Natural Language Processing CA#2

Student Name:
Pouya Haji Mohammadi Gohari

SID:810102113

Date of deadline
Tuesday 2nd April, 2024

Dept. of Computer Engineering

University of Tehran

Contents

1 Question 1

1.1 Dataset

We will start by loading our dataset using Python 'pandas' library.The code below will accomplishes what we want:

```
    path = r'Data\training.1600000.processed.noemoticon.csv'
    df = pd.read_csv(path, encoding='latin1')
    df.head()
```

Our dataset comprises 1.6 million tweets, with each tweet containing multiple feature columns. The features columns are not defined by these dataset.Thus we add header row to our dataframe.

```
    columns = ['target', 'ids', 'date', 'flag', 'user', 'text']
    path = r'Data\training.1600000.processed.noemoticon.csv'
    df = pd.read_csv(path, names=columns, header=None, encoding='latin1')
    df.head()
```

The header names were obtained from the dataset explanination by sentiment140 where the features descriptions are:

- 'target': The polarity of tweet($0 =$ negative, $4 =$ positive)
- 'ids': The id of the tweet
- 'data': The date of the tweet
- 'flag': The query
- 'user': The user that tweeted
- 'text': The text of the tweet

1.2 Preprocessing and resampling

For getting sample from each polarity target we consider 'groupby' method in 'pandas' library.
The code:

```
    sampled_df = df.groupby('target').apply(lambda x: x.sample(5000, random_state
       ↪ =42))
    sampled_df = sampled_df.reset_index(drop=True)
    sampled_df.head()
```

This piece of code simply group by 'target' and apply a function that getting sample from each unique 'target' feature.To make sure sampling is correct the following code will assure this:

```
    print(len(sampled_df[sampled_df['target'] == 0]))
    print(len(sampled_df[sampled_df['target'] == 4]))
```

Where each classes have 5000 samples based on the table 5.

Positive Polarity	Negative Polarity
5000	5000

Table 1: length of each class's sample

Every NLP[1] tasks needs preprocess phase. Specially when we are dealing with sentimental analysis.In this question we use some common preprocess methods in sentimental analysis. In each method, we consider explaining why each of them will be useful.

- Lower Case:
  - Lowercasing make sure that the same word in different cases like 'HAPPY', 'happy' is recognized as same word.But to be honest these is not good for sentimenal analysis since 'GOOD' or 'good' has a very different meanings in sentiment.However this method has this disadvantage, it has also more advantages for us where will be discussed later.
  - One of the most advantages of lowercasing is reducing vocabulary size.By converting all letters to lowercase, the total number of unique words in the dataset decreases.
  - This method will normalize our data also.Furthure more, this mehod invloves standardizing textula data so that variants of the same word are treated as identical.
- Remove user mention: Consider a situtaion where I mentioned by a random user like $@ < user >$.Since this mention has lack of sentiment and has nothing to do with sentimental analysis, we consider to remove them.
- Remove URLs: Like mentioning user, URLs has no concept in sentimental analysis.
- Remove Emails
- Replace Emoticons: Deleting emoticons from dataset is very naive thing to do since emoticons has signifacnt meaning they carry and indeed is a very crucial in sentinemtal analysis.Therefore instead of discarding them, we will substitute them with their actual meanings(dictionary will handle this later).
- Replace Emojies: Replacing emojies to words can be very useful in sentimental analysis(SA). Since they carry strong emotional weight, translating them to word can be indeed enhancing the impact on SA.
- Handling Hashtags: Extracting the word in hashtags are also a good move toward enhancing the SA.
- Removing Repeated Characters: This is optional method since it has positive and negative effect on our SA task.
  - Normalization: It will help us normalizing words by reducing variations.
  - Noise Reduction: It can reduce noise in the data by elimination unnecessary repetitions that don't contribute to the sentiment.

[1]Natural Language Processing

  - Model Effiency: With normalizing and reducing vocab size, model can become more effcient in processing and analysing text.(faster training and prediction times)

  Negative Side:

  - Loss of Intense Information: Repeated characters in social media texts can indicate emphasis of a user.Like 'soooo happppy' has stronger positive sentiment than 'so happy'.
  - Missunderstanding: Assume word 'cool'. It removals make it has no meaning and confusion.

  We will use two preprocess one with this and another without. We will discuss the accuracy later.

- Remove Punctuations: We'll eliminate punctuation marks from the text, with the exception of '!', given its significant impact on sentiment analysis tasks.

- Remove Non-ASCII Characters: We investigate through dataset and some of unusall characters were detected like �. We consider delete these since it has no meaning(I guess).

- Remove Numbers: Since numbers has no meaning in the SA tasks, we consider to delete them.

- Expanding Accronyms: We consider use a dictionary to converts accronyms like 'LOL' to 'laughing out loud'.There are several reasons that this method will effect the SA tasks:

  - Clearer Semantic Meaning: Expanding "LOL" to "laughing out loud" clarifies the meaning and sentiment of the text, making it easier for sentiment analysis algorithms to detect positivity or humor in the message.
  - Consistency: It standardizes variations of expressions of laughter (like "lol", "LOL", "Lol") into a consistent form, reducing ambiguity and improving the model's ability to recognize sentiment consistently.

- Tokenizing: We will use 'word tokenizer' from 'nltk' library and it is very main of NLP tasks since we must dealing with tokens.

- Lematize tokens: We use lematizer to have more normal data and reduce the vocab size for us.

We have implement a class for our preprocess data using 'nltk' and 're' library in order to apply above preprocess methods.

```python
class preprocess:
    def __init__(self, emoticons_dict, stop_words, lemmatizer, acronyms_dict):
        self.text = None
        self.map = emoticons_dict
        self.stop_words_set = stop_words
        self.lemmatizer = lemmatizer
        self.accronyms = acronyms_dict

    def assign_text(self, text):
        self.text = text

    def normalize_tokenize(self):
        self.text = self._lower_case()
        self.text = self._remove_user_mention()
        self.text = self._remove_URLs()
        self.text = self._remove_emalis()
        self.text = self._replace_emoticons()
        self.text = self._replace_emojies()
        self.text = self._remove_URLs()
        self.text = self._handling_hashtags()
        self.text = self._repeated_characters()
        self.text = self._remove_punctuations()
        self.text = self._remove_non_ascii_chars()
        self.text = self._remove_numbers()
        self.text = self._expand_accronyms()
        tokens = self._tokenizing()
        tokens = self._lemmatize_tokens(tokens)
        return tokens

    def _lower_case(self):
        return self.text.lower()

    def _remove_user_mention(self):
        return re.sub(r'@\w+', '', self.text)

    def _remove_URLs(self):
        return re.sub(r'http\S+|www\S+|https\S+', '', self.text, flags=re.
            ↪ MULTILINE)

    def _replace_emoticons(self):
        return " ".join(self.map.get(word, word) for word in self.text.split()
            ↪ )

    def _replace_emojies(self):
        return emoji.demojize(self.text, delimiters=("",""))

    def _handling_hashtags(self):
        return self.text.replace('#', '')

    def _repeated_characters(self):
        return re.sub(r'(.)\1+', r'\1\1', self.text)

    def _remove_punctuations(self): # Except important one like !
        return self.text.translate(str.maketrans('', '', string.punctuation.
            ↪ replace('!', '')))

    def _remove_numbers(self):
        return re.sub(r'\b\d+(?:,\d+)*(?:\.\d+)?\b', '', self.text)

    def _remove_non_ascii_chars(self):
        return self.text.encode('ascii', 'ignore').decode('ascii')

    def _expand_accronyms(self):
        return " ".join(self.accronyms.get(word, word) for word in self.text.
           ↪ split())

    def _tokenizing(self):
        return word_tokenize(self.text)

    def _lemmatize_tokens(self, tokens):
        return [self.lemmatizer.lemmatize(token) for token in tokens]

    def _remove_emalis(self):
        email_pattern = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'
        return re.sub(email_pattern, '', self.text)
```

