GLOVE

What is GloVe?

- Global Vectors for Word Representation.
- It is an unsupervised learning algorithm that generates vector representations, or embeddings, of words.
- Researchers Richard Socher, Christopher D. Manning, and Jeffrey Pennington first presented it in 2014.
- By using the statistical co-occurrence data of words in a given corpus, GloVe is intended to capture the semantic relationships between words.

- The fundamental concept underlying GloVe is the representation of words as vectors in a continuous vector space, where the angle and direction of the vectors correspond to the semantic connections between the appropriate words.

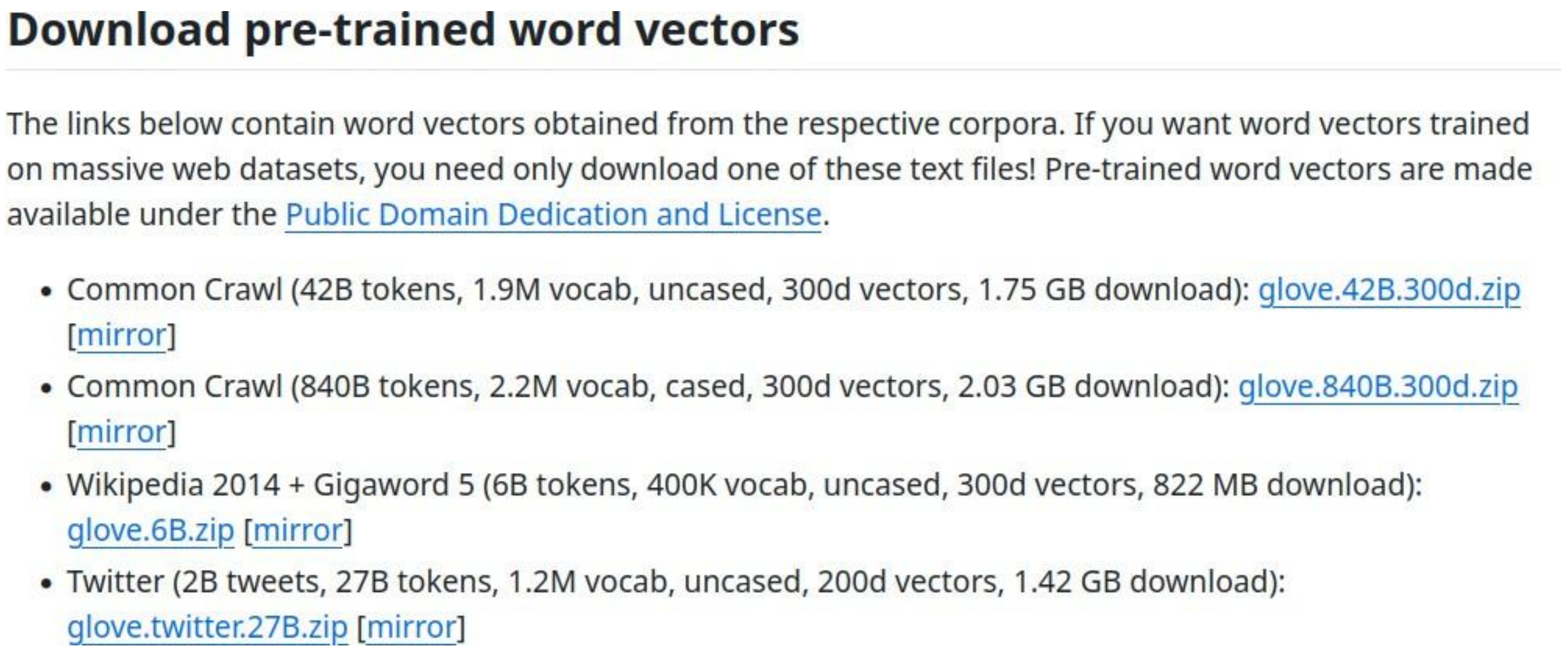
What should I do ?

- GloVe builds a co-occurrence matrix using word pairs and then optimizes the word vectors to minimize the difference between the pointwise mutual information of the corresponding words and the dot product of vectors.

- The glove has pre-defined dense vectors for around every 6 billion words of English along with many other general-use characters like commas, braces, and semicolons.
- The algorithm’s developers frequently make the pre-trained GloVe embeddings available.
- Users can select a pre-trained GloVe embedding in a dimension (e.g., 50-d, 100-d, 200-d, or 300-d vectors) that best fits their needs in terms of computational resources and task specificity.

- Here d stands for dimension. 100d means, in this file each word has an equivalent vector of size 100.
- Glove files are simple text files in the form of a dictionary. Words are key and dense vectors are values of key.

