INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD

Introduction to NLP (CS7.401)

Assignment 3

Submitted by:

Bhanuj Gandhi

2022201068

March 26, 2023

Contents

1 Part 1: Theory **1**

2 Part 2: Implementation **3**

3 Part 3: Analysis **6**

3.1 Display the top-10 word 6

3.2 Comparison of words beteen SVD and CBOW 11

3.3 Comparison with pre-trained word2vec 12

1 Part 1: Theory

`Q1.) Explain negative sampling. How do we approximate the word2vec training computation using this technique?`

Negative sampling is a technique used in training word embeddings using the word2vec model. The main idea behind negative sampling is to approximate the softmax function, which is computationally expensive when the size of the vocabulary is large. That means when training word2vec model, we consider only context words to be true words, rest all words are considered to be false words. Now

In the word2vec model, given a context word, the task is to predict the target word that appears in the context. This is done by training a neural network to maximize the probability of observing the target word given the context words. The probability is calculated using the softmax function over all words in the vocabulary, which requires computing the exponentials of a large number of scores. This becomes computationally expensive when the vocabulary size is large.

Negative sampling provides a way to approximate the softmax function. Instead of computing the probability of observing the target word for all words in the vocabulary, we sample a small number of "negative" words from the vocabulary and compute the probability of not observing these words as the negative examples. This reduces the computational cost significantly, as we only need to compute the probabilities for a small number of words.

The negative sampling technique works by training the network to distinguish between the true target word and randomly sampled negative words. Specifically, given a context word, we sample a few negative words from the vocabulary and train the network to predict that these words are not the target word. This is done by minimizing the loss function, which is a binary cross-entropy loss between the predicted probabilities and the true labels. The true label for the target word is 1, and for the negative samples, it is 0. By doing this, the network learns to distinguish between the true target word and the negative samples, and the resulting word embeddings capture the semantic and syntactic relationships between the words in the corpus.

In summary, negative sampling is a technique used in training word embeddings that approximates the softmax function, which is computationally expensive when the vocabulary size is large. The technique works by sampling a small number of negative words from the vocabulary and training the network to distinguish between the true target word and the negative samples, using a binary cross-entropy loss function. The resulting word embeddings capture the semantic and syntactic relationships between the words in the corpus.

Q2.) Explain the concept of semantic similarity and how it is measured using word embedding. Describe at least two techniques for measuring semantic similarity using word embedding.

Semantic similarity is a metric used to evaluate how closely related the meanings of two phrases or words are. In Natural Language Processing (NLP), it is essential to measure the level of semantic similarity between words or phrases to identify synonyms or compare the similarity of meanings in different contexts. To represent words in NLP, word embeddings are commonly used, which capture the distributional features of words in a given corpus of text. These embeddings portray words as dense vectors in a high-dimensional space where words with similar meanings lie closer to each other.

- **Cosine Similarity**: To measure semantic similarity using word embeddings, a common approach is to compute the cosine similarity between the vectors representing the two words. The cosine similarity measures the cosine of the angle between two vectors and ranges from -1 to 1, where a score of 1 means that the vectors are identical, 0 means that the vectors are unrelated, and -1 suggests that they have opposite meanings. To determine the cosine similarity between two words, their word embeddings are first acquired, and then the cosine of the angle between the two embeddings is calculated, which yields a value between -1 and 1 that indicates the degree of semantic similarity between the two words. Higher scores indicate greater similarity, while lower scores suggest less similarity.
- **Euclidean distance**: Another method for measuring semantic similarity using word embeddings is to compute the Euclidean distance between the vectors representing two words. The Euclidean distance is simply the straight-line distance between two points in a multidimensional space, which in this case is the space defined by the word embeddings. If two words have similar meanings, they will have embeddings that are close to each other in this space, and their Euclidean distance will be small. Conversely, if two words have very different meanings, their embeddings will be far apart in space, and their Euclidean distance will be large.

2 Part 2: Implementation

Q1.) Implement a word embedding model and train word vectors by first building a Co-occurrence Matrix followed by the application of SVD.

In this method I have created **Co-Occurrence Matrix with a fixed context window**. Idea is similar words tend to occur together and will have a similar context for example — Apple is a fruit. Mango is a fruit.