The provided code are simple and readable and I think thats enough for preprocesing phase. We will create another feature 'tokens' for samples dataframe and from the function below, we will pass all the 'text' in sample df to this function in order to get tokens corresponding to that text:

```
def preprocess_text(text):
    preprocess_cls.assign_text(text)
    return preprocess_cls.normalize_tokenize()

sampled_df['tokens'] = sampled_df['text'].apply(preprocess_text)
```

Let's proceed to divide our sample dataframe into training and testing sets. To ensure effective training, we'll split our data so that both the training and testing sets have an equal distribution based on 'target' values. This is very important and crucial since an imbalance in classes' distributions can lead the model to be biased towards the class that is more.

```
negative_class = sampled_df[sampled_df['target'] == 0]
positive_class = sampled_df[sampled_df['target'] == 4]
X_pos_train, X_pos_test, y_pos_train, y_pos_test = train_test_split(
   ↪ positive_class['tokens'], positive_class['target'], test_size=0.2,
   ↪ random_state=2)
X_neg_train, X_neg_test, y_neg_train, y_neg_test = train_test_split(
   ↪ negative_class['tokens'], negative_class['target'], test_size=0.2,
   ↪ random_state=2)
X_train = pd.concat([X_pos_train, X_neg_train])
X_test = pd.concat([X_pos_test, X_neg_test])
y_train = pd.concat([y_pos_train, y_neg_train])
y_test = pd.concat([y_pos_test, y_neg_test])
```

Attention: As question said 20% test set therefore:

$$0.20 * 10000 = 0.20(5000 + 5000)$$

each class contains 5000 sample and overall 10000 sample therfore 20% of 10000 is equal to 20% of each class!
Attention: We did not use stop words since it include some words that are crucial to remain in SA tasks.(e,g, 'didn't' or 'could'nt').

1.3 Term Frequency

Before diving to create TF[2] for train and test we must declare how we are going to handle OOV[3]. In train TF matrix we are going to use the mean of vocab set to assign the 'OOV' words. In test TF matrix whenever we faced with OOV in a single document we will add 1 to the entry of corresponding in that doc and 'OOV'.
First from all tokens in trainset we are going to create a set that gives us unique vocoabs in all train documents.

```
    vocab_set = {}
    all_tokens = []
    for index,tokens in X_train.items():
        all_tokens.extend(tokens)

    print(len(all_tokens))
    vocab_set = set(all_tokens)
    print(len(vocab_set))
```

The number of tokens in all of the train documents and vocab set are in table 6:

Table 2: Tokens and Unique ones

Number of Tokens in train docs	Number of Unique Tokens
104489	11644

Attention:The numbers in table 6 will be variate if random state in sampling function changes.
In ordert to create each matrix for this part and following parts we are going to define a 'freq matrix' class be more robust and readable code:

```
    class freq_matrix:
        def __init__(self, X_train, X_test, vocab_set):
            self.train = X_train
            self.test = X_test
            self.vocab_set = vocab_set
            self.vocab_set.add('OOV')
            print(len(self.vocab_set))
            self.unique_words_index = list(self.vocab_set)
```

For getting instance of 'freq matrix' class simply you shoud pass train and test set as well as the set of vocabulary. In order to addressing the issue of 'OOV' words we consider adding this particular 'OOV' marker. Additionally, a seperate list will be created during creation of instance in order to index words more straightforward.TF matrix

[2]Term-Frequency
[3]Out of Vocabulary

for training set:

```
    def term_frequency_matrix_train(self):
        real_index = 0
        tf_matrix = np.zeros(shape=(len(self.train), len(self.vocab_set)),
           ↪ dtype=float)
        for _,tokens in self.train.items():
            for token in tokens:
                word_index = self.unique_words_index.index(token)
                tf_matrix[real_index][word_index] += 1
            word_index = self.unique_words_index.index('OOV')
            tf_matrix[real_index][word_index] = np.mean(tf_matrix[real_index])
            real_index += 1
        tf_matrix = tf_matrix + 1
        return np.where(tf_matrix > 1, 1 + np.log(tf_matrix), 0), tf_matrix
```

As implemented code above, tf_matrix will be initialzied with 0 with the dimension of:

$$\begin{aligned} rows &= len(docs) \\ cols &= len(vocab\_set) \end{aligned} \tag{1}$$

In processing each document we will counting how many times each token appears within that document. Once counting process for that doc was completed, we will get mean of that row for 'OOV' entry.(for that doc) At last we apply add-1 smoothing technique to the matrix.(Without add-1 we will face a problem in TF-IDF since log of something small would be negative!) Finally we will return TF matrix alongside of log-transfromed TF.
Test TF:

```
    def term_frequncy_matrix_test(self):
        real_index = 0
        unseen_data_set = set()
        tf_matrix = np.zeros(shape=(len(self.test), len(self.vocab_set)),
           ↪ dtype=float)
        for _,tokens in self.test.items():
            for token in tokens:
                if token in self.vocab_set:
                    word_index = self.unique_words_index.index(token)
                    tf_matrix[real_index][word_index] += 1
                else:
                    unseen_data_set.add(token)
                    word_index = self.unique_words_index.index('OOV')
                    tf_matrix[real_index][word_index] += 1
            real_index += 1
        tf_matrix = tf_matrix + 1
        return np.where(tf_matrix > 1, 1 + np.log(tf_matrix), 0), tf_matrix,
           ↪ unseen_data_set
```

The same process will be applied to the test set, with the distinct approach of managing 'OOV' (Out-of-Vocabulary) tokens by adding 1 to the corresponding document's count. Unseen data would be counted in order to see how much 'OOV' words are within testing set.

```
    my_freq_matrix = freq_matrix(X_train, X_test, vocab_set)
    log_train_tf, ordinary_tf_Train = my_freq_matrix.term_frequency_matrix_train()
    log_test_tf, ordinary_tf_test, unseen_data = my_freq_matrix.
       ↪ term_frequncy_matrix_test()
```

From provided code we can finally create TF matrix for training and testing set. The proportion of unseen data is about 15.6% of set of vocabulary.

1.4 Term Frequency-Inverse Document Frequency

We will use the following equations in order to caluclate TF-IDF[4] matrix:

$$tf_{t,d} = \begin{cases} 1 + \log_{10} count(t,d) & if\ count(t,d) > 0 \\ 0 & Otherwise \end{cases} \tag{2}$$

$$idf_t = \log_{10}\left(\frac{N}{df_t}\right)$$

$$w_{t,d} = tf_{t,d} * idf_t \tag{3}$$

But for addressing this issue that might df_t would be zero, we will add-1 smoothing to the df_t vector.Moreover, in train TF-IDF matrix for 'OOV' we will consider very rare document frequency due to we didn't see any 'OOV' word in our training documents.
Train TF-IDF:

```
    def tf_idf_matrix_train(self, log_tf, train_tf):
        train_tf_idf = np.zeros(shape=(log_tf.shape))
        term_freq = np.copy(train_tf)
        df = np.zeros(shape=(train_tf.shape[1]))
        idf = np.zeros(shape=(train_tf.shape[1]))
        for i,_ in enumerate(self.unique_words_index):
                df[i] = (term_freq[:,i] > 1).sum()
                idf[i] = np.log(train_tf.shape[0]/(df[i] + 1))
        idf[self.unique_words_index.index('OOV')] = np.log(train_tf.shape[0]/1)
        for i in range(len(self.unique_words_index)):
                train_tf_idf[:,i] = log_tf[:, i] * idf[i]
        return train_tf_idf
```