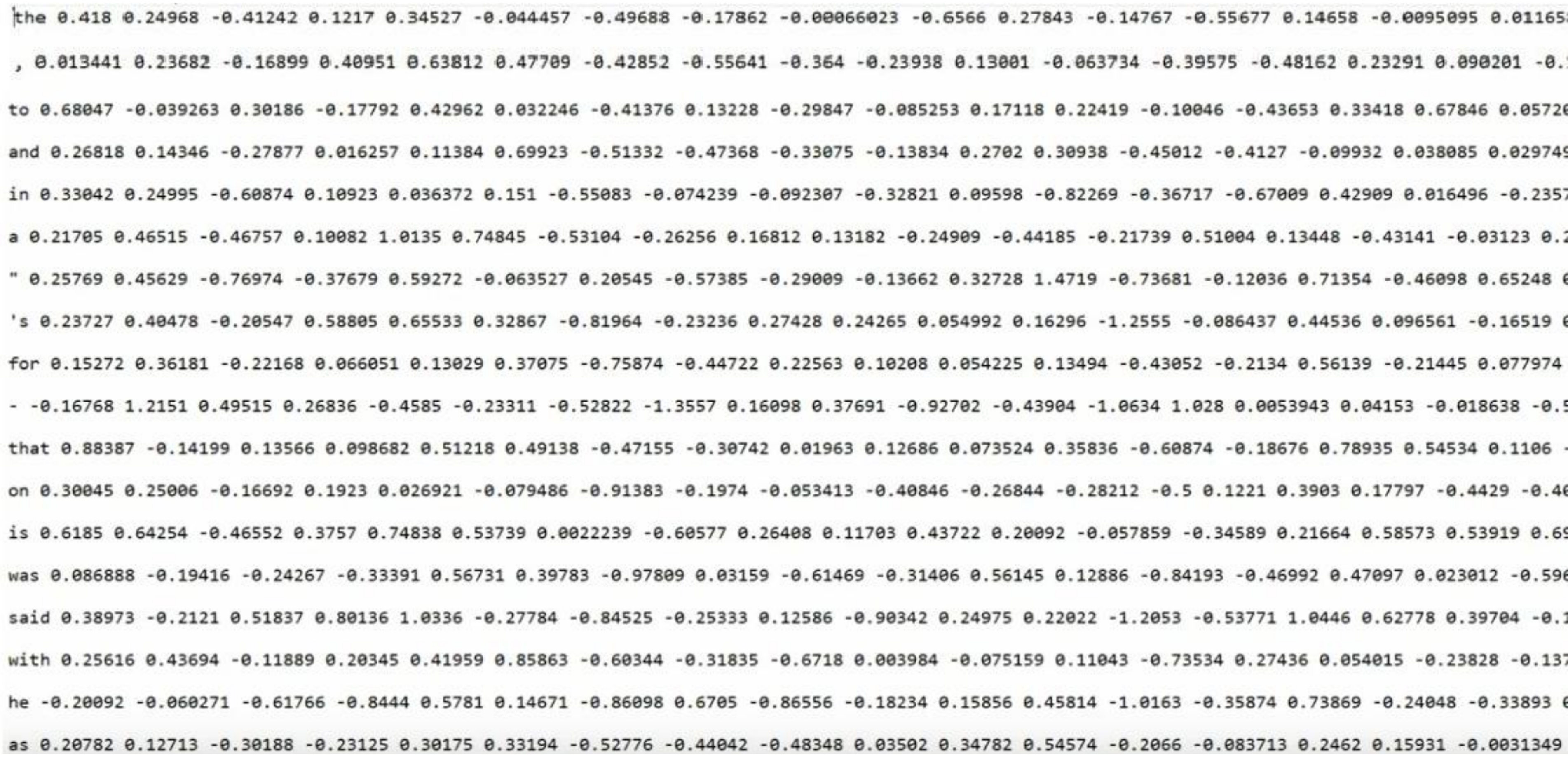
https://github.com/stanfordnlp/GloVe



Download pre-trained word vectors

The links below contain word vectors obtained from the respective corpora. If you want word vectors trained on massive web datasets, you need only download one of these text files! Pre-trained word vectors are made available under the Public Domain Dedication and License.

