In [1]:

```
# %%shell
# jupyter nbconvert --to html /content/HW1_CSCI544.ipynb
```

In [2]:

```
import pandas as pd
import numpy as np

import pickle

import gensim
import gensim.downloader

import nltk
nltk.download('wordnet')
import re
import string
from bs4 import BeautifulSoup
from nltk.stem import WordNetLemmatizer
from nltk.tag import pos_tag
nltk.download('omw-1.4')
nltk.download('averaged_perceptron_tagger')

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split

from sklearn.svm import LinearSVC
from sklearn.linear_model import LogisticRegression
from sklearn.linear_model import Perceptron
from sklearn.naive_bayes import MultinomialNB
from nltk.metrics.scores import precision
import sklearn.metrics as metrics
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score
from sklearn.preprocessing import LabelEncoder, OneHotEncoder

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils import data
# from torch.utils.data import Dataset, DataLoader
```

```
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\ADMIN\AppData\Roaming\nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package omw-1.4 to
[nltk_data]     C:\Users\ADMIN\AppData\Roaming\nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     C:\Users\ADMIN\AppData\Roaming\nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
```

In [3]:

```
# ! pip install bs4 # in case you don't have it installed
# Dataset: https://s3.amazonaws.com/amazon-reviews-pds/tsv/amazon_reviews_us_Beauty_v1_00.tsv.gz
```

Helper Function (Length)

In [4]:

```
# Average Length Function
def average_length(review_column) -> str:
  return review_column.apply(len).mean()

def average_length_print(before_cleaning, review_column) -> str:
  return str(before_cleaning) + ", " + str(review_column.apply(len).mean())
```

1. Dataset Generation

Read Data

Imported the data as a Pandas frame using Pandas package and only kept the Reviews and Ratings fields in the input data frame to generate data.
Reference - https://www.geeksforgeeks.org/how-to-load-a-tsv-file-into-a-pandas-dataframe/ (https://www.geeksforgeeks.org/how-to-load-a-tsv-file-into-a-pandas-dataframe/)

In [5]:

```
dataset = pd.read_table('data.tsv', on_bad_lines='skip', low_memory=False)
```

Dropped the column with the null value and converted the datatype of star_rating column to int for consistency.
Reference - https://www.statology.org/pandas-create-dataframe-from-existing-dataframe/ (https://www.statology.org/pandas-create-dataframe-from-existing-dataframe/)

In [6]:

```
InputDataFrame = dataset[['star_rating','review_body']].copy()
InputDataFrame.to_csv('Dataset_Input_Data_Frame.csv', index=False)
InputDataFrame = pd.read_csv('Dataset_Input_Data_Frame.csv', low_memory=False)
InputDataFrame.head()
```

Out[6]:

	star_rating	review_body
0	5	Love this, excellent sun block!!
1	5	The great thing about this cream is that it do...
2	5	Great Product, I'm 65 years old and this is al...
3	5	I use them as shower caps & conditioning caps....
4	5	This is my go-to daily sunblock. It leaves no ...

In [7]:

```
InputDataFrame.isnull().sum()
```

Out[7]:

```
star_rating     10
review_body    400
dtype: int64
```

In [8]:

```
# InputDataFrame.info(verbose = True, show_counts = True)
InputDataFrame.dropna(inplace = True)

InputDataFrame['star_rating'] = InputDataFrame['star_rating'].astype('float').astype('int')
# InputDataFrame.dtypes
```

In [9]:

```
InputDataFrame.isnull().sum()
```

Out[9]:

```
star_rating    0
review_body    0
dtype: int64
```

Created a three-class classification problem according to ratings with ratings with the values of 1 and 2 from class 0, ratings with the value of 3 form class 1, and ratings with the values of 4 and 5 form class 2.
Reference - https://www.geeksforgeeks.org/create-a-new-column-in-pandas-dataframe-based-on-the-existing-columns/ (https://www.geeksforgeeks.org/create-a-new-column-in-pandas-dataframe-based-on-the-existing-columns/)

In [10]:

```
def assign_class(value):
    if value == 1 or value == 2:
        return 0
    if value == 3:
        return 1
    if value == 4 or value == 5:
        return 2

InputDataFrame['class'] = InputDataFrame['star_rating'].map(assign_class)

# InputDataFrame['class'].value_counts()
# display(InputDataFrame)
InputDataFrame.head()
```

Out[10]:

	star_rating	review_body	class
0	5	Love this, excellent sun block!!	2
1	5	The great thing about this cream is that it do...	2
2	5	Great Product, I'm 65 years old and this is al...	2
3	5	I use them as shower caps & conditioning caps....	2
4	5	This is my go-to daily sunblock. It leaves no ...	2

In [11]:

```
before_cleaning = average_length(InputDataFrame["review_body"])
print("Review Average Length : " + str(before_cleaning))
```

```
Review Average Length : 253.43061308343476
```

In [12]:

```
# InputDataFrame.to_csv('Dataset_Data_Frame.csv', index=False)
```

In [13]:

```
InputDataFrameHW1 = InputDataFrame.copy()
InputDataFrameHW3 = InputDataFrame.copy()

# InputDataFrameHW1 = pd.read_csv('Dataset_Data_Frame.csv')
# InputDataFrameHW3 = pd.read_csv('Dataset_Data_Frame.csv')
```

In [14]:

```
InputDataFrameHW1['class'].value_counts()
```

Out[14]:

```
2    3979156
0     717991
1     396760
Name: class, dtype: int64
```

In [15]:

```
del dataset, InputDataFrame
```

Helper Function Class (Data Cleaning)

Data cleaning steps to preprocess the dataset you created

Contraction()

code to perform contractions on the reviews where different regexes are used to decontract the words in the review. Two general contraction regex are used and 8 specific regexes for contraction are used.

Lowercase()

code to convert all reviews into lowercase where each word of the data frame is lowered with the help of python function .lower()

RemoveHTMLURL()

code to remove the HTML and URLs from the reviews where two different regex is used to remove the html tag and the url by passing each review text to html_url function

RemoveNonAlphabeticalCharacter()

code to remove non-alphabetical characters where single regex expression which remove the characters except the a-z, A-Z and spaces between words

RemoveExtraSpaces()

code to remove extra spaces where regex is used to remove extra white spaces

Reference - https://www.kaggle.com/code/benroshan/sentiment-analysis-amazon-reviews/notebook (https://www.kaggle.com/code/benroshan/sentiment-analysis-amazon-reviews/notebook)
Reference - https://stackoverflow.com/questions/19790188/expanding-english-language-contractions-in-python (https://stackoverflow.com/questions/19790188/expanding-english-language-contractions-in-python)

In [16]:

```
class Contraction():
    def __init__(self, InputDataFrame) -> None:
        self.InputDataFrame = InputDataFrame

    def contraction_function(self, review):
        review = re.sub(r"won\'t", "will not", review)
        review = re.sub(r"can\'t", "can not", review)
        review = re.sub(r"n\'t", " not", review)
        review = re.sub(r"\'re", " are", review)
        review = re.sub(r"\'s", " is", review)
        review = re.sub(r"\'d", " would", review)
        review = re.sub(r"\'ll", " will", review)
        review = re.sub(r"\'t", " not", review)
        review = re.sub(r"\'ve", " have", review)
        review = re.sub(r"\'m", " am", review)
        return review

    def clean(self) -> None:
        self.InputDataFrame['review_body'] = self.InputDataFrame['review_body'].apply(lambda text: 

class Lowercase():
    def __init__(self, InputDataFrame) -> None:
        self.InputDataFrame = InputDataFrame

    def clean(self) -> None:
        self.InputDataFrame['review_body'] = self.InputDataFrame['review_body'].apply(lambda text: 

class RemoveHTMLURL():
    def __init__(self, InputDataFrame) -> None:
        self.InputDataFrame = InputDataFrame

    def html_url(self, review):
        review = re.sub('https?://\S+|www\.\S+', '', review) # html
        review = re.sub('<[^<]+?>', '', review)              # url
        return review

    def clean(self) -> None:
        self.InputDataFrame['review_body'] = self.InputDataFrame['review_body'].apply(lambda text: 

class RemoveNonAlphabeticalCharacter():
    def __init__(self, InputDataFrame) -> None:
        self.InputDataFrame = InputDataFrame

    def clean(self) -> None:
        self.InputDataFrame['review_body'] = self.InputDataFrame['review_body'].apply(lambda text: 

class RemoveExtraSpaces():
    def __init__(self, InputDataFrame) -> None:
        self.InputDataFrame = InputDataFrame

    def clean(self) -> None:
        self.InputDataFrame['review_body'] = self.InputDataFrame['review_body'].apply(lambda text: 
```

TF-IDF Dataset Generation

Data Cleaning

In [17]:

```
contraction = Contraction(InputDataFrameHW1)
contraction.clean()

data_cleaning_lowercase = Lowercase(InputDataFrameHW1)
data_cleaning_lowercase.clean()

data_cleaning_remove_html_url = RemoveHTMLURL(InputDataFrameHW1)
data_cleaning_remove_html_url.clean()

remove_non_alphabetical_character = RemoveNonAlphabeticalCharacter(InputDataFrameHW1)
remove_non_alphabetical_character.clean()

remove_white_space = RemoveExtraSpaces(InputDataFrameHW1)
remove_white_space.clean()
```

In [18]:

```
InputDataFrameHW1.isnull().sum()
```

Out[18]:

```
star_rating    0
review_body    0
class          0
dtype: int64
```

To avoid the computational burden, select 20,000 random reviews from each rating class and create a balanced dataset to perform the required tasks on the downsized dataset.

