3725520208_HW2_CSCI544

February 7, 2024

Name: Anne Sai Venkata Naga Saketh USC ID: 3725520208 USC Email: annes@usc.edu NLP HW2

0.1 Import Statements

[21]:
```
# Importing necessary libraries
import pandas as pd  # For data manipulation and analysis
import numpy as np   # For numerical operations
import re            # Regular expressions for text processing
from bs4 import BeautifulSoup  # For HTML parsing
from sklearn.model_selection import train_test_split  # For splitting data into␣
 ↪training and testing sets
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import Perceptron
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
 ↪f1_score
from sklearn.naive_bayes import MultinomialNB
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression

import nltk          # Natural Language Toolkit for text processing
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
nltk.download('wordnet')  # Download WordNet data
nltk.download('stopwords')   # Download StopWords data

import warnings      # To handle warnings
warnings.filterwarnings("ignore")  # Ignore warnings for the remainder of the␣
 ↪code
warnings.filterwarnings("default")  # Set warnings back to default behavior

import torch
from torch.utils.data import Dataset, DataLoader, TensorDataset
from gensim.models import KeyedVectors
from sklearn.feature_extraction.text import CountVectorizer
from gensim.models import Word2Vec

from nltk.tokenize import word_tokenize
```

```
from tqdm import tqdm
# from tensorflow.keras.utils import to_categorical
from sklearn.linear_model import Perceptron
from sklearn.svm import LinearSVC

import torch.nn as nn
import torch.optim as optim
```

```
[nltk_data] Downloading package wordnet to
[nltk_data]     /Users/sakethanne/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/sakethanne/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

0.2 1. Data Set Generation

0.3 Read Data from the file

Ignoring the bad lines or the rows in the TSV file that contain in-correct data

[22]:
```
# Reading the data from the tsv (Amazon Kitchen dataset) file as a Pandas frame
full_data = pd.read_csv("./amazon_reviews_us_Office_Products_v1_00.tsv",␣
 ↪delimiter='\t', encoding='utf-8', error_bad_lines=False)
```

```
/var/folders/xx/d1tzxzhj3fzbmswz4z7m_40w0000gn/T/ipykernel_31112/1150709939.py:2
: FutureWarning: The error_bad_lines argument has been deprecated and will be
removed in a future version. Use on_bad_lines in the future.


  full_data = pd.read_csv("./amazon_reviews_us_Office_Products_v1_00.tsv",
delimiter='\t', encoding='utf-8', error_bad_lines=False)
Skipping line 20773: expected 15 fields, saw 22
Skipping line 39834: expected 15 fields, saw 22
Skipping line 52957: expected 15 fields, saw 22
Skipping line 54540: expected 15 fields, saw 22

Skipping line 80276: expected 15 fields, saw 22
Skipping line 96168: expected 15 fields, saw 22
Skipping line 96866: expected 15 fields, saw 22
Skipping line 98175: expected 15 fields, saw 22
Skipping line 112539: expected 15 fields, saw 22
Skipping line 119377: expected 15 fields, saw 22
Skipping line 120065: expected 15 fields, saw 22
Skipping line 124703: expected 15 fields, saw 22

Skipping line 134024: expected 15 fields, saw 22
```

```
Skipping line 153938: expected 15 fields, saw 22
Skipping line 156225: expected 15 fields, saw 22
Skipping line 168603: expected 15 fields, saw 22
Skipping line 187002: expected 15 fields, saw 22

Skipping line 200397: expected 15 fields, saw 22
Skipping line 203809: expected 15 fields, saw 22
Skipping line 207680: expected 15 fields, saw 22
Skipping line 223421: expected 15 fields, saw 22
Skipping line 244032: expected 15 fields, saw 22

Skipping line 270329: expected 15 fields, saw 22
Skipping line 276484: expected 15 fields, saw 22
Skipping line 304755: expected 15 fields, saw 22

Skipping line 379449: expected 15 fields, saw 22
Skipping line 386191: expected 15 fields, saw 22
Skipping line 391811: expected 15 fields, saw 22

Skipping line 414348: expected 15 fields, saw 22
Skipping line 414773: expected 15 fields, saw 22
Skipping line 417572: expected 15 fields, saw 22
Skipping line 419496: expected 15 fields, saw 22
Skipping line 430528: expected 15 fields, saw 22
Skipping line 442230: expected 15 fields, saw 22
Skipping line 450931: expected 15 fields, saw 22

Skipping line 465377: expected 15 fields, saw 22
Skipping line 467685: expected 15 fields, saw 22
Skipping line 485055: expected 15 fields, saw 22
Skipping line 487220: expected 15 fields, saw 22
Skipping line 496076: expected 15 fields, saw 22
Skipping line 512269: expected 15 fields, saw 22

