Lect 5:Word Embedding *Fast text*&GloVe

Limitations of Word2Vec

[1] Out of Vocabulary(OOV) Words:
In Word2Vec, an embedding is created for each word. Any words that not encountered during its training can not ber handeled. It is out of vocab.

For example,
words such as "**Fast**" and "**Text**" are present in the vocabulary of Word2Vec. But if you try to get embedding for the compound word "**Fasttext**", you will get an out of vocabulary error.

[2] Morphology:
For words with same "root" such as "eat", "eaten", "eating", "eats"
In Word2Vec *each word is learned uniquely* based on the context it appears in.

Objective: *utilizing the internal structure of the word to make the process more efficient.*

What is FastText

FastText is an NLP Library developed by the Facebook research team for text classification and Word Embeddings.

Original Paper:

Enriching Word Vectors with Subword Information

Piotr Bojanowski • and Edouard Grave • and Armand Joulin and Tomas Mikolov

Facebook AI Research

June 2017

There are two frameworks of FastText:

- Text Representation (fastText Word Embedding)
- Text Classification

The main difference between Baseline Word2Vec and FastText is:

In "Word2Vec Baseline" representation is word based

In "FastText" each word is representation is character n-gram based.

Why FastText is bettern than Baseline CBOW and Skipgram

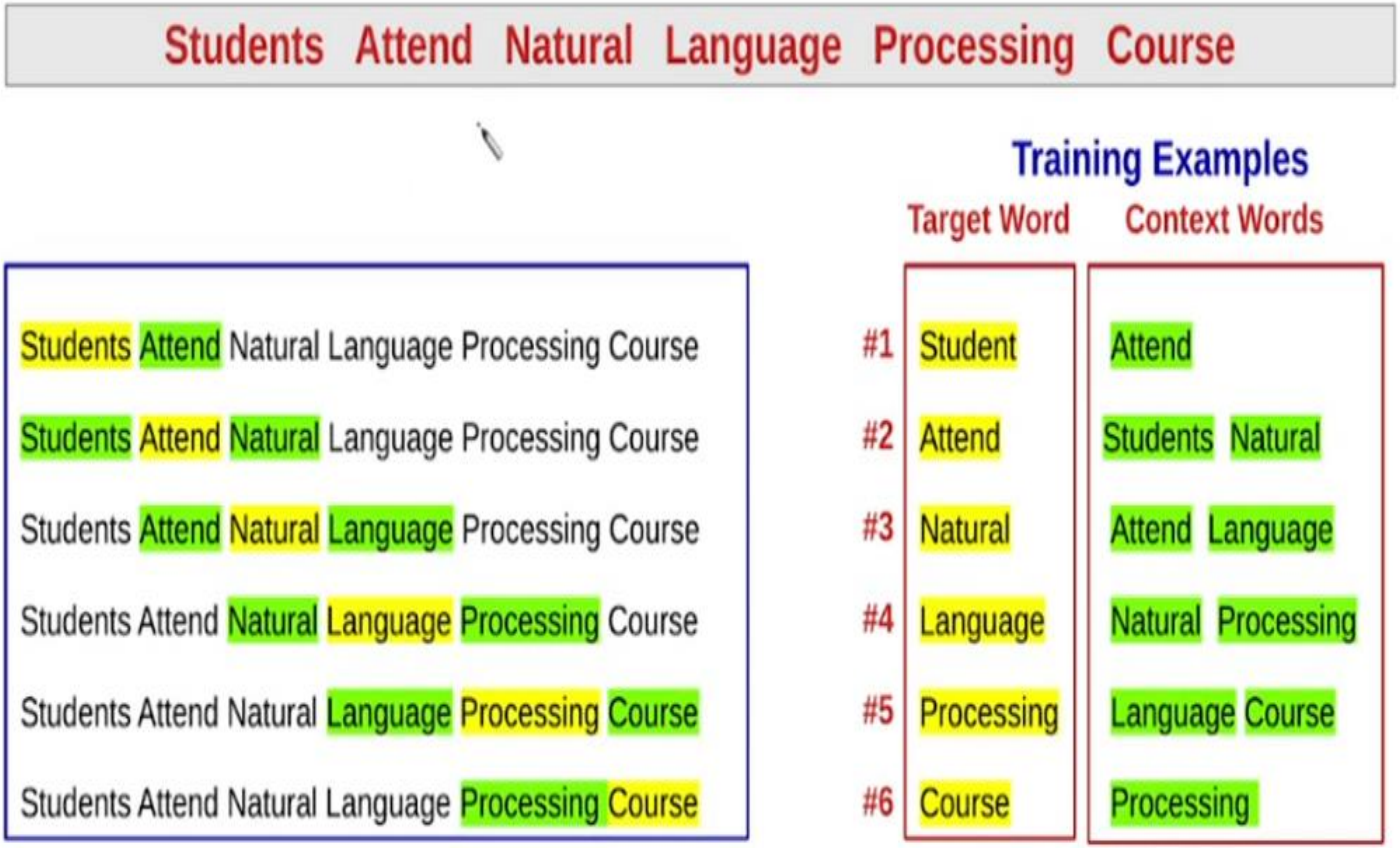
FastText uses sub-word information with character-ngrams.