We will pass obtained log-transformed TF and TF from previous part to this function.
Attention: Since we use add-1 smoothing in TF matrix, for calculating DF we assume that word exist in that document whenever corresponding entry in TF matrix is greather than 1. We alsp assign very rare number (e,g, 1) to 'OOV' in df vector.
Test TF-IDF matrix:

```
    def tf_idf_matrix_test(self, log_tf, test_tf):
        train_tf_idf = np.zeros(shape=(log_tf.shape))
        term_freq = np.copy(test_tf)
```

[4]Term Frequency-Inverse Document Frequency

```
        df = np.zeros(shape=(test_tf.shape[1]))
        idf = np.zeros(shape=(test_tf.shape[1]))
        for i,_ in enumerate(self.unique_words_index):
                df[i] = (term_freq[:,i] > 1).sum()
                idf[i] = np.log(test_tf.shape[0]/(df[i] + 1))
        for i in range(len(self.unique_words_index)):
                train_tf_idf[:,i] = log_tf[:, i] * idf[i]
        return train_tf_idf
```

Same process were apply to test TF-IDF.(With out assign any numbers to ’OOV’ since we will see ’OOV’ word in test documents) Finally we can have our TF-IDF matrixes:

```
    tf_idf_train = my_freq_matrix.tf_idf_matrix_train(log_train_tf,
       ↪ ordinary_tf_Train)
    tf_idf_test = my_freq_matrix.tf_idf_matrix_test(log_test_tf, ordinary_tf_test)
```

1.5 Positive Pointwise Mutual Infromation

Pointwise mutual information is one of the most important concepts in NLP. It is a measure of how often two events x and y occur, compared with what we would expect if they were independent:

$$I(x,y) = \log_2 \frac{P(x,y)}{P(x)P(y)} \tag{4}$$

The PMI[5] between a target word w and a context word c is then defined as:

$$PMI_{(w,c)} = \log_2 \frac{P(w,c)}{p(w)p(c)} \tag{5}$$

PPMI[6] can be calculated as follows:

$$PPMI_{(w,c)} = \max\left(\log_2 \frac{P(w,c)}{P(w)P(c)}, 0\right) \tag{6}$$

There will be a question that needed to be answer. How can we use PPMI in documnet-word concept?
The answer is we can calculate the PPMI for word-word matrix and then calculate the centroid of words embedding vectors that exist in that particular document.
In order to calculate the PPMI, first we should calculate the following equations:

$$\begin{aligned} p_{ij} &= \frac{f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}} \\ p_{i*} &= \frac{\sum_{j=1}^{C} f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}} \\ p_{i*} &= \frac{\sum_{i=1}^{W} f_{ij}}{\sum_{i=1}^{W}\sum_{j=1}^{C} f_{ij}} \end{aligned} \tag{7}$$

[5]Pointwise Mutual Information
[6]Positive PMI

Now how can we obtain frequency matrix between word-word?
Don't go too far, we have bigrams! bigrams comming handy in order to calculate the co-occurence matrix between words from a corpus.
We will collect all tokens in training documents and use bigrams in 'nltk' library and FreqDist method to calculate how many time two words occurs with each other.

```
    def bigram_freq(self):
        all_tokens = []
        for _, tokens in self.train.items():
            all_tokens.extend(tokens)
        return FreqDist(list(ngrams(all_tokens, 2)))
```

Assume that 'love the bird' is our corpus therefore bigrams just saying 'love, the', 'the, bird' is one time.But what about 'the love'?
For addressing this issue we can simply use following code:

```
    def co_occurence_matrix(self):
        co_matrix = np.zeros(shape=(len(self.vocab_set), len(self.vocab_set)))
        bigrams = self.bigram_freq()
        for bigram,value in bigrams.items():
            word_index1 = self.unique_words_index.index(bigram[0])
            word_index2 = self.unique_words_index.index(bigram[1])
            co_matrix[word_index1, word_index2] += value
        word_index = self.unique_words_index.index('OOV')
        co_matrix[word_index, :] = np.mean(co_matrix, axis=0)
        co_matrix[:, word_index] = np.mean(co_matrix, axis=1)
        co_matrix += 1
        return co_matrix
```

We will initialize a matrix with zero,where the dimension of the matrix are:

$$\begin{aligned} rows &= \text{length of vocab set} \\ cols &= \text{length of vocab set} \end{aligned} \tag{8}$$

We will assign each word-word in the bigrams to the corresponding entry in co-matrix and leave the reamin entries to be zero. Moreover, we will assign 'OOV' row(and also col) to be mean of remain embedding vectors. Finally we apply add-1 smoothing and return the co-occurence matrix.
Calculating PPMI matrix:

```
    def ppmi_matrix(self, occurence_matrix):
        co_matrix = np.copy(occurence_matrix)
        total_sums = np.sum(co_matrix)
        co_matrix = co_matrix / total_sums
        p_rows = np.sum(co_matrix, axis=1) / total_sums
        p_cols = np.sum(co_matrix, axis=0) / total_sums
        pmi_matrix = np.log2(np.divide(co_matrix, np.outer(p_rows, p_cols)))
        ppmi_matrix = np.maximum(pmi_matrix, 0)
        return ppmi_matrix
```

In this code snippet, we calculate the total frequency (serving as the denominator in equation number 7). Additionally, we compute the sum across rows and the sum across columns from the co-occurrence matrix to derive p_{i*}, p_{i*} respectively. Now we can calculate the ppmi-matrix:

```python
    occurence_matrix = my_freq_matrix.co_occurence_matrix()
    ppmi_matrix = my_freq_matrix.ppmi_matrix(occurence_matrix)
```

Now it is time in order to calculate the doc-word ppmi using centroid of words exists in that document.

```python
    def doc_embeding_train(self, word_ppmi):
        doc_word_ppmi = np.zeros(shape=(len(self.train), len(self.vocab_set)),
           ↪ dtype=float)
        real_index = 0
        for _,tokens in self.train.items():
            counter = 0
            for token in tokens:
                doc_word_ppmi[real_index] += word_ppmi[self.unique_words_index.
                   ↪ index(token), :]
                counter += 1
            if counter != 0:
                doc_word_ppmi[real_index] = doc_word_ppmi[real_index] / counter
            real_index += 1
        return doc_word_ppmi
```

Just careful the counter is represented by how many words exist in that document and it will help us to calculate the centroid.
But why we use condition if counter is not zero?
Because some of tweets just having mention while we removed mentions in the Preprocessing phase.So the list of tokens would be empty for that document.Therefore we use that condition to avoid this problem.
Test doc-embedding:

```python
    def doc_embedding_test(self, word_ppmi):
        doc_word_ppmi = np.zeros(shape=(len(self.test), len(self.vocab_set)),
           ↪ dtype=float)
        real_index = 0
        for _,tokens in self.test.items():
            counter = 0
            for token in tokens:
                if token in self.vocab_set:
                    doc_word_ppmi[real_index] += word_ppmi[self.unique_words_index
                       ↪ .index(token), :]
                else:
                    doc_word_ppmi[real_index] += word_ppmi[self.unique_words_index
                       ↪ .index('OOV'), :]
                counter += 1
            if counter!=0:
                doc_word_ppmi[real_index] = doc_word_ppmi[real_index] / counter
            real_index += 1
        return doc_word_ppmi
```

Precess would be same for test doc-embedding, with the distinct approach of managing unseen vocab tokens. If token was in our vocab set we will add the corresponding embedded vector, otherwise we will add 'OOV' vector.

```
    train_doc_ppmi = my_freq_matrix.doc_embeding_train(ppmi_matrix)
    test_doc_ppmi = my_freq_matrix.doc_embedding_test(ppmi_matrix)
```