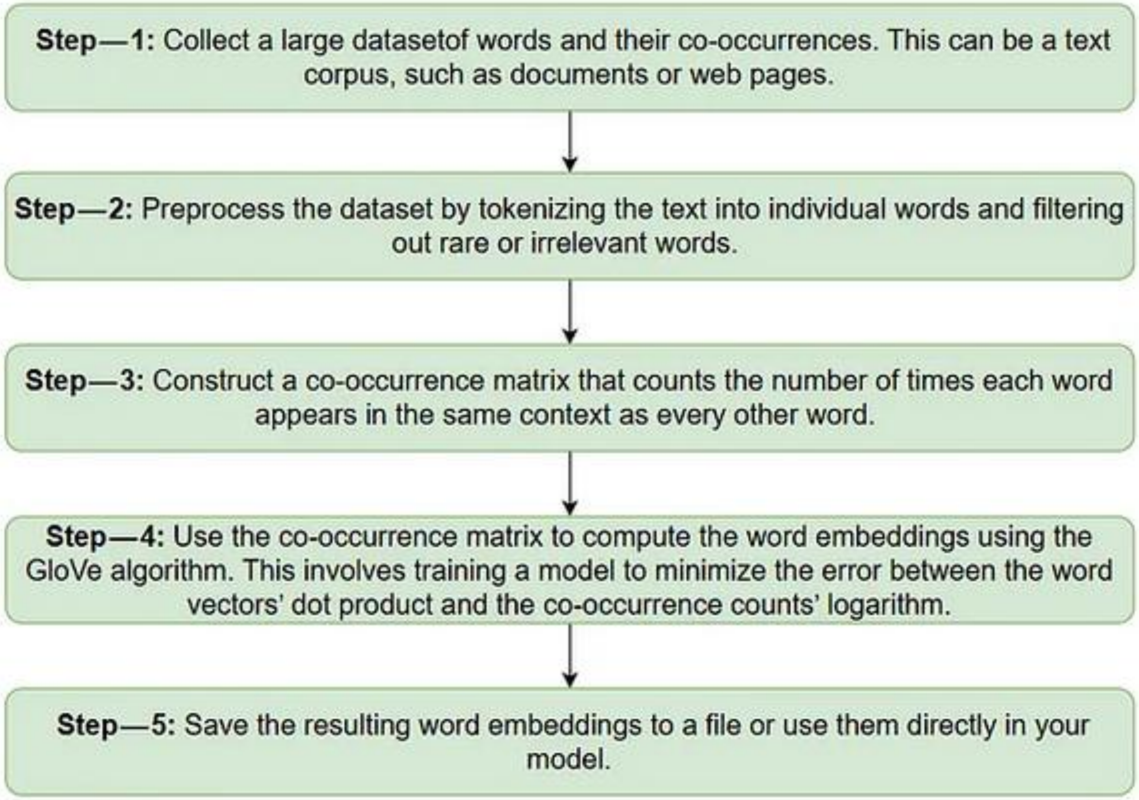
- Common Crawl (42B tokens, 1.9M vocab, uncased, 300d vectors, 1.75 GB download): glove.42B.300d.zip [mirror]
- Common Crawl (840B tokens, 2.2M vocab, cased, 300d vectors, 2.03 GB download): glove.840B.300d.zip [mirror]
- Wikipedia 2014 + Gigaword 5 (6B tokens, 400K vocab, uncased, 300d vectors, 822 MB download): glove.6B.zip [mirror]
- Twitter (2B tweets, 27B tokens, 1.2M vocab, uncased, 200d vectors, 1.42 GB download): glove.twitter.27B.zip [mirror]



```
the 0.418 0.24968 -0.41242 0.1217 0.34527 -0.044457 -0.49688 -0.17862 -0.00066023 -0.6566 0.27843 -0.14767 -0.55677 0.14658 -0.0095095 0.01165
, 0.013441 0.23682 -0.16899 0.40951 0.63812 0.47709 -0.42852 -0.55641 -0.364 -0.23938 0.13001 -0.063734 -0.39575 -0.48162 0.23291 0.090201 -0.
to 0.68047 -0.039263 0.30186 -0.17792 0.42962 0.032246 -0.41376 0.13228 -0.29847 -0.085253 0.17118 0.22419 -0.10046 -0.43653 0.33418 0.67846 0.0572
and 0.26818 0.14346 -0.27877 0.016257 0.11384 0.69923 -0.51332 -0.47368 -0.33075 -0.13834 0.2702 0.30938 -0.45012 -0.4127 -0.09932 0.038085 0.02974
in 0.33042 0.24995 -0.60874 0.10923 0.036372 0.151 -0.55083 -0.074239 -0.092307 -0.32821 0.09598 -0.82269 -0.36717 -0.67009 0.42909 0.016496 -0.235
a 0.21705 0.46515 -0.46757 0.10082 1.0135 0.74845 -0.53104 -0.26256 0.16812 0.13182 -0.24909 -0.44185 -0.21739 0.51004 0.13448 -0.43141 -0.03123 0.
" 0.25769 0.45629 -0.76974 -0.37679 0.59272 -0.063527 0.20545 -0.57385 -0.29009 -0.13662 0.32728 1.4719 -0.73681 -0.12036 0.71354 -0.46098 0.65248
's 0.23727 0.40478 -0.20547 0.58805 0.65533 0.32867 -0.81964 -0.23236 0.27428 0.24265 0.054992 0.16296 -1.2555 -0.086437 0.44536 0.096561 -0.16519
for 0.15272 0.36181 -0.22168 0.066051 0.13029 0.37075 -0.75874 -0.44722 0.22563 0.10208 0.054225 0.13494 -0.43052 -0.2134 0.56139 -0.21445 0.077974
- -0.16768 1.2151 0.49515 0.26836 -0.4585 -0.23311 -0.52822 -1.3557 0.16098 0.37691 -0.92702 -0.43904 -1.0634 1.028 0.0053943 0.04153 -0.018638 -0.
that 0.88387 -0.14199 0.13566 0.098682 0.51218 0.49138 -0.47155 -0.30742 0.01963 0.12686 0.073524 0.35836 -0.60874 -0.18676 0.78935 0.54534 0.1106
on 0.30045 0.25006 -0.16692 0.1923 0.026921 -0.079486 -0.91383 -0.1974 -0.053413 -0.40846 -0.26844 -0.28212 -0.5 0.1221 0.3903 0.17797 -0.4429 -0.4
is 0.6185 0.64254 -0.46552 0.3757 0.74838 0.53739 0.0022239 -0.60577 0.26408 0.11703 0.43722 0.20092 -0.057859 -0.34589 0.21664 0.58573 0.53919 0.6
was 0.086888 -0.19416 -0.24267 -0.33391 0.56731 0.39783 -0.97809 0.03159 -0.61469 -0.31406 0.56145 0.12886 -0.84193 -0.46992 0.47097 0.023012 -0.59
said 0.38973 -0.2121 0.51837 0.80136 1.0336 -0.27784 -0.84525 -0.25333 0.12586 -0.90342 0.24975 0.22022 -1.2053 -0.53771 1.0446 0.62778 0.39704 -0.
with 0.25616 0.43694 -0.11889 0.20345 0.41959 0.85863 -0.60344 -0.31835 -0.6718 0.003984 -0.075159 0.11043 -0.73534 0.27436 0.054015 -0.23828 -0.13
he -0.20092 -0.060271 -0.61766 -0.8444 0.5781 0.14671 -0.86098 0.6705 -0.86556 -0.18234 0.15856 0.45814 -1.0163 -0.35874 0.73869 -0.24048 -0.33893
as 0.20782 0.12713 -0.30188 -0.23125 0.30175 0.33194 -0.52776 -0.44042 -0.48348 0.03502 0.34782 0.54574 -0.2066 -0.083713 0.2462 0.15931 -0.0031349
```