One problem of Baseline Word2Vec is Out Of Vocab (OOV).
It is represented by NULL vector resulting in reduction of performance.

In FastText, Representing out-of-vocab words by summing their sub-words has better performance than assigning null vectors.

In addition to:
test Results show that FastText generally has better performance than Baseline Word2Vec algorithms (CBOW and SkipGram) even with OOV.

Skip-Gram Representation Of FastText Embedding



Students Attend Natural Language Processing Course

Students Attend Natural Language Processing Course

Students Attend Natural Language Processing Course

Students Attend Natural Language Processing Course

Students Attend Natural Language Processing Course

Students Attend Natural Language Processing Course

Students Attend Natural Language Processing Course

Training Examples

	Target Word	Context Words
#1	Student	Attend
#2	Attend	Students Natural
#3	Natural	Attend Language
#4	Language	Natural Processing
#5	Processing	Language Course
#6	Course	Processing

Skip-Gram Representation Of FastText Embedding

Tri Gram [n-gram with n=3]

Student	<Students>	<Students>	<St Stu tud ude den ent nts ts>
Attend	<Attend>	<Attend>	<At Att tte ten end nd>
Natural	<Natural>	<Natural>	<Na Nat atu tur ura ral al>
Language	<Language>	<Language>	<La Lan ang ngu gua uag age ge>
Processing	<Processing>	<Processing>	<Pr Pro roc oce ces ssi sin ing ng>
Course	<Course>	<Course>	<Co Cou our rse se>

Note: *Denote Begin and End of Word by Angular Brackets* < >

Skip-Gram Representation Of FastText Embedding

1- Select Target and Context Words

- Define Sliding window size “C”. (C is in range from 3 to 6)
- Slide a window with length “2C+1”. The target word is the word at the center, Context words are other words inside the window
- Select “C” words to the right and to the left of the Target word.

2- Select Negative Samples for each context word

(Skip-gram with negative sampling)

- Collect negative samples randomly with probability proportion to the square root of the uni-gram frequency.
- For one actual context word, “N” random negative words are sampled (e.g. N=5)

Skip-Gram Representation Of FastText Embedding

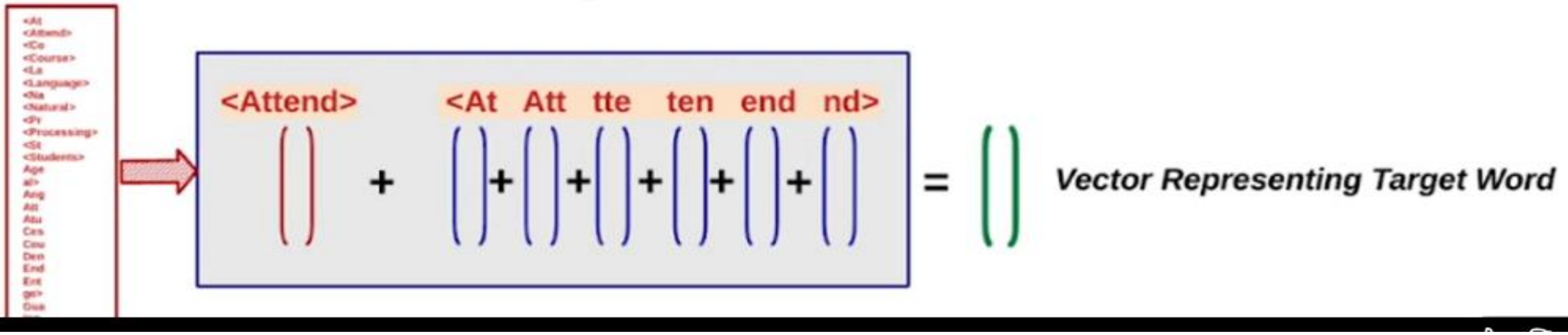
3- Sub-word generation

< >

- Add angular brackets to each target word to denote the beginning and end of a word
- For a word, generate character n-grams of length 3 to 6.
- Generate character n-grams of length **n** by sliding a window of **n** characters from the start of the angular bracket till the ending angular bracket is reached.

