Section 1: Properties of Word Embeddings

Question 1.1 Read the documentation of the Vocab class of Torchtext that you can find here: https://torchtext.readthedocs.io/en/latest/vocab.html and then read the A1_Section1_starter.ipynb code. Run the notebook and make sure you understand what each step does.

Question 1.2 Write a new function, similar to print_closest_words called print_closest_cosine_words that prints out the N-most (where N is a parameter) similar words using cosine similarity rather than euclidean distance. Provide a table that compares the 5-most cosine-similar words to the word 'dog', in order, alongside to the 10 closest words computed using euclidean distance. Give the same kind of table for the word 'computer.' Looking at the two lists, does one of the metrics (cosine similarity or euclidean distance) seem to be better than the other? Explain your answer. Submit the specific code for the print_closest_cosine_words function that you wrote in a separate Python file named A1P1_2.py. [2 points]

Word	**Cosine Similarity**	**Word**	**Euclidean Distance**
Cat	0.92	Cat	1.88
Dogs	0.85	Dogs	2.65
Horse	0.79	Puppy	3.15
Puppy	0.78	Rabbit	3.18
pet	0.77	Pet	3.23
		Horse	3.25
		Pig	3.39
		Pack	3.43
		Cats	3.44
		Bite	3.46

Table 1.2.1: similar words to "dog" by embedding.

Word	**Cosine Similarity**	**Word**	**Euclidean Distance**
Computer	0.92	Computers	2.44
Software	0.88	Software	2.93
Technology	0.85	Technology	3.19
Electronic	0.81	Electronic	3.51
Internet	0.81	Computing	3.60
		Devices	3.67
		Hardware	3.68
		Internet	3.69
		Applications	3.69
		Digital	3.70

Table 1.2.2: similar words to "computer" by embedding.

For "dog": the cosine similarity gives us close semantic relationships like "dogs", "puppy", and "pet", which all relate directly to the animal as a concept. Euclidean distance seems to capture words that are close in terms of numerical proximity, but they aren't always semantically related. For example, words like *pack* and *bite* are included in the Euclidean distance list for "dog" but don't have as strong a semantic relationship as those in the cosine similarity list.

For "computer": cosine similarity shows that words like “software”, “technology”, and “internet” are highly related in meaning, while Euclidean distance provides a broader range of related terms, including “devices” and “hardware”.

To conclude, cosine similarity seems to be a better measure in this case because it captures semantic relationships more effectively. Euclidean distance, while useful for proximity in the vector space, might not always capture meaning as well.

Question 1.3 The section of A1_Section1_starter.ipynb that is labelled Analogies shows how relation- ships between pairs of words is captured in the learned word vectors. Consider, now, the word-pair relationships given in Figure 1 below, which comes from Table 1 of the Mikolov[2] paper. Choose one of these relationships, but not one of the ones already shown in the starter notebook, and report which one you chose. Write and run code that will generate the second word given the first word. Generate 10 more examples of that same relationship from 10 other words, and comment on the quality of the results. Submit the specific code that you wrote in a separate Python file, A1P1_3.py. [4 points]

```
# exampple from teh table: plural verbs
print_analgous_word('work', 'works', 'speaks')

# 10 more examples using the same relationship
print('\n10 more words using the same relationship:')
print_analgous_word('work', 'works', 'runs')
print_analgous_word('work', 'works', 'walks')
print_analgous_word('work', 'works', 'rides')
print_analgous_word('work', 'works', 'leaves')
print_analgous_word('work', 'works', 'pushes')
print_analgous_word('work', 'works', 'starts')
print_analgous_word('work', 'works', 'eats')
print_analgous_word('work', 'works', 'reads')
print_analgous_word('work', 'works', 'probes')
print_analgous_word('work', 'works', 'taxes')
```

```
speak    2.84

10 more words using the same relationship:
run      2.83
walk     3.01
ride     2.62
leave    3.96
push     2.89
start    2.64
eat      3.66
read     3.37
probe    3.05
tax      3.16
```

Example 1.3: Demonstration of embedding recognizing pluralization of verbs.

For the 10 examples generated, the results are consistent. However, not all words exhibited this stability; some examples were unclear or nonsensical. These 10 examples were specifically chosen because they aligned with the expected outcomes based on the analogy relationships. It's important to note that this method did not work for every word tested, although it was effective for most of them, the selected examples are only the successful ones.

Question 1. 4 The section of A1_Section1_starter.ipynb that is labelled Bias in Word Vectors illus- trates examples of bias within word vectors in the notebook, as also discussed in class. Choose a context that you're aware of (different from those already in the notebook), and see if you can find evidence of a bias that is built into the word vectors. Report the evidence and the conclusion you make from the evidence. [2 points]

In addition to the commonly observed gender biases (demonstrated in the notebook), the language used to train word models can also reflect socioeconomic biases inherent in our word choices. Terms associated with poorer economic status might be further from these positive descriptors and closer to words like "lazy." This illustrates a classist bias in how we frame socioeconomic status, particularly when models can reinforce stereotypes by portraying poor individuals as inherently lazy, suggesting that their struggles are a result of their own shortcomings rather than institutional or inherited disadvantages of an unfair system.

```
[2] torch.norm(glove['rich'] - glove['lazy'])

    tensor(5.7811)

[3] torch.norm(glove['poor'] - glove['lazy'])

    tensor(5.1195)
```

Example 1.4.1: Comparative distance of wealth terms with negative connotations.

Even more troubling is when these socioeconomic biases intersect with racial lines, especially when concerning the dynamics between dominant and privileged groups and historically marginalized groups. Perhaps most troubling is the models' apparent alignment with historical racial prejudices, particularly along lines of class and perceived intellect. This alignment may reflect the data on which the embeddings were trained, as well as the negative stereotypes present in the text overall.

```
[13] print_closest_words(glove['white'] - glove['rich'] + glove['poor'], n = 1)

     black    4.93

[14] print_closest_words(glove['white'] - glove['smart'] + glove['dumb'], n = 1)

     black    4.23
```

Example 1.4.2: Racial and socioeconomic bias within vector embeddings on an analogous basis.

Question 1.5 Change the embedding dimension (also called the vector size) from 50 to 300 and re-run the notebook including the new cosine similarity function from part 2 above. How does the euclidean difference change between the various words in the notebook when switching from d=50 to d=300? How does the cosine similarity change? Does the ordering of nearness change? Is it clear that the larger size vectors give better results - why or why not? [5 points]

Cosine similarity appears to decrease across the examples demonstrated previously.

