

WordVectors

February 19, 2026

0.1 Word Vectors

This assignment is comprised of three parts:

1. **Theory:** Prove simple properties of cosine similarity. 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. **Classification:** You will use LLM embeddings or moving reviews to train a sentiment analyzer.

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:

```
pip install --upgrade gensim
```

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. **Do not submit the data folder!**
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.

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

2.1.1 Solution 1 goes here.

2.1.2 Solution 2 goes here.

2.1.3 Solution 3 goes here.

2.2 Phrase embeddings (20p)

Given a phrase consisting of a sequence of M words, $phrase = [word_1, word_2, \dots, word_M]$, 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:

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

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:

$$z(good) > z(not\ good) \quad (2)$$

$$z(bad) < z(not\ bad) \quad (3)$$

2.2.1 YOUR SOLUTION goes here.

2.3 Time and memory complexity (bonus 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.

2.3.1 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 similarity 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.

```
[ ]: # We will use as context 4 words to the left and 4 words to the right.
window = 4

# We will only consider as context the most common 10,000 words in the
↳ vocabulary.
```

```
vocabSize = 10000
```

3.1.1 Question 1.1: Load corpus and create document frequency dictionary (10p)

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

Create empty word representations For each of the `vocabSize` most common words in the data, create a word representation that is initialized to an empty dictionary. Return the `word_representations` as a dictionary mapping each word to its empty word representation.

```
[ ]: def create_representations(data, vocabSize):
    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_representations = create_representations(wiki_data, vocabSize)
```

```
[ ]: print(word_representations['falcon'])
```

Update word representations Traverse the data from left to right, and for each word occurrence update its word representation by incrementing the counts for words appearing window words to the left and window words to the right.

```
[ ]: def update_representations(data, word_representations, window):
    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
```

```
[ ]: update_representations(wiki_data, word_representations, window)
```

```
[ ]: print(word_representations['falcon'])
```

Normalize word representations Currently, `word_representations` is a dictionary that maps words to their representation, where each representation is itself a dictionary mapping a words to counts. This dictionary can be seen as a sparse vector representation, where each position in the vector corresponds to a word in the vocabulary. Since most words would have a co-occurrence count of 0, the dictionary only stores words for which the counts are greater than 0.

In the function below, we normalize each word representation such that its norm is 1. This is done by first computing the norm of the representation, and then dividing each count in the vector by the norm. We do this normalization such that dot-product between normalized vectors is equivalent with cosine similarity.

```
[ ]: def normalize(word_representations):
    for word in word_representations:
        total = 0
        for key in word_representations[word]:
            total += word_representations[word][key] *  $\square$ 
         $\hookrightarrow$ word_representations[word][key]

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

```
[ ]: normalize(word_representations)
```

```
[ ]: print(word_representations['falcon'])
```

3.1.2 Question 1.2: Cosine Similarity (10p)

Write a function `dictionary_dot_product(dict1, dict2)` that computes the dot-product between the two normalized word representations stored as dictionaries.

```
[ ]: def dictionary_dot_product(dict1, dict2):
    dot = 0
    # YOUR CODE HERE

    return dot
```

```
[ ]: print(dictionary_dot_product(word_representations['falcon'],
    ↪word_representations['bird']))
print(dictionary_dot_product(word_representations['falcon'],
    ↪word_representations['bridge']))
```

Nearest Neighbors Write a function `find_nearest_neighbors(word_representations, query, K)` that takes as input a query word and returns the `K` most similar words.

We first define a helper function `compute_sim(word_representations, query)` that takes as input a query word and computes, for each word in the vocabulary, the cosine similarity between the representation of that word and the representation of the query word.

```
[ ]: def compute_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 = compute_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))
```

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

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

3.1.3 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
```

```
[ ]: tf_idf_word_representations = create_representations(wiki_data, vocabSize)
    update_representations(wiki_data, tf_idf_word_representations, window)
    tfidf(tf_idf_word_representations)
    normalize(tf_idf_word_representations)
```

3.1.4 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 = "falcon" # "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)
```

3.1.5 YOUR ANSWER goes here.

3.2 Part 2: Prediction-Based Word Vectors

As discussed in class, prediction-based dense word embeddings have come to dominate NLP. 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.