How GloVe works?

Algorithm for GloVe



Step—1: Collect a large datasetof words and their co-occurrences. This can be a text corpus, such as documents or web pages.

Step—2: Preprocess the dataset by tokenizing the text into individual words and filtering out rare or irrelevant words.

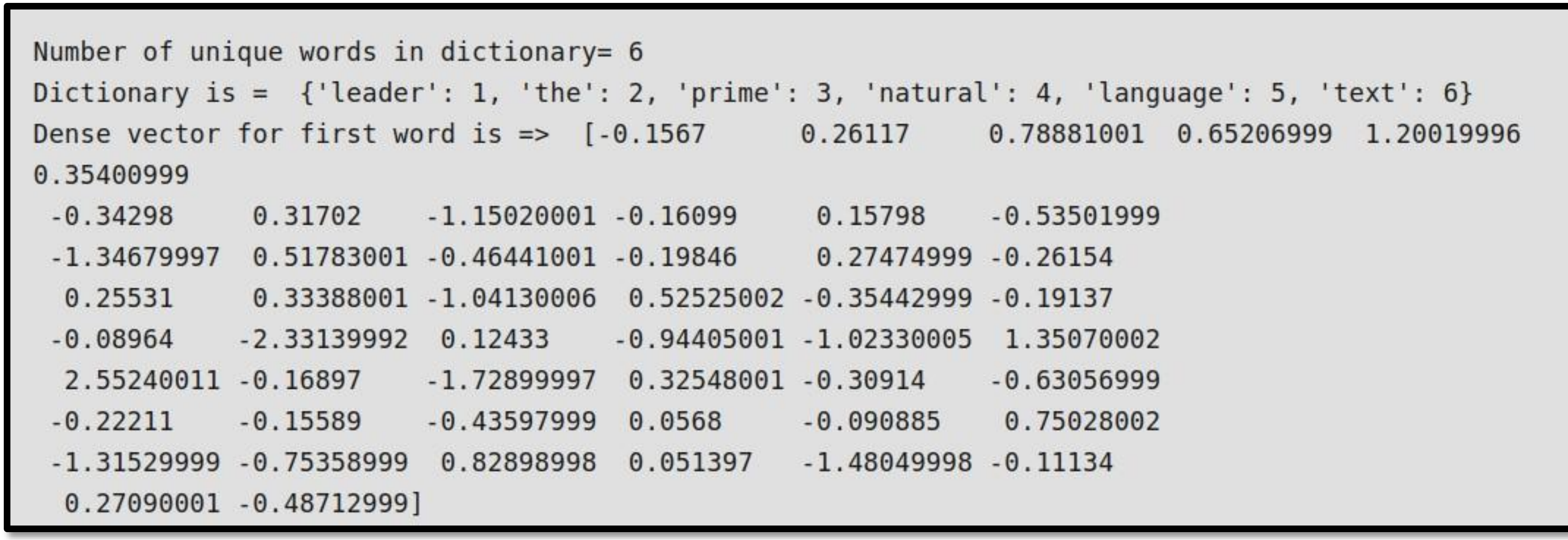
Step—3: Construct a co-occurrence matrix that counts the number of times each word appears in the same context as every other word.