Co-occurrence: For a given corpus, the co-occurrence of a pair of words say w1 and w2 is the number of times they have appeared together in a Context Window.

Context Window: Context window is specified by a number and the direction.

Implementation of Co-occurrence matrix

```
def build_co_occ(sentence):
        sent = tokenizer(sentence)
        for idx, word in enumerate(sent):
            for context_id in range((max(0, idx - WINDOW_LENGTH)), (min(len(sent
    ), idx + WINDOW_LENGTH + 1))):
                row = vocab[word]
                col = vocab[sent[context_id]]
                co_occ_matrix[row][col] += 1
```

There is one problem with this approach, for a huge corpus, this co-occurrence matrix could become really (high-dimension). For eg: if `vocabulary_size` is 50000, then we need to store 50000x50000 matrix in the memory, which is *sparse* as we are taking only a certain length context window size.

The solution to this problem is Singular value decomposition(SVD) and principal component analysis(PCA) are two eigenvalue methods used to reduce a high-dimensional dataset into fewer dimensions while retaining important information.

For this assignment, I have experimented both the methods, for SVD I have used **Truncated SVD** which is faster but still took almost **15 minutes** to build on a `vocabulary_size = 27000` for `100 dimension` embedding representation.

```
    U, s, VT = svds(co_occ_np, k=300)
    # U is the SVD matrix
```

Similary, for PCA, I have used **Incremental PCA** which takes in the batch size, I used 1000 batch size for 300 dimension embedding. It took `5 minutes` to create the matrix.

```
    incr_pca = IncrementalPCA(n_components=300, batch_size=1000)
    batches = np.array_split(np.nan_to_num(co_occ_matrix), int(len(co_occ_matrix
    ) / 1000))

    for batch in tqdm(batches):
        incr_pca.partial_fit(batch)

    U = incr_pca.components_.T
```

I found that results were better when I was using `IncrementalPCA`, one reason could be using

PCA, I was able to get 300 dimension embeddings, which are able to retain more relation than 100 dimension using SVD.

Q2.) Implement the word2vec model and train word vectors using the CBOW model with Negative Sampling.

CBOW (continuous bag-of-words) uses the context or surrounding words as input. For instance, if the context window C is set to C=5, then the input would be words at positions w(t-2), w(t-1), w(t+1), and w(t+2). Basically the two words before and after the center word w(t). Given this information, CBOW then tries to predict the target word.

For this assignment, I have experimented **Vanilla CBOW** as well as **CBOW with Negative Sampling** which is computationally better as explained above.

Below is the model implementation of **Vanilla CBOW**. We just need the ouput from embedding layer, after the model is converged and the loss is minimized.

```
class CBOW(nn.Module):
    def __init__(self, vocab, embedding_dim, max_norm):
        super(CBOW, self).__init__()
        self.embeddings = nn.Embedding(vocab, embedding_dim, max_norm)
        self.linear = nn.Linear(embedding_dim, vocab)

    def forward(self, input):
        x = self.embeddings(input)
        x = x.mean(axis=1)
        x = self.linear(x)
        return x
```

Below is the implementation of **CBOW with Negative Sampling**, here I have randomly created negative words, so that model does not calculate loss for all the words in the vocabulary and back propagate for them. My trainloader give Input as context words, target words as focused word + negative words, and labels has 1's and 0's for the positive and negative words respectively.

```
class CBOW_1(nn.Module):
    def __init__(self, vocab_size, embedding_dim, device):
        super(CBOW, self).__init__()
        self.device = device
        self.context_embeddings = nn.Embedding(vocab_size, embedding_dim)
        self.neg_embeddings = nn.Embedding(vocab_size, embedding_dim)

    def forward(self, input, focus_word, weight_mask, labels):
        temp_src_embedding = [
            self.context_embeddings(torch.tensor(src_word, dtype=torch.long).to(
    self.device)).sum(dim=0)
            for src_word in input
        ]
        src_embeds = torch.stack(temp_src_embedding)
        target_embeds = self.neg_embeddings(torch.tensor(focus_word, dtype=torch
    .long).to(self.device))
        weight_mask = torch.tensor(weight_mask, dtype=torch.float).to(self.
    device)
        labels = torch.tensor(labels, dtype=torch.float).to(self.device)

        pred = torch.bmm(src_embeds.unsqueeze(1), target_embeds.permute(0, 2, 1)
    ).squeeze(1)

        loss = nn.functional.binary_cross_entropy_with_logits(
            pred.float(), labels, reduction="none", weight=weight_mask
```

```
        )
        loss = (loss.mean(dim=1) * weight_mask.shape[1] / weight_mask.sum(dim=1)
    ).mean()

        return loss
```

3 Part 3: Analysis

3.1 Display the top-10 word

Q1.) Display the top-10 word vectors for five different words (a combination of nouns, verbs, adjectives, etc.) using t-SNE (or such methods) on a 2D plot.