4- Prepare Vectors for Target, Context, and Negative Words

- For the context words: take their word vectors from the embedding table
- For the Negative words: take their word vectors from the embedding table.
- For Target Words (words at the Center): take a sum of vectors representing the character n-grams enclosed in the word) and the vector representing the whole word itself. All vectors are from the embedding Table.



Skip-Gram Representation Of FastText Embedding

5- Train Neural Network with architecture:

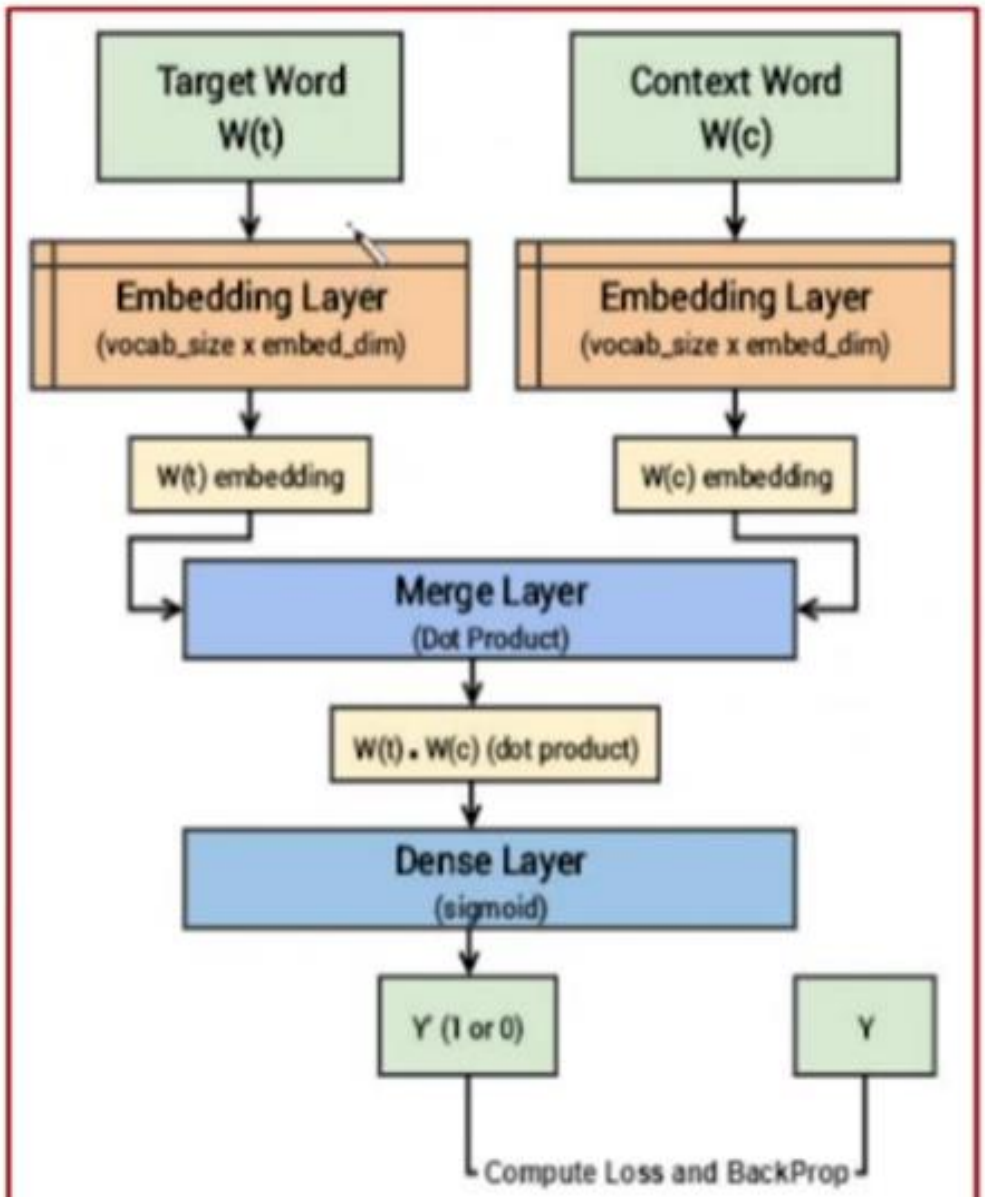
Two inputs, One hidden layer, and One sigmoid output

- Calculate dot product between Target word vector and actual context words (one by one) and apply sigmoid function to get a match score between 0 and 1. (the target output is "1")

- Calculate dot product between Target word vector and Negative words (one by one) and apply sigmoid function to get a match score between 0 and 1. (the target output is "0")

- Based on the loss, update the embedding vectors with SGD optimizer to bring actual context words closer to the target word and increase distance to the negative samples.