Step—4: Use the co-occurrence matrix to compute the word embeddings using the GloVe algorithm. This involves training a model to minimize the error between the word vectors' dot product and the co-occurrence counts' logarithm.

Step—5: Save the resulting word embeddings to a file or use them directly in your model.

How GloVe works?

- Create Vocabulary Dictionary:
- Vocabulary is the collection of all unique words present in the training dataset.
- The dataset is tokenized into words, then all the frequency of each word is counted.
- Then words are sorted in decreasing order of their frequencies.
- Words having high frequency are placed at the beginning of the dictionary.



```
Number of unique words in dictionary= 6
Dictionary is =  {'leader': 1, 'the': 2, 'prime': 3, 'natural': 4, 'language': 5, 'text': 6}
Dense vector for first word is =>  [-0.1567      0.26117     0.78881001  0.65206999  1.20019996
0.35400999
 -0.34298     0.31702    -1.15020001 -0.16099     0.15798    -0.53501999
 -1.34679997  0.51783001 -0.46441001 -0.19846     0.27474999 -0.26154
  0.25531     0.33388001 -1.04130006  0.52525002 -0.35442999 -0.19137
 -0.08964    -2.33139992  0.12433    -0.94405001 -1.02330005  1.35070002
  2.55240011 -0.16897    -1.72899997  0.32548001 -0.30914    -0.63056999
 -0.22211    -0.15589    -0.43597999  0.0568     -0.090885    0.75028002
 -1.31529999 -0.75358999  0.82898998  0.051397   -1.48049998 -0.11134
  0.27090001 -0.48712999]
```