1.6 Fitting Model

Now we will use Multinomial naive bayes in order to fit to our model.Multinomial naive bayes is a classifier that are used in most NLP tasks. Since we consider the counts,thus it is positive and also discrete therefore MultinomialNB will be best option for fitting our model.
At first model we will remove repeated characters as we promised to check the metrics with or without the removing repeated characters. TF and TF-IDF and PPMI metric results with removing repeated characters are in table 7:

Table 3: Metrics Result

	Accuracy	Recall	Percision	F1-score
TF	0.737	0.712	0.7494736842105263	0.7179487179487181
TF-IDF	0.6935	0.663	0.7060702875399361	0.6809623430962343
PPMI	0.5795	0.543	0.5857605177993528	0.5388951521984217

These metrics definitions can be obtained by figure 2:

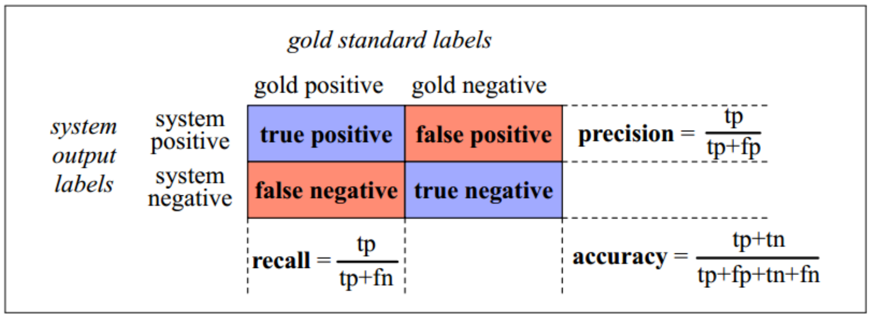

Figure 1: Metrics

F-measure comes from a weighted harmonic mean of precision and recall. The harmonic mean of a set of numbers is the reciprocal of the arithmetic mean of reciprocals:

$$HarmonicMean(a_1, a_2, \cdots, a_n)\frac{n}{\frac{1}{a_1}+\frac{1}{a_2}+\cdots+\frac{1}{a_n}} \tag{9}$$

and hence F-measure is

$$F = \frac{1}{\alpha\frac{1}{P}+(1-\alpha)\frac{1}{R}} \quad or(with\ \beta^2 = \frac{1-\alpha}{\alpha})\ F = \frac{(\beta^2+1)PR}{\beta^2 P + R} \tag{10}$$

Interpretation of metrics for TF:

- Accuracy: This metric indicates that the model correctly predicted the class labels for approximately 73.7% of the test dataset. It's a general measure of how often the model is correct.
- Precision: Precision indicates how many of the items labeled as positive by the model are actually positive. With a precision of approximately 74.9%, this means that out of all the instances the model predicted as positive, around 74.9% were indeed positive.
- F1-score: An F1-score of approximately 71.8% is a weighted average of precision and recall which considers both false positives and false negatives. It is especially useful when the cost of false positives and false negatives are roughly equivalent, or when the classes are imbalanced.
- Recall: Recall measures the ability of the model to find all the relevant instances in the dataset. A recall of 0.712 means the model identified 71.2% of all the positive cases in the test data. In other words, when the actual class label is positive, the model predicts it correctly 71.2% of the time.

Conclusion:The F1-score confirms this balance, showing that the model is reasonably effective at this classification task with the given Term-Frequency input.
Interpretation of metrics for TF-IDF:

- Accuracy: An accuracy of 69% means that, out of all predictions made by model, 69% were correct. This gives us a general idea of the model's overall performance but doesn't distinguish between its ability to predict different classes.
- Recall: A recall of 66.3% means that the model correctly identified 66.3% of the positive cases. In the context of text classification, this could mean how many relevant documents were retrieved out of all relevant documents available.
- Precision:A precision of 70.6% means that, when our model predicted a document to be in a certain category, it was correct about 70.6% of the time. Precision is important when the cost of a false positive is high.
- F1-Score: An F1-score of 68.09% indicates that our model has a fairly balanced performance between precision and recall. This is particularly useful when we want to find a balance between catching as many positives as possible (high recall) and ensuring that the predictions are accurate (high precision).

Interpretation of metrics for PPMI:

- Accuracy: Compared to the previous model, there is a decrease in overall performance. This suggests that the model is less effective at correctly identifying both positive and negative cases in your dataset when using PPMI and centroid-based features.
- Recall: This is a decrease from the previous model, indicating that using PPMI and centroids might be less effective at retrieving all relevant documents.
- Percision: A precision of 58.57% means that when the model predicts a document to be in a certain category, it is correct about 58.57% of the time. This is slightly lower than the precision of the previous model, suggesting a decrease in the reliability of the model's positive predictions with the current feature engineering method.
- F1-Score: An F1-score of 53.88% is lower than what was observed with the TF-IDF based model, indicating that the balance between precision and recall has worsened with the use of PPMI and centroids for features.

What we observed in the metrics between TF, TF-IDF, PPMI based are very out of our expectations. Since generally PPMI must be better than TF-IDF, TF-IDF better than TF. But why this happens?
The reasons can be attributed to several factors:
1)Tweets are short and informal.The effectiveness of TF in such a dataset might be because raw term frequencies, despite their simplicity, are quite effective in capturing the essence of sentiment in short texts.
2)In sentiment analysis, espically in short text like tweets specific words can have a high impact on the overall sentiment of the text. TF might captures the raw impact of those words more directly than TF-IDF or PPMI.
3)TF-IDF decrease the weights for terms that appear frequently in documents.However, in SA common words across positive and negative classes, for example exclamations or specific emoticonds, might be crucial in sentiment.
4) If in dataset similar terms are used in different documents, IDF might not add much value over simple term frequency.
5)In order to calculate the PPMI, co-occurence matrix must be calculated.Therefore with short documents as tweets can cause PPMI-matrix would be more sparse.
6)PPMI is a good technique to identifying significant assosiations between words.However, it may lose context that is very necessary in sentimenal analysis tasks.
7)Calculating of centroids of tokens within documents based on PPMI can introduce noise or lose important information. (specially when embedding vectors did not capture sentiment features!)
Overally from observed performance in this experminet could be becuase of Twitter data. The table 8 is the result of ignoring repeated charcters:

Table 4: Metrics based on ignoring repeated characters

	Accuracy	Recall	Percision	F1-score
TF	0.7285	0.68	0.7530454042081949	0.7146610614818708
TF-IDF	0.6935	0.648	0.7128712871287128	0.678889470927187
PPMI	0.5905	0.454	0.624484181568088	0.5257672264041691

As you can see our metrics are did not variate too much while for TF method the accuracy is a little bit lower,TF-IDF method is still unchanged and for PPMI accuracy is higher!

2 Question 2

2.1 Dataset

We will use 'pandas' library to load our dataset called 'sarcasm.json'. Simply we will use following code where accomplishes our purpose:

```
path = r'sarcasm.json'
df = pd.read_json(path, lines=True)
df.head()
```

Description of features in our dataset:

- The 'is_sarcastic' field marked as 1 if the corresponding row's data is sarcastice, otherwise, it is set to 0.
- 'headline' is a documnet where we must faced with.
- 'artilce_link' is where the documents comes from.

While investigating the data with following code:

```
print(df[df['is_sarcastic'] == 1].shape, df[df['is_sarcastic'] == 0].shape)
```

we findout that the number of sarcastic documents is a little bit less 5 than the number of not sarcastic docs.