Negative Sampling implementation Model



Disadvantages of FastText

Increased Memory Usage:

•Subword information requires storing embeddings for a large number of n-grams, leading to higher memory consumption.

Complexity:

•Generating embeddings involves aggregating subword embeddings, which adds computational overhead.

Insensitive to Context:

•**Like Word2Vec and GloVe, FastText embeddings are static and do not adapt to the context in which a word appea**rs.

Similar vectors for different words

FastText breaks words into smaller parts called **n-grams**. However, sometimes these n-grams can be too similar between different words, leading to confusion. For example:

•The word **“house”** might be broken into the n-grams: **“<ho”, “hou”, “ous”, “use**.**”**

•The word **“horse”** might be broken into the n-grams: **“<ho”, “hor”, “ors”, “rse**.**”**

Both “house” and “horse” share the n-grams **“<ho”** and **“hou”**, so FastText might mistakenly think “horse” is similar to “house” when they are actually different.

This overlap of n-grams can sometimes lead to misclassifications, where the model mixes up words that are not related.

if **all subwords** in the new word are completely unique, FastText cannot generate embeddings for them.

What Does “Static” Mean?

- **Single Representation:** A word always has the same embedding vector. For example:
- The word **“bank”** (as in a financial institution) and **“bank”** (as in the edge of a river) will both have the same embedding.
- **No Context Awareness:** The vector does not change based on how the word is used in a sentence.

Why Is FastText Static?

- FastText improves over Word2Vec by breaking words into **subwords** (e.g., “bank” → “ba”, “ank”, etc.), which helps represent rare or OOV words. However, the resulting embeddings are still based on pre-trained data and do not adjust dynamically to different sentence contexts.