Example of co-occurrence matrix

I love Learnerea

I love DataScience

Window = 1

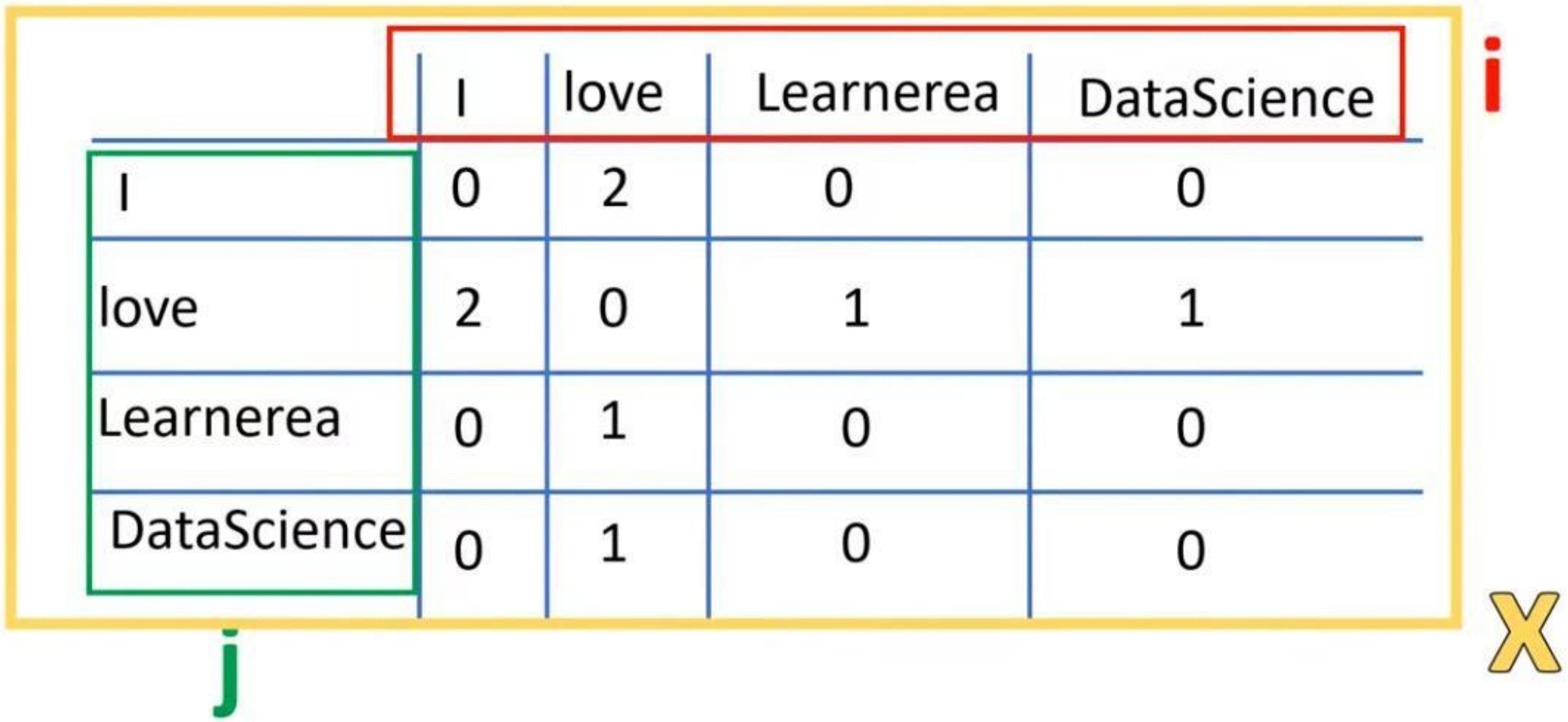
	I	love	Learnerea	DataScience
I				
love				
Learnerea				
DataScience				

	I	love	Learnerea	DataScience
I	0	2	0	0
love	2	0	1	1
Learnerea	0	1	0	0
DataScience	0	1	0	0

	I	love	Learnerea	DataScience
I	0	2	0	0
love	2	0	1	1
Learnerea	0	1	0	0
DataScience	0	1	0	0

i

j



	I	love	Learnerea	DataScience
I	0	2	0	0
love	2	0	1	1
Learnerea	0	1	0	0
DataScience	0	1	0	0

i

	I	love	Learnerea	DataScience
I	0	2	0	0
love	2	0	1	1
Learnerea	0	1	0	0
DataScience	0	1	0	0

j

I love Learnerea

I love DataScience

Step 1: Collect a Large Dataset

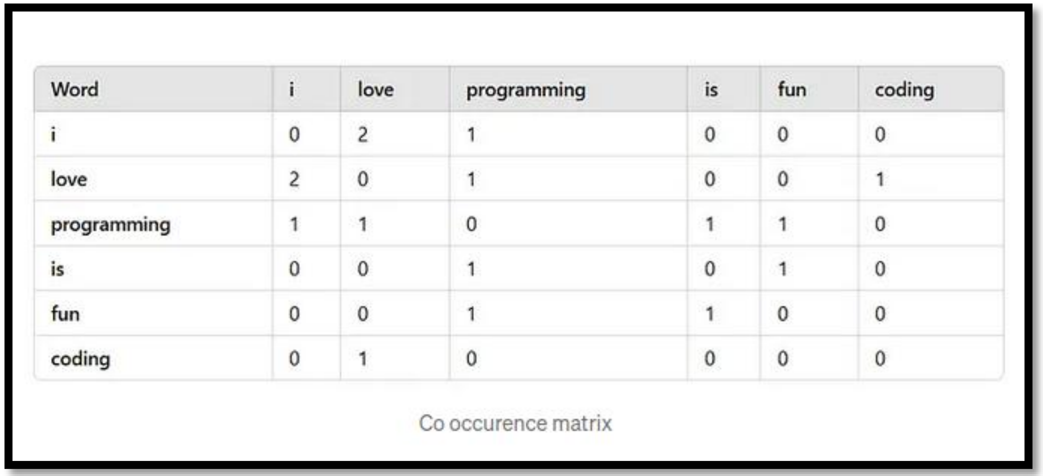
- For simplicity, let’s use this small example corpus:
- *“I love programming. Programming is fun. I love coding.”*

Step 2: Preprocess the Dataset

- **Tokenize the text**: Split the sentences into words: ["I", "love", "programming", "Programming", "is", "fun", "I", "love", "coding"]
- **Normalize the text**: Convert words to lowercase to ensure consistency: ["i", "love", "programming", "programming", "is", "fun", "i", "love", "coding"]
- **Filter out rare words**: Keep only meaningful words (ignore very rare or irrelevant words, like stopwords, if needed).