Table 5: Number of documents

Not Sarcastic	Sarcastic
14985	13634

2.2 Part 1, Preprocessing Phase

For getting tokens from each documents, we have implemented a class called 'preprocess':

```
class preprocess:
    def __init__(self, lematizer):
        self.stop_words = set(stopwords.words('english'))
        self.lemmatizer = lematizer
        self.text = None

    def assign_text(self, text):
        self.text = text

    def normalize_tokenize(self):
        self.text = self._lower_case()
        self.text = self._remove_URLs()
        self.text = self._remove_emalis()
        self.text = self._remove_punctuations()
        self.text = self._remove_non_ascii_chars()
        self.text = self._remove_numbers()
```

```
            tokens = self._tokenizing()
            self.text = self._remove_stopwords(tokens)
            tokens = self._lemmatize_tokens(tokens)
            return tokens

        def _lower_case(self):
            return self.text.lower()

        def _remove_URLs(self):
            return re.sub(r'http\S+|www\S+|https\S+', '', self.text, flags=re.
               ↪ MULTILINE)

        def _remove_punctuations(self):
            return self.text.translate(str.maketrans('', '', string.punctuation))

        def _remove_numbers(self):
            return re.sub(r'\b\d+(?:,\d+)*(?:\.\d+)?\b', '', self.text)

        def _remove_non_ascii_chars(self):
            return self.text.encode('ascii', 'ignore').decode('ascii')

        def _tokenizing(self):
            return word_tokenize(self.text)

        def _lemmatize_tokens(self, tokens):
            return [self.lemmatizer.lemmatize(token) for token in tokens]

        def _remove_emalis(self):
            email_pattern = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'
            return re.sub(email_pattern, '', self.text)

        def _remove_stopwords(self, tokens):
            return [word for word in tokens if word not in self.stop_words]
```

While see the raw dataset in 'sarcasm.json' we detect some URLs but it was not necessary due to the number of docs having URL's is few.
Reasons we use this class in our preprocess:

- Lowercasing:

    1. Lowercasing make sure that the same word in different cases like 'HAPPY', 'happy' is recognized as same word.
    2. one of the most advantages of lowercasing is reducing vocabulary size.By converting all letters to lowercase, the total number of unique words in the dataset decreases.
    3. This method will normalize our data also.Furthure more, this method invloves standardizing textual data so that variants of the same word are treated as identical.

- Remove URLs, Emails: We did not investigate all documents to see if they have email or urls in them or not. However, for assuring, we will also delete them since they have no important features in our sarcastic task.
- Remove Punctuations, Numbers, Non-ASCII Chars: For the purpose of sarcastic classification, we must delete all punctuations, numbers and also non-ASCII characters, as they do not contribute significantly to our task.
- In our task we are going to use Glove vectors as an embedding vectors for each word. So tokenizing and lemmatization would be necessary.
- Also stopwords will not contribute significantly to our classification task, so we consider to remove them.

The following code will utilize our preprocessing method on each 'headline' in order to extract its tokens within the document.

```
preprocess_cls = preprocess(WordNetLemmatizer())
def preprocess_headline(headline):
    preprocess_cls.assign_text(headline)
    return preprocess_cls.normalize_tokenize()

df['preprocess_headling'] = df['headline'].apply(preprocess_headline)
df.head()
```

For save each tokens of each document, we consider creating a new column called 'preprocess_headline'. Now it's time to proceed with splitting our dataset, allocating 20% of it for the test set.To achive a more balanced dataset in terms of the 'is_sarcastic' values, we have implemented the following code:

```
not_sarcastic_df = df[df['is_sarcastic'] == 0]
is_sarcastic_df = df[df['is_sarcastic'] == 1]
X_pos_train, X_pos_test, y_pos_train, y_pos_test = train_test_split(
   ↪ is_sarcastic_df['preprocess_headling'], is_sarcastic_df['is_sarcastic'],
   ↪  test_size=0.2, random_state=2)
X_neg_train, X_neg_test, y_neg_train, y_neg_test = train_test_split(
   ↪ not_sarcastic_df['preprocess_headling'], not_sarcastic_df['is_sarcastic'
   ↪ ], test_size=0.2, random_state=2)
X_train = pd.concat([X_pos_train, X_neg_train])
X_test = pd.concat([X_pos_test, X_neg_test])
y_train = pd.concat([y_pos_train, y_neg_train])
y_test = pd.concat([y_pos_test, y_neg_test])
```

At the end we concatinate the positive and negative train, test.The shape of test set in terms of 'is_sarcastive' values are in table 6:

Table 6: Shape of test documents

Sarcastic docs	Not-sarcastic docs
2727	2997

Attention: Since we had less sarcastic documents in raw dataset, it was expected that for splitting we will also see this difference. But the best strategy for getting more balanced data it was what we implemented in code above.

2.3 Part 2, Loading GLOVE

Glove Vectors:
The GloVe model was developed by researchers at Stanford University. It is designed to aggregate global word-word co-occurrence statistics from a corpus, and then uses these statistics to map words into a space where the distance between words corresponds to semantic similarity. The method emphasizes both the aggregation of global statistics (like overall word co-occurrences across the entire corpus) and local context (capturing relational meaning from nearby word pairs), aiming to capture a wide array of linguistic features, including both syntactic and semantic relationships.
For embedding our tokens(words), we will use GLOVE Vectors. Firstly, we have downloaded Glove Vectors from 6b version and then create a dictionary based to store the words embeddings with following code:

```
path_glove_300 = r'Glove/glove.6B.300d.txt'
def load_glove_vectors(path):
    embeddings = {}
    with open(path, 'r', encoding='utf-8') as my_file:
        for line in my_file:
            values = line.split()
            word = values[0]
            vector = np.asarray(values[1:], dtype='float32')
            embeddings[word] = vector
    return embeddings
embeddings = load_glove_vectors(path_glove_300)
```

This code simply will load the path of Glove vectors where each line contains a word and 300 float numbers as embedding vector for that particular word.
Question: How we are going to embede each document?
There are various methods to answer this question, one of the most common approach is the calculation of the centroid of word embedding vectors within a document. This technique involves averaging the embeddings of all words in the document to represent its overall semantic content. For pursuing this puprose, the code below was implemented:

```
def create_glove_matrix(data, embeddings):
    first_key = list(embeddings.keys())[0]
    my_glove_matrix = np.zeros(shape=(len(data), len(embeddings[first_key])))
    real_index = 0
    for _,tokens in data.items():
        counter = 0
        for token in tokens:
            if token in embeddings:
                my_glove_matrix[real_index, :] += embeddings[token]
                counter += 1
        if counter > 0:
            my_glove_matrix[real_index, :] = my_glove_matrix[real_index, :]/
               ↪ counter
        real_index += 1
    return my_glove_matrix
```

This code will get the words in the embedding dictionary and for each token present in that document, if that token was in our embedding dictionary, we will add the corresponding embedding vector for that token. Finally, we will consider averaging the embeddings of all words in the document.
We will construct 'train glove matrix' and 'test glove matrix' with our pre-created train and test documents with the function implemented above.

```
    train_glove_matrix = create_glove_matrix(X_train, embeddings)
    test_glove_matrix = create_glove_matrix(X_test, embeddings)
```

Now we are ready to train the data in the next part!

