with the python numpy package, we suggest walking yourself through this tutorial.

```python
def compute_co_occurrence_matrix(corpus, window_size=4):
    """ Compute co-occurrence matrix for the given corpus and window_size (default of 4).

        Note: Each word in a document should be at the center of a window. Words near edges will have a smaller
              number of co-occurring words.

              For example, if we take the document "START All that glitters is not gold END" with window size of 4,
              "All" will co-occur with "START", "that", "glitters", "is", and "not".

        Params:
            corpus (list of list of strings): corpus of documents
            window_size (int): size of context window
        Return:
            M (numpy matrix of shape (number of unique words in the corpus , number of unique words in the corpus)):
                Co-occurrence matrix of word counts.
                The ordering of the words in the rows/columns should be the same as the ordering of the words given by the distinct_words function.
            word2Ind (dict): dictionary that maps word to index (i.e. row/column number) for matrix M.
    """
    words, num_words = distinct_words(corpus)
    M = None
    word2Ind = {}

    # ### START CODE HERE ###


    # ### END CODE HERE ###

    return M, word2Ind
```

Objective: Build a co-occurrence matrix based on context windows

What it does:

- For each word, count how often other words appear within a sliding window
- Default window size = 4
- Returns: co-occurrence matrix M + word-to-index mapping

Overview:

- **STEP 1:** set up the data structures
- **STEP 2:** iterate each document in the corpus
- **STEP 3:** iterate for each center word
- **STEP 4:** look at surrounding words within window
- **STEP 5:** apply boundary checks w/ conditional statements
- **STEP 6:** count the co-occurrence

1. Computing Co-occurrence Embeddings

(c) [2 points]

Question: Implement the reduce to k dim function in src/submission.py.

```python
def reduce_to_k_dim(M, k=2):
    """ Reduce a co-occurrence 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 unique words in the corpus , number of unique words in the corpus)): co-occurrence matrix of word counts
            k (int): embedding size of each word after dimension reduction
        Return:
            M_reduced (numpy matrix of shape (number of unique words in the corpus, k)): matrix of k-dimensioal word embeddings.
                    In terms of the SVD from math class, this actually returns U * S
    """
    np.random.seed(4355)
    n_iter = 10     # Use this parameter in your call to `TruncatedSVD`
    M_reduced = None
    print("Running Truncated SVD over %i words..." % (M.shape[0]))

    # ### START CODE HERE ###



    # ### END CODE HERE ###

    print("Done.")
    return M_reduced
```

Objective: Apply SVD dimensionality reduction to the co-occurrence matrix

What it does:

- Takes high-dimensional co-occurrence matrix (vocab_size × vocab_size)
- Uses Truncated SVD to reduce to k dimensions (default k=2 for visualization)
- Returns the reduced embeddings (vocab_size × k)

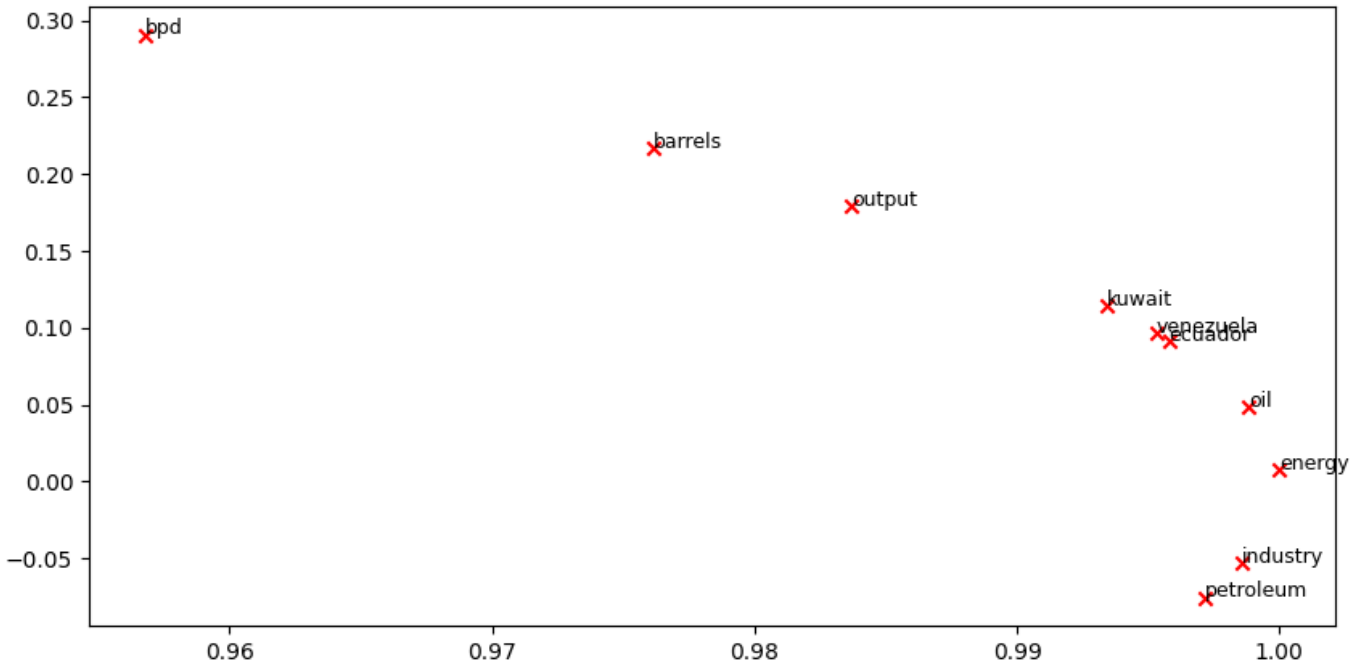
Overview:

- **Why SVD:** captures latent semantic relationships in fewer dimensions

1. Computing Co-occurrence Embeddings

(d) [0 points]

Question: Show time! Now we are going to load the Reuters corpus and compute some word vectors with everything you just implemented! There is no additional code to write for this part; just run python submission.py from within the src/ subdirectory. This will generate a plot of the embeddings for hand-picked words which have been dimensionally reduced to 2-dimensions. Try and take note of some of the clusters and patterns you do or don't see.



1. Computing Co-occurrence Embeddings

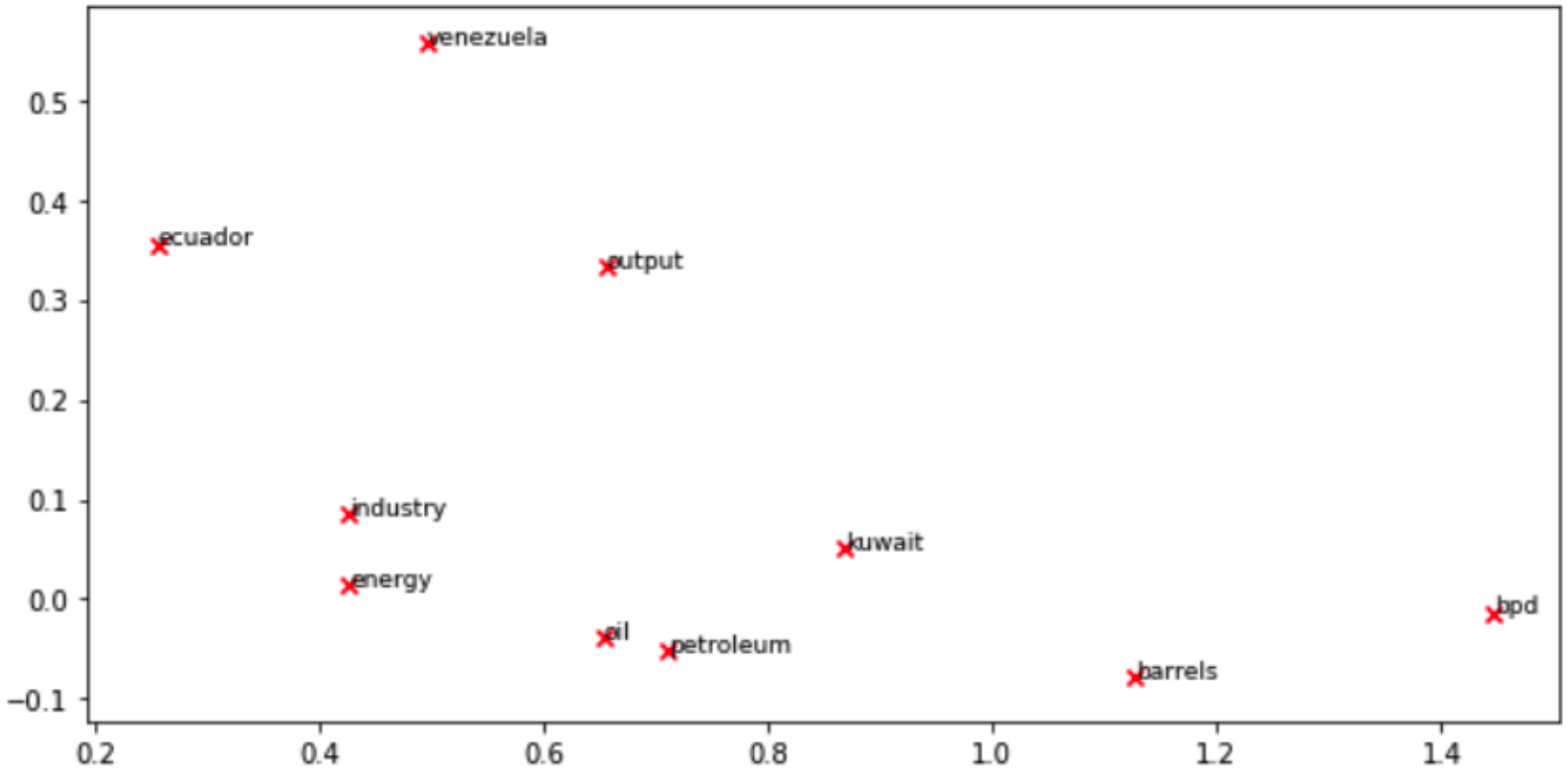
Common Pitfalls & Debugging Tips

- 1(a): Forgetting to sort or using incorrect data structures
- 1(b): Off-by-one errors in window bounds, not handling document boundaries
- 1(c): Using wrong SVD implementation (must use TruncatedSVD from sklearn)

2. Experimenting with Word2Vec Embeddings

Google Colab NoteBook

1. Why aren't countries "venezuela", "ecuador" and "kuwait" clustered together in the Word2Vec plot while as they in your co-occurence plot? - Select all the reasonable possibilities



2. Experimenting with Word2Vec Embeddings

Google Colab NoteBook

2. For the following homonyms, respond True if the top 10 most similar words represent more than one meaning for the homonym and False otherwise. These are case sensitive so please make sure to input the word exactly as specified below