Word embedded based on two methods

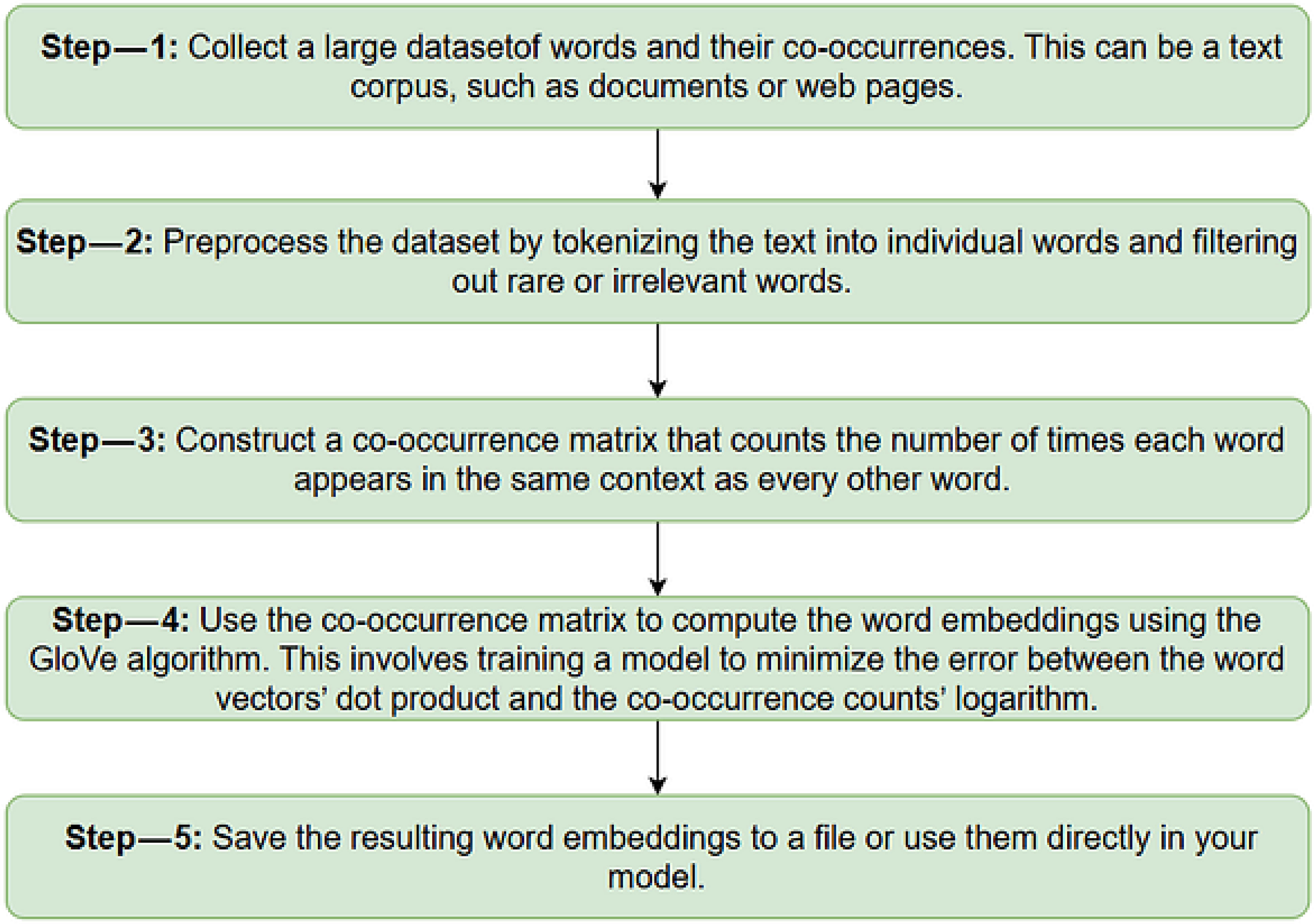
1.Global matrix factorization

This method focuses on capturing the **global statistics** of word co-occurrences in a large text corpus by using **matrix factorization**.

2. **Local context window.**

This method focuses on capturing **local relationships** between words by considering only the nearby words in a sentence. This is where methods like **CBOW** and **Skip-Gram** from Word2Vec come into play.

GloVe (Global Vector)



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.

Word co-occurrence

We will follow the original GloVe paper here

- https://nlp.stanford.edu/pubs/glove.pdf

Notations:

Notations:

- X_{ij} — number of times word j occurs in the context of word i
- $X_i = \sum_j X_{ij}$ — number of times any word occurs in the context of word i.
- $P_{ij} = P(j|i) = \frac{X_{ij}}{X_i}$ — probability that word j occurs in the context of word i.

Step 1: Collect a Large Dataset

You need a **large text corpus** where words frequently co-occur.

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:

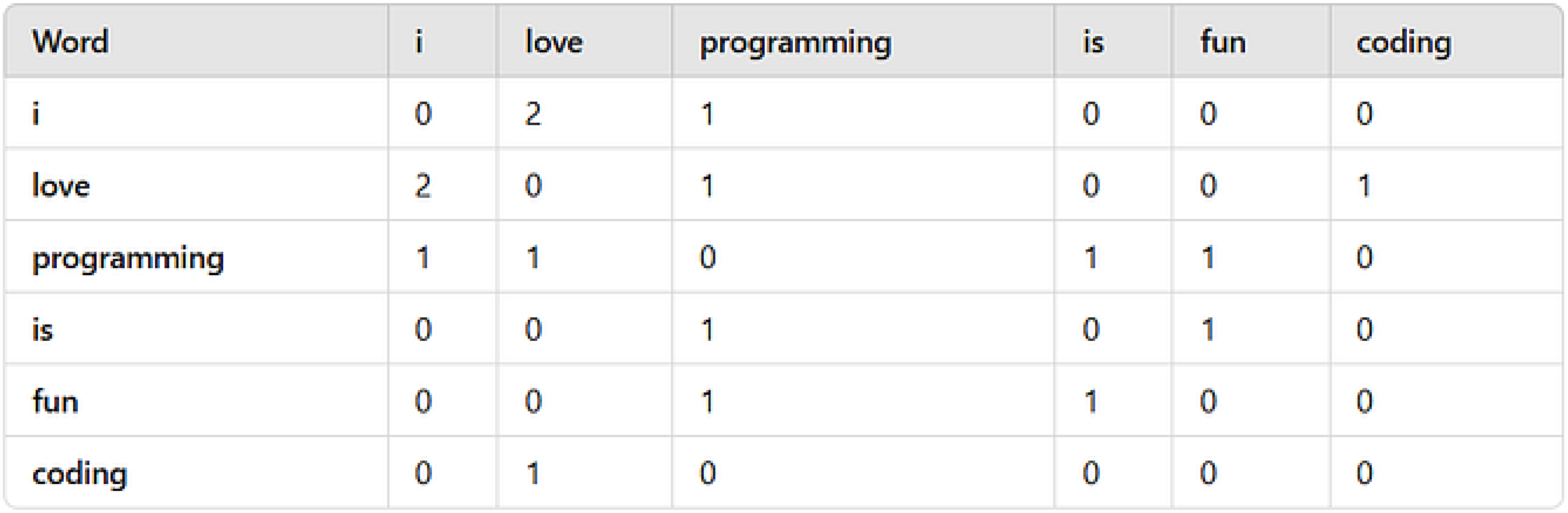
"]i", "love", "programming", "programming", "is", "fun", "i", "love", "coding["

•**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.

