Tribhuwan University

Institute of Engineering

Pulchowk Campus

A Project Report on:

Nepali Language Processing

Submitted By:

Amrit Baral(076bct006)

Nabin Da Shrestha(076bct037)

Nirajan Bekoju(076bct039)

Nishant Luitel (076bct041)

Submitted To:

Department of Electronics and Computer Engineering

Submission Date:

8^{th} November, 2022

Contents

List of Figures

List of Tables

Chapter 1

Word Embeddings

1.1 Introduction

A very basic definition of a word embedding is a real number, vector representation of a word. Typically, these days, words with similar meaning will have vector representations that are close together in the embedding space (though this hasn't always been the case).

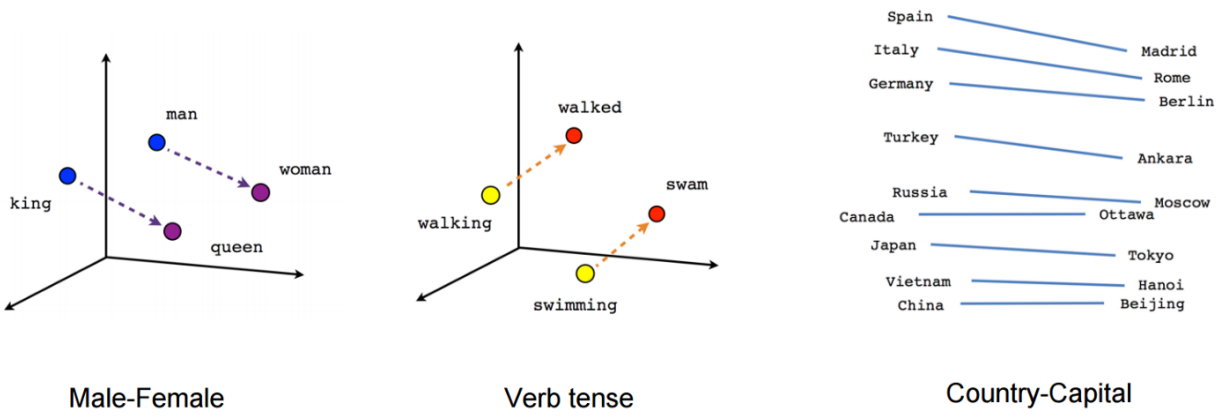

Figure 1.1: Word2Vec

When constructing a word embedding space, typically the goal is to capture some sort of relationship in that space, be it meaning, morphology, context, or some other kind of relationship.

By encoding word embeddings in a densely populated space, we can represent words numerically in a way that captures them in vectors that have tens or hundreds of dimensions instead of millions (like one-hot encoded vectors).

Different word embeddings are created either in different ways or using different text corpora to map this distributional relationship, so the end result are word embeddings that help us on different down-stream tasks in the world of NLP.

1.2 Why Word Embeddings?

Words aren't things that computers naturally understand. By encoding them in a numeric form, we can apply mathematical rules and do matrix operations to them. This makes them amazing in the world of machine learning, especially.

Take deep learning for example. By encoding words in a numerical form, we can take many deep learning architectures and apply them to words. Convolutional neural networks have been applied to NLP tasks using word embeddings and have set the state-of-the-art performance for many tasks.

Moreover, word embeddings can be pre-trained and can be used for variety of NLP applications like text classification, question-answering, auto-complete, spelling correction, speech processing, etc.

1.3 Word Embeddings Methods

1.3.1 One-Hot Encoding(Count Vectorizing)

One of the most basic ways we can numerically represent words is through the one-hot encoding method (also sometimes called count vectorizing).

In one-hot encoding, to represent a word, a vector of dimension equal to the number of unique words in the corpus is created. Each unique word has a unique dimension and will be represented by 1 in that dimension with 0s everywhere else.