a. left

b. nuts

c. pen

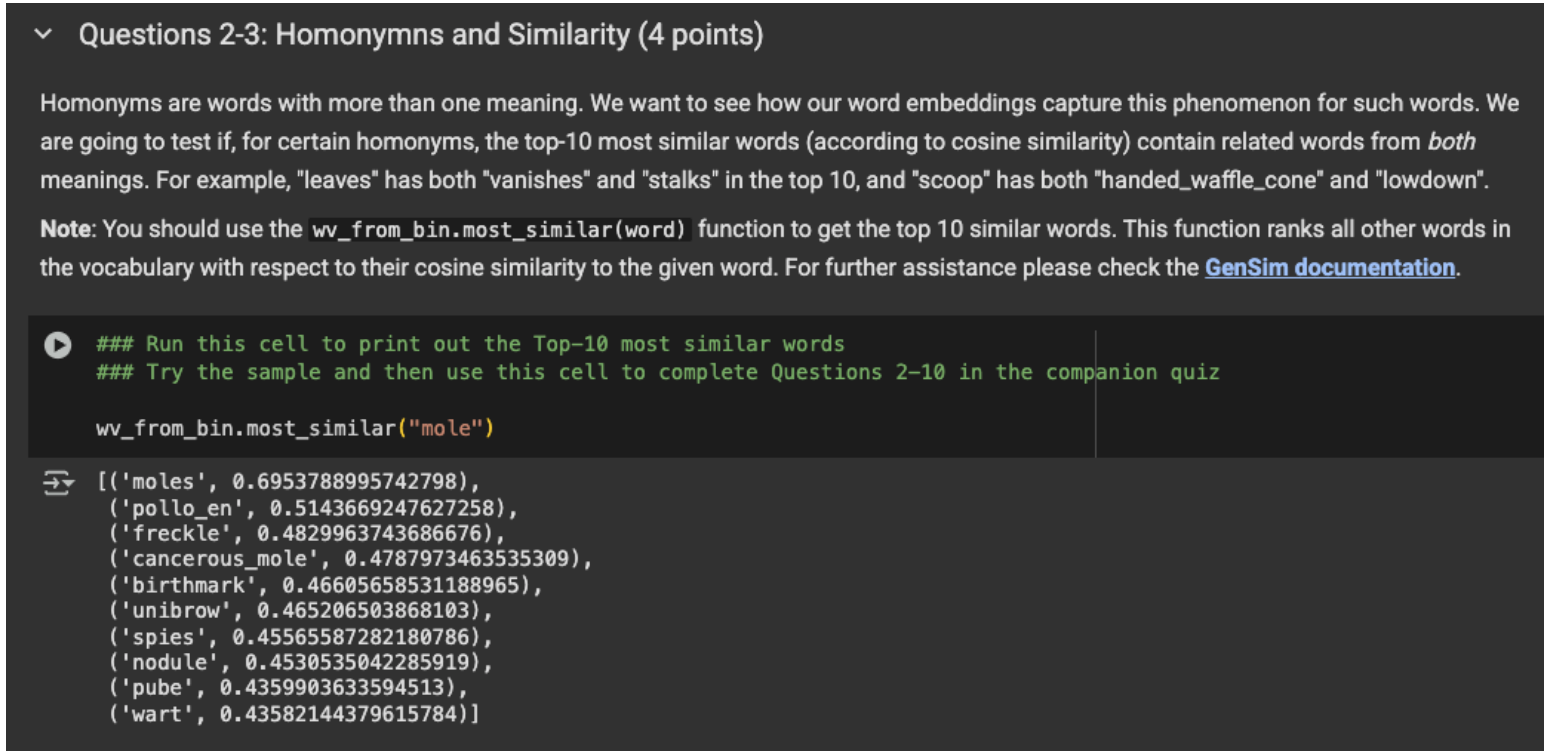
d. right

e. drive

f. rose

g. mean

h. saw



Questions 2-3: Homonymns and Similarity (4 points)

Homonyms are words with more than one meaning. We want to see how our word embeddings capture this phenomenon for such words. We are going to test if, for certain homonyms, the top-10 most similar words (according to cosine similarity) contain 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".

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.

```
### Run this cell to print out the Top-10 most similar words
### Try the sample and then use this cell to complete Questions 2-10 in the companion quiz

wv_from_bin.most_similar("mole")
```

```
[('moles', 0.6953788995742798),
 ('pollo_en', 0.5143669247627258),
 ('freckle', 0.4829963743686676),
 ('cancerous_mole', 0.4787973463535309),
 ('birthmark', 0.46605658531188965),
 ('unibrow', 0.465206503868103),
 ('spies', 0.45565587282180786),
 ('nodule', 0.4530535042285919),
 ('pube', 0.4359903633594513),
 ('wart', 0.43582144379615784)]
```

2. Experimenting with Word2Vec Embeddings

Google Colab NoteBook

3. Please input answer to question 3 of the Gradescope online assessment A1 (Google Colab) Online Assessment. Why do you think many of the homonyms you tried did not represent more than one meaning? - Select all the reasonable possibilities.

a. A different word vector is trained for each meaning of the homonym, so synonyms for a particular meaning of the word will be most similar depending on the which vector we are using for the original word.

b. In some scenarios, one meaning of the homonym is much more common in the training data than the other meanings, resulting in the vector for the homonym is much closer to words from one meaning than the other(s).

c. When a particular meaning of a homonym is much more common than the others, the algorithm that trains the word vectors is able to recognize this and only learns to encode one of the definitions in the word vector.

2. Experimenting with Word2Vec Embeddings

Google Colab NoteBook

4. Mark True if the antonyms pair has a smaller cosine distance than the synonyms pair. Mark False otherwise. Please make sure to input the words exactly as specified below:

a. happy, sad | happy, cheerful

b. right, wrong | right, correct

c. big, small | big, large

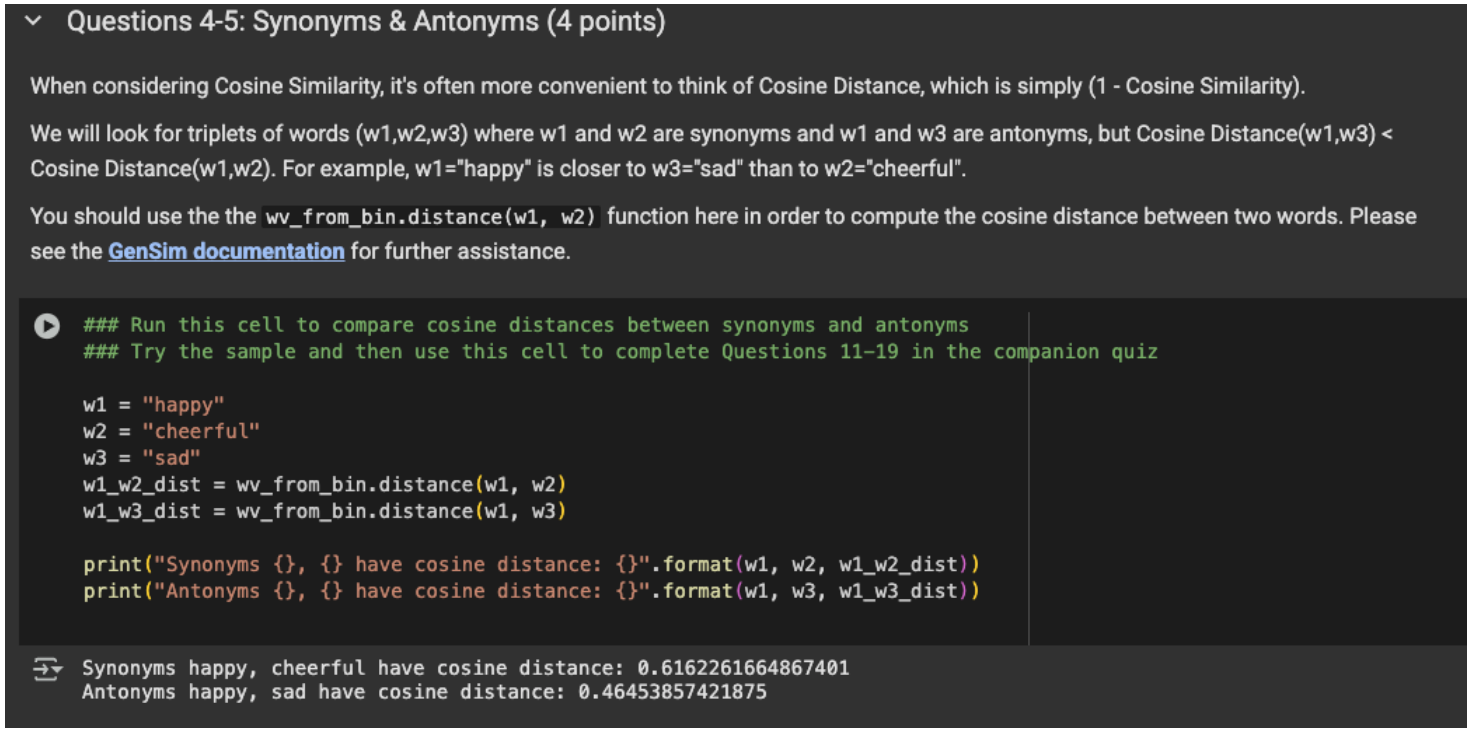
d. good, bad | good, nice

e. day, night | day, daytime

f. insane, sane |insane, crazy

g. several, one |several, numerous

h. antonym, synonym |antonym, opposite



2. Experimenting with Word2Vec Embeddings

Google Colab NoteBook

5. Why do you think there are times where synonym-antonym pairs have a smaller cosine distance than the synonym-synonym pairs? - Select all the reasonable possibilities.

a. Sometimes, the synonym-antonym pair has two words that are found with much higher frequency in the corpus than the other synonym word. The word vectors capture this frequency, making the synonym-antonym pair more similar than the synonym-synonym pair.

b. In some cases, the antonym is a homonym and carries multiple meanings, some of which may have definitions slightly similar to the synonym, making them overall fairly similar.

c. We may find that the synonym-antonym words are actually used in similar contexts, more so than the synonym-synonym words, making the synonym-antonym words more similar.