```
[ ]: from gensim.models import KeyedVectors

wv_from_bin = KeyedVectors.load_word2vec_format('../data/
    ↪ word2vec-google-news-300.gz', binary = True)
vocab = list(wv_from_bin.key_to_index.keys())
print("Loaded vocab size %i" % len(vocab))
```

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, using 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("falcon") # "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-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 corpus words, number of corpus
      ↪ words)): 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 corpus words, k)):
      ↪ matrix of k-dimensional 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.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.get_vector(w))
            word2Ind[w] = curInd
            curInd += 1
        except KeyError:
            continue
    for w in required_words:
        try:
            M.append(wv_from_bin.get_vector(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-Dimensional 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("leaves")  
  
# -----
```

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 $1 - \text{Cosine Similarity}$.

Find three words (w_1, w_2, w_3) where w_1 and w_2 are synonyms and w_1 and w_3 are antonyms, but $\text{Cosine Distance}(w_1, w_3) < \text{Cosine Distance}(w_1, w_2)$. For example, $w_1 = \text{“happy”}$ is closer to $w_3 = \text{“sad”}$ than to $w_2 = \text{“cheerful”}$.

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

You should use the `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 = "happy"
w2 = "cheerful"
w3 = "sad"
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: Sentiment Analysis using LLM Embeddings

In this part of the assignment, you will use an LLM's embeddings API, such as [Gemini Embeddings](#), to map movie reviews into embeddings, and then use these embeddings as feature vector to train a sentiment analyzer. Follow the sequence of steps below:

3.3.1 Data Preprocessing (50 points)

1. Read the training examples from `imdb_sentiment_train.txt`. Store the labels in an array `y_train` and the corresponding movie reviews in the list of strings `x_train`.
2. Pass the list of reviews `x_train` to the Gemini Embeddings API and store the resulting embeddings matrix in a NumPy array `embeddings_train`. Use a size of 512 for the embeddings.
3. Save the `embeddings_train` array in `../data/embeddings.train.npy` using the `numpy.save()` function. Save the labels `y_train` into `../data/labels.train.txt`.
4. Repeat steps 1, 2, and 3 above for the test examples in `imdb_sentiment_test.txt` and create the corresponding `y_test`, `embeddings_test`, and the files `../data/embeddings.test.npy` and `../data/labels.test.txt`.

It is recommended that you split the examples into batches, e.g., each batch contains 100 examples, then provide each batch of examples to the Gemini API, which will return a batch of embeddings. If you use an API key associated with the \$300 free credits, this should not incur any rate-limit issues. If you use a generic, free Gemini account, you may want to insert a 10s delay between batches.

For your convenience, if you still encounter issues, we created and saved the Gemini embeddings in the `../data` folder. Nevertheless, submit your code and the output cell you get from it showing any issues you encounter.

```
[ ]: import numpy as np
      from google import genai

      # YOUR CODE HERE
```

3.3.2 LR Model Training and Testing (50 points)

5. Read the training embeddings from `../data/embeddings.train.npy` into a NumPy array `x_train`, and read the training labels from `../data/labels.train.txt` into `y_train`.
6. Train a binary classifier on `x_train` and `y_train`, using the `sklearn.linear_model.LogisticRegression` class.
7. Read the test embeddings from `../data/embeddings.test.npy` into a NumPy array `x_test`, and read the training labels from `../data/labels.test.txt` into `y_test`.
8. Use the trained classifier to predict the labels of the test embeddings in `x_test` and store the predictions in `pred_test`.
9. Compute and report the test accuracy by comparing the labels in `y_test` and `pred_test`.

10. Compare this accuracy with the accuracy obtained at homework 3.

Our trained logistic regression model obtained **96.4% test accuracy**.

```
[ ]: from sklearn.linear_model import LogisticRegression

lr_model = LogisticRegression(penalty = 'l2', C = 1.0, solver = 'lbfgs',
                               ↪max_iter = 1000)

# YOUR CODE HERE
```

3.3.3 Comparison of LR with Embeddings vs. LR with Engineered features.

YOUR ANALYSIS GOES HERE:

3.4 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.
3. (40p) Use the development examples in `../data/imdb_sentiment_dev.txt` to tune the hyperparameters of the LogisticRegression model, e.g., by using the K-fold cross-validation function from sklearn. Report how much this improves performance on the test examples.
4. (20p) Tune the embedding size (from 256 to 3072) on the development data and report which one obtains best performance on development, then use it to train and test.

```
[ ]:
```