Let, **Corpus :** {I have a pen}

Then, the vector representation of words will be

I : [1, 0, 0, 0]

have : [0, 1, 0, 0]

a : [0, 0, 1, 0]

pen : [0, 0, 0, 1]

As in above example, each unique word is assigned a vector with its dimension equal 1 and all other equal 0.

Disadvantage of one-hot encoding is words are represented by huge and sparse vectors that captures no relational information.

1.3.2 Word2Vec

In 2013, with Word2Vec , Mikolov et al. at Google completely changed the embedding paradigm: from then on, embedding will be the weights of a neural network that are adjusted to minimize some loss, depending on the task. Embedding had become a neural network algorithm.

Word2vec is not a singular algorithm, rather, it is a family of model architectures and optimizations that can be used to learn word embeddings from large datasets. Embeddings learned

through word2vec have proven to be successful on a variety of downstream natural language processing tasks.

There are two major learning approaches for Word2Vec.

Continuous Bag-of-Words (CBOW)

It predicts the middle word based on surrounding context words. The context consists of a few words before and after the current (middle) word. This architecture is called a bag-of-words model as the order of words in the context is not important.

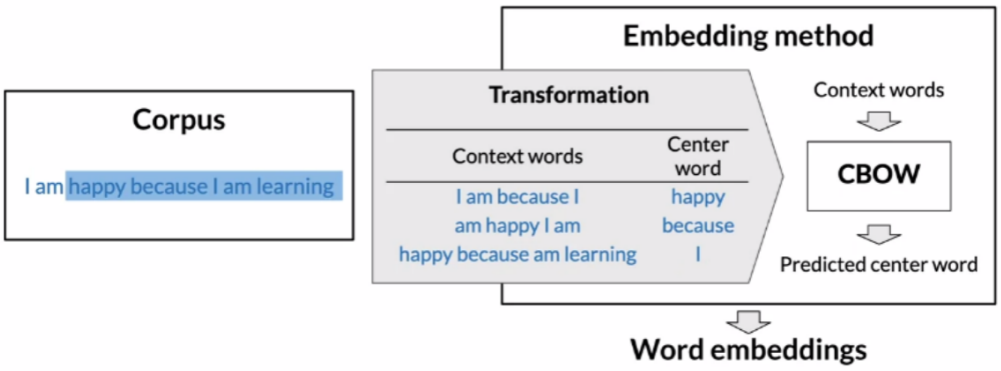

Figure 1.2: CBOW Model

Continuous Skip-Gram

This method learns an embedding by predicting the surrounding words given the context. The context is the current word.

Both of these learning methods use local word usage context (with a defined window of neighboring words). The larger the window is, the more topical similarities that are learned by the embedding. Forcing a smaller window results in more semantic, syntactic, and functional similarities to be learned.

Using Word2Vec model, high quality embeddings can be learned pretty efficiently, especially when comparing against neural probabilistic models. That means low space and low time complexity to generate a rich representation.

More than that, the larger the dimensionality, the more features we can have in our representation. But still, we can keep the dimensionality a lot lower than some other methods. It also allows us to efficiently generate something like a billion word corpora, but encompass a bunch of generalities and keep the dimensionality small.

1.3.3 GloVe

GloVe is an extension of word2vec, and a much better one at that. There are a set of classical vector models used for natural language processing that are good at capturing global statistics of a corpus, like LSA (matrix factorization). They're very good at global information, but they don't capture meanings so well and definitely don't have the cool analogy features built in.