2. Experimenting with Word2Vec Embeddings

Google Colab NoteBook

6. Mark True if the word vectors are able to successfully complete the analogy. Mark False otherwise. These are case sensitive so please make sure to input the word exactly as specified below:

a. man:king::woman:QUEEN
b. air:plane::sea:BOAT
c. happy:laugh::sad:CRY
d. cat:kitten::dog:PUPPY
e. France:Paris::Germany:BERLIN
f. carnivore:meat::herbivore:VEGETABLES
g. dog:bark::cat:MEOW
h. bat:baseball::racquet:TENNIS
i. mosque:Islam::church:CHRISTIANITY
j. long:longer::short:SHORTER
k. longer:longest::more:MOST
l. talk:talked::help:HELPED

2. Experimenting with Word2Vec Embeddings

Google Colab NoteBook

7. What factors might contribute to the biases observed in the word vectors presented in the notebook examples?
- Select all the reasonable possibilities.

a. Text data is generated by people, and thus word vectors trained on a corpus are prone to have the same biases of the people who generated the corpus

b. The training corpus may have many instances of sentences that relate certain races, genders, etc. to certain properties or behaviours

4. Extra Credit Challenge

how to submit written assignments

(a) Show that $\mathbf{A} = \sum_{i=1}^{r} \sigma_i \mathbf{u}_i \mathbf{v}_i^T$

High-level idea: "Rank-1 building blocks''

- Write U D V^T as (u_1 ... u_r) diag(σ_1...σ_r) (v_1 ... v_r)^T.
- Multiplying U by D scales each column u_i by σ_i.
- The product with V^T is then a sum of outer products σ_1 u_1 v_1^T + ... + σ_r u_r v_r^T.

Thus: SVD is literally telling us that any matrix can be decomposed as a sum of rank-1 "tiles'' whose directions are the singular vectors and whose magnitudes are the singular values.

4. Extra Credit Challenge

how to submit written assignments

(b) Show that $$\mathbf{u}_i = \frac{1}{\sigma_i}\mathbf{A}\mathbf{v}_i$$

In particular, the components of ui represent the size of the projection of the rows of A onto vi (scaled by σi).

High-level idea: "Left vectors are scaled projections''

- Recall that v_i are orthonormal you can use the result from (a)
- Multiply both sides of $A = U D V^T$ on the right by v_j.
  → $A v_j = U D V^T v_j$.
- Work your way to get the desired formula.

4. Extra Credit Challenge

how to submit written assignments

(c) One way of finding a reduced rank approximation of A is by hard-setting all but the k largest σi to 0. This approximation is called the truncated SVD, and by (a) we see it can be written as

$$\mathbf{A}_k := \sum_{i=1}^{k} \sigma_i \mathbf{u}_i \mathbf{v}_i^T$$

From (a), we see the truncated SVD can also be written as $A_k = U_k D_k V T_k$, where $U_k \in R^{n \times k}$, $V \in R^{d \times k}$ are the first k columns of U,V, and $D \in R^{k \times k}$ has the first k singular values.

Show that the rows of A_k are the projections of the rows of A onto the subspace of V_k spanned by the first k right singular vectors.

Hint: Recall that the projection of a vector a onto a subspace spanned by $v_1,\ldots,v_k$ where the vi are pairwise orthogonal is given by the sum of projections of a onto the individual vi.

High-level idea: "Truncated SVD = project rows onto top-k subspace"

- Each outer product σ_i u_i v_i^T adds back the component of A that lies in direction v_i.
- Because the v_i's are orthonormal, summing the first k of them gives exactly the orthogonal projection onto their span.
- Therefore every original row a_j is replaced by $\text{proj}_{Vk}(a_j)$...

4. Extra Credit Challenge

how to submit written assignments

(d) The Frobenius norm of a matrix M ∈Rn×m is defined as

$$||\mathbf{M}||_F = \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{m} M_{ij}^2}$$

Show that

$$\mathbf{A}_k = \underset{\text{rank}(\mathbf{B})=k}{\arg\min} ||\mathbf{A} - \mathbf{B}||_F$$

where the arg min is taken over matrices of rank k.
Hint: Use the fact that Vk is the best-fit k-dimensional subspace for the rows of A.

High-level intuition: "Why truncated SVD is optimal"

- **Best-fit subspace:** the span of {v_1,…,v_k} is the unique k-dimensional subspace that minimizes squared reconstruction error of the rows of A (proved earlier from SVD theory).
- **Projection gives smallest error within that subspace:** once you commit to that subspace, the orthogonal projection minimizes the distance from each row to the subspace. That projection is exactly A_k.
- **No other rank-k matrix can beat that error:** any rank-k matrix B lives in some k-dimensional row-space; if that space is not V_k, its total squared error is larger. Hence A_k is globally optimal in Frobenius norm.

Explains why dimensionality-reduction methods based on SVD are principled: they guarantee the closest possible rank-k summary of the data.