In [19]:

```
InputDataFrame_1 = InputDataFrameHW1.loc[InputDataFrameHW1['class'] == 0].sample(n = 20000)
InputDataFrame_2 = InputDataFrameHW1.loc[InputDataFrameHW1['class'] == 1].sample(n = 20000)
InputDataFrame_3 = InputDataFrameHW1.loc[InputDataFrameHW1['class'] == 2].sample(n = 20000)
```

In [20]:

```
## Reference - https://pandas.pydata.org/docs/user_guide/merging.html

InputDataFrames = [InputDataFrame_1, InputDataFrame_2, InputDataFrame_3]
InputDataFrameFinalHW1 = pd.concat(InputDataFrames)

del InputDataFrame_1, InputDataFrame_2, InputDataFrame_3
InputDataFrameFinalHW1['class'].value_counts()
```

Out[20]:

```
0    20000
1    20000
2    20000
Name: class, dtype: int64
```

In [21]:

```
print("Average length of reviews before and after data cleaning")
print(average_length_print(before_cleaning, InputDataFrameHW1["review_body"]))
```

```
Average length of reviews before and after data cleaning
253.43061308343476, 243.177856800291
```

Pre-processing

Using NLTK package to process the dataset by remove the stop words and tokenizing the reviews and applying part of speech tagging technique to identify part of speech for each word in the review for accurate performance of lemmatization

lemmatization()

code to perform lemmatization where each review text is split into list of words and each word is assigned a part of speech tag and based on the part of speech tag each word is lemmatized with the help of WordNetLemmatizer of nltk

In [22]:

```
before_preprocessing = average_length(InputDataFrameFinalHW1["review_body"])
```

In [23]:

```
def lemmatization(review) -> None:
    lemmatizer = WordNetLemmatizer()
    lemmatized_sentence = []
    for word, tag in pos_tag(review):
        if (review is None):
            return review
        else:
            if tag.startswith('NN'):
                pos = 'n'
            elif tag.startswith('VB'):
                pos = 'v'
            else:
                pos = 'a'
            lemmatized_sentence.append(lemmatizer.lemmatize(word, pos))
    return lemmatized_sentence

InputDataFrameFinalHW1['review_body'] = InputDataFrameFinalHW1['review_body'].apply(lambda text: '
```

In [24]:

```
print("Average length of reviews before and after data preprocessing")
print(average_length_print(before_preprocessing, InputDataFrameFinalHW1["review_body"]))
```

```
Average length of reviews before and after data preprocessing
258.4317, 248.80425
```

In [25]:

```
# InputDataFrameFinalHW1.to_csv('Dataset_Final_Data_HW1.csv', index=False)
# InputDataFrameFinalHW1 = pd.read_csv('Dataset_Final_Data_HW1.csv')
InputDataFrameFinalHW1['class'].value_counts()
```

Out[25]:

```
0    20000
1    20000
2    20000
Name: class, dtype: int64
```

In [26]:

```
InputDataFrameFinalHW1.isnull().sum()
```

Out[26]:

```
star_rating    0
review_body    0
class          0
dtype: int64
```

In [27]:

```
InputDataFrameFinalHW1.head()
```

Out[27]:

	star_rating	review_body	class
962866	1	roll my pressure sock into a painful band half...	0
2814727	1	buy these to replace another pair of lacross t...	0
163608	1	ridiculous price i purchase the exact same pro...	0
1226023	2	i read a lot of review before buy good on fine...	0
3093646	1	i have a hair that like a lot of moisture and ...	0

Word2Vec Dataset Generation

Data Cleaning

In [33]:

```
data_cleaning_remove_html_url = RemoveHTMLURL(InputDataFrameHW3)
data_cleaning_remove_html_url.clean()

remove_non_alphabetical_character = RemoveNonAlphabeticalCharacter(InputDataFrameHW3)
remove_non_alphabetical_character.clean()

remove_white_space = RemoveExtraSpaces(InputDataFrameHW3)
remove_white_space.clean()
```

In [34]:

```
print("Average length of reviews before and after data cleaning")
print(average_length_print(before_cleaning, InputDataFrameHW3["review_body"]))
```

```
Average length of reviews before and after data cleaning
253.43061308343476, 241.223850965477
```

In [35]:

```
InputDataFrameHW3.isnull().sum()
```

Out[35]:

```
star_rating    0
review_body    0
class          0
dtype: int64
```

In [36]:

```
InputDataFrameHW3.dropna(inplace = True)
```

In [37]:

```
InputDataFrame_1 = InputDataFrameHW3.loc[InputDataFrameHW3['class'] == 0].sample(n = 20000)
InputDataFrame_2 = InputDataFrameHW3.loc[InputDataFrameHW3['class'] == 1].sample(n = 20000)
InputDataFrame_3 = InputDataFrameHW3.loc[InputDataFrameHW3['class'] == 2].sample(n = 20000)
```

In [38]:

```
## Reference - https://pandas.pydata.org/docs/user_guide/merging.html

InputDataFrames = [InputDataFrame_1, InputDataFrame_2, InputDataFrame_3]
InputDataFrameFinalHW3 = pd.concat(InputDataFrames)

del InputDataFrame_1, InputDataFrame_2, InputDataFrame_3
InputDataFrameFinalHW3['class'].value_counts()
```

Out[38]:

```
0    20000
1    20000
2    20000
Name: class, dtype: int64
```

In [39]:

```
# InputDataFrameFinalHW3.to_csv('Dataset_Final_Data_HW3.csv', index=False)
# InputDataFrameFinalHW3 = pd.read_csv('Dataset_Final_Data_HW3.csv')
```

In [40]:

```
InputDataFrameFinalHW3.isnull().sum()
```

Out[40]:

```
star_rating    0
review_body    0
class          0
dtype: int64
```

In [41]:

```
InputDataFrameFinalHW3.head()
```

Out[41]:

	star_rating	review_body	class
2133250	1	had to return did not include small travel bot...	0
847125	1	this product have me a rash that lasted for da...	0
4535979	1	This paper leaves on the skin traces of colore...	0
2347591	2	Just OK stick with the towel version The glove...	0
4810657	1	I had a this done a year ago I was happy when ...	0

2. Word Embedding

Part (a)

Word2Vec features are extracted using the pretrained dataset generated using the Gensim library and the semantic similarities are check for three example using the pretrained model '**Word2Vec_Pretrained_Model**'
Reference - https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html (https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html)

In [42]:

```
Word2Vec_Pretrained_Model = gensim.downloader.load("word2vec-google-news-300")
```

In [51]:

```
# Compute vector arithmetic
result1 = Word2Vec_Pretrained_Model.most_similar(positive=['Woman', 'King'], negative=['Man'], topn=
# result2 = Word2Vec_Pretrained_Model.most_similar(positive=['Madrid', 'France'], negative=['Spain'
# result3 = Word2Vec_Pretrained_Model.most_similar(positive=['Actor', 'Female'], negative=['Male'],

# Display the results
print("King - Man + Woman =", result1[0][0])
# print("Madrid - Spain + France =", result2[0][0])
# print("Actor - Male + Female =", result3[0][0])
```

```
King - Man + Woman = Queen
```

In [54]:

```
result4 = Word2Vec_Pretrained_Model.most_similar('great', topn=1)
result5 = Word2Vec_Pretrained_Model.most_similar('bad', topn=1)
print("great ~", result4[0][0])
print("bad ~", result5[0][0])
```

```
great ~ terrific
bad ~ good
```

Part (b)

Word2Vec features are extracted using the dataset generated. The semantic similarities are check for same three example as Part (a) using the trained model '**Word2Vec_Trained_Model**'

In [55]:

```
InputDataFrameFinalHW3.head()
```

Out[55]:

	star_rating	review_body	class
2133250	1	had to return did not include small travel bot...	0
847125	1	this product have me a rash that lasted for da...	0
4535979	1	This paper leaves on the skin traces of colore...	0
2347591	2	Just OK stick with the towel version The glove...	0
4810657	1	I had a this done a year ago I was happy when ...	0

Empty list called review_list to store the review texts is created. The function Loops over the rows of the dataframe '**InputDataFrameFinalHW3**' using the iterrows() function and appends the review_body column of each row to the review_list using the append() function. The review_list should contain all the review texts from the original dataframe.