For instance, the cosine similarity between "apple" and "banana" decreased from 0.5608 to 0.3924. Similarly, the cosine similarity between "good" and "bad" dropped from 0.7965 to 0.6445. The sharpest decline is observed in the pair "good" and "perfect," where the similarity fell significantly from 0.8376 to 0.5893. This decline reflects a deepening of semantic isolation; while "good" and "perfect" might seem closely related on the surface, there is a

substantial semantic gap between them. In contrast, "good" and "bad," though they are opposites, still share similar semantic concepts and can be compared within a broader categorical context, despite their lower similarity score.

```
[ ] torch.cosine_similarity(glove['good'].unsqueeze(0),
                            glove['perfect'].unsqueeze(0))
```

```
tensor([0.8376])
```

```
[ ] torch.cosine_similarity(glove['good'].unsqueeze(0),
                             glove['perfect'].unsqueeze(0))
```

```
tensor([0.5893])
```

Example 1.5.1: “Good” and “Perfect” difference before and after dimension adjustment.

The ordering changed dramatically as well. Words with more expansive connections, such as "nurse" or "health," showed revised lists that included terms more directly related, such as "nursing" and "nurses." Additionally, "benefits" and "care" were added, reflecting how "health" is often associated with the healthcare system in usage. Conversely, more obscure words like the names "Elizabeth" and "Michael" experienced less significant changes, with only one or two outlier adjustments. Overall, the increase in vector size appears to benefit words with more links and greater contextual relevance in usage.

```
[ ] print_closest_words(glove['health'])
```

```
care      4.64
healthcare        5.61
education         5.77
medical           5.91
benefits          6.44
```

Example 1.5.2: “Health” Adjacent words after dimension adjustment.

The ordering exercise conducted in question 1.2 also revealed significant changes. For the "dog" example, "dogs" and "cat" swapped positions in both cosine similarity and Euclidean distance rankings. This suggests that, in natural comparisons, the plural form of a word may be more similar to the singular form than other words. Other notable changes include the higher ranking of "pet" and "puppy" over less relevant terms like "horse." The appearance of "hound" in both lists reflects a deeper semantic connection due to the increased context provided by more dimensions in the word embedding vectors.

Similarly, the word "computer" saw several notable improvements. As with the previous example, the plural form "computers" rose to the top position, followed closely by "software" and the newly emergent term "pc," which is a common synonym for "computer." Additionally, the verb form "computing" was added to the list, demonstrating the various ways the base word "computer" can be used. Overall, the final ordering for "computer" is: "computers," "software," "pc," "technology," and "computing." This represents a much-improved list compared to the output from word embeddings with 50 dimensions.

Adjacent to “Dog”	Cosine Similarity	Adjacent to “Computer”	Cosine Similarity
Dogs	0.79	Computers	0.82
Cat	0.68	Software	0.73
Pet	0.63	Pc	0.62
Puppy	0.59	Technology	0.62
Hound	0.55	Computing	0.62

Table 1.5: “Dog” and “Computer” adjacent words after dimension adjustment.

Question 1.6 There are many different pre-trained embeddings available, including one that tokenizes words at a sub-word level called FastText. These pre-trained embeddedings are available from Torchtext. Modify the notebook to use the FastText embeddings. State any changes that you see in the Bias section of the notebook. [2 points]

It is apparent that FastText embeddings exhibit much less gender-based bias compared to the GloVe embeddings used previously. A notable example of this can be seen in the analogy comparisons between "men" and "doctor" and "women" and "nurse." In the GloVe example, the words "nurse," "pregnant, "child” and "mother" were used. In contrast, the FastText example employs "doctoress" as the most neutral comparative word, despite its less common usage compared to "doctor."

```
[ ] print_closest_words(glove['doctor'] - glove['man'] + glove['woman'])

nurse    3.14
pregnant         3.78
child    3.78
woman    3.86
mother   3.92
```

Example 1.6.1: Resulting output of “Doctor” – “Man” + “Women” using GloVe Embeddings.

```
[ ] print_closest_words(glove['doctor'] - glove['man'] + glove['woman'])

doctoress        4.27
woman    4.32
doctors          4.35
doctor/physician         4.39
doctory          4.43
```

Example 1.6.2: Resulting output of “Doctor” – “Man” + “Women” using FastText Embeddings.

This suggests that FastText shows a gender-neutral bias when assigning the profession of "doctor," as opposed to relating gender to the profession.

Similarly, the "programmer" example in FastText also yields more neutral results compared to GloVe embeddings. The list includes terms directly related to "programmer," with many words incorporating the "programmer/" prefix, indicating that gender is almost entirely irrelevant in the embedding's consideration of the term.

```
[ ] print_closest_words(glove['programmer'] - glove['man'] + glove['woman'])

prodigy          3.67
psychotherapist          3.81
therapist        3.81
introduces       3.91
swedish-born     4.12
```

Example 1.6.3: Resulting output of “Programmer” – “Man” + “Women” using GloVe Embeddings.

```
[ ] print_closest_words(glove['programmer'] - glove['man'] + glove['woman'])

programmers      3.84
programmer/analyst       4.02
programmer/developer     4.04
programmering    4.06
brogrammer       4.11
```

Example 1.6.4: Resulting output of “Programmer” – “Man” + “Women” using FastText Embeddings.

Section 2: Computing Meaning From Word Embeddings

Question 2.1 Write a PyTorch-based function called compare words to category that takes as input

- The meaning category given by a set of words (as discussed above) that describe the category, and
- A given word to ‘measure’ against that category.

The function should compute the cosine similarity of the given word in the category in two ways:

a. By averaging the cosine similarity of the given word with every word in the category, and
b. By computing the cosine similarity of the word with the average of the embeddings of all of the words in the category.