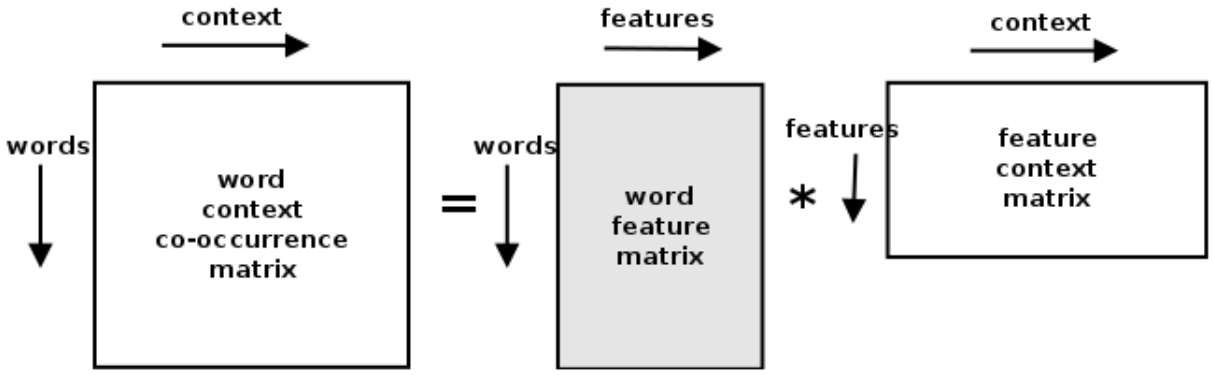

Figure 1.3: Glove Model

GloVe's contribution was the addition of global statistics in the language modeling task to generate the embedding. There is no window feature for local context. Instead, there is a word-context/word co-occurrence matrix that learns statistics across the entire corpora.

1.3.4 Others

Some other word embedding methods are fastText(Facebook, 2016), BERT (Google, 2018), ELMo(Allen Institute for AI, 2018) and GPT-2(OpenAI, 2018).

1.4 Feature Vectors and Similarity Scores

Word vectors represent words in a way that encodes their meaning. A vector is simply an array of fractions. Here's a vector taken from the Google News model. It's the vector for the word "fast". It consists of 300 floating point values: [0.0575, -0.0049, 0.0474, ..., -0.0439]

The straight line distance between two points is called the "Euclidean" or "L2" distance, and it can be extended to any number of dimensions. Here's the formula for the distance between two vectors a and b with an arbitrary number of dimensions (n dimensions):

$$dist_{L2}(a,b) = \sqrt{\sum_{i=0}^{n}(a_i - b_i)^2} \tag{1.1}$$

Here's the important insight, word2vec learns a vector for each word in a vocabulary such that words with similar meanings are close together and words with different meanings are farther apart. That is, the Euclidean distance between a pair of word vectors becomes a measure of how dissimilar they are.

The Euclidean distance is what we intuitively understand as distance, and it's a workable distance metric for comparing word vectors, but in practice another metric called the Cosine similarity gives better results.

$$Similarity(A, B) = \frac{A.B}{||A|| * ||B||} = \frac{\sum_{i=1}^{n} A_i * B_i}{\sum_{i=1}^{n} A_i^2 * \sum_{i=1}^{n} B_i^2} \tag{1.2}$$

The Cosine similarity between two vectors a and b is found by calculating their dot product, and dividing this by their magnitudes. The cosine similarity is always a value between -1.0 and 1.0.

1.5 Data Cleaning and Tokenization

Following details should be followed for proper word embedding while data pre-processing and tokenization.

1. **Letter case :** In enligsh language, the letter case (eg: 'Eagle'or 'eagle') doesn't change the meaning of the words. Hence, it is better to have all data in lower case for word embedding.
2. **Punctuation :** Punctuation symbols like ", ', !, ?, etc. should be handled before word embeddings.
3. **Numbers :** Numbers (0, 1, 2, 3 ....) should be removed before word embeddings if they are not important for semantic meaning.
4. **Special characters :** Special characters like $\delta, \Delta, \alpha, \beta$, etc. should be handled before word embeddings.
5. **Special words :** Special words like '#nlp', emojis, etc. should be cleaned.

1.6 Evaluation of Word Embeddings Model

1.6.1 Extrinsic Evaluation

In extrinsic evaluation, word embeddings are tested on external tasks like named entity recognition, parts-of-speech tagging, etc. It evaluates the actual usefulness of embeddings. However, it is time consuming and more difficult to troubleshoot.