Skipping line 529505: expected 15 fields, saw 22
Skipping line 531286: expected 15 fields, saw 22
Skipping line 535424: expected 15 fields, saw 22
Skipping line 569898: expected 15 fields, saw 22
Skipping line 586293: expected 15 fields, saw 22

Skipping line 593880: expected 15 fields, saw 22
Skipping line 599274: expected 15 fields, saw 22
Skipping line 607961: expected 15 fields, saw 22
Skipping line 612413: expected 15 fields, saw 22
Skipping line 615913: expected 15 fields, saw 22

Skipping line 677580: expected 15 fields, saw 22
Skipping line 687191: expected 15 fields, saw 22
```

```
Skipping line 710819: expected 15 fields, saw 22

Skipping line 728692: expected 15 fields, saw 22
Skipping line 730216: expected 15 fields, saw 22
Skipping line 758397: expected 15 fields, saw 22
Skipping line 760061: expected 15 fields, saw 22
Skipping line 768935: expected 15 fields, saw 22
Skipping line 769483: expected 15 fields, saw 22

Skipping line 822725: expected 15 fields, saw 22
Skipping line 823621: expected 15 fields, saw 22

Skipping line 857041: expected 15 fields, saw 22
Skipping line 857320: expected 15 fields, saw 22
Skipping line 858565: expected 15 fields, saw 22
Skipping line 860629: expected 15 fields, saw 22
Skipping line 864033: expected 15 fields, saw 22
Skipping line 868673: expected 15 fields, saw 22
Skipping line 869189: expected 15 fields, saw 22

Skipping line 938605: expected 15 fields, saw 22
Skipping line 940100: expected 15 fields, saw 22
Skipping line 975137: expected 15 fields, saw 22
Skipping line 976314: expected 15 fields, saw 22

Skipping line 985597: expected 15 fields, saw 22
Skipping line 990873: expected 15 fields, saw 22
Skipping line 991806: expected 15 fields, saw 22
Skipping line 1019808: expected 15 fields, saw 22
Skipping line 1021526: expected 15 fields, saw 22
Skipping line 1023905: expected 15 fields, saw 22
Skipping line 1044207: expected 15 fields, saw 22

Skipping line 1084683: expected 15 fields, saw 22
Skipping line 1093288: expected 15 fields, saw 22

Skipping line 1136430: expected 15 fields, saw 22
Skipping line 1139815: expected 15 fields, saw 22

Skipping line 1179821: expected 15 fields, saw 22
Skipping line 1195351: expected 15 fields, saw 22
Skipping line 1202007: expected 15 fields, saw 22
Skipping line 1224868: expected 15 fields, saw 22
Skipping line 1232490: expected 15 fields, saw 22
Skipping line 1238697: expected 15 fields, saw 22

Skipping line 1258654: expected 15 fields, saw 22
Skipping line 1279948: expected 15 fields, saw 22
```

```
Skipping line 1294360: expected 15 fields, saw 22
Skipping line 1302240: expected 15 fields, saw 22

Skipping line 1413654: expected 15 fields, saw 22

Skipping line 1687095: expected 15 fields, saw 22

Skipping line 1805966: expected 15 fields, saw 22

Skipping line 1892134: expected 15 fields, saw 22

/var/folders/xx/d1tzxzhj3fzbmswz4z7m_40w0000gn/T/ipykernel_31112/1150709939.py:2
: DtypeWarning: Columns (7) have mixed types. Specify dtype option on import or
set low_memory=False.
  full_data = pd.read_csv("./amazon_reviews_us_Office_Products_v1_00.tsv",
delimiter='\t', encoding='utf-8', error_bad_lines=False)
```

0.4 Extract only Review and Ratings

[23]:
```
# Printing the data frame that contains the entire dataset from the tsv file
print(full_data)

# Keep only the Reviews and Ratings fields from the full data
df = full_data[['review_body', 'star_rating']]

# Converting 'star_rating' to numeric values
df['star_rating'] = pd.to_numeric(full_data['star_rating'], errors='coerce')
```

```
        marketplace  customer_id       review_id  product_id  product_parent  \
0                US     43081963  R18RVCKGH1SSI9  B001BM2MAC       307809868
1                US     10951564  R3L4L6LW1PUOFY  B00DZYEXPQ        75004341
2                US     21143145  R2J8AWXWTDX2TF  B00RTMUHDW       529689027
3                US     52782374  R1PR37BR7G3M6A  B00D7H8XB6       868449945
4                US     24045652  R3BDDDZMZBZDPU  B001XCWP34        33521401
...             ...          ...             ...         ...             ...
2640249          US     53005790   RLI7EI10S7SN0  B00000DM9M       223408988
2640250          US     52188548  R1F3SRK9MHE6A3  B00000DM9M       223408988
2640251          US     52090046  R23V0C4NRJL8EM  0807865001       307284585
2640252          US     52503173  R13ZAE1ATEUC1T  1572313188       870359649
2640253          US     52585611   RE8J5O2GY04NN  1572313188       870359649