Submit the specific code that you wrote in a separate Python file, A1P2_1.py. [2 points]

Question 2.2 Let’s define the colour meaning category using these words: “colour”, “red”, “green”, “blue”, “yellow.” Compute the similarity (using both methods (a) and (b) above) for each of these words: “greenhouse”, “sky”, “grass”, “azure”, “scissors”, “microphone”, “president” and present them in a table. Do the results for each method make sense? Why or why not? What is the apparent difference between method 1 and 2? [4 points]

Word	**Cosine similarity by method (a)**	**Cosine similarity by method (b)**
greenhouse	0.1831	0.2017
sky	0.6019	0.6702
grass	0.5060	0.5579
azure	0.4078	0.4556
scissors	0.2890	0.3203
microphone	0.3077	0.3431
president	0.2986	0.3292

Table 1.5: Various cosine similarities computed compared by methods (a) and (b) outlined in 2.1

The results generally align with our intuitive understanding of how closely these words relate to the concept of "colour". “Azure” stands out as one of the closest terms, yet it is still relatively distant compared to words like "sky" and "grass," which feel more nebulously connected to colour. In contrast, "azure" is explicitly a colour word. Additionally, the method avoids pitfalls like false positives; for example, "greenhouse" contains "green" but is only tangentially related to colour as a concept.

There is a noticeable difference between methods (a) and (b), with method (b) consistently yielding higher results. This can likely be attributed to the averaging of total vectors, which produces outcomes that better represent the overall concept rather than focusing solely on colour. Furthermore, averaging after calculating the cosine similarity can sometimes introduce outliers in each category that skew the results. In contrast, averaging the vectors before computing similarity tends to create a more accurate connection to the concept itself, smoothing over outliers and leading to higher cosine similarity scores overall, along with improved accuracy regarding the concept.

Question 2.3 Create a similar table for the meaning category temperature by defining your own set of category words, and test a set of 10 words that illustrate how well your category works as a way to determine how much temperature is “in” the words. You should explore different choices and try to make this succeed as much as possible. Comment on how well your approach worked. [4 points]

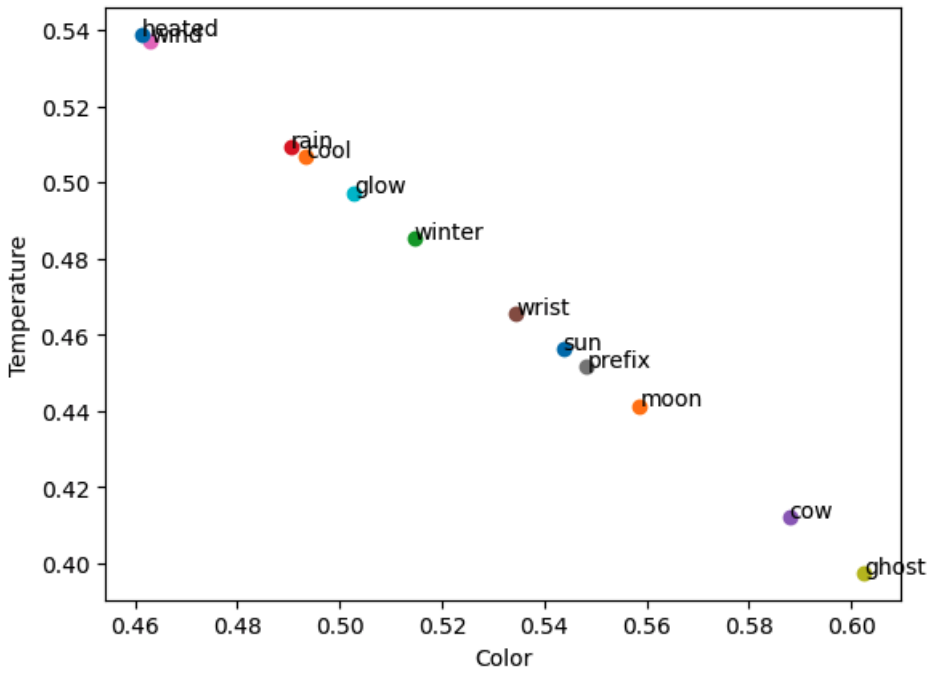
Temperature category words selected: "temperature", "heat", "warmth", "coolness", "climate", "thermal", "degree", "hotness", "coldness", "warmness"

Word	**Cosine similarity by method (a)**	**Cosine similarity by method (b)**
sun	0.2051	0.4111
moon	0.1314	0.2717
winter	0.2350	0.4662
rain	0.2853	0.5234
wrist	0.0885	0.1637
wind	0.3160	0.5870
ghost	-0.0036	0.0219
glow	0.3641	0.5760
heated	0.2709	0.5044
cool	0.3371	0.6061

Table 1.5: Various cosine similarities computed compared by methods (a) and (b) outlined in 2.1

This approach worked relatively well. While method (a) produced more varied results, method (b) consistently showed promise. When the words are considered collectively before calculating cosine similarity, the results reflect their relative closeness to the concept of temperature. For instance, the vectors for “rain,” “heated,” and “cool” are closely related, while concepts we might expect to be less connected, such as “wrist” and “ghost,” show minimal association. The category was designed with common usage of temperature in mind, incorporating words like “climate” and “degree” to reference the idea of temperature and our overall weather. Similarly, pairing “heat” and “cold” helps maintain balance when creating the amalgamated vector for the concept of temperature.

Question 2.4 Use these two categories (colour & temperature) to create a new word vector (of dimension 2) for each of the words given in Table 1, in the following way: for each word, take its (colour, temperature) cosine similarity numbers (try both methods and see which works better), and apply the softmax function to convert those numbers into a probabilities. Plot each of the words in two dimensions (one for colour and one for temperature) using matplotlib. Do the words that are similar end up being plotted close together? Why or why not? [2 points]



Example 2.4: Table 1 words plotted along colour temperature axis.

Similar words tend to cluster closely together; for example, “rain” and “cool,” “sun” and “moon,” and “heated” and “wind” all appear condensed on the plot. Additionally, there is a clear trend: words closely related to temperature are found on one end of the diagonal plot, while words more associated with color (or, in this case, less related to temperature) are positioned further away.

Overall, because the words related to temperature were more deliberately chosen, this resulted in better alignment with the concept of temperature, while the words associated with color were less distinct overall. Similarly, the most neutral words (like "prefix" and "wrist"), which don’t relate to either concept, tend to hover around the middle of the plot.