In [56]:

```
# Empty list to store the review texts
review_list = []

# Loop over the rows of the dataframe and append the review texts to the list
for index, row in InputDataFrameFinalHW3.iterrows():
    review_list.append(row['review_body'])

# print(review_list[1])
```

review_list dataset is loaded as a list of strings and each document in the dataset is tokenized using the gensim.utils.simple_preprocess() function.

In [57]:

```
# Load the dataset - review_list

# Tokenize the dataset
tokenized_data = [gensim.utils.simple_preprocess(doc) for doc in review_list]

# print(tokenized_data[45])
```

Word2Vec model is trained on the tokenized data using the gensim.models.Word2Vec() function. Here, we set the size parameter to 300 to create 300-dimensional word embeddings, the window parameter to 13 to set the maximum distance between the current and predicted word within a sentence, and the min_count parameter to 9 to only consider words that appear at least nine time in the dataset.

In [58]:

```
# Train the Word2Vec model
Word2Vec_Trained_Model = gensim.models.Word2Vec(tokenized_data, vector_size=300, window=13, min_cou
```

In [59]:

```
# Compute vector arithmetic
result1 = Word2Vec_Trained_Model.wv.most_similar(positive=['woman', 'king'], negative=['man'], topn
# result2 = Word2Vec_Trained_Model.wv.most_similar(positive=['Madrid', 'France'], negative=['Spain'
# result3 = Word2Vec_Trained_Model.wv.most_similar(positive=['actor', 'female'], negative=['male'],

# Display the results
print("King - Man + Woman =", result1[0][0])
# print("Madrid - Spain + France =", result2[0][0])
# print("Actor - Male + Female =", result3[0][0])
```

```
King - Man + Woman = excelent
```

In [62]:

```
result4 = Word2Vec_Trained_Model.wv.most_similar('great', topn=1)
result5 = Word2Vec_Trained_Model.wv.most_similar('bad', topn=1)
print("great ~", result4[0][0])
print("bad ~", result5[0][0])
```

```
great ~ good
bad ~ terrible
```

What do you conclude from comparing vectors generated by yourself and the pretrained model?

The Word2Vec_Pretrained_Model is pre-trained on a large dataset and should have a more generalized understanding of word relationships, whereas the Word2Vec_Trained_Model is trained on a specific dataset and may have a more specialized understanding of word relationships specific to that dataset.

Which of the Word2Vec models seems to encode semantic similarities between words better?

Word2Vec_Pretrained_Model seems to encode semantic similarities better, as it produces the well-known semantic analogy: King - Man + Woman = Queen. In contrast, the Word2Vec_Trained_Model produces a less coherent result for the same analogy, i.e., King - Man + Woman = excellent, which doesn't make sense semantically.

Furthermore, based on the given information, the Word2Vec_Pretrained_Model also seems to encode semantic similarities better for the word pairs "great ~ terrific" and "bad ~ good," as it produces the expected relationships, whereas the Word2Vec_Trained_Model produces the opposite relationship for "bad ~ good," which is not ideal. Therefore, the Word2Vec_Pretrained_Model seems to be a better model for encoding semantic similarities between words.

3. Simple models

In [24]:

```
#  Load the data into a pandas dataframe
# - InputDataFrameFinalHW1
# - InputDataFrameFinalHW3

# Load the pre-trained Word2Vec model from Google News dataset
# - Word2Vec_Pretrained_Model
```

TF-IDF Feature

In [28]:

```
TF_IDF = TfidfVectorizer(ngram_range=(1,4))
X_tf = TF_IDF.fit_transform(InputDataFrameFinalHW1["review_body"])
Y_tf = InputDataFrameFinalHW1['class']
```

In [29]:

```
x_train_tf ,x_test_tf, y_train_tf, y_test_tf = train_test_split(X_tf, Y_tf, test_size=0.2, random_s
```

Word2Vec Feature

averageWord2VecVector()

The average Word2Vec vectors for each review as the input feature (x = $\frac{1}{N}\sum_{i=1}^{N} W_i$ for a review with N words) is calculated using the averageWord2VecVector() function

The loop iterates through each word in the review, splits the review string into individual words, and checks if each word is present in the pre-trained **Word2Vec_Pretrained_Model** model. If the word is present, its corresponding vector is added to the vectors list.

If there are no words in the review that are present in the pre-trained Word2Vec_Pretrained_Model model, then the function returns a zero vector with the same dimensionality as the Word2Vec_Pretrained_Model vectors (word2vec_dim=300).

Finally, the function returns the average vector of all the vectors in the vectors list. np.mean calculates the mean of all the vectors along the first axis (which corresponds to the individual dimensions of the vectors), resulting in a single 300-dimensional vector representing the review's average Word2Vec vector.

In [43]:

```
word2vec_dim=300

def averageWord2VecVector(review):
    vectors = []
    for word in review.split():
        if word in Word2Vec_Pretrained_Model:
            vectors.append(Word2Vec_Pretrained_Model[word])
    if len(vectors) == 0:
        return np.zeros(word2vec_dim)
    return np.mean(vectors, axis=0)
```

In [44]:

```
# Compute the average Word2Vec vectors for all reviews in the dataset
X = np.vstack(InputDataFrameFinalHW3['review_body'].apply(lambda x: averageWord2VecVector(x)))

# Define the target variable
Y = InputDataFrameFinalHW3['class'].values
```

In [45]:

```
X.shape
```

Out[45]:

```
(60000, 300)
```

In [46]:

```
Y.shape
```

Out[46]:

```
(60000,)
```

In [47]:

```
# Split the data into training and test sets random_state=42
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.2, random_state=100)
```

Perceptron (TF-IDF Feature)

In [30]:

```
model_perceptron = Perceptron()
model_perceptron.fit(x_train_tf,y_train_tf)
y_pred_tf = model_perceptron.predict(x_test_tf)
print("Perceptron accuracy (TF-IDF Feature):", accuracy_score(y_test_tf,y_pred_tf))
```

```
Perceptron accuracy (TF-IDF Feature): 0.72275
```

Perceptron (Word2Vec Feature)

In [48]:

```
model_perceptron = Perceptron()
model_perceptron.fit(x_train,y_train)
y_pred = model_perceptron.predict(x_test)
print("Perceptron accuracy (Word2Vec Feature):", accuracy_score(y_test,y_pred))
```

```
Perceptron accuracy (Word2Vec Feature): 0.54825
```

SVM (TF-IDF Feature)

In [32]:

```
model_svm = LinearSVC()
model_svm.fit(x_train_tf,y_train_tf)
y_pred_tf = model_svm.predict(x_test_tf)
print("SVM accuracy (TF-IDF Feature):", accuracy_score(y_test_tf,y_pred_tf))
```

```
SVM accuracy (TF-IDF Feature): 0.7465833333333334
```

SVM (Word2Vec Feature)

In [49]:

```
model_svm = LinearSVC()
model_svm.fit(x_train,y_train)
y_pred = model_svm.predict(x_test)
print("SVM accuracy (Word2Vec Feature):", accuracy_score(y_test,y_pred))
```

```
SVM accuracy (Word2Vec Feature): 0.6455833333333333
```

What do you conclude from comparing performances for the models trained using the two different feature types (TF-IDF and your trained Word2Vec features)?

The models trained using the TF-IDF feature outperform the models trained using the pretrained Word2Vec feature. The SVM classifier achieves an accuracy of 0.7465834 using the TF-IDF feature, which is about 10% higher than the accuracy achieved by the same classifier using the Word2Vec feature (0.645583). Similarly, the Perceptron classifier achieves an accuracy of 0.72275 using the TF-IDF feature, which is about 17% higher than the accuracy achieved by the same classifier using the Word2Vec feature (0.54825).

Hence, the TF-IDF feature is more effective than the pretrained Word2Vec feature in representing the sentiment of the amazon product reviews. It's important to note that the effectiveness of a feature representation can vary depending on the task and dataset, and it's possible that the Word2Vec feature may perform better on other sentiment analysis tasks or datasets.

4. Feedforward Neural Networks

In [67]:

```
#  Load the data into a pandas dataframe
# - InputDataFrameFinalHW3

# Load the pre-trained Word2Vec model from Google News dataset
# - Word2Vec_Pretrained_Model
```

Feedforward Multilayer Perceptron Network for classification is trained using Word2Vec_Pretrained_Model. The network with two hidden layers, each with 100 and 10 nodes, respectively is constructed.