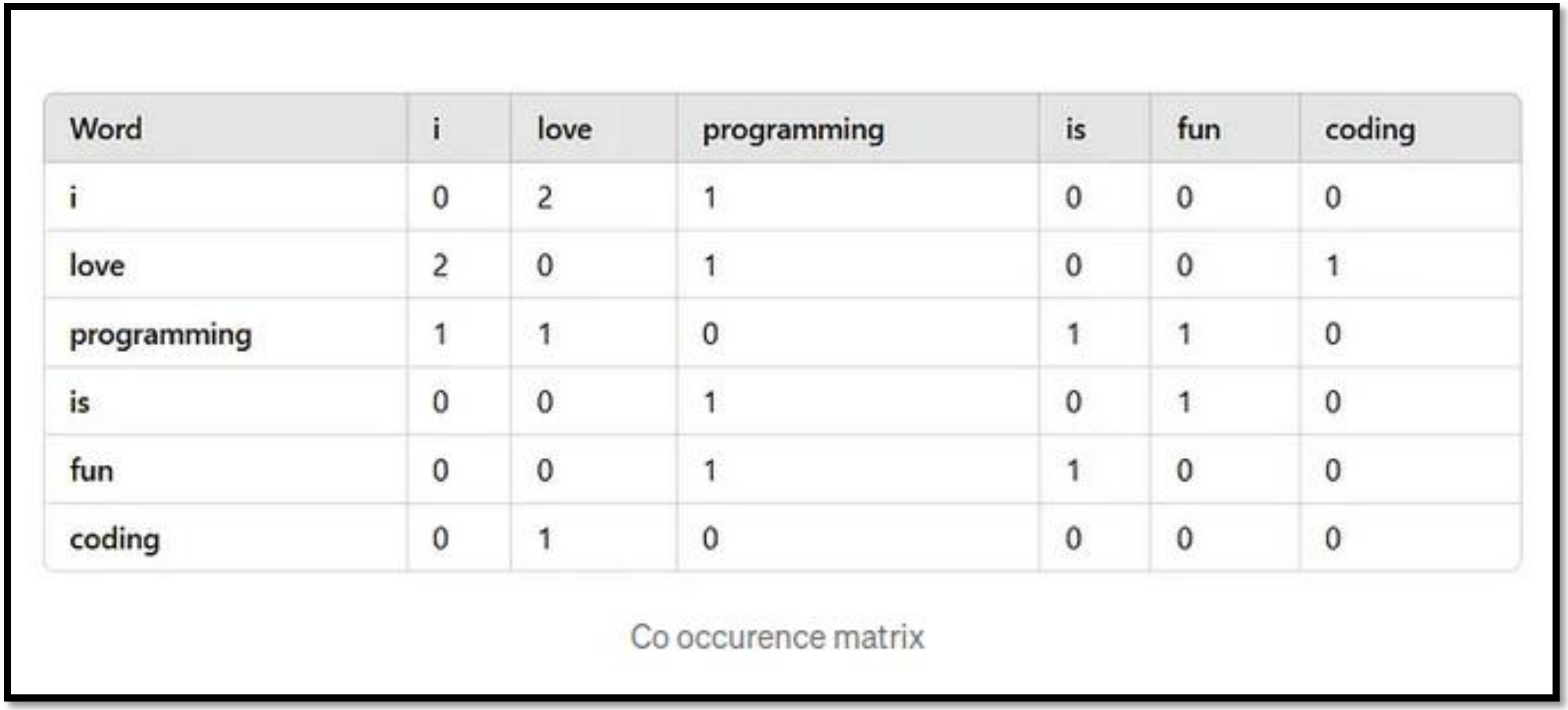
Step 3: Construct a Co-occurrence Matrix

- Define a **context window** (e.g., 2 words to the left and right).
- Count how often each word co-occurs with every other word within this window.
- Each cell in the table represents how many times two words appeared together within the context window.