Section 3: Training A Word Embedding Using the Skip-Gram Method on a Small Corpus

Question 3.1 First, read the file SmallSimpleCorpus.txt so that you see what the sequence of sentences is. Recalling the notion "you shall know a word by the company it keeps," find three pairs of words that this corpora implies have similar or related meanings. For example, 'he' and 'she' are one such example – which you cannot use in your answer! [1 point]

3 commonly used related words are:

- dog, cat
- rub, hold
- the, a

Question 3.2 The prepare_texts function in the starter code is given to you and fulfills several key functions in text processing, a little bit simplified for this simple corpus. Rather than full tokenization (covered in Section 4 below, you will only lemmatize the corpus, which means converting words to their root - for example the word "holds" becomes "hold", whereas the word "hold" itself stays the same (see the Jurafsky [4] text, section 2.6 for a discussion of lemmatization). The prepare_texts function performs lemmatization using the spaCy library, which also performs parts of speech tagging. That tagging determines the type of each word such as noun, verb, or adjective, as well as detecting spaces and punctuation. Jurafsky [4] Section 17.1 and 17.2 describes parts-of-speech tagging. The function prepare_texts uses the parts-of-speech tag to eliminate spaces and punctuation from the vocabulary that is being trained.

Review the code of prepare_texts to make sure you understand what it is doing. Write the code to read the corpus SmallSimpleCorpus.txt, and run the prepare_texts on it to return the text (lemmas) that will be used next. Check that the vocabulary size is 11. Which is the most frequent word in the corpus, and the least frequent word? What purpose do the v2i and i2v functions serve? [2 points]

```
[('and', 160), ('hold', 128), ('dog', 128), ('cat', 128), ('rub', 128),

('can', 104), ('she', 96), ('he', 96), ('I', 80)]
```

Example 3.2: Ordered output of words by count

After running the function, the word with the most frequent occurrence is "and" with 60 occurrences. The least frequent is "I" at 80.

The v2i and i2v variables create a mapping between words and their corresponding indices

- v2i (word-to-index): This dictionary maps each unique word in the vocabulary to a unique integer index. Used for converting words into numerical representations
- i2v (index-to-word): This dictionary maps each index back to its corresponding word, useful for interpreting model outputs or visualizing results, allowing you to map back from the numerical representations to the actual words.

Together, these mappings facilitate the transition between text data and numerical formats needed for model training and evaluation.

Question 3.3 Write a new function called tokenize_and_preprocess_text the skeleton of which is given in the starter code, but not written. It takes the lemmatized small corpus as input, along with v2i (which serves as a simple, lemma-based tokenizer) and a window size window. You should write it so that its output should be the Skip Gram training dataset for this corpus: pairs of words in the corpus that "belong" together, in the Skip Gram sense. That is, for every word in the corpus a set of training examples are generated with that word serving as the (target) input to the predictor, and all the words that fit within a window of size window surrounding the word would be predicted to be in the "context" of the given word. The words are expressed as tokens (numbers). To be clear, this definition of window means that only

For example, if the corpus contained the sequence then the brown cow said moo, and if the current focus word was cow, and the window size was window=3, then there would be two training examples generated for that focus word: (cow, brown) and (cow, said). You must generate all training examples across all words in the corpus within a window of size window. Test that your function works, and show with examples of output (submitted) that it does. [2 points]

```
text = "the brown cow said moo"
lemmas, v2i, i2v = prepare_texts(text)
X, Y = tokenize_and_preprocess_text(lemmas, v2i, 3)
print("X:", X)
print("Y:", Y)
```

```
[('the', 1), ('brown', 1), ('cow', 1), ('say', 1), ('moo', 1)]
X: [0, 1, 1, 2, 2, 3, 3, 4]
Y: [1, 0, 2, 1, 3, 2, 4, 3]
```

Example 3.3: Output of processing "the brown cow said moo"

Given the output we can see that the output example of indices for index 2, corresponding to the word "cow" has 2 indices, of 1 and 3. 1 Corresponds to "brown", and 3 corresponds to "said". Creating 2 pairings of (cow, brown) and (cow, said) respectively.

Question 3.4 Next you should define the model to be trained, the skeleton for which is give in the starter code class Word2vecModel. Portions of the weights in this model, once trained, provides the trained embeddings we are seeking. Recall that the input to the model is a token (a number) representing which word in the vocabulary is being predicted from. The output of the model is of size |V|, where |V| is the size of the vocabulary set V, and each individual output in some sense represents the probability of that word being the correct output. That prediction is based directly on the embedding for each word, and the embeddings are quantities being determined during training. Set the embedding size to be 2, so that will be the size of our word embeddings/vectors. What is the total number of parameters in this model with an embedding size of 2 - counting all the weights and biases? Submit your code for the Word2VecModel class in the file A1P3_4.py. [2 points]

Setting the embedding size to 2, each word will be represented by a vector of size 2. So the total embedding layer will be size 2V.

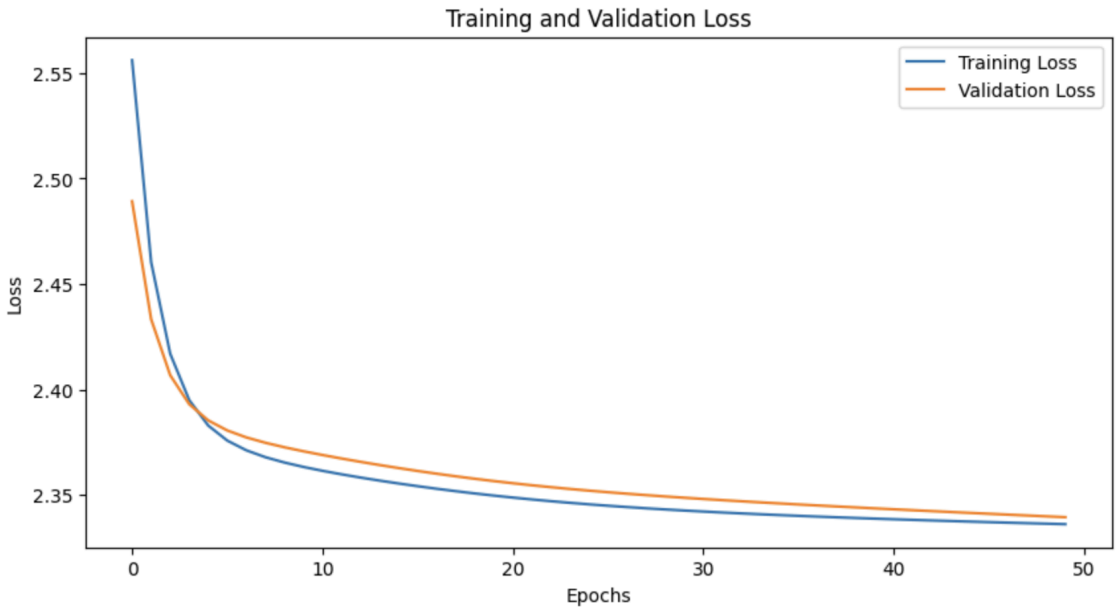
Given an embedding of size 2, the total number of parameters in this model includes the embedding layer and the linear layer. The total number of parameter in the embedding layer is 2V, as outlined before. The total number of parameters includes 2V weights from the input, with an additiona V of vectors for biases.