“I love programming. Programming is fun. I love coding”.



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

Each cell in the table represents how many times two words appeared together within the context window.

The following real example is taken from the GloVe paper.

```
## # A tibble: 4 x 4
##   Expression 'P(k|ice)' 'P(k|steam)' 'P(k|ice) / P(k=steam)'
##   <chr>          <dbl>        <dbl>                     <dbl>
## 1 k=solid      0.00019      0.000022                  8.9
## 2 k=gas        0.000066     0.00078                   0.085
## 3 k=water      0.003        0.0022                    1.36
## 4 k=fashion    0.000017     0.000018                  0.96
```

It shows that it is important to consider how often two different words occur in the context of another word and then take ratios of probabilities.

GloVe

In GloVe model, we have word vectors w and context vectors $\tilde{w}$.

Let w_i and w_j be word vectors for words i and j and let $\tilde{w}_k$ be the context vector for word k. The general model is

$$F(w_i, w_j, \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}$$

To reflect the vector space structure on $\mathbb{R}^d$, we choose F so that it only depends on the difference of target vectors, i.e.,

$$F(w_i - w_j, \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}$$

The function F has a vector input and a scalar output. To further simplify it, we will make the assumption that it actually depends on the dot product of the vectors, i.e.,

$$F\left((w_i - w_j)^T \tilde{w}_k\right) = \frac{P_{ik}}{P_{jk}}$$

$$F\left((w_i - w_j)^T \tilde{w}_k\right) = \frac{\overset{w_i{}^T \tilde{w}_k \text{ relate to (high probability if they are similar)}}{P_{ik}}}{\underset{w_j{}^T \tilde{w}_k}{P_{jk}}}$$

where we just need to compute the difference and the similarity of word embedding for the parameters in *F*. In addition, their relation is symmetrical. (a.k.a. relation(*a, b*) = relation(*b, a*)). To enforce such symmetry, we can have

$$F\left((w_i - w_j)^T \tilde{w}_k\right) = \frac{F(w_i^T \tilde{w}_k)}{F(w_j^T \tilde{w}_k)}$$

Intuitively, we are maintaining the linear relationship among all these embedding vectors. To fulfill this relation, *F*(*x*) would be an exponential function, i.e. *F*(*x*) = exp(*x*). Combine the last two equations, we get

$$F(w_i^T \tilde{w}_k) = P_{ik} = \frac{X_{ik}}{X_i}$$

$$w_i^T \tilde{w}_k = \log(P_{ik}) = \log(X_{ik}) - \log(X_i)$$

$$w_i^T \tilde{w}_k + b_i + \tilde{b}_k = \log(\underbrace{X_{ik}}_{\text{co-occurrence count for word w}i\text{ and w}k})$$

We can absorb log(X_i) as a constant bias term since it is invariant of *k*. But to maintain the symmetrical requirement between *i* and *k*, we will split it into two bias terms above. This *w* and *b* form the embedding matrix. Therefore, the dot product of two embedding matrices predicts the log co-occurrence count.

And then we minimize the loss function:

$$L(w, \tilde{w}) = \sum_{i,k=1}^{V} f(X_{ik}) \left(w_i^T \tilde{w}_k + b_i + \tilde{b}_k - \log X_{ik} \right)^2 ,$$

where V is the vocabulary size and $f(X_{ik})$ is a weighting function chosen carefully to avoid overweighting too rare and too common co-occurrences.