2.4 Part 3, Training Model

So far, we have created embedding matrix for each documents using Glove Vectors.In order to train our model we will use Logistic Regression from 'sklearn' library in Python.

```
    model = LogisticRegression(max_iter=10000)
    model.fit(train_glove_matrix, y_train)
```

We assigned 1000 iteration as a maximum iterations.Then we will fit our Logistic Regression with our 'train glove matrix' and their labels created in previous sections.
In next step, we will pass the 'test glove matrix' to our model in order to predict for us either that particular test document is sarcastic or not.

```
    predicitons = model.predict(test_glove_matrix)
```

Let's dive into the metrics for evaluating our model now. Our intereseted metrics are shown in table 7: Interpret

Table 7: Evaluation Metrics

Accuracy	Recall	Percision	F1-Score
0.756	0.728	0.752	0.74

the Metrics:

- Accuracy (75.59%): Reflects the overall percentage of correct predictions (both sarcastic and non-sarcastic).
- Recall (72.83%): Indicates that the model correctly identifies 72.83% of all actual sarcastic cases. This means it misses some sarcastic documents, failing to label them as sarcastic.
- Precision (75.17%): Shows that out of all the documents the model labels as sarcastic, 75.17% are indeed sarcastic. This suggests the model is quite reliable in its positive classifications but still makes some errors.
- F1 Score (73.98%): Balances precision and recall, indicating the model has a harmonious performance between identifying most sarcastic cases and maintaining a high accuracy rate in these identifications.Further, The F1 score reflects a strong overall performance, demonstrating model's ability to manage the trade-off between recall and precision effectively.

3 Question 3

3.1 Dataset

Our dataset including stories of Sherlock Holmes.At first we are going to load dataset in Python with following code:

```python
path = 'advs.txt'
with open(path, 'r', encoding='utf-8') as file:
    data = file.read()
data
```

Through investigate on raw dataset we first consider to remove backslash n's and remove white spaces with following code:

```python
data = data.replace('\n', '')
data = re.sub(r'\s+', ' ', data).strip()
data
```

Before jumping to further preprocesses, we tokenized data respect to their sentences. Splitting our corpus into sentences before applying preprocessing steps is very beneficial for several reasons:(specially dealing with skip-gram models)

- Contextual Boundaries: Words are more likely to be related to other words within the same sentence than to words in the next sentence. This will be crucial and necessary step for skip-gram models, where the context of a word (the surrounding words within a specified window) is used to learn its embedding.
- Accurate Windowing:If we use a window size of 5, the five words before and after the target word should be form the same sentences.

Since we saw two main reasons, why should we tokenized sentences, the 'nltk' library can help us to splitting our corpus into sentences:

```python
sentences = sent_tokenize(data)
```

It is very precise in tokenizing sentences, for example from given following two sentences:

- Input: 'Dr.Khosravi have some briliant ideas. avenue st. has very complex road.'
- Output: 'Dr.Khosravi have some briliant ideas.', 'avenue st. has very complex road'.

As example above illustrate that 'sent_tokenize' can split our corpus into sentences. Now let's proceed to preprocess each sentences with function implemented below:

```python
# stop_words = set(stopwords.words('english'))
def preprocessing(sentence):
    my_sentence = sentence
    tokens = word_tokenize(my_sentence)
    tokens = [token.lower() for token in tokens]
    tokens = [word for word in tokens if word not in string.punctuation]
    # tokens = [word for word in tokens if word not in stop_words]
```

```python
        tokens = [word for word in tokens if word.isalpha()]
        tokens = [WordNetLemmatizer().lemmatize(word) for word in tokens]
        tokens = [token for token in tokens if token not in {"r's", "e's", "r", "e
            ↪ ", "g", "p"}]
        return tokens
```

For each sentence, we are going to tokenized their words.Then lowercasing would be beneficial for our model sinse it would reduce vocabulary size.Further we will delete all punctuations since they are not even word to obtain their embedding vector. We will also consider to delete all numbers with 'word.isalpha()' function with the same reason we have deleted all punctuations. Finally we consider to lemmatize each word in order to have less vocabulary size.
Attention: Through investigating the dataset, we faced some characters in line 225.Therefore the last line of code is simply delete these unique characters.
And we decided not to delete stopwords since it can indeed effect the real or precived window size in terms of original sentence structure.But lets us clarify this by an example:
Original Sentence:"The queen of the kingdom"
Real Window Size:With a window size of 2 around "queen", we will consider ['The', 'of'] as contexts.
Preprocessed Tokens:['queen', 'kingdom'] after removing stopwords. It is obvious that there is just one context for 'queen' if we consider to delete stopwords.
Now it's time to preprocess each sentence:

```python
    tokens = [preprocessing(sentence) for sentence in sentences]
    print(sentences[2], tokens[2])
```

The tokens for third sentence is:

- Sentence: 'In his eyes she eclipses and predominates the whole of her sex.'
- Tokens: ['in', 'his', 'eye', 'she', 'eclipse', 'and', 'predominates', 'the', 'whole', 'of', 'her', 'sex']

3.2 Part 1, Implementing Skip-Gram

First we are going to obtain how many positive neighbors(positive context) a unique word has with the following code:

```python
    def assign_neighbors(tokenized_sentence, window_size=2):
    word_neighbors = {}
    for tokens in tokenized_sentence:
        for i, word in enumerate(tokens):
            start = max(0, i-window_size)
            end = min(len(tokens), i + window_size + 1)
            neighbors = set(tokens[start:i] + tokens[i+1:end])
            if word not in word_neighbors:
                word_neighbors[word] = set()
            word_neighbors[word].update(neighbors)
    return word_neighbors
```

This code will return a dictionary the keys are unique words in all tokens.The value of each key is a set of their positive context with a pre-defined window size.

For example for word 'queen' with window size of five, the positive contexts are:
{'a', 'admirable', 'an', 'have', 'made', 'not', 'she', 'what', 'would'}
Now we have a dictionary that keys are representing target and the values are representing positive context. Before jumping to get negative sample for each pair of (target, positive context). We will use unigrams to obtain frequency of each tokens.Negative sampels or noise words are chosen according to their weighted unigrams frequency p_α, where α is a weight.For using weighted unigrams simply we will use the following equation:

$$P_\alpha(w) = \frac{count(w)^\alpha}{\sum_{w'} count(w')^\alpha} \tag{11}$$

The most common valu for α is 0.75 where gives us better performance since it gives rare noise words slightly higher probablity.
First, we will collect each token of sentence.

```
    whole_tokens = []
    for token_list in tokens:
        whole_tokens.extend(token_list)
```

Next step, using 'ngram' and 'FreqDist' in nltk library to calcualte the frequency of each word:

```
    unigram = list(ngrams(whole_tokens, 1))
    uni_freq = FreqDist(unigram)
    uni_freq
```

The frequency for some common words are illustrated in table 8: Applying α counts:

Table 8: Frequency of most common words

Word	Frequency
the	5615
a	3494
i	3035

```
    alpha = 0.75
    total_freqs = np.sum(np.array([value**alpha for _,value in uni_freq.items()]))
    for key,value in uni_freq.items():
        uni_freq[key] = (uni_freq[key] ** alpha) / total_freqs
    weighted_probs = [value for _,value in uni_freq.items()]
    unique_words = [key[0] for key,_ in uni_freq.items()]
```

Now based on implemented code above, we have weighted probabilities to get negative samples in more effecient way.
There is one thing to do before proceeding to getting negatve sample.We must create (target word, context word) pairs then get 4 negative sample for each pair:

```
    target_neighbors_pairs = []
    for target_index, neighbors in context_neigbor.items():
        pairs = [(target_index, neighbor) for neighbor in neighbors]
        target_neighbors_pairs.extend(pairs)
```

From creating pairs, we can now get negative samples for each pair:

```
def get_negative_samples(pairs, context_neigbor, unique_words, weighted_probs,
    ↪ n_samples=4):
    target_neigbor = list(context_neigbor[pairs[0]]) + list(context_neigbor[
        ↪ pairs[1]])
    negative_samples = []
    while len(negative_samples) < n_samples:
        neg_candidates = random.choices(unique_words, weighted_probs, k=
            ↪ n_samples)
        for neg in neg_candidates:
            if neg not in target_neigbor + [pairs[0]] + negative_samples + [
                ↪ pairs[1]]:
                negative_samples.append(neg)
            if len(negative_samples) == n_samples:
                break
    return negative_samples
```