$$p_{embedding} = s_{embed} \times V = 2V$$

$$p_{linear} = s_{embed} \times V + s_{bias} \times V = 2V + V$$

$$p_{total} = p_{embedding} + p_{linear} = 2V + 3V = 5V$$

Question 3.5 Write the training loop function, given in skeleton form in the starter code as function train_word2vec. It should call the function tokenize_and_preprocess_text to obtain the data and labels, and split that data to be 80% training and 20% validation data. It should use a Cross Entropy loss function, a batch size of 4, a window size of 5, and 50 Epochs of training. Using the default Adam optimizer, find a suitable learning rate, and report what that is. Show the training and validation curves (loss vs. Epoch), and comment on the apparent success (or lack thereof) that these curves suggest. Submit your code for the training function train_word2vec in the file A1P3_5.py [4 points]

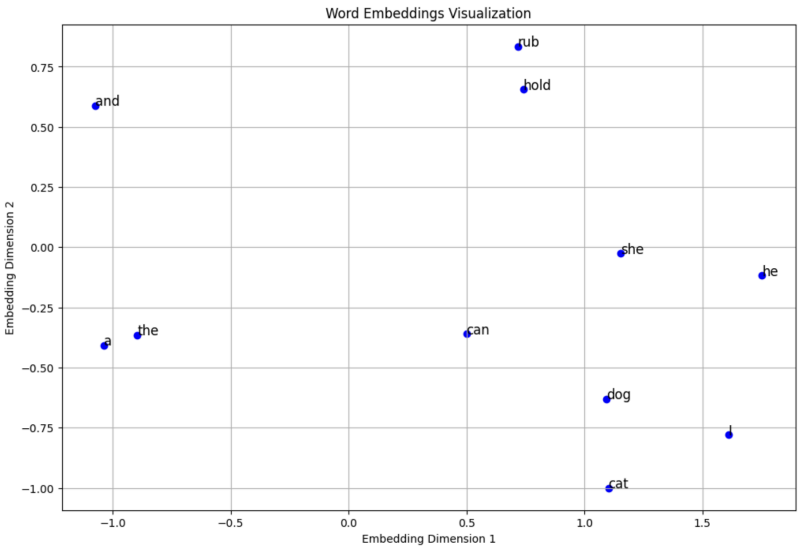


Example 3.5: Training and validation loss of training data over 50 epochs, lr = 0.001

The optimal learning rate was found to be between 0.003 and 0.001. These rates effectively mitigated the risk of severe overfitting that can occur with higher learning rates, despite the randomness of individual runs. Overall, the curves indicate that the training and validation losses were relatively close and matched each other throughout the training process. The shape of the curve aligns with expectations, showing an initial sharp decrease followed by gradual refinement in the remaining epochs, particularly from epochs 20 to 50.

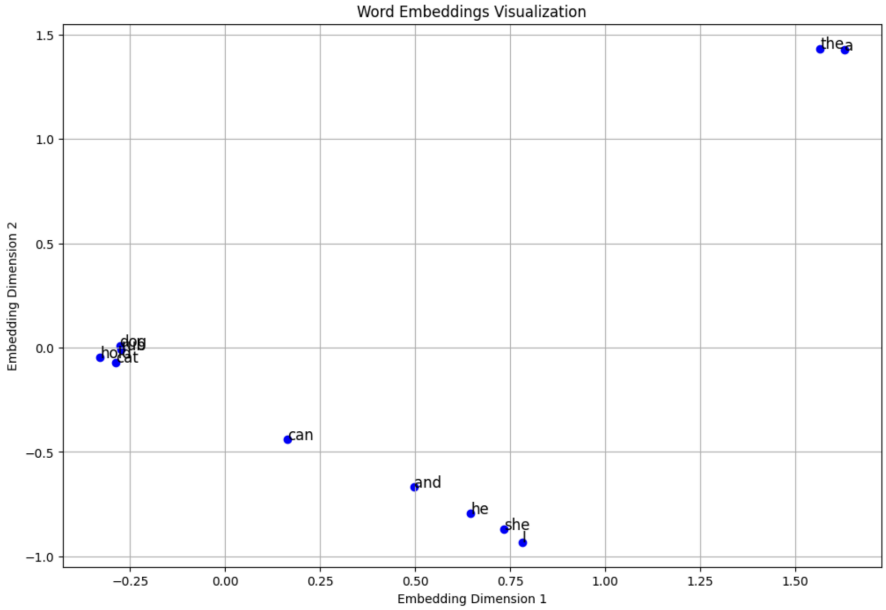
Question 3.6 For your best learning rate, display each of the embeddings in a 2-dimensional plot using Matplotlib. Display both a point for each word, and the word itself. Submit this plot, and answer this question: Do the results make sense, and confirm your choices from part 1 of this Section? What would happen when the window size is too large? At what value would window become too large for this corpus? [5 points]

At a window size of 5, the output displays the relationships we expect to see. For example, "dog" and "cat" are relatively similar, "he" and "she" are paired, "a" and "the" are also paired, and "rub" and "hold" are paired. The remaining words are placed relatively arbitrarily, as they hold less contained relationships compared to the other words based on their semantic meanings.