Here is a small example:

```
## [1] "baa baa black sheep have you any wool"
## [2] "yes sir yes sir three bags full"
```

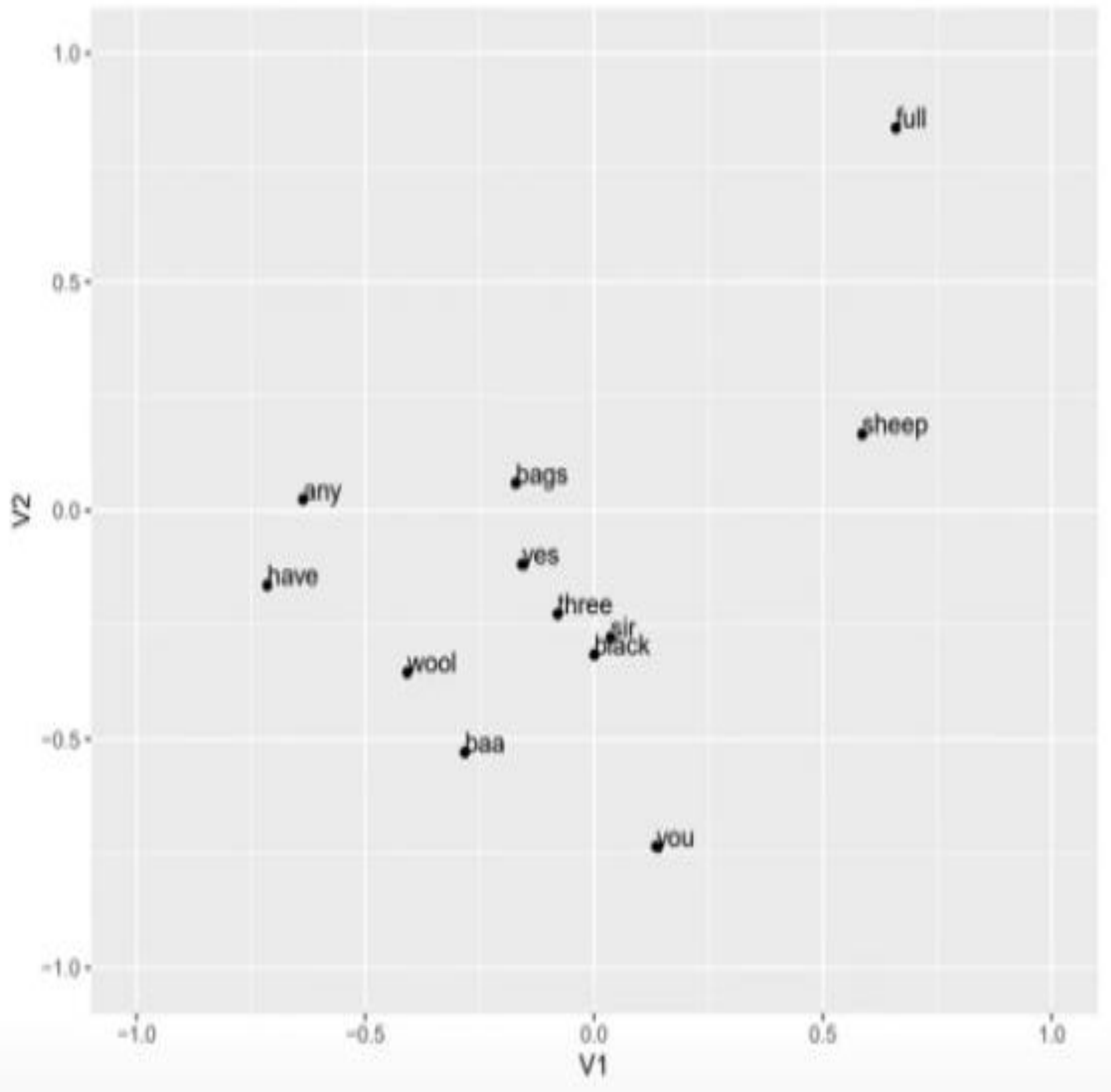
We train the model for 5 epochs with window size 2 and 2-dimensional vectors.

Here are v_i:

```
##                [,1]        [,2]
## any   -0.6361937611  0.02398498
## bags  -0.1713014948  0.05952789
## black  0.0003231433 -0.31565438
## full   0.6596298553  0.83645426
## have  -0.7144603245 -0.16380399
## sheep  0.5859912801  0.16685294
## three -0.0797902889 -0.22663150
## wool  -0.4086090348 -0.35431686
## you    0.1356754443 -0.73474876
## baa   -0.2825158936 -0.52919429
## sir    0.0358659152 -0.27769032
## yes   -0.1576986445 -0.11813807
```

Cosine similarity between "yes" and "sir":

```
## [1] 0.4921018
```



Gradient for GloVe

For simplicity, let $f(X_{ij}) = 1$. We will find the gradient of

$$L(w_1, \ldots, w_V, \tilde{w}_1, \ldots, \tilde{w}_V) = \sum_{i,k=1}^{V} \left(w_i^T \tilde{w}_k + b_i + \tilde{b}_k - \log X_{ik} \right)^2$$

We have found that

$$\frac{\partial L}{\partial w_i} = 2 \sum_{k=1}^{V} \left(w_i^T \tilde{w}_k + b_i + \tilde{b}_k - \log X_{ik} \right) \tilde{w}_k$$

The algorithm goes as follows: we choose the dimensions of the word vectors, the window size (the default in R is 5) that defines "being in the context of another word" and the minimum word frequency to be included into the vocabulary. Then we construct the term co-occurence matrix to find all X_{ij}. Then we find the minimum of the loss function L by some form of gradient descent.