We will make sure that negative samples are not from the neighbors of that particualr target and context neighbors.We will use a while loop statement to get 4 negative sample from unique words with their corresponding wighted probabilities.
if negative sample in 4 candidate samples are not in the list of target and context neighbors and we will append it to negative samples list.
With utilizing dictionary structure, for every keys(pair) we will get 4 negative sample as that keys' values. This will takes around 6 minutes if we used window size of 5.

```
negative_samples = {pair: get_negative_samples(pair, context_neigbor,
    ↪ unique_words, weighted_probs) for pair in target_neighbors_pairs}
```

Clarifing with example:

```
negative_samples[('sherlock', 'holmes')]
```

Four negative sample for that examples are: ['vincent', 'supper', 'force', 'camp']. Now we are ready to implement our skip-gram model.First we have defined a class for skip-gram model as follows:

```
class skip_gram():
    def __init__(self, unique_words, embedding_dim, contexts) -> None:
        self.unique_words = unique_words
        self.embedding_dim = embedding_dim
        self.neighbors_neg_context = list(contexts.items())
        self.lr = None
        self.loss = None
        self.context_matrix = np.random.uniform(-0.5/embedding_dim, 0.5/
            ↪ embedding_dim, (len(unique_words), embedding_dim))
        self.target_matrix = np.random.uniform(-0.5/embedding_dim, 0.5/
            ↪ embedding_dim, (len(unique_words), embedding_dim))
```

With instantiation of our Skip-gram model class, it will define unique words and an embedding dimension. A dictionary for pairs and negative sampling is prepared as a list and allocated to 'neighbor neg context'. Initially, learning rate and loss are set to None. Both the context and target matrices are randomly initialized using a uniform distribution across a structure where rows represent words and columns represent the embedding dimensions. Sigmoid function is used to calculate loss fucntion and update weights:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{12}$$

We implement a method to calculate the sigmoid function(it is a private method):

```
    def _sigmoid_function(self, x):
        return 1 / (1 + np.exp(-x))
```

From the source book we know that loss function is:

$$\begin{aligned} L_{CE} &= -\log\left[P(+|w,c_{pos})\prod_{i=1}^{k}P(-|w,c_{neg_i})\right] \\ &= -\left[\log P(+|w,c_{pos}) + \sum_{i=1}^{k}\log P(-|w,c_{neg_i})\right] \\ &= -\left[\log P(+|w,c_{pos}) + \sum_{i=1}^{k}\log\left(1-P(+|w,c_{neg_i})\right)\right] \\ &= -\left[\log \sigma(c_{pos}.w) + \sum_{i=1}^{k}\log\sigma(-c_{neg_i}.w)\right] \end{aligned} \tag{13}$$

This loss function is used to maximize the dot product of the word with the actual context words, and minimize the dot products of the word with the k negative sampled non-neighbor words.
We will minimize the loss fucntion using stochastic gradiend descent.To get the gradient, we need to to take derivative of loss function with respect to the different embeddings.

$$\frac{\partial L_{CE}}{\partial c_{pos}} = [\sigma(c_{pos}.w) - 1]\, w \tag{14}$$

$$\frac{\partial L_{CE}}{\partial c_{neg}} = [\sigma(c_{neg}.w)]\, w \tag{15}$$

$$\frac{\partial L_{CE}}{\partial w} = [\sigma(c_{pos}.w) - 1]\, c_{pos} + \sum_{i=1}^{k}[\sigma(c_{neg_i}.w)]\, c_{neg_i} \tag{16}$$

Now from equations above we can update weights from time step t to t+1.

$$c_{pos}^{t+1} = c_{pos}^{t} - \eta\left[\sigma(c_{pos}^{t}.w^{t}) - 1\right] w^{t} \tag{17}$$

$$c_{neg}^{t+1} = c_{neg}^{t} - \eta\left[\sigma(c_{neg}^{t}).w^{t}\right] w^{t} \tag{18}$$

$$w^{t+1} = w^{t} - \eta\left[\left[\sigma(c_{pos}^{t}.w^{t}) - 1\right] c_{pos}^{t} + \sum_{i=1}^{k}\left[\sigma(c_{neg_i}^{t}.w^{t})\right] c_{neg}^{t}\right] \tag{19}$$

Now we can implement code such that calculate the loss function and also update the weights corresponding to each pair, and its' 4 negative samples.
The following private method is for update weights alongside calculation of loss fucntion:

```
    def _update_weights_with_loss(self, pair, negative_samples):
        cache = 0

        target_index, context_index = self.unique_words.index(pair[0]), self.
            ↪ unique_words.index(pair[1])
        dot_product = np.dot(self.context_matrix[context_index, :], self.
            ↪ target_matrix[target_index, :])
        sigmoid_score = self._sigmoid_function(dot_product)
        self.loss -= (np.log(sigmoid_score))
        cache += ((sigmoid_score-1) * self.context_matrix[context_index, :])
        self.context_matrix[context_index, :] -= (self.lr * (sigmoid_score -
            ↪ 1) * self.target_matrix[target_index, :])

        for neg_sample in negative_samples:
            word_index = self.unique_words.index(neg_sample)
            dot_product = np.dot(self.context_matrix[word_index, :], self.
                ↪ target_matrix[target_index, :])
            self.loss -= (np.log(self._sigmoid_function(-dot_product)))
            sigmoid_score = self._sigmoid_function(dot_product)
            cache += (sigmoid_score * self.context_matrix[word_index, :])
            self.context_matrix[word_index, :] -= (self.lr * (sigmoid_score) *
                ↪  self.target_matrix[target_index, :])
        self.target_matrix[target_index, :] -= (self.lr * cache)
```

We initalized cache to zero since we dont want to lose previous c_{pos}, c_{neg} since they will be updated after each time.First we calculate the loss for positive context and next $\sigma(c_{pos}.w) - 1$ will be cached and at the end we will update the positive context in context matrix.Once the the calculation of positive context word was finished, we proceed the same procedure for negative sample with loop on them.

Attention: cache is simply used for calculation of $\left[\left[\sigma(c^t_{pos}.w^t) - 1\right] c^t_{pos} + \sum_{i=1}^{k} \left[\sigma(c^t_{neg_i}.w^t)\right] c^t_{neg}\right]$.

It is time to train our model in order to fit the weights with the following code:

```
    def fit(self, learning_rate, epochs):
        loss_per_epoch = np.zeros(shape=(epochs))
        self.lr = learning_rate
        for epoch in range(epochs):
            self.loss = 0
            random.shuffle(self.neighbors_neg_context)
            for pair, negative_samples in self.neighbors_neg_context:
                self._update_weights_with_loss(pair, negative_samples)
            print(f'Epoch number : {epoch}, Loss : {self.loss}')
            loss_per_epoch[epoch] = self.loss
        return loss_per_epoch
```

This code will get the learning rate and epochs.For the purpose of plotting the loss per epoch we defined a numpy array.In each epoch we will use random library to shuffle the pairs with their correspondin negative samples due to have better generalization and letting the model to learn more robust features.Loss will be set to zero after each epoch to calculate the loss for each epoch seperately.