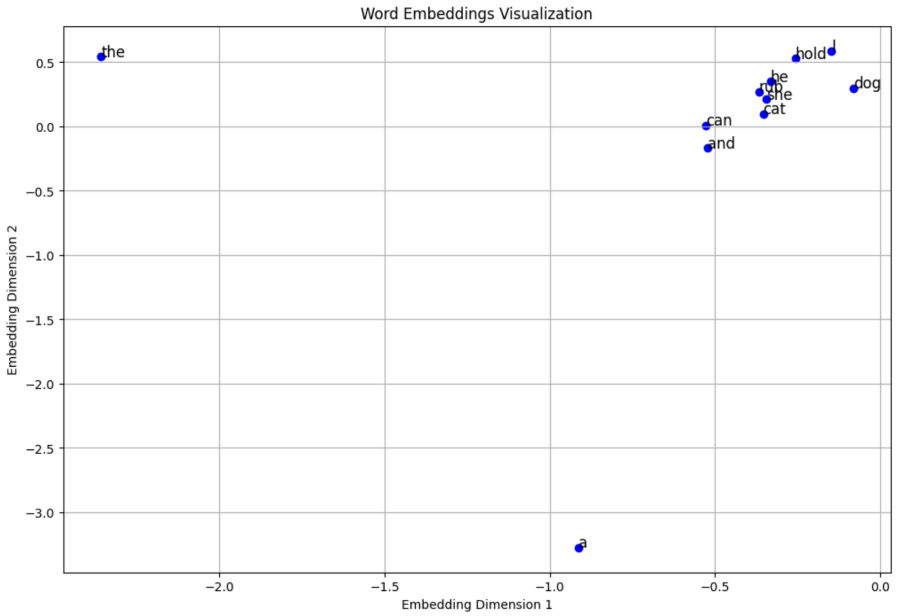
Example 3.6.2 Embedding output at a window size of 5

Larger window sizes of 7 to 9 produce outputs similar to those of size 5, but with a more pronounced clustering of words. “He”, “She”, and “I” show more condensed clustering, as do the proposition words and the context words of “cat”, “dog”, “hold” and “rub”. This may be due to the greater emphasis on word positioning in longer sentences. This includes the positions of propositions and the 2 verbs included. The window sizes of 7 to 9 are effective because they align well with the average sentence length in this dataset.



Example 3.6.2 Embedding output at a window size of 9

However, larger sizes tend to diminish this effect. Specifically, at a window size of 13, the output of the embeddings begins to lose context, likely because this size encompasses multiple sentences, given that most sentences range from 5 to 7 words. As a result, the embeddings start to lose coherence.



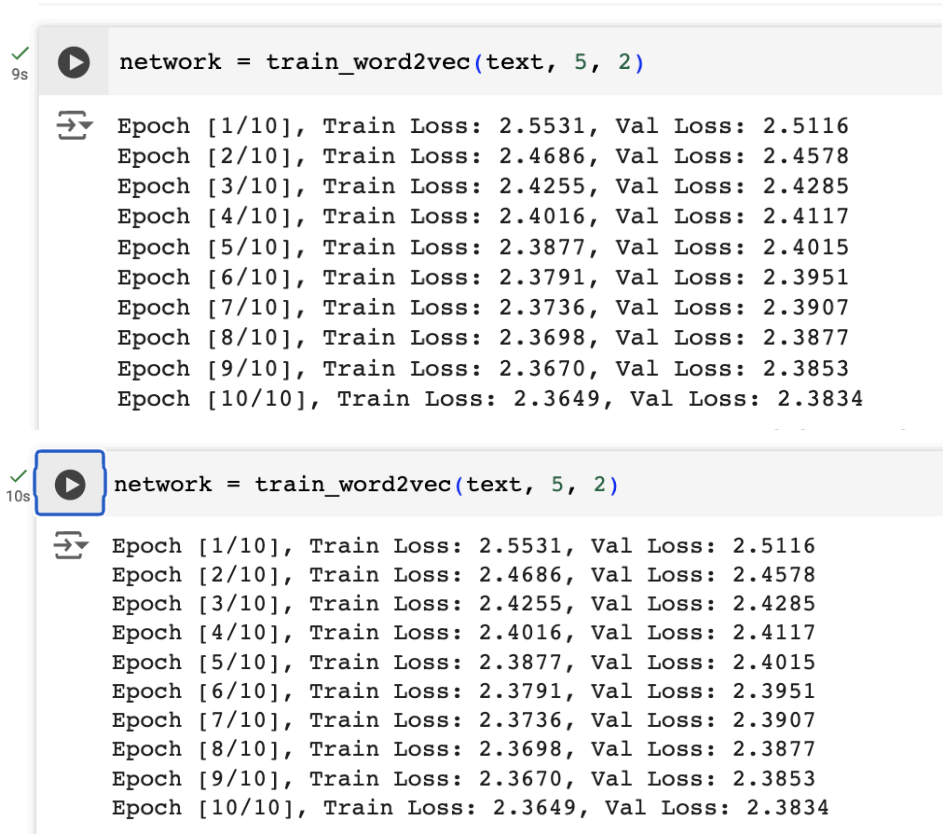
Example 3.6.3 Embedding output at a window size of 13

Question 3.7 Run the training sequence twice - and observe whether the results are identical or not. Then set the random seeds that are used in, separately in numpy and torch as follows (use any number you wish, not necessary 43, for the seed):

np.random.seed(43)
torch.manual_seed(43)

Verify (and confirm this in your report) that the results are always the same every time you run your code if you set these two seeds. This will be important to remember when you are debugging code, and you want it to produce the same result each time.

Before manual seeding, each training cycle would produce different results even given identical parameters, post manual seedings, the results are identical each time you rerun the training function.



Example 3.7: Two separate runs with manual seeding.

Section 4: Skip-gram with Negative Sampling

Question 4.1 Take a quick look through LargerCorpus.txt to get a sense of what it is about. Give a 3 sentence summary of the subject of the document. [1 point]

LargerCorpus.txt discusses the evolution of money, from bartering to the use of various commodities like tin, iron, and silk as currency, and eventually the widespread use of metals like gold, silver, and copper in coinage. It discusses ancient coin-making techniques, starting with crude methods of engraving, and details the advancements in creating more sophisticated coins with both obverse and reverse designs. Additionally, the document highlights how coins and medals serve as historical records, reflecting significant events, figures, and even cultural aspects like portraiture and heraldic symbols.

* Completed with the aid of summarization AI.

Question 4.2 The prepare_texts function in the starter code is a more advanced version of that same function given in Section 3. Read through it and make sure you understand what it does. What are the functional differences between this code and that of the same function in Section 3? [1 point]

Text Tokenization and Cleaning: Section 4 code uses both nltk for sentence tokenization and spaCy for word tokenization, performing more thorough cleaning by removing punctuation and special characters at the sentence level.

Handling of Infrequent Words: Section 4 code filters out words that occur less than a specified minimum frequency, which reduces noise. Section 3 code processes all words, regardless of frequency, keeping them all in the vocabulary.

Out of Vocabulary Handling: Section 4 code introduces an OOV token to replace infrequent or unknown words not included in the vocabulary, ensuring the model can handle unseen terms. The section 3 function does not handle OOV words.