MLP(nn.Module)

The **init** method is called when an instance of the MLP class is created. It initializes the model's layers, activation functions, and other parameters. Here, the method takes four arguments:

input_size - The size of the input vector for the model - 300
hidden_layer_1_size - The number of nodes in the first hidden layer - 100
hidden_layer_2_size - The number of nodes in the second hidden layer - 10
output_size The size of the output vector for the model - 3
The super() function is used to call the constructor of the parent class nn.Module, which initializes the module.

The fully connected layers (also known as linear layers) of the MLP model. nn.Linear is a PyTorch module that represents a linear transformation of the input data. Each nn.Linear object is assigned to an instance variable in the MLP class, self.fc1, self.fc2, and self.fc3, respectively.

The activation functions for the model. nn.ReLU() creates an instance of the rectified linear unit (ReLU) activation function, which applies the element-wise rectified linear function to the input data. nn.Softmax(dim=1) creates an instance of the softmax activation function, which applies a normalized exponential function to the input data to produce a probability distribution over the output classes.

forward(self, x)

This method defines how the input data x flows through the layers of the model. The forward method takes the input tensor x as its argument, and applies the fully connected layers and activation functions to it. The output tensor is then returned by the method. Specifically, the input tensor x is passed through the first hidden layer (self.fc1) and the ReLU activation function (self.relu), then through the second hidden layer (self.fc2) and the ReLU activation function, and finally through the output layer (self.fc3) and the softmax activation function (self.softmax). The resulting tensor is returned by the method.

Reference - https://www.kaggle.com/mishra1993/pytorch-multi-layer-perceptron-mnist (https://www.kaggle.com/mishra1993/pytorch-multi-layer-perceptron-mnist)

In [16]:

```
# Define the neural network architecture
# This defines a PyTorch Module called MLP, which will represent multi-layer perceptron (MLP) model
class MLP(nn.Module):
    def __init__(self, input_size, hidden_layer_1_size, hidden_layer_2_size, output_size):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_layer_1_size)
        self.fc2 = nn.Linear(hidden_layer_1_size, hidden_layer_2_size)
        self.fc3 = nn.Linear(hidden_layer_2_size, output_size)
        self.relu = nn.ReLU()
        # self.softmax = nn.Softmax(dim=1)

    def forward(self, x):
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        # x = self.softmax(x)
        return x
```

In [70]:

```
# Define a Dataset class for data
class Dataset(data.Dataset):
    def __init__(self, X, Y):
        self.X = X
        self.Y = Y

    def __getitem__(self, index):
        return torch.FloatTensor(self.X[index]), torch.FloatTensor(self.Y[index])

    def __len__(self):
        return len(self.X)
```

Part (a) - Average

The input features are generated, use the average Word2Vec vectors similar to the '**Simple models**' section

In [71]:

```
word2vec_dim=300

def averageWord2VecVector(review):
    vectors = []
    for word in review.split():
        if word in Word2Vec_Pretrained_Model:
            vectors.append(Word2Vec_Pretrained_Model[word])
    if len(vectors) == 0:
        return np.zeros(word2vec_dim)
    return np.mean(vectors, axis=0)
```

In [72]:

```
# Compute the average Word2Vec vectors for all reviews in the dataset
X = np.vstack(InputDataFrameFinalHW3['review_body'].apply(lambda x: averageWord2VecVector(x)))

# Define the target variable
Y = InputDataFrameFinalHW3['class'].values
```

In [73]:

```
X.shape
```

Out[73]:

```
(60000, 300)
```

In [74]:

```
Y
```

Out[74]:

```
array([0, 0, 0, ..., 2, 2, 2], dtype=int64)
```

In [75]:

```
# Perform one-hot encoding
Y_tensor = torch.tensor(Y)
Y_onehot = torch.nn.functional.one_hot(Y_tensor)
Y= Y_onehot.numpy()
```

In [76]:

```
Y
```

Out[76]:

```
array([[1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       ...,
       [0, 0, 1],
       [0, 0, 1],
       [0, 0, 1]], dtype=int64)
```

In [77]:

```
# Split the data into training and test sets
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.2, random_state=100)
```

In [17]:

```
# Define the hyperparameters
input_size = 300 # 300
hidden_layer_1_size = 100
hidden_layer_2_size = 10
output_size = 3
learning_rate = 0.001
num_workers = 6
batch_size = 64
num_epochs = 100
```

In [18]:

```
# Initialize the model, loss function, and optimizer
model = MLP(input_size, hidden_layer_1_size, hidden_layer_2_size, output_size)
print(model)
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

```
MLP(
  (fc1): Linear(in_features=300, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=10, bias=True)
  (fc3): Linear(in_features=10, out_features=3, bias=True)
  (relu): ReLU()
)
```

In [80]:

```
# Create instances of the Dataset class for the training and testing data
# Create DataLoader objects for the training and testing data

training_set = Dataset(x_train, y_train)
training_generator = data.DataLoader(training_set, batch_size = batch_size, shuffle = True)
testing_set = Dataset(x_test, y_test)
testing_generator = data.DataLoader(testing_set, batch_size=batch_size)
```

In [81]:

```
# Train the model
for epoch in range(num_epochs):
    running_loss = 0.0
    for inputs, labels in training_generator:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = loss_fn(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print('Epoch %d \t Training Loss: %.6f' % (epoch + 1, running_loss/len(training_generator)))
```

```
Epoch 1         Training Loss: 0.969062
Epoch 2         Training Loss: 0.904044
Epoch 3         Training Loss: 0.893738
Epoch 4         Training Loss: 0.888281
Epoch 5         Training Loss: 0.886398
Epoch 6         Training Loss: 0.883098
Epoch 7         Training Loss: 0.881297
Epoch 8         Training Loss: 0.879152
Epoch 9         Training Loss: 0.877107
Epoch 10        Training Loss: 0.875518
Epoch 11        Training Loss: 0.874424
Epoch 12        Training Loss: 0.872270
Epoch 13        Training Loss: 0.871687
Epoch 14        Training Loss: 0.869544
Epoch 15        Training Loss: 0.867678
Epoch 16        Training Loss: 0.866549
Epoch 17        Training Loss: 0.864641
Epoch 18        Training Loss: 0.864032
Epoch 19        Training Loss: 0.861388
Epoch 20        Training Loss: 0.859432
Epoch 21        Training Loss: 0.858595
Epoch 22        Training Loss: 0.857860
Epoch 23        Training Loss: 0.855605
Epoch 24        Training Loss: 0.854491
Epoch 25        Training Loss: 0.851912
Epoch 26        Training Loss: 0.851597
Epoch 27        Training Loss: 0.849830
Epoch 28        Training Loss: 0.847800
Epoch 29        Training Loss: 0.846442
Epoch 30        Training Loss: 0.844867
Epoch 31        Training Loss: 0.843248
Epoch 32        Training Loss: 0.841522
Epoch 33        Training Loss: 0.839250
Epoch 34        Training Loss: 0.837589
Epoch 35        Training Loss: 0.835953
Epoch 36        Training Loss: 0.834678
Epoch 37        Training Loss: 0.832780
Epoch 38        Training Loss: 0.830681
Epoch 39        Training Loss: 0.828928
Epoch 40        Training Loss: 0.827160
Epoch 41        Training Loss: 0.825529
Epoch 42        Training Loss: 0.824154
Epoch 43        Training Loss: 0.821762
Epoch 44        Training Loss: 0.819895
Epoch 45        Training Loss: 0.818884
Epoch 46        Training Loss: 0.816695
Epoch 47        Training Loss: 0.814809
Epoch 48        Training Loss: 0.813742
Epoch 49        Training Loss: 0.813971
Epoch 50        Training Loss: 0.810880
Epoch 51        Training Loss: 0.808421
Epoch 52        Training Loss: 0.807973
Epoch 53        Training Loss: 0.806545
Epoch 54        Training Loss: 0.804237
Epoch 55        Training Loss: 0.803231
Epoch 56        Training Loss: 0.802416
Epoch 57        Training Loss: 0.800652
Epoch 58        Training Loss: 0.799986
Epoch 59        Training Loss: 0.796794
Epoch 60        Training Loss: 0.796971
Epoch 61        Training Loss: 0.794583
Epoch 62        Training Loss: 0.792888
Epoch 63        Training Loss: 0.792471
Epoch 64        Training Loss: 0.790828
Epoch 65        Training Loss: 0.791018
Epoch 66        Training Loss: 0.788433
```