1.6.2 Intrinsic Evaluation

Similarity and analogy test are used for intrinsic evaluation.

Chapter 2

Probabilistic Language Model

2.1 Introduction

Language model is used to estimate the probability of the word sequences and to estimate the probability of a word following the sequence of words. The concept can be used to auto-complete a sentence with most likely suggestions as shown below:

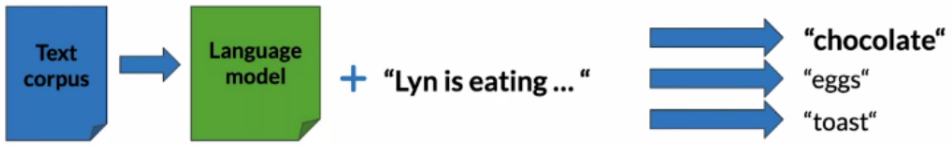

Figure 2.1: Language Model for Auto-complete

Some other applications of the language model are:

1. Speech Recognition
2. Spelling Correction
3. Augmentative Communication

2.2 N-gram

A N-gram is a sequence of N-words. For example:

Corpus : I am happy because I am learning.

Uni-grams : {I, am, happy, because, I, am, learning }

Bi-grams : {I am, am happy, happy because, . . . }

Tri-grams : {I am happy, am happy because, . . . }

2.3 Sequence Notation

Let's define a standard sequence notation to represent the sequence of words in our corpus. Let our corpus be as shown below where n^{th} is represented as W_n. Let the number of words in the corpus be m = 500

Corpus: This is great. ... teacher drinks tea.

Hence, W_1 = This, W_2 = is, W_3 = great and so on...

Let us represent the sequence of words as W_s^e where s represent the starting index and e represent the ending index of the words in the corpus. So,

$W_1^m = W_1\ W_2\ W_3$ ..... $W_{m-1}\ W_m$

Similarly, $W_1^3 = W_1\ W_2\ W_3$ = This is great

2.4 N-gram Probability

N-gram probability is the probability of the occurrence of the sequence of N-words in a corpus. From uni-gram, bi-gram and tri-gram probability, we can deduce for N-gram probability denoted by $P(W_N|W_1^{N-1})$ as

$$P(W_N|W_1^{N-1}) = \frac{C(W_1^{N-1}\,W_N)}{C(W_1^{N-1})} = \frac{C(W_1^N)}{C(W_1^{N-1})} \tag{2.1}$$

2.4.1 Uni-gram Probability

Uni-gram Probability is the probability of occurrence of a word in a given corpus.

Corpus : I am happy because I am learning.

Size of corpus : m = 7

$$P(I) = \frac{2}{7} \tag{2.2}$$

$$P(happy) = \frac{1}{7} \tag{2.3}$$

Hence, we can see that Probability of uni-gram is given by:

$$P(w) = \frac{C(w)}{m} \tag{2.4}$$

where C(w) = frequency of the word in the corpus

2.4.2 Bi-gram Probability

Bi-gram Probability is the probability of occurrence of sequence of two words in a corpus.
Corpus : I am happy because I am learning.
Size of corpus : m = 7

$$P(am|I) = \frac{C(I\,am)}{C(I)} = \frac{2}{2} = 1 \tag{2.5}$$

$$P(happy|I) = \frac{C(I\,happy)}{C(I)} = \frac{0}{7} = 0 \tag{2.6}$$

$$P(learning|am) = \frac{C(am\,learning)}{C(am)} = 1/2 \tag{2.7}$$

Hence, we can see that Probability of bi-gram is given by:

$$P(y|x) = \frac{C(x\,y)}{C(x)} \tag{2.8}$$

where C(w) = frequency of the word in the corpus

2.4.3 Tri-gram Probability

Tri-gram Probability is the probability of the occurrence of the sequence of three words in a corpus.
Corpus : I am happy because I am learning.
Size of corpus : m = 7