There are technicalities that were important in the success of GloVe, like calculating the term co-occurence matrix effectively and choosing a good weighting function $f(X_{ij})$. But we won't discuss them in our course.

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 $f(X_{ij}) = 1$.
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.

Why Should We Use GloVe?

1. Large Datasets

GloVe works well when you have a **large dataset** because it captures **global relationships** between words by analyzing the entire corpus at once.

Example:

Suppose you are analyzing a dataset like Wikipedia, where words like **“king”**, **“queen”**, and **“royalty”** co-occur frequently. GloVe considers these co-occurrences globally, ensuring the embeddings reflect relationships across the entire text.

2. Semantic Similarity

GloVe captures the **meaning** of words and how they are related based on their co-occurrence. Words with similar contexts will have similar vectors.

Example:

If **“doctor”** and **“nurse”** often appear in similar sentences, GloVe will create embeddings for these words that are close in vector space. This helps in tasks like finding synonyms or clustering similar words.

3.Efficient for Large Datasets

GloVe computes word embeddings based on a **pre-built co-occurrence matrix**. This means the heavy lifting happens once, making it computationally efficient for large datasets compared to methods that process data repeatedly, like Word2Vec.

Example:

When processing billions of words, GloVe requires less computational power because it leverages the precomputed co-occurrence statistics instead of processing the data iteratively like Word2Vec.

4. Great for Specific Problems

GloVe excels in tasks like: **Word Analogies** (finding relationships between words).

Named Entity Recognition (identifying entities like names, locations, etc.).

Example:

For a word analogy task:

GloVe can solve relationships like: “king” — “man” + “woman”≈”queen”

Here, the embeddings capture the underlying relationships and patterns between words.

Why GloVe Stands Out

Global Understanding: Instead of just focusing on local context (like Word2Vec), GloVe takes into account how often words co-occur across the **entire corpus**.

Efficient: It’s great for scenarios with limited computational power but large datasets.

Limitations of GloVe

While GloVe is a powerful word embedding technique, it has its limitations:

1. Static Word Embeddings

GloVe assigns a single embedding to each word, regardless of the context in which the word appears. This leads to ambiguity for words with multiple meanings (polysemy).

Example: The word **“bank”** will have the same embedding whether it’s used in the context of a **riverbank** or a **financial institution**. This limits its ability to handle contextual nuances.

2. Dependence on Precomputed Data

GloVe relies on a precomputed **co-occurrence matrix**, which requires significant memory for very large datasets. This can make it challenging to process extremely large corpora.

For a dataset with millions of unique words, the co-occurrence matrix becomes enormous, requiring significant storage and computational resources.

3. Inability to Handle Out-of-Vocabulary (OOV) Words

GloVe cannot generate embeddings for words not present in the training vocabulary. OOV words will either be ignored or represented by a generic “unknown” vector.

Example: If the word **“bioinformatics”** was not in the training corpus, GloVe won’t be able to generate a meaningful embedding for it, limiting its adaptability to new words.

4. Global Focus Over Local Context

GloVe emphasizes **global word co-occurrence statistics**, which might overlook finer details of local context that methods like Word2Vec (using skip-grams) capture better.

Example: GloVe might miss subtle word pairings that occur infrequently in the corpus but are critical in specific applications, such as technical or domain-specific texts.

5. Lack of Subword Information

Unlike FastText, GloVe does not break words into subword units. This limits its ability to handle morphologically rich languages or rare, compound words.

Example: For the word **“unhappiness”**, GloVe won’t leverage subword components like **“un-”**, **“happy”**, and **“-ness”**, which could provide additional semantic information.

6. Training Time and Resources

Constructing the co-occurrence matrix and training embedding can be computationally expensive for large datasets, especially compared to pre-trained embeddings.

Example:

Training GloVe on a dataset like Wikipedia requires substantial computational power and time, making it less practical for smaller teams or projects.