```
Epoch 67        Training Loss: 0.787657
Epoch 68        Training Loss: 0.785272
Epoch 69        Training Loss: 0.784601
Epoch 70        Training Loss: 0.783962
Epoch 71        Training Loss: 0.781805
Epoch 72        Training Loss: 0.782032
Epoch 73        Training Loss: 0.779557
Epoch 74        Training Loss: 0.780551
Epoch 75        Training Loss: 0.777851
Epoch 76        Training Loss: 0.776338
Epoch 77        Training Loss: 0.776182
Epoch 78        Training Loss: 0.776348
Epoch 79        Training Loss: 0.774145
Epoch 80        Training Loss: 0.772683
Epoch 81        Training Loss: 0.771658
Epoch 82        Training Loss: 0.770589
Epoch 83        Training Loss: 0.770590
Epoch 84        Training Loss: 0.769045
Epoch 85        Training Loss: 0.770361
Epoch 86        Training Loss: 0.769410
Epoch 87        Training Loss: 0.766801
Epoch 88        Training Loss: 0.766122
Epoch 89        Training Loss: 0.765456
Epoch 90        Training Loss: 0.764921
Epoch 91        Training Loss: 0.764369
Epoch 92        Training Loss: 0.763091
Epoch 93        Training Loss: 0.763989
Epoch 94        Training Loss: 0.762474
Epoch 95        Training Loss: 0.761444
Epoch 96        Training Loss: 0.761521
Epoch 97        Training Loss: 0.759038
Epoch 98        Training Loss: 0.759968
Epoch 99        Training Loss: 0.759930
Epoch 100       Training Loss: 0.758555
```

In [82]:

```
def Model_Evaluate(model, testing_generator):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in testing_generator:
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            _, labels = torch.max(labels, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print('Accuracy Feedforward Neural Networks (Average): %.6f' % (correct / total))
```

In [83]:

```
Model_Evaluate(model, testing_generator)
```

```
Accuracy Feedforward Neural Networks (Average): 0.660667
```

Part (b) - Concatenate

The input features are generated by concatenating the first 10 Word2Vec vectors for each review as the input feature ($x = [W_T^1, \ldots, W_T^{10}]$) and train the neural network.

concatenateWord2VecVector(review)

This function takes a review and the Word2Vec model as inputs and returns the input feature for the review. The function first splits the review into words and checks if each word is in the **Word2Vec_Pretrained_Model** model. If a word is in the model, the corresponding word vector is added to a vectors list. If the list has less than 10 vectors, it is padded with zero vectors. Finally, the first 10 vectors are concatenated into a single input feature vector.

The input features are constructed for all reviews in the dataset by applying the concatenateWord2VecVector function to each review in the 'review_body' column of the dataframe. The resulting input features are stacked vertically into a numpy array.

In [84]:

```
# Define a function to generate the input feature for a review
word2vec_dim = 300
review_length = 10

def concatenateWord2VecVector(review):
    vectors = []
    for word in review.split():
        if word in Word2Vec_Pretrained_Model:
            vectors.append(Word2Vec_Pretrained_Model[word])
    if len(vectors) < review_length:
        vectors += [np.zeros(word2vec_dim)] * (review_length - len(vectors))
    return np.concatenate(vectors[:review_length], axis=0)
```

In [85]:

```
# Generate the input features for all reviews in the dataset
X = np.vstack(InputDataFrameFinalHW3['review_body'].apply(lambda x: concatenateWord2VecVector(x)))

# Define the target variable
Y = InputDataFrameFinalHW3['class'].values

# Perform one-hot encoding
Y_tensor = torch.tensor(Y)
Y_onehot = torch.nn.functional.one_hot(Y_tensor)
Y = Y_onehot.numpy()
```

In [86]:

```
X.shape
```

Out[86]:

```
(60000, 3000)
```

In [87]:

```
Y.shape
```

Out[87]:

```
(60000, 3)
```

In [88]:

```
# Split the data into training and test sets
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.2, random_state=100)
```

In [19]:

```
# Define the hyperparameters
input_size = 3000 # 300 * 10
hidden_layer_1_size = 100
hidden_layer_2_size = 10
output_size = 3
learning_rate = 0.001
batch_size = 64
num_epochs = 100
```

In [90]:

```
# Create instances of the Dataset class for the training and testing data
# Create DataLoader objects for the training and testing data

training_set = Dataset(x_train, y_train)
training_generator = data.DataLoader(training_set, batch_size = batch_size, shuffle = True)
testing_set = Dataset(x_test, y_test)
testing_generator = data.DataLoader(testing_set, batch_size = batch_size)
```

In [20]:

```
# Initialize the model, loss function, and optimizer
model = MLP(input_size, hidden_layer_1_size, hidden_layer_2_size, output_size)
print(model)
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

```
MLP(
  (fc1): Linear(in_features=3000, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=10, bias=True)
  (fc3): Linear(in_features=10, out_features=3, bias=True)
  (relu): ReLU()
)
```

In [92]:

```
# Train the model
for epoch in range(num_epochs):
    running_loss = 0.0
    for inputs, labels in training_generator:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = loss_fn(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print('Epoch %d \t Training Loss: %.6f' % (epoch + 1, running_loss/len(training_generator)))
```

```
Epoch 1         Training Loss: 0.991617
Epoch 2         Training Loss: 0.938220
Epoch 3         Training Loss: 0.913522
Epoch 4         Training Loss: 0.889798
Epoch 5         Training Loss: 0.861253
Epoch 6         Training Loss: 0.831912
Epoch 7         Training Loss: 0.805143
Epoch 8         Training Loss: 0.781526
Epoch 9         Training Loss: 0.763044
Epoch 10        Training Loss: 0.750060
Epoch 11        Training Loss: 0.739314
Epoch 12        Training Loss: 0.730753
Epoch 13        Training Loss: 0.725482
Epoch 14        Training Loss: 0.722934
Epoch 15        Training Loss: 0.717653
Epoch 16        Training Loss: 0.714221
Epoch 17        Training Loss: 0.712332
Epoch 18        Training Loss: 0.709636
Epoch 19        Training Loss: 0.709847
Epoch 20        Training Loss: 0.706351
Epoch 21        Training Loss: 0.703935
Epoch 22        Training Loss: 0.701922
Epoch 23        Training Loss: 0.700064
Epoch 24        Training Loss: 0.700092
Epoch 25        Training Loss: 0.698028
Epoch 26        Training Loss: 0.697225
Epoch 27        Training Loss: 0.696344
Epoch 28        Training Loss: 0.695427
Epoch 29        Training Loss: 0.694209
Epoch 30        Training Loss: 0.693827
Epoch 31        Training Loss: 0.691924
Epoch 32        Training Loss: 0.690885
Epoch 33        Training Loss: 0.690948
Epoch 34        Training Loss: 0.690324
Epoch 35        Training Loss: 0.690382
Epoch 36        Training Loss: 0.689833
Epoch 37        Training Loss: 0.687816
Epoch 38        Training Loss: 0.687214
Epoch 39        Training Loss: 0.687058
Epoch 40        Training Loss: 0.686728
Epoch 41        Training Loss: 0.685815
Epoch 42        Training Loss: 0.684504
Epoch 43        Training Loss: 0.684581
Epoch 44        Training Loss: 0.685118
Epoch 45        Training Loss: 0.684330
Epoch 46        Training Loss: 0.682759
Epoch 47        Training Loss: 0.684158
Epoch 48        Training Loss: 0.682885
Epoch 49        Training Loss: 0.683139
Epoch 50        Training Loss: 0.681542
Epoch 51        Training Loss: 0.681358
Epoch 52        Training Loss: 0.680023
Epoch 53        Training Loss: 0.680851
Epoch 54        Training Loss: 0.680704
Epoch 55        Training Loss: 0.679377
Epoch 56        Training Loss: 0.679753
Epoch 57        Training Loss: 0.679794
Epoch 58        Training Loss: 0.678314
Epoch 59        Training Loss: 0.678333
Epoch 60        Training Loss: 0.678940
Epoch 61        Training Loss: 0.677255
Epoch 62        Training Loss: 0.676968
Epoch 63        Training Loss: 0.676314
Epoch 64        Training Loss: 0.676018
Epoch 65        Training Loss: 0.676588
Epoch 66        Training Loss: 0.677185
```

```
Epoch 67        Training Loss: 0.678946
Epoch 68        Training Loss: 0.675874
Epoch 69        Training Loss: 0.675460
Epoch 70        Training Loss: 0.676012
Epoch 71        Training Loss: 0.674379
Epoch 72        Training Loss: 0.675307
Epoch 73        Training Loss: 0.675314
Epoch 74        Training Loss: 0.673304
Epoch 75        Training Loss: 0.672854
Epoch 76        Training Loss: 0.674813
Epoch 77        Training Loss: 0.673954
Epoch 78        Training Loss: 0.673098
Epoch 79        Training Loss: 0.673512
Epoch 80        Training Loss: 0.672227
Epoch 81        Training Loss: 0.673151
Epoch 82        Training Loss: 0.671578
Epoch 83        Training Loss: 0.672107
Epoch 84        Training Loss: 0.672461
Epoch 85        Training Loss: 0.670870
Epoch 86        Training Loss: 0.671205
Epoch 87        Training Loss: 0.671387
Epoch 88        Training Loss: 0.671024
Epoch 89        Training Loss: 0.672044
Epoch 90        Training Loss: 0.670935
Epoch 91        Training Loss: 0.670806
Epoch 92        Training Loss: 0.669426
Epoch 93        Training Loss: 0.669923
Epoch 94        Training Loss: 0.669113
Epoch 95        Training Loss: 0.669808
Epoch 96        Training Loss: 0.668948
Epoch 97        Training Loss: 0.669379
Epoch 98        Training Loss: 0.669464
Epoch 99        Training Loss: 0.669266
Epoch 100       Training Loss: 0.668856
```

In [93]:

```
def Model_Evaluate(model, testing_generator):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in testing_generator:
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            _, labels = torch.max(labels, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print('Accuracy Accuracy Feedforward Neural Networks (Concatenate): %.6f' % (correct / total))
```

In [94]:

```
Model_Evaluate(model, testing_generator)
```

```
Accuracy Accuracy Feedforward Neural Networks (Concatenate): 0.562333
```

What do you conclude by comparing accuracy values you obtain with those obtained in the “Simple Models” section ?