Output and Dictionaries: Both functions generate dictionaries mapping words to indices and vice versa. However, the section 4 code includes more sophisticated filtering, ensuring that only frequent words are mapped, while the section 3 code simply maps all words without filtering based on frequency.

Question 4.3 Write the code to read in LargerCorpus.txt and run prepare_texts on it. Determine the number of words in the text, and the size of the filtered vocabulary, and the most frequent 20 words in the filtered vocabulary, and report those. Of those top 20 most frequent words, which one(s) are unique to the subject of this particular text? [1 point]

```
print(len(w2i))
```

```
2569
```

Example 4.3.1: Length of words registered in the word to index

There are 2569 words (after filtering)

```
for i, (key, value) in enumerate(w2i.items()):
    if i < 20:
        print(value, key)
```

```
0 the
1 of
2 be
3 and
4 in
5 to
6 a
7 for
8 as
9 by
10 he
11 with
12 coin
13 this
14 on
15 his
16 which
17 at
18 it
19 from
```

Example 4.3.2: Top 20 words in frequency in the Larger Corpus

From this most common list, the word specific to this text is the word coin, as the text largely discusses the role of currency and exchange.

Question 4.4 Write the function tokenize_and_preprocess_text which generates both positive and neg- ative samples for training in the following way: first, use w2i to create a tokenized version of the corpus. Next, for every word in the corpus, generate: a set of positive examples within a window of size window similar to the example generation in Section 3. Set the label for each positive example to be +1, in the list Y produced by the function. Also, for each word, gen- erate the same number of negative samples as positive examples, by choosing that number of randomly-chosen words from the entire corpus. (You can assume randomly chosen words are very likely to be not associated with the given word). Set the label of the negative examples to be -1. Submit your code for this function in the file named A1P4_4.py. How many total examples were created? [2 points]

Using a window size of 5, (similar to the output in Section 3). A total of 49082 samples were created.

```
[65] X, T, Y = tokenize_and_preprocess_text(textlist, w2i, 5)
     print(len(X))

     498028
```

Example 4.4: Number of Examples after Tokenizing and Preprocessing

Question 4.5 OPTIONAL: The training can be made more efficient by reducing the number of examples for the most frequent words, as the above method creates far more examples connected to those words than are necessary for successful training. Revise your function to reduce the number of examples. Submit your code for this function in the file named A1P4_5.py, and state how many examples remain for the corpus using this reduction. [2 bonus points]

Utilize an inverse sampler that considers word frequency: the higher the frequency of a word, the fewer samples are taken for each window occurrence. After re-running the process, the total length of the training vector decreased by 109 157

```
X, T, Y = tokenize_and_preprocess_text(textlist, w2i, 5)
X2, T2, Y2 = tokenize_and_preprocess_text_with_frequency_cutoff(textlist, w2i, 5)
print(len(X) - len(X2))

109157
```

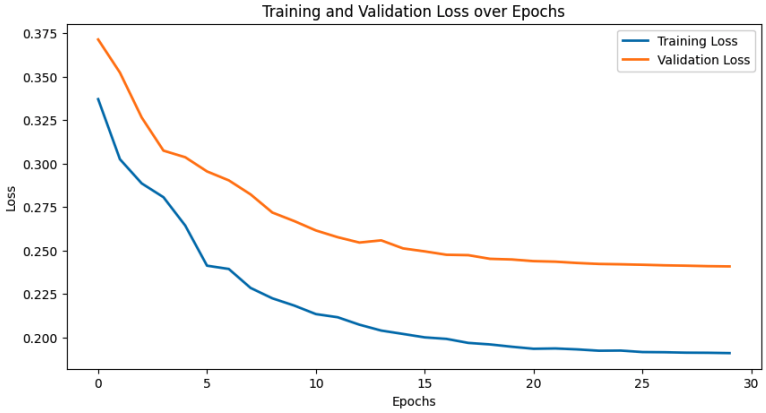
Example: 4.5 Preprocessing with Frequency Cutoff

Question 4.6 Write the model class, from the given skeleton class SkipGramNegativeSampling. The model's input are the tokens of two words, the word and the context. The model stores the embedding that is being trained (just one embedding per token), which is set up in the _init_ method. The output of the forward function is a binary prediction, but it should only compute the raw prediction of the network (which is the dot product of the the two embeddings of the input tokens). Submit your model class in the file A1P4_6.py. [2 points]

Question 4.7 Write the training function, given in skeleton form in the starter code as the function train_sgns. This function should call the tokenize_and_preprocess_text function to obtain the data and labels, and split that data to be 80% training and 20% validation data. It should use an embedding size of 8, a window size of 5, a batch size of 4, and 30 Epochs of training. The loss function should be log(σ(prediction)) for positive examples, and log(σ(−prediction)) where σ is the sigmoid function. Since it is possible for the prediction to be 0, to prevent the log function from having an infinite result, we typically add a small constant the output of the sigmoid to prevent this - typically 1e-5.

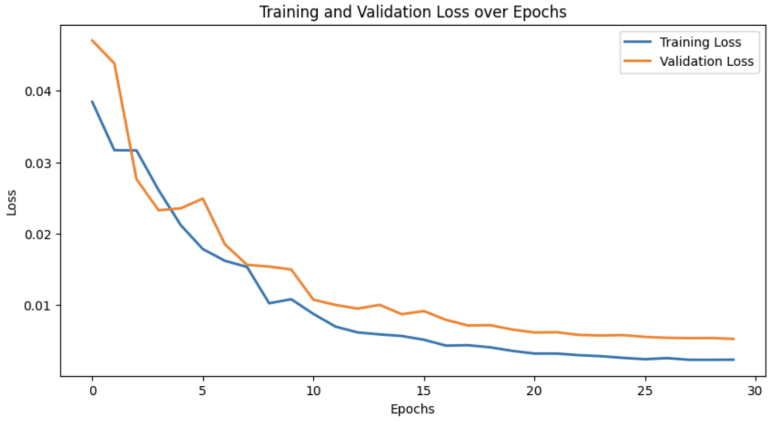
Using the default Adam optimizer, find a suitable learning rate, and report what that is. Show the training and validation curves vs. Epoch, and comment on the apparent success (or lack thereof) that these curves suggest. Submit your training function in the file A1P4_7.py. [4 points]

The overall model ran very slowly due to the large number of parameters. Increasing the batch size helped alleviate this issue and reduce overfitting. With the default batch size of 4, a larger window size of 5 in combination with a meager learning rate of 1e-7 was necessary to prevent overfitting in later epochs, where memorization of the training data caused the validation results to diverge significantly.