Then we will loop on the pairs and update the target word and context matrix based on them. At end of each epoch we will print the loss and epoch number.

```
        def matrix_sums(self):
            return self.context_matrix + self.target_matrix

        def cosine_between_two_vec(self, x, y):
            return np.dot(x,y)/ np.sqrt(np.dot(x,x)*np.dot(y,y))
```

For the next parts we implemented methods above to return the sum of context embedding and target embedding and similarity of two vector based on their embeddings.
Now let's get an instance of our class and train the data:

```
    embedding_dim = 100
    model = skip_gram(unique_words, embedding_dim, negative_samples)
    learning_rate = 0.05
    epochs = 20
    loss_per_epoch = model.fit(learning_rate, epochs)
```

The loss per epoch is illustrated in figure 2:(Each epoch will take 50 second to train!)

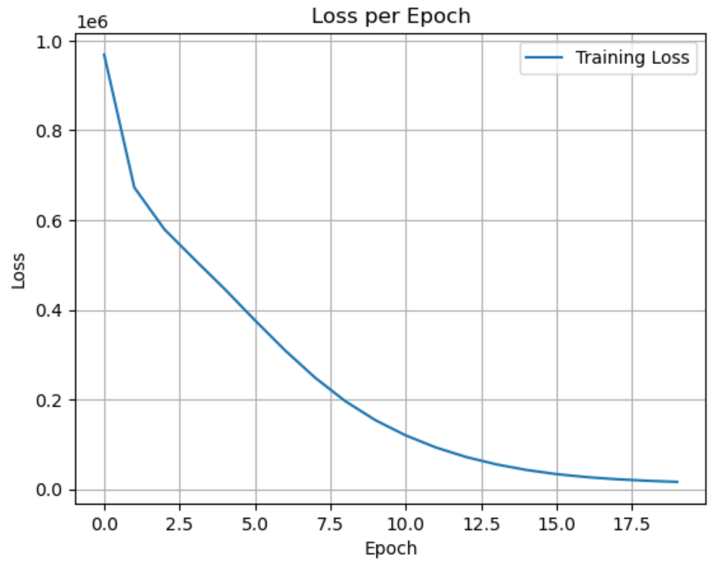

Figure 2: Loss per Epoch

As you can see loss will be minimized after each epoch.It is time to get sum of context and target matrix for future purposes.

```
    embedding_vectors = model.matrix_sums()
```

3.3 Part 2, Similarity

We can notice two words are similar only if the dot product between their corresponding vectors divided by multiplication of their norm$_2$s will be close to one, Otherwise they are less similar. From the following code we can can obtain the similarity between the provided vectors:

```
    king_index = unique_words.index('king')
    man_index = unique_words.index('man')
    woman_index = unique_words.index('woman')
    queen_index = unique_words.index('queen')
    print(king_index, man_index, woman_index, queen_index)
    vec_1 = embedding_vectors[king_index, :] - embedding_vectors[man_index, :] +
       ↪ embedding_vectors[woman_index, :]
    vec_2 = embedding_vectors[queen_index, :]
    cosin_vec1_vec2 = model.cosine_between_two_vec(vec_1, vec_2)
```

The cosine similarity between these two vectors is approximately around 0.13, indicating they are not highly similar.Although there is slight degree of similarity between them. Moreover, word 'queen' has come twice in our corpus so its neighbors are few.It might effect our model to learn the words. Through investigate the dataset, we found out that 'queen' is similar to 'she' since they are neighbors and we can see how much similary the following vectors are:

$$\begin{aligned} vector_1 &= vec('king') - vec('he') + vec('she') \\ vector_2 &= vec('queen') \end{aligned} \tag{20}$$

The similarity between these two vectors are around 0.24 which gives us better intution about our model.Let's see another example and see how much similar are the words 'sherlock' and 'holmes'.The similarity between these two vectors is around 0.69 where this value is quite higher since they are really similar words.

3.4 Part 3, PCA

Now from PCA in sklearn library we will reduced the embedding dimenstion to 2d vectors with following code:

```
    pca = PCA(n_components=2)
    embeddings_2d = pca.fit_transform(embedding_vectors)
```

Now from 2d embedding vectors we will get the vectors corresponding to 'brother', 'sister', 'uncle', 'aunt':

```
    word1_index = unique_words.index('sister')
    word2_index = unique_words.index('brother')
    word3_index = unique_words.index('aunt')
    word4_index = unique_words.index('uncle')
    word1_embedding_2d = embeddings_2d[word1_index]
    word2_embedding_2d = embeddings_2d[word2_index]
    word3_embedding_2d = embeddings_2d[word3_index]
    word4_embedding_2d = embeddings_2d[word4_index]
```

The differne for these vectors are:

```
    diff_vector_2d = word2_embedding_2d - word1_embedding_2d
    diff_vector_2d_2 = word4_embedding_2d - word3_embedding_2d
```

Now from matplotlib we will plot them:

```python
    plt.figure(figsize=(8, 8))

    plt.scatter([word2_embedding_2d[0], word1_embedding_2d[0]], [word2_embedding_2d[1], word1_embedding_2d[1]], color=['red', 'blue'])
    plt.scatter([word4_embedding_2d[0], word3_embedding_2d[0]], [word4_embedding_2d[1], word3_embedding_2d[1]], color=['red', 'blue'])
    plt.text(word1_embedding_2d[0], word1_embedding_2d[1], 'sister', fontsize=12, ha='right')
    plt.text(word2_embedding_2d[0], word2_embedding_2d[1], 'brother', fontsize=12,  ha='right')
    plt.text(word3_embedding_2d[0], word3_embedding_2d[1], 'aunt', fontsize=12, ha='right')
    plt.text(word4_embedding_2d[0], word4_embedding_2d[1], 'uncle', fontsize=12, ha='right')

    plt.arrow(word1_embedding_2d[0], word1_embedding_2d[1], diff_vector_2d[0], diff_vector_2d[1], head_width=0.005, head_length=0.03, fc='green', ec='green')
    plt.arrow(word3_embedding_2d[0], word3_embedding_2d[1], diff_vector_2d_2[0], diff_vector_2d_2[1], head_width=0.005, head_length=0.03, fc='green', ec= 'green')

    plt.show()
```

The figure 3 illustrate the difference vectors: If the difference vectors were parallel, it would indicate a very high

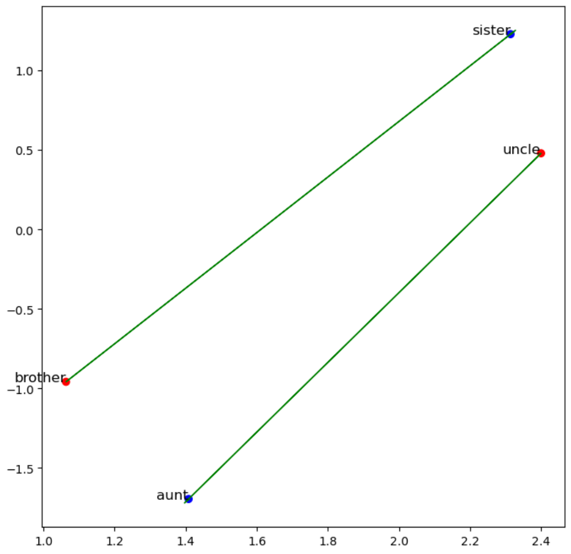

Figure 3: Diff Vectors

similarity. The fact that they are opposite could suggest several reasons:

- The skip-gram model may not have learned a consistent representation for gender differences.

- The word embeddings might be capturing more complex semantic relationships than just gender. For example, ”uncle” and ”aunt” might have other contextual associations that influence the direction of their difference vector.
- The process of reducing the dimensionality to 2D with PCA can sometimes create artifacts. PCA finds a projection that maximizes variance and may not always preserve the original semantic relationships. In high-dimensional space, these vectors might be more aligned, and the opposition could be a result of the PCA projection.
- If the words ”brother” and ”sister” appear in different contexts than ”uncle” and ”aunt,” this could affect the directionality of the difference vectors.