The Feedforward Neural Networks (FNN) model using the word embeddings outperforms the "Simple Models" (Perceptron and SVM) using the same word embeddings in terms of accuracy. The Perceptron model achieves an accuracy of 0.54825, the SVM model achieves an accuracy of 0.6455834, whereas the Feedforward Neural Networks model using the averaging method achieves an accuracy of 0.660667. The FNN model is more effective than the Perceptron and SVM models in representing the sentiment of the reviews in this particular task and dataset.

The two different methods for combining word embeddings are being compared: averaging and concatenating. The accuracy of the Feedforward Neural Networks (FNN) model is reported for both methods, and the accuracy of the model using the averaging method (0.660667) is higher than the accuracy of the model using the concatenation method (0.562333).

The averaging method is more effective than the concatenation method in representing the sentiment of the product reviews in this particular task and dataset. The difference in accuracy between the two methods could be due to the fact that the averaging method captures the overall sentiment of the sentence by taking the mean of the word embeddings, whereas the concatenation method concatenates the word embeddings into a longer vector, which may not capture the overall sentiment as effectively.

5. Recurrent Neural Networks

In [95]:

```
#  Load the data into a pandas dataframe
# - InputDataFrameFinalHW3

# Load the pre-trained Word2Vec model from Google News dataset
# - Word2Vec_Pretrained_Model
```

Recurrent Neural Network (RNN) using **Word2Vec_Pretrained_Model** features for classification. For Training a simple RNN an RNN cell with the hidden state size of 20 is considered. To feed your data into our RNN, review length is limited the maximum 20 by truncating longer reviews and padding shorter reviews with a null value (0). To perform the preprocessing rnnWord2VecVector is created
Reference - https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html (https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html)

rnnWord2VecVector(review)

The loop iterates through each word in the review, splits the review string into individual words, and checks if each word is present in the pre-trained Word2Vec_Pretrained_Model model. If the word is present, its corresponding vector is added to the vectors list.
If the length of the vectors list is greater than review_length (20), the function truncates it to review_length by keeping only the first review_length vectors.
If the length of the vectors list is less than review_length, the function pads it with zero vectors of the same dimensionality as the Word2Vec vectors to make its length equal to review_length.
Finally, the function returns an array of vectors representing the review. The length of the array is review_length, and each element is a Word2Vec vector corresponding to a word in the review (or a zero vector if the word is not present in the pre-trained Word2Vec_Pretrained_Model model).

In [103]:

```
# Define a function to generate the input feature for a review

word2vec_dim = 300
review_length = 20

def rnnWord2VecVector(review):
    vectors = []
    for word in review.split():
        if word in Word2Vec_Pretrained_Model:
            vectors.append(Word2Vec_Pretrained_Model[word])
    if len(vectors) > review_length:
        vectors = vectors[:review_length]
    elif len(vectors) < review_length:
        vectors += [np.zeros(word2vec_dim)] * (review_length - len(vectors))
    return np.array(vectors)
```

In [105]:

```
# The input features for all reviews in the dataset
X = np.vstack(InputDataFrameFinalHW3['review_body'].apply(lambda x: rnnWord2VecVector(x)))
X = X.reshape((len(InputDataFrameFinalHW3), review_length, word2vec_dim))

# Define the target variable
Y = InputDataFrameFinalHW3['class'].values

# Perform one-hot encoding
Y_tensor = torch.tensor(Y)
Y_onehot = torch.nn.functional.one_hot(Y_tensor)
Y = Y_onehot.numpy()
```

In [106]:

```
X.shape
```

Out[106]:

```
(60000, 20, 300)
```

In [107]:

```
Y.shape
```

Out[107]:

```
(60000, 3)
```

In [108]:

```
Y
```

Out[108]:

```
array([[1, 0, 0],
       [1, 0, 0],
       [1, 0, 0],
       ...,
       [0, 0, 1],
       [0, 0, 1],
       [0, 0, 1]], dtype=int64)
```

In [109]:

```
# Define a Dataset class for our data
class Dataset(data.Dataset):
    def __init__(self, X, Y):
        self.X = X
        self.Y = Y

    def __getitem__(self, index):
        return torch.FloatTensor(self.X[index]), torch.FloatTensor(self.Y[index])

    def __len__(self):
        return len(self.X)
```

In [110]:

```
# Split the data into training and test sets
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.2, random_state=42)
```

In [111]:

```
# Create instances of the Dataset class for the training and testing data
# Create DataLoader objects for the training and testing data
batch_size = 64

training_set = Dataset(x_train, y_train)
training_generator = data.DataLoader(training_set, batch_size = batch_size, shuffle = True)
testing_set = Dataset(x_test, y_test)
testing_generator = data.DataLoader(testing_set, batch_size=batch_size)
```

Part (a) - Recurrent Neural Networks

In [112]:

```
# Define the recurrent neural network architecture
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        # Initialize hidden state
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size)
        # Forward propagate RNN
        output, _ = self.rnn(x, h0)
        # Decode the hidden state of the last time step
        output = self.fc(output[:, -1, :])
        return output
```

In [113]:

```
# Define the hyperparameters
input_size = 300 # 300 * 20
hidden_size = 20
num_layers = 1
output_size = 3
learning_rate = 0.001
num_epochs = 100
```

In [114]:

```
# Initialize the model, loss function, and optimizer
model = RNN(input_size, hidden_size, num_layers, output_size)
print(model)
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

```
RNN(
  (rnn): RNN(300, 20, batch_first=True)
  (fc): Linear(in_features=20, out_features=3, bias=True)
)
```

In [115]:

```
# Train the model
for epoch in range(num_epochs):
    running_loss = 0.0
    for inputs, labels in training_generator:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = loss_fn(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print('Epoch %d \t Training Loss: %.6f' % (epoch + 1, running_loss/len(training_generator)))
```

```
Epoch 1         Training Loss: 1.012442
Epoch 2         Training Loss: 0.947200
Epoch 3         Training Loss: 0.896272
Epoch 4         Training Loss: 0.869501
Epoch 5         Training Loss: 0.854658
Epoch 6         Training Loss: 0.847672
Epoch 7         Training Loss: 0.841372
Epoch 8         Training Loss: 0.833769
Epoch 9         Training Loss: 0.829456
Epoch 10        Training Loss: 0.824625
Epoch 11        Training Loss: 0.821032
Epoch 12        Training Loss: 0.816276
Epoch 13        Training Loss: 0.810729
Epoch 14        Training Loss: 0.809639
Epoch 15        Training Loss: 0.807310
Epoch 16        Training Loss: 0.807869
Epoch 17        Training Loss: 0.800729
Epoch 18        Training Loss: 0.798637
Epoch 19        Training Loss: 0.794758
Epoch 20        Training Loss: 0.795091
Epoch 21        Training Loss: 0.789704
Epoch 22        Training Loss: 0.785946
Epoch 23        Training Loss: 0.786505
Epoch 24        Training Loss: 0.782175
Epoch 25        Training Loss: 0.782911
Epoch 26        Training Loss: 0.777744
Epoch 27        Training Loss: 0.775959
Epoch 28        Training Loss: 0.775452
Epoch 29        Training Loss: 0.773066
Epoch 30        Training Loss: 0.769983
Epoch 31        Training Loss: 0.769847
Epoch 32        Training Loss: 0.769126
Epoch 33        Training Loss: 0.765537
Epoch 34        Training Loss: 0.761262
Epoch 35        Training Loss: 0.763123
Epoch 36        Training Loss: 0.760006
Epoch 37        Training Loss: 0.758837
Epoch 38        Training Loss: 0.755128
Epoch 39        Training Loss: 0.755949
Epoch 40        Training Loss: 0.752199
Epoch 41        Training Loss: 0.748793
Epoch 42        Training Loss: 0.747613
Epoch 43        Training Loss: 0.750846
Epoch 44        Training Loss: 0.745376
Epoch 45        Training Loss: 0.746813
Epoch 46        Training Loss: 0.741758
Epoch 47        Training Loss: 0.746276
Epoch 48        Training Loss: 0.742908
Epoch 49        Training Loss: 0.741629
Epoch 50        Training Loss: 0.748495
Epoch 51        Training Loss: 0.737899
Epoch 52        Training Loss: 0.738889
Epoch 53        Training Loss: 0.739005
Epoch 54        Training Loss: 0.738511
Epoch 55        Training Loss: 0.735105
Epoch 56        Training Loss: 0.733150
Epoch 57        Training Loss: 0.732751
Epoch 58        Training Loss: 0.734759
Epoch 59        Training Loss: 0.733019
Epoch 60        Training Loss: 0.730641
Epoch 61        Training Loss: 0.727878
Epoch 62        Training Loss: 0.727334
Epoch 63        Training Loss: 0.726205
Epoch 64        Training Loss: 0.730510
Epoch 65        Training Loss: 0.725871
Epoch 66        Training Loss: 0.724586
```