Below five images are top-10 words generated using Co-occurrence matrix method. We can notice that in each of them 8 words are close to each other while 2 are away, the reason behind it is I have used window size of 8, so my context for each word is previous 4 words and next 4 words.

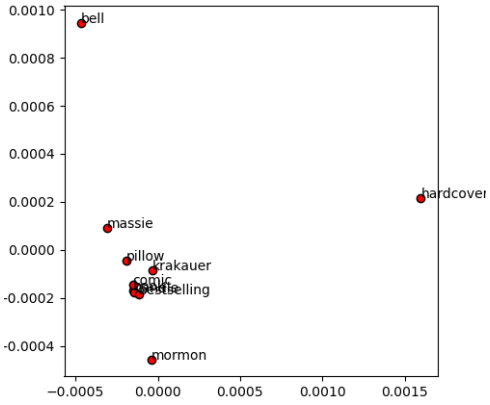

Figure 1: Word: book

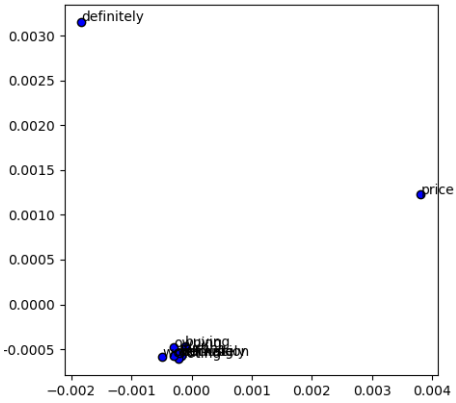

Figure 2: Word: definitely

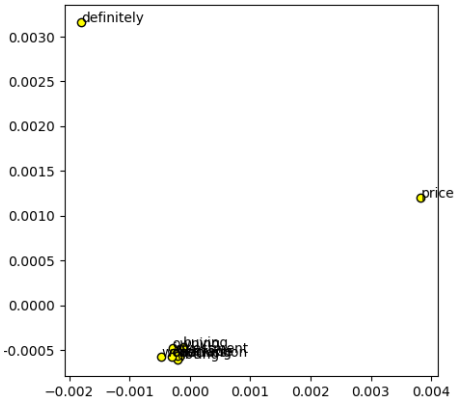

Figure 3: Word: worth

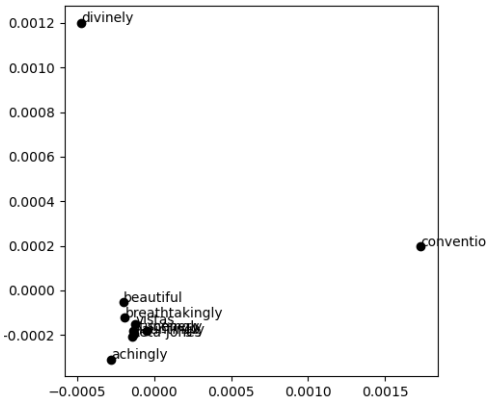

Figure 4: Word: beautiful

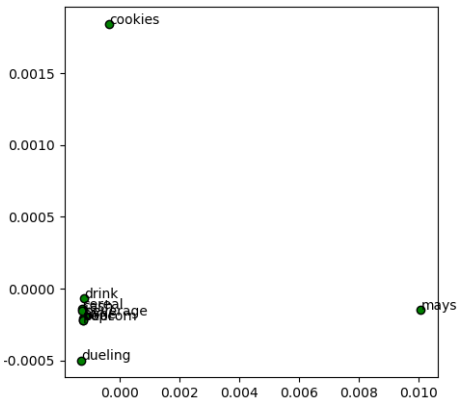

Figure 5: Word: drink

Below are the five images plot for same word using CBOW method and negative sampling. I have used the same words as I have used in the co-occurrence matrix method.