$$P(happy|I\,am) = \frac{C(I\,am\,happy)}{C(I\,am)} = \frac{1}{2} \tag{2.9}$$

Hence, we can see that Probability of bi-gram is given by:

$$P(W_3|W_1^2) = \frac{C(W_1^2\,W_3)}{C(W_1^2)} = \frac{C(W_1^3)}{C(W_1^2)} \tag{2.10}$$

where C(w) = frequency of the word in the corpus

2.5 Probability of a sequence

From the conditional probability and chain rule, we have

$$P(B|A) = \frac{P(A,B)}{P(A)} \tag{2.11}$$

From (2.11), we get

$$P(A,B) = P(A).P(B|A) \tag{2.12}$$

Using (2.12), we can deduce

$$P(A,B,C,D) = P(A).P(B|A).P(C|A,B).P(D|A,B,C) \tag{2.13}$$

Now, the probability of the sequence of words: "the teacher drinks tea" is given by:

$$\begin{aligned} &P(the\,teacher\,drinks\,tea) = P(the).P(teacher|the) \\ &P(drinks|the\,teacher).P(tea|the\,teacher\,drinks) \end{aligned} \tag{2.14}$$

2.6 Approximation of Sequence Probability

When using n-gram language model, the exact sentence for the required n-gram might not be present exactly in corpus due to which the frequency of the words sequence becomes zero and hence the probability. For example:

Input : the teacher drinks tea

P(the teacher drinks tea) = P(the) . P(teacher | the) P(drinks | the teacher) . P(tea | the teacher drinks)

Here,

$$P(tea|the\,teacher\,drinks) = \frac{C(the\,teacher\,drinks\,tea)}{C(the\,teacher\,drinks)} \tag{2.15}$$

In the above equation (2.15), the frequency of "the teacher drinks tea" and "the teacher drinks" are both likely 0.

Hence, in order to calculate the required probability of the sequence of the words, as per Markov assumption, only last N words matter. Suppose, only last one words matter then, we have

P(the teacher drinks tea)
= P(the) . P(teacher | the) P(drinks | the teacher) . P(tea | the teacher drinks)
≈ P(the) . P(teacher | the) P(drinks | teacher) . P(tea | drinks)

Hence, we can model entire sentence with n-gram following Markov Assumption as follow

$$P(W_n|W_1^{n-1}) \approx P(W_n|W_{n-N+1}^{n-1}) \tag{2.16}$$

2.7 Starting and Ending Sentences

2.7.1 Start of sentence symbol $\langle s \rangle$

For an input sentence, "the teacher drinks tea", we have

P(the teacher drinks tea) ≈ P(the) . P(teacher | the) P(drinks | teacher) . P(tea | drinks)

After, we add a start token $\langle s \rangle$, we have

Input : $\langle s \rangle$ the teacher drinks tea

P(the teacher drinks tea) ≈ P(the | $\langle s \rangle$) . P(teacher | the) P(drinks | teacher) . P(tea | drinks)

In this way, we can add start token to determine the possible starting words and calculate the probability of sequence considering first word in N-gram language model. For Tri-gram and N-gram, we add (N - 1) start token $\langle s \rangle$ in the beginning of the sequence.

2.7.2 End of sentence symbol $\langle /s \rangle$

For an input sentence, "the teacher drinks tea", we have

P(the teacher drinks tea) ≈ P(the) . P(teacher | the) P(drinks | teacher) . P(tea | drinks)

After, we add a start token $\langle s \rangle$ and an end token $\langle /s \rangle$, we have

Input : $\langle s \rangle$ the teacher drinks tea $\langle /s \rangle$

P(the teacher drinks tea) $\approx$ P(the | $\langle s \rangle$) . P(teacher | the) P(drinks | teacher) . P(tea | drinks) . P ($\langle /s \rangle$ | tea)

For bi-gram, tri-gram or any N-gram model, we add end token just once at the end of sentence unlike start token

2.8 Count Matrix

Count matrix is the matrix containing the frequency of occurrence of N-gram. Rows in count matrix represent the unique corpus of (N - 1) gram and columns represent the unique corpus word.