```
Epoch 67        Training Loss: 0.725887
Epoch 68        Training Loss: 0.721822
Epoch 69        Training Loss: 0.716466
Epoch 70        Training Loss: 0.719863
Epoch 71        Training Loss: 0.719421
Epoch 72        Training Loss: 0.721713
Epoch 73        Training Loss: 0.719197
Epoch 74        Training Loss: 0.715349
Epoch 75        Training Loss: 0.715734
Epoch 76        Training Loss: 0.714410
Epoch 77        Training Loss: 0.716169
Epoch 78        Training Loss: 0.711369
Epoch 79        Training Loss: 0.710989
Epoch 80        Training Loss: 0.710635
Epoch 81        Training Loss: 0.715226
Epoch 82        Training Loss: 0.709205
Epoch 83        Training Loss: 0.709120
Epoch 84        Training Loss: 0.712725
Epoch 85        Training Loss: 0.711025
Epoch 86        Training Loss: 0.706673
Epoch 87        Training Loss: 0.705504
Epoch 88        Training Loss: 0.706507
Epoch 89        Training Loss: 0.707053
Epoch 90        Training Loss: 0.711752
Epoch 91        Training Loss: 0.706707
Epoch 92        Training Loss: 0.703764
Epoch 93        Training Loss: 0.702694
Epoch 94        Training Loss: 0.703600
Epoch 95        Training Loss: 0.703202
Epoch 96        Training Loss: 0.702079
Epoch 97        Training Loss: 0.701186
Epoch 98        Training Loss: 0.706495
Epoch 99        Training Loss: 0.703194
Epoch 100       Training Loss: 0.698692
```

In [116]:

```
def Model_Evaluate(model, testing_generator):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in testing_generator:
            outputs = model(inputs)
            _, predicted = torch.max(outputs, dim=1)
            _, labels = torch.max(labels, dim=1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print('Accuracy Recurrent Neural Networks: %.6f' % (correct / total))
```

In [117]:

```
Model_Evaluate(model, testing_generator)
```

```
Accuracy Recurrent Neural Networks: 0.623833
```

What do you conclude by comparing accuracy values you obtain with those obtained with feedforward neural network models?

The Recurrent Neural Network (RNN) model is lower than the accuracy of the Feedforward Neural Network (FNN) models. The FNN models achieve an accuracy of 0.660667 using the averaging method. In contrast, the Recurrent Neural Network (RNN) model achieves a higher accuracy (0.623833) compared to the Feedforward Neural Network (FNN) model that uses concatenation method (accuracy of 0.562333).

However, it's important to note that the effectiveness of different models can vary depending on the task and dataset, and it's possible that the RNN model may perform better on other sentiment analysis tasks or datasets.

Part (b) - Recurrent Neural Networks (GRU cell)

In [118]:

```
# Define the neural network architecture with GRU cell
class GRU(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(GRU, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.gru = nn.GRU(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size)
        output, _ = self.gru(x, h0)
        output = self.fc(output[:, -1, :])
        return output
```

In [119]:

```
# Define the hyperparameters
input_size = 300 # 20 * 300
hidden_size = 20
num_layers = 1
output_size = 3
learning_rate = 0.001
num_epochs = 100
```

In [120]:

```
# Initialize the model, loss function, and optimizer
model = GRU(input_size, hidden_size, num_layers, output_size)
print(model)
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

```
GRU(
  (gru): GRU(300, 20, batch_first=True)
  (fc): Linear(in_features=20, out_features=3, bias=True)
)
```

In [121]:

```
# Train the model
for epoch in range(num_epochs):
    running_loss = 0.0
    for inputs, labels in training_generator:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = loss_fn(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print('Epoch %d \t Training Loss: %.6f' % (epoch + 1, running_loss/len(training_generator)))
```

```
Epoch 1         Training Loss: 0.957988
Epoch 2         Training Loss: 0.811878
Epoch 3         Training Loss: 0.776203
Epoch 4         Training Loss: 0.756927
Epoch 5         Training Loss: 0.741676
Epoch 6         Training Loss: 0.729803
Epoch 7         Training Loss: 0.718652
Epoch 8         Training Loss: 0.708747
Epoch 9         Training Loss: 0.700041
Epoch 10        Training Loss: 0.692355
Epoch 11        Training Loss: 0.685226
Epoch 12        Training Loss: 0.677274
Epoch 13        Training Loss: 0.669224
Epoch 14        Training Loss: 0.664576
Epoch 15        Training Loss: 0.657392
Epoch 16        Training Loss: 0.650452
Epoch 17        Training Loss: 0.645329
Epoch 18        Training Loss: 0.639979
Epoch 19        Training Loss: 0.634853
Epoch 20        Training Loss: 0.628883
Epoch 21        Training Loss: 0.625003
Epoch 22        Training Loss: 0.619841
Epoch 23        Training Loss: 0.613468
Epoch 24        Training Loss: 0.610178
Epoch 25        Training Loss: 0.604051
Epoch 26        Training Loss: 0.600960
Epoch 27        Training Loss: 0.595957
Epoch 28        Training Loss: 0.592083
Epoch 29        Training Loss: 0.586774
Epoch 30        Training Loss: 0.583204
Epoch 31        Training Loss: 0.579907
Epoch 32        Training Loss: 0.573618
Epoch 33        Training Loss: 0.570702
Epoch 34        Training Loss: 0.566138
Epoch 35        Training Loss: 0.565185
Epoch 36        Training Loss: 0.557722
Epoch 37        Training Loss: 0.555917
Epoch 38        Training Loss: 0.550937
Epoch 39        Training Loss: 0.547399
Epoch 40        Training Loss: 0.543049
Epoch 41        Training Loss: 0.539767
Epoch 42        Training Loss: 0.537474
Epoch 43        Training Loss: 0.533748
Epoch 44        Training Loss: 0.528361
Epoch 45        Training Loss: 0.525844
Epoch 46        Training Loss: 0.521466
Epoch 47        Training Loss: 0.519077
Epoch 48        Training Loss: 0.515237
Epoch 49        Training Loss: 0.510918
Epoch 50        Training Loss: 0.509838
Epoch 51        Training Loss: 0.505732
Epoch 52        Training Loss: 0.501264
Epoch 53        Training Loss: 0.498677
Epoch 54        Training Loss: 0.495516
Epoch 55        Training Loss: 0.492651
Epoch 56        Training Loss: 0.492592
Epoch 57        Training Loss: 0.488120
Epoch 58        Training Loss: 0.484016
Epoch 59        Training Loss: 0.482403
Epoch 60        Training Loss: 0.479262
Epoch 61        Training Loss: 0.475009
Epoch 62        Training Loss: 0.473592
Epoch 63        Training Loss: 0.468755
Epoch 64        Training Loss: 0.467171
Epoch 65        Training Loss: 0.465142
Epoch 66        Training Loss: 0.462384
```

```
Epoch 67        Training Loss: 0.460089
Epoch 68        Training Loss: 0.462849
Epoch 69        Training Loss: 0.458796
Epoch 70        Training Loss: 0.455339
Epoch 71        Training Loss: 0.451939
Epoch 72        Training Loss: 0.451605
Epoch 73        Training Loss: 0.447317
Epoch 74        Training Loss: 0.445424
Epoch 75        Training Loss: 0.443188
Epoch 76        Training Loss: 0.438841
Epoch 77        Training Loss: 0.442507
Epoch 78        Training Loss: 0.435662
Epoch 79        Training Loss: 0.433615
Epoch 80        Training Loss: 0.429910
Epoch 81        Training Loss: 0.430357
Epoch 82        Training Loss: 0.430183
Epoch 83        Training Loss: 0.424403
Epoch 84        Training Loss: 0.423044
Epoch 85        Training Loss: 0.421001
Epoch 86        Training Loss: 0.422558
Epoch 87        Training Loss: 0.418164
Epoch 88        Training Loss: 0.419339
Epoch 89        Training Loss: 0.418565
Epoch 90        Training Loss: 0.413254
Epoch 91        Training Loss: 0.408127
Epoch 92        Training Loss: 0.409080
Epoch 93        Training Loss: 0.412083
Epoch 94        Training Loss: 0.406501
Epoch 95        Training Loss: 0.406538
Epoch 96        Training Loss: 0.402153
Epoch 97        Training Loss: 0.398575
Epoch 98        Training Loss: 0.402369
Epoch 99        Training Loss: 0.399033
Epoch 100       Training Loss: 0.394391
```

In [122]:

```
def Model_Evaluate(model, testing_generator):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in testing_generator:
            outputs = model(inputs)
            _, predicted = torch.max(outputs, dim=1)
            _, labels = torch.max(labels, dim=1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print('Accuracy Recurrent Neural Networks (Gated Recurrent unit cell): %.6f' % (correct / total
```

In [123]:

```
Model_Evaluate(model, testing_generator)
```

```
Accuracy Recurrent Neural Networks (Gated Recurrent unit cell): 0.614833
```

What do you conclude by comparing accuracy values you obtain with those obtained with feedforward neural network models?