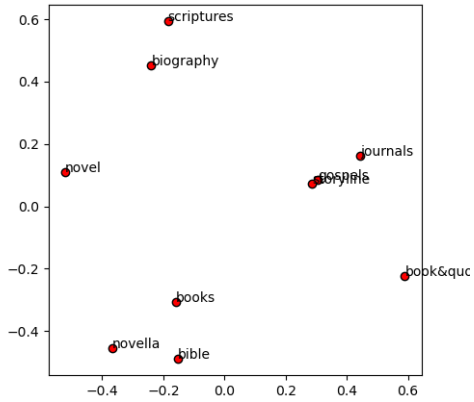

Figure 6: Word: book

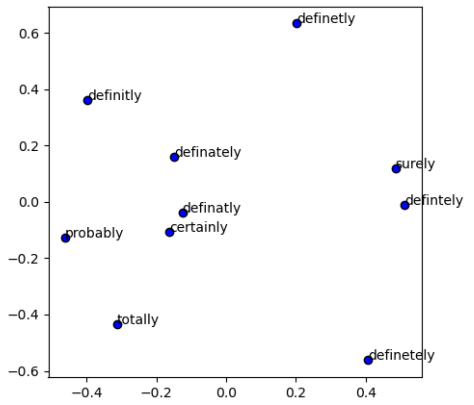

Figure 7: Word: definitely

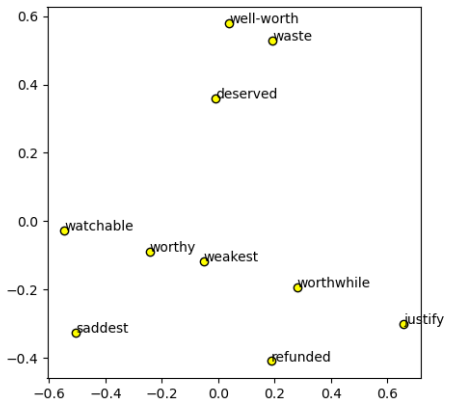

Figure 8: Word: worth

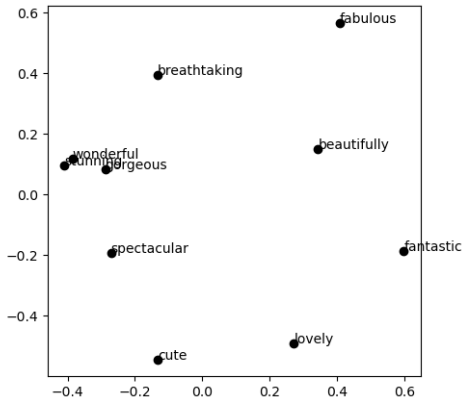

Figure 9: Word: beautiful

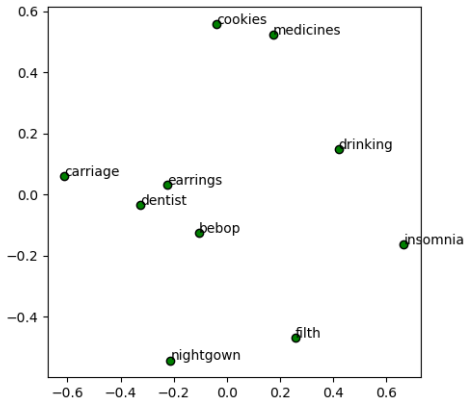

Figure 10: Word: drink

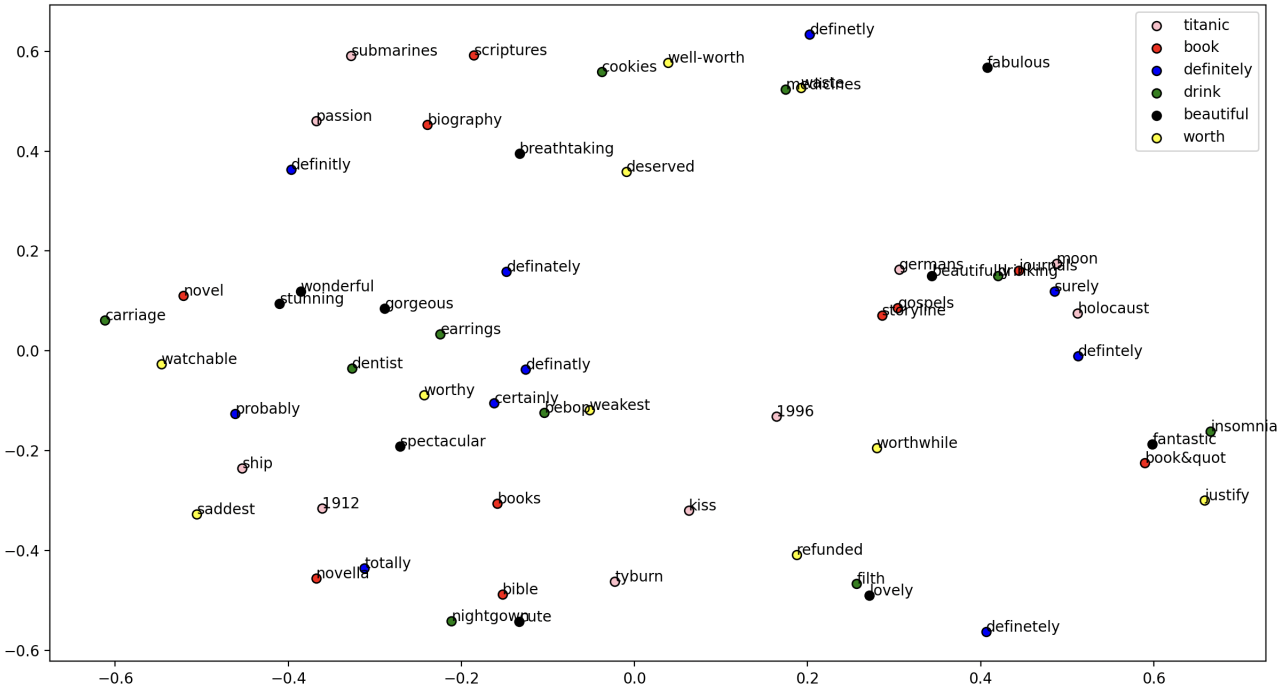

Figure 11: All 5 words using CBOW

3.2 Comparison of words beteen SVD and CBOW

book	
SVD	**CBOW**
book	novel
comic	books
pillow	bible
candle	gospels
hardcover	journals
bell	storyline
krakauer	novella
mormon	bookquot
massie	scriptures
bestselling	biography

definitely	
SVD	**CBOW**
definitely	definately
worth	certainly
owning	definetly
admission	probably
definately	definetely
noting	surely
buying	definitly
checking	totally
renting	defintely
price	definatly

drink	
SVD	**CBOW**
drink	drinking
beverage	cookies
cereal	medicines
cookies	earrings
mine	filth
popcorn	insomnia
beer	dentist
dueling	nightgown
mays	carriage
cash	bebop

beautiful	
SVD	**CBOW**
beautiful	gorgeous
stunningly	lovely
hauntingly	wonderful
breathtakingly	stunning
scenery	breathtaking
achingly	fabulous
zeta-jones	fantastic
conventionally	beautifully
vistas	spectacular
divinely	cute

worth	
SVD	**CBOW**
worth	worthwhile
owning	well-worth
admission	worthy
noting	refunded
investment	waste
price	justify
buying	weakest
definitely	deserved
upgrade	watchable
checking	saddest

Difference between Similar words between SVD and CBOW

3.3 Comparison with pre-trained word2vec

Q2.) What are the top 10 closest words for the word 'titanic' in the embeddings generated by your program? Compare them against the pre-trained word2vec embeddings that you can download off-the-shelf.

titanic using Co-occurrence Matrix

titanic
since
1912
1999
public
bulldog
midsummer
citizen
1990s
fever

titanic using CBOW

ship
holocaust
moon
tyburn
1996
germans
1912
submarines
passion
kiss

titanic using GLoVe

sinking
dicaprio
rms
voyage
sunk
epic
starship
winslet
r.m.s.
iceberg

These are the most related word to `titanic` as GLoVe embeddings are trained on billion of data. I have used wikipedia text, so it has show actors name as well. To train on this much data a lot of compute is required. Due to lack of compute, our results are not as accurate as this but still comparable.