Corpus : $\langle s \rangle$ I study I learn $\langle /s \rangle$

Bi-gram Count Matrix is as follow:

	$\langle s \rangle$	$\langle /s \rangle$	I	study	learn
$\langle s \rangle$	0	0	1	0	0
$\langle /s \rangle$	0	0	0	0	0
I	0	0	0	1	1
study	0	0	1	0	0
learn	0	1	0	0	0

Table 2.1: Count Matrix for bi-gram model

2.9 Probability Matrix

Probability matrix is the matrix containing the probability of the occurrence of N-gram words sequence. It can be created by dividing each elements of the count matrix by total sum of all row elements. For example:

Corpus : $\langle s \rangle$ I study I learn $\langle /s \rangle$

	$\langle s \rangle$	$\langle /s \rangle$	I	study	learn	sum
$\langle s \rangle$	0	0	1	0	0	1
$\langle /s \rangle$	0	0	0	0	0	0
I	0	0	0	1	1	2
study	0	0	1	0	0	1
learn	0	1	0	0	0	1

Table 2.2: Count Matrix for bi-gram model with row-sum

	$\langle s \rangle$	$\langle /s \rangle$	I	study	learn
$\langle s \rangle$	0	0	1	0	0
$\langle /s \rangle$	0	0	0	0	0
I	0	0	0	0.5	0.5
study	0	0	1	0	0
learn	0	1	0	0	0

Table 2.3: Probability Matrix for bi-gram model

From the above probability matrix, we can calculate the probability of sentence and also predict the next word based on higher probability.

For example :

Input : $\langle s \rangle$ I learn $\langle /s \rangle$

P(Input) = P($\langle s \rangle$ I learn $\langle /s \rangle$)

= P(I | $\langle s \rangle$) . P(learn | I). P ($\langle /s \rangle$ | learn)

= $1 \times 0.5 \times 1$

= 0.5

2.10 Generative Language Model

Generative language model is used to generate a most likely sentences.

Algorithm for the generative language model is:

Step 1 : Choose sentence start

Step 2 : Choose next bi-gram starting with the previous word

Step 3 : Continue until $\langle /s \rangle$ is picked

For example:

Assume our corpus be:

$\langle s \rangle$ Lyn drinks chocolate $\langle /s \rangle$

$\langle s \rangle$ John drinks tea $\langle /s \rangle$

$\langle s \rangle$ Lyn eats chocolate $\langle /s \rangle$

Then, the generative language model can generate a sentence as follow:

Step 1: ($\langle s \rangle$, Lyn) or ($\langle s \rangle$, John)

Step 2: (Lyn, eats) or (Lyn drinks)

Step 3: (drinks, tea) or (drinks, chocolate)

Step 4: (tea, $\langle /s \rangle$)

2.11 Train, Validation and Test Split

The recommended split of corpus into train, validation and test data is as follow

Type of corpus	Train	Validation	Test
Small	80%	10%	10%
Large	98%	1%	1%

Table 2.4: Train, Validation and Test Split

2.12 Language Model Evaluation

We can use two different approaches to evaluate and compare language models:

2.12.1 Extrinsic evaluation

This involves evaluating the models by employing them in an actual task (such as machine translation) and looking at their final loss/accuracy. This is the best option as it's the only way to tangibly see how different models affect the task we're interested in. However, it can be computationally expensive and slow as it requires training a full system.

2.12.2 Intrinsic evaluation

This involves finding some metric to evaluate the language model itself, not taking into account the specific tasks it's going to be used for. While intrinsic evaluation is not as "good" as extrinsic evaluation as a final metric, it's a useful way of quickly comparing models. Perplexity is an intrinsic evaluation method.