Word	i	love	programming	is	fun	coding
i	0	2	1	0	0	0
love	2	0	1	0	0	1
programming	1	1	0	1	1	0
is	0	0	1	0	1	0
fun	0	0	1	1	0	0
coding	0	1	0	0	0	0

Co occurence matrix



Word	i	love	programming	is	fun	coding
i	0	2	1	0	0	0
love	2	0	1	0	0	1
programming	1	1	0	1	1	0
is	0	0	1	0	1	0
fun	0	0	1	1	0	0
coding	0	1	0	0	0	0

Co occurence matrix

Step 4: Use the Co-occurrence Matrix to Compute Embeddings

- GloVe aims to learn word vectors wi and wj such that their dot product approximates the logarithm of the co-occurrence probability:
- The **dot product** of word vectors (for word pairs, e.g., “i” and “love”).
- The **logarithm of the co-occurrence count** (e.g., log(2) for "i" and "love").

Formula:

$$J = \sum_{i,j} f(X_{ij}) \left(\mathbf{w}_i \cdot \mathbf{w}_j + b_i + b_j - \log(X_{ij}) \right)^2$$

Where:

- X_{ij}: Co-occurrence count for words i and j.
- $\mathbf{w}_i$: Embedding for word i.
- b_i: Bias term for word i.
- $f(X_{ij})$: Weighting function to reduce the impact of very frequent words.

The algorithm adjusts the word vectors $\mathbf{w}_i$ and $\mathbf{w}_j$ to minimize this error.

Example Calculation

Using the example co-occurrence matrix, let's calculate for the pair ("i", "love") with a co-occurrence count $X_{ij} = 2$.

1. **Co-occurrence Value:** $\log(X_{ij}) = \log(2) = 0.693$.
2. **Dot Product:** Assume we initialize the word vectors $\mathbf{w}_i = [w_{i1}, w_{i2}]$ and $\mathbf{w}_j = [w_{j1}, w_{j2}]$, each with 2 dimensions. The dot product:

$$\mathbf{w}_i \cdot \mathbf{w}_j = w_{i1} \cdot w_{j1} + w_{i2} \cdot w_{j2}$$

3. **Error for the Pair:** Substituting into the loss function for this pair:

$$\text{Loss} = f(X_{ij}) \cdot (\mathbf{w}_i \cdot \mathbf{w}_j + b_i + b_j - \log(X_{ij}))^2$$

4. **Weighting Function $f(X_{ij})$:** $f(X_{ij})$ ensures the algorithm focuses on meaningful co-occurrence pairs. For example:

$$f(X_{ij}) = \min\left(\left(\frac{X_{ij}}{X_{\max}}\right)^{\alpha}, 1\right)$$

Where $X_{\max}$ and α are hyperparameters. For simplicity, let's assume $f(X_{ij}) = 1$.

4. Weighting Function $f(X_{ij})$: $f(X_{ij})$ ensures the algorithm focuses on meaningful co-

5. **Substitute Values**: For word pair ("i", "love"):

- Assume initial vectors $\mathbf{w}_i = [0.5, 0.8]$ and $\mathbf{w}_j = [0.6, 0.7]$.
- Dot product:
$$\mathbf{w}_i \cdot \mathbf{w}_j = (0.5 \cdot 0.6) + (0.8 \cdot 0.7) = 0.3 + 0.56 = 0.86$$
- Bias terms $b_i = 0.1, b_j = 0.2$.
- Loss for this pair:
$$\text{Loss} = (0.86 + 0.1 + 0.2 - 0.693)^2 = (1.16 - 0.693)^2 = (0.467)^2 = 0.218$$

6. **Gradient Updates**: Gradients are calculated to update the embedding vectors $\mathbf{w}_i, \mathbf{w}_j$, and bias terms b_i, b_j to reduce the loss:

$$\mathbf{w}_i \leftarrow \mathbf{w}_i - \eta \cdot \frac{\partial J}{\partial \mathbf{w}_i}$$

Where η is the learning rate.

Generalizing for All Pairs

- The process is repeated for all pairs in the co-occurrence matrix. Each word pair contributes a small part to the overall objective function J. Over multiple iterations, the embeddings wi and wj converge to values that represent meaningful relationships between words.

Resulting Embeddings

- After optimization:
- The word "i" might have a vector: [0.45, 0.8].
- The word "love" might have a vector: [0.6, 0.7].
- These vectors can now be used in NLP tasks like similarity comparisons or downstream applications.
- This process ensures that frequent co-occurrences (e.g., "i" and "love") result in embeddings that are close in vector space, while less frequent pairs (e.g., "i" and "fun") are farther apart.

Step 5: Save or Use the Word Embeddings

- After training, each word is represented by a vector. For example:

  “i” → [0.12, -0.45, 0.67, ...]

  “love” → [0.56, -0.23, 0.34, ...]

- Save these embeddings for downstream tasks like sentiment analysis, text classification, or similarity calculations.