The Gated Recurrent Unit (GRU) model achieves a higher accuracy (0.614833) compared to the Feedforward Neural Network (FNN) model that uses the concatenation method (accuracy of 0.562333). However, the FNN model that uses the average method achieves a higher accuracy of 0.660667 compared to both the GRU and concatenation-based FNN models. However, as previously mentioned, the effectiveness of different models can vary depending on the task and

dataset.

Part (c) - Recurrent Neural Networks (LSTM unit cell)

In [134]:

```
# Define the LSTM neural network architecture
class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, output_size):
        super(LSTM, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.lstm = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size)
        output, _ = self.lstm(x, (h0, c0))
        output = self.fc(output[:, -1, :])
        return output
```

In [140]:

```
# Define the hyperparameters
input_size = 300 # 20 * 300
hidden_size = 20
num_layers = 1
output_size = 3
learning_rate = 0.001
num_epochs = 100
```

In [141]:

```
# Initialize the model, loss function, and optimizer
model = LSTM(input_size, hidden_size, num_layers, output_size)
print(model)
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```

```
LSTM(
  (lstm): LSTM(300, 20, batch_first=True)
  (fc): Linear(in_features=20, out_features=3, bias=True)
)
```

In [142]:

```
# Train the model

for epoch in range(num_epochs):
    running_loss = 0.0
    for inputs, labels in training_generator:
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = loss_fn(outputs, labels)
        loss.backward()
        optimizer.step()
        running_loss += loss.item()
    print('Epoch %d \t Training Loss: %.6f' % (epoch + 1, running_loss/len(training_generator)))
```

```
Epoch 1         Training Loss: 0.955152
Epoch 2         Training Loss: 0.836005
Epoch 3         Training Loss: 0.800330
Epoch 4         Training Loss: 0.777768
Epoch 5         Training Loss: 0.758508
Epoch 6         Training Loss: 0.745380
Epoch 7         Training Loss: 0.733737
Epoch 8         Training Loss: 0.722880
Epoch 9         Training Loss: 0.713477
Epoch 10        Training Loss: 0.704263
Epoch 11        Training Loss: 0.693429
Epoch 12        Training Loss: 0.688016
Epoch 13        Training Loss: 0.680503
Epoch 14        Training Loss: 0.672582
Epoch 15        Training Loss: 0.665070
Epoch 16        Training Loss: 0.659323
Epoch 17        Training Loss: 0.651528
Epoch 18        Training Loss: 0.644219
Epoch 19        Training Loss: 0.638402
Epoch 20        Training Loss: 0.631826
Epoch 21        Training Loss: 0.627155
Epoch 22        Training Loss: 0.620642
Epoch 23        Training Loss: 0.614215
Epoch 24        Training Loss: 0.608928
Epoch 25        Training Loss: 0.604388
Epoch 26        Training Loss: 0.598268
Epoch 27        Training Loss: 0.593018
Epoch 28        Training Loss: 0.586868
Epoch 29        Training Loss: 0.582748
Epoch 30        Training Loss: 0.576093
Epoch 31        Training Loss: 0.572630
Epoch 32        Training Loss: 0.565219
Epoch 33        Training Loss: 0.562517
Epoch 34        Training Loss: 0.557180
Epoch 35        Training Loss: 0.552393
Epoch 36        Training Loss: 0.548276
Epoch 37        Training Loss: 0.541716
Epoch 38        Training Loss: 0.537752
Epoch 39        Training Loss: 0.534650
Epoch 40        Training Loss: 0.530412
Epoch 41        Training Loss: 0.524608
Epoch 42        Training Loss: 0.520337
Epoch 43        Training Loss: 0.519939
Epoch 44        Training Loss: 0.511880
Epoch 45        Training Loss: 0.510461
Epoch 46        Training Loss: 0.504140
Epoch 47        Training Loss: 0.500175
Epoch 48        Training Loss: 0.496463
Epoch 49        Training Loss: 0.494161
Epoch 50        Training Loss: 0.489503
Epoch 51        Training Loss: 0.488891
Epoch 52        Training Loss: 0.484351
Epoch 53        Training Loss: 0.481915
Epoch 54        Training Loss: 0.477115
Epoch 55        Training Loss: 0.472646
Epoch 56        Training Loss: 0.471841
Epoch 57        Training Loss: 0.468486
Epoch 58        Training Loss: 0.463189
Epoch 59        Training Loss: 0.457424
Epoch 60        Training Loss: 0.456549
Epoch 61        Training Loss: 0.455135
Epoch 62        Training Loss: 0.453537
Epoch 63        Training Loss: 0.444415
Epoch 64        Training Loss: 0.445547
Epoch 65        Training Loss: 0.441859
Epoch 66        Training Loss: 0.439820
```

```
Epoch 67        Training Loss: 0.439644
Epoch 68        Training Loss: 0.432544
Epoch 69        Training Loss: 0.431664
Epoch 70        Training Loss: 0.427088
Epoch 71        Training Loss: 0.420421
Epoch 72        Training Loss: 0.426360
Epoch 73        Training Loss: 0.416108
Epoch 74        Training Loss: 0.418398
Epoch 75        Training Loss: 0.415948
Epoch 76        Training Loss: 0.411684
Epoch 77        Training Loss: 0.407255
Epoch 78        Training Loss: 0.410648
Epoch 79        Training Loss: 0.404858
Epoch 80        Training Loss: 0.401680
Epoch 81        Training Loss: 0.397976
Epoch 82        Training Loss: 0.398328
Epoch 83        Training Loss: 0.391327
Epoch 84        Training Loss: 0.390739
Epoch 85        Training Loss: 0.389554
Epoch 86        Training Loss: 0.389599
Epoch 87        Training Loss: 0.383531
Epoch 88        Training Loss: 0.380884
Epoch 89        Training Loss: 0.381205
Epoch 90        Training Loss: 0.379850
Epoch 91        Training Loss: 0.376009
Epoch 92        Training Loss: 0.377902
Epoch 93        Training Loss: 0.373447
Epoch 94        Training Loss: 0.370378
Epoch 95        Training Loss: 0.366742
Epoch 96        Training Loss: 0.370829
Epoch 97        Training Loss: 0.362051
Epoch 98        Training Loss: 0.357922
Epoch 99        Training Loss: 0.362277
Epoch 100       Training Loss: 0.357349
```

In [143]:

```
def Model_Evaluate(model, testing_generator):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in testing_generator:
            outputs = model(inputs)
            _, predicted = torch.max(outputs, dim=1)
            _, labels = torch.max(labels, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print('Accuracy Recurrent Neural Networks (LSTM unit cell): %.6f' % (correct / total))
```

In [144]:

```
Model_Evaluate(model, testing_generator)
```

```
Accuracy Recurrent Neural Networks (LSTM unit cell): 0.626667
```

What do you conclude by comparing accuracy values you obtain by GRU, LSTM, and simple RNN?

The LSTM outperforms GRU and simple RNN in terms of accuracy. The accuracy of the simple RNN model is the lowest among the three, while the accuracy of the GRU model is slightly lower than that of the LSTM model.

LSTM and GRU are more advanced RNN architectures that are specifically designed to address the vanishing gradient problem that can occur in simple RNNs. They have additional mechanisms such as gates that allow them to selectively retain or forget information from past inputs, which can be very helpful in processing long sequences of data. This often leads to better performance compared to simple RNNs. In the case of the specific task you are working on, it seems that LSTM was able to outperform both GRU and simple RNN, possibly because it was better able to capture the complex relationships between the text inputs and the corresponding labels.

This suggests that the LSTM model is better at capturing long-term dependencies and relationships between words in the input text, which is important for sentiment analysis tasks.