2.12.3 Perplexity

Perplexity is an evaluation metric for language models. For better language model, perplexity value should be smaller. Perplexity can be calculated as follow:

$$PP(W) = P(s_1, s_2, s_3, s_4, ..., s_m)^{\frac{-1}{m}} \tag{2.17}$$

where,

$\mathbf{W}$ = test set containing m sentences s

$S_i = i^{th}$ sentence in the test set, each ending with $\langle /s \rangle$

$\mathbf{m}$ = number of all words in entire test set $\mathbf{W}$ including $\langle /s \rangle$ but not including $\langle s \rangle$

Similarly, Perplexity for the bi-gram model can be calculated as:

$$PP(W) = \sqrt[m]{\prod_{i=1}^{m} \prod_{j=1}^{|S_i|} \frac{1}{P(w_j^{(i)} | w_{j-1}^{(i)})}} \tag{2.18}$$

where, $w_j^{(i)} = j^{th}$ word in i^{th} sentence

If we concatenate all the sentences in W, we can get perplexity as

$$PP(W) = \sqrt[m]{\prod_{i=1}^{m} \frac{1}{P(w_i|w_{i-1})}} \tag{2.19}$$

where, $w_i = i^{th}$ word in test set

2.12.4 Log Perplexity

Log perplexity is calculated by taking logarithm on both sides of equation (2.19) and we get

$$logPP(W) = \frac{-1}{m} \sum_{i=1}^{m} log_2(P(w_i|w_{i-1})) \tag{2.20}$$

2.13 Vocabulary

Vocabulary is a set of words from the training corpus. We can create vocabulary from training corpus using following criteria.

1. Min word frequency f
2. Max $|V|$, include words by frequency
3. Using $\langle UNK \rangle$ for unknown token

For example: Suppose our corpus is as follow

$\langle s \rangle$ Lyn drinks chocolate $\langle /s \rangle$

$\langle s \rangle$ John drinks tea $\langle /s \rangle$

$\langle s \rangle$ Lyn eats chocolate $\langle /s \rangle$

let min frequency to be added in vocabulary be f = 2. So, now corpus will be as follow:

$\langle s \rangle$ Lyn drinks chocolate $\langle /s \rangle$

$\langle s \rangle$ $\langle UNK \rangle$ drinks $\langle UNK \rangle$ $\langle /s \rangle$

$\langle s \rangle$ Lyn $\langle UNK \rangle$ chocolate $\langle /s \rangle$

Hence, vocabulary = $\{Lyn, drinks, chocolate\}$

Suppose,

Input Query : $\langle s \rangle$ Adam drinks chocolate $\langle /s \rangle$

So, the input query after processing becomes

Ouput : $\langle s \rangle$ $\langle UNK \rangle$ drinks chocolate $\langle /s \rangle$

In this way, vocabulary can be created and unknown words can be handled.

2.14 Missing N-gram in training corpus

N-grams made of known words still might be missing in the training corpus. N-gram probability is given as:

$$P(w_n|w_{n-N+1}^{n-1}) = \frac{C(w_{n-N+1}^{n-1}, w_n)}{C(w_{n-N+1}^{n-1})} \tag{2.21}$$

In equation (2.21), both $C(w_{n-N+1}^{n-1}, w_n)$ and $C(w_{n-N+1}^{n-1})$ can be 0 in the probability matrix which causes the probability to be indefinite. It can be solved by using smoothing.

2.14.1 Laplacian Smoothing

In Laplacian smoothing, probability is calculated as follow:

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}, w_n) + 1}{C(w_{n-1}) + V} \tag{2.22}$$

2.14.2 K-smoothing

In K-smoothing, probability is calculated as follow

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}, w_n) + K}{C(w_{n-1}) + k * V} \tag{2.23}$$