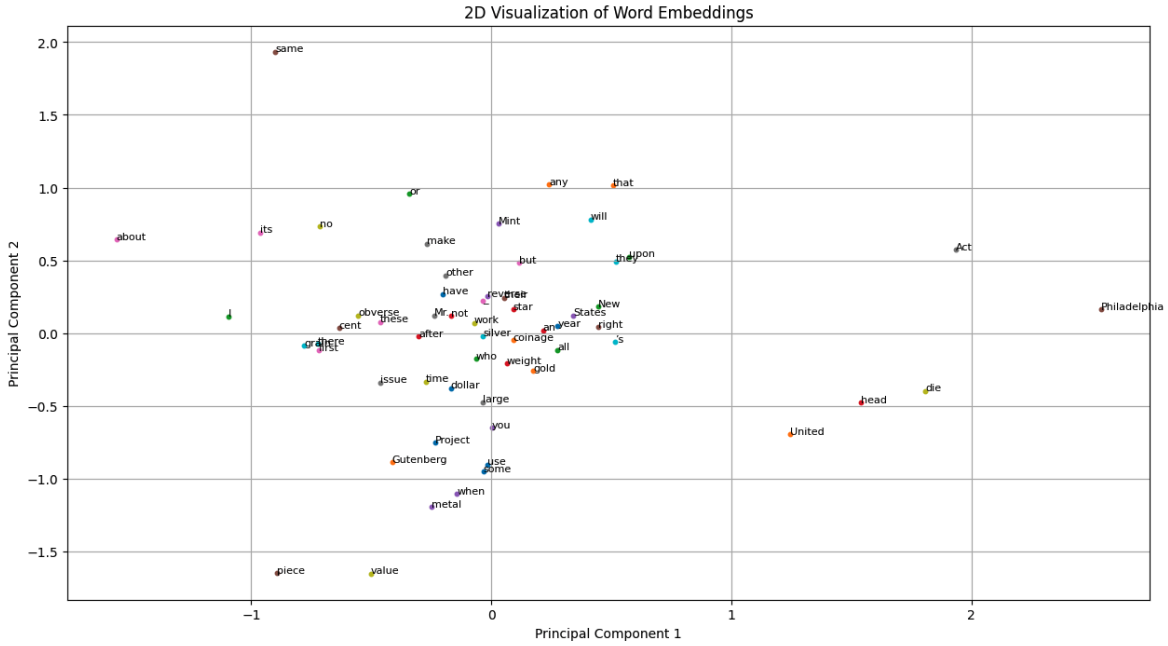
Example 4.7: Training with a batch size of 4 and learning rate 1e-7

In contrast, using a larger batch size of 128 allowed for a higher learning rate of 5e-5, resulting in much faster convergence, and allowed the use of a larger window size of 9 which can create more context and training examples, resulting in a better overall learning curve.



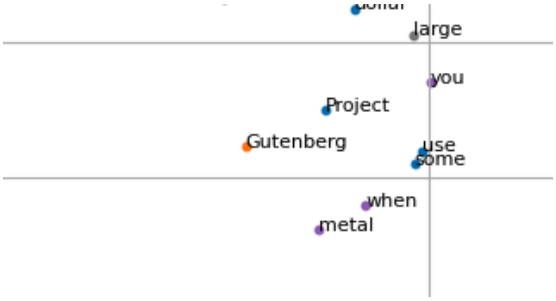
Example 4.7: Training with batch size of 512 and learning rate 1e-4

Question 4.8 Write a function that reduces the dimensionality of the embeddings from 8 to 2, using principle component analysis (PCA) as shown in the partially-written function visualize_embedding. Since we cannot visualize the embeddings of the entire vocabulary, the function is written to select a range of the most frequent words in the vocabulary. (Too frequent are not interesting, but too infrequent are also less interesting). Comment on how well the embeddings worked, finding two examples each of embeddings that appear correctly placed in the plot, and two examples where they are not. [3 points]



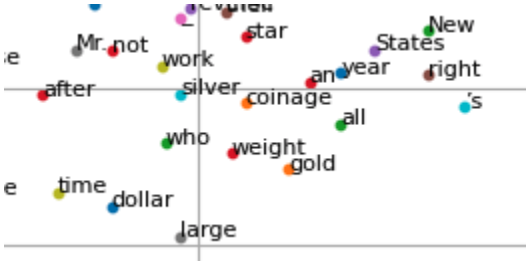
Example 4.8.1 Overall plot (too small to see, below, focus regions are highlighted)

In this region, we can see that “Gutenberg” and “Project”—the e-book initiative mentioned frequently in the text—are grouped closely together.



Example 4.8.2: Focus Region 1 - Successful pairing

Here, we observe a cluster comprising “gold”, “weight”, “coinage”, and “silver,” all of which are closely related in the context of the text. These terms are associated with early coinage and highlight how precious metals were considered valuable as exchange currencies.

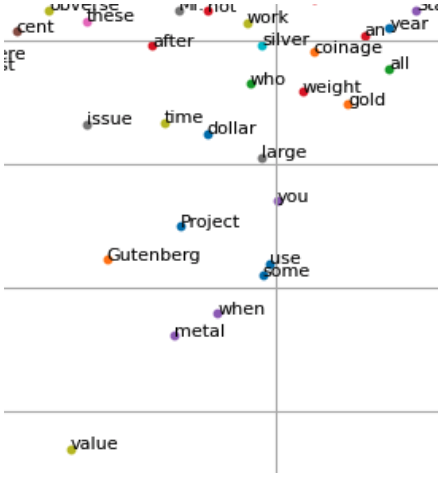


Example 4.8.3: Focus Region 2 - Successful clustering

In this case, the model appears to have failed to classify two closely related words: “United” and “States.” The capitalized versions refer to the country, indicating that they are almost always used together. However, in this plot, they are positioned relatively far apart.

Example 4.8.3 Focus Region 3 - Unsuccessful pairing

As discussed previously, the central cluster related to currency is missing three important terms: “metal”, “value”, and “cent”. These words should be closely associated with the initial cluster observed in the upper right corner of the focus region. However, they are not appropriately grouped with the other related terms; instead, they are dispersed and positioned quite far apart.



Example 4.8.4 - Focus Region 4 - Unsuccessful clustering