                                              product_title product_category  \
0           Scotch Cushion Wrap 7961, 12 Inches x 100 Feet  Office Products
1                Dust-Off Compressed Gas Duster, Pack of 4  Office Products
2         Amram Tagger Standard Tag Attaching Tagging Gu...  Office Products
3         AmazonBasics 12-Sheet High-Security Micro-Cut ...  Office Products
4         Derwent Colored Pencils, Inktense Ink Pencils,...  Office Products
```

```
...                                                    ...               ...
2640249                PalmOne III Leather Belt Clip Case  Office Products
2640250                PalmOne III Leather Belt Clip Case  Office Products
2640251                 Gods and Heroes of Ancient Greece  Office Products
2640252  Microsoft EXCEL 97/ Visual Basic Step-by-Step ...  Office Products
2640253  Microsoft EXCEL 97/ Visual Basic Step-by-Step ...  Office Products

        star_rating  helpful_votes  total_votes vine verified_purchase  \
0                 5            0.0          0.0    N                 Y
1                 5            0.0          1.0    N                 Y
2                 5            0.0          0.0    N                 Y
3                 1            2.0          3.0    N                 Y
4                 4            0.0          0.0    N                 Y
...             ...            ...          ...  ...               ...
2640249           4           26.0         26.0    N                 N
2640250           4           18.0         18.0    N                 N
2640251           4            9.0         16.0    N                 N
2640252           5            0.0          0.0    N                 N
2640253           5            0.0          0.0    N                 N

                                        review_headline  \
0                                           Five Stars
1        Phfffffffft, Phfffffft. Lots of air, and it's C...
2                          but I am sure I will like it.
3        and the shredder was dirty and the bin was par...
4                                           Four Stars
...                                                    ...
2640249  Great value! A must if you hate to carry thing...
2640250          Attaches the Palm Pilot like an appendage
2640251  Excellent information, pictures and stories, I...
2640252                                       class text
2640253                              Microsoft's Finest

                                          review_body review_date
0                                       Great product.  2015-08-31
1        What's to say about this commodity item except...  2015-08-31
2          Haven't used yet, but I am sure I will like it.  2015-08-31
3        Although this was labeled as &#34;new&#34; the...  2015-08-31
4                          Gorgeous colors and easy to use  2015-08-31
...                                                    ...         ...
2640249  I can't live anymore whithout my Palm III. But...  1998-12-07
2640250  Although the Palm Pilot is thin and compact it...  1998-11-30
2640251  This book had a lot of great content without b...  1998-10-15
2640252  I am teaching a course in Excel and am using t...  1998-08-22
2640253  A very comprehensive layout of exactly how Vis...  1998-07-15

[2640254 rows x 15 columns]
```

```
/var/folders/xx/d1tzxzhj3fzbmswz4z7m_40w0000gn/T/ipykernel_31112/79198575.py:8:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df['star_rating'] = pd.to_numeric(full_data['star_rating'], errors='coerce')
```

0.5 Prepare a Balanced Data Set and prepare a Testing and Training Set

From the dataset we have extracted in the previous step, creating a balanced data set for each of the rating that we have from 1 to 5. Then I had added the sentiment column based on the ratings that we have and then I have split it into training and testing datasets.

[24]:
```python
# Check unique values in 'star_rating' column
unique_ratings = df['star_rating'].unique()
# print("Unique ratings:", unique_ratings)

# Convert 'star_rating' column to integer, handling errors
df['star_rating'] = pd.to_numeric(df['star_rating'], errors='coerce')

# Drop rows with NaN values in 'star_rating' column
df = df.dropna(subset=['star_rating'])

# Convert 'star_rating' column to integer
df['star_rating'] = df['star_rating'].astype(int)

# Define a custom dataset class
class AmazonReviewsDataset(Dataset):
    def __init__(self, reviews, ratings):
        self.reviews = reviews
        self.ratings = ratings

    def __len__(self):
        return len(self.reviews)

    def __getitem__(self, idx):
        review = self.reviews[idx]
        rating = self.ratings[idx]
        return review, rating

# Build balanced dataset
ratings = df['star_rating'].unique()
balanced_data = pd.DataFrame(columns=df.columns)
for rating in ratings:
    subset = df[df['star_rating'] == rating]
```

```
    if len(subset) >= 50000:
        subset = subset.sample(n=50000, random_state=42)
    balanced_data = pd.concat([balanced_data, subset])

# Create ternary labels
balanced_data['sentiment'] = np.where(balanced_data['star_rating'] > 3, 1,
                                      np.where(balanced_data['star_rating'] < 3,␣
 ↪2, 3))

print("Checking if the Dataset has been balanced out:\n")
print("Star_rating  Count")
print(balanced_data['star_rating'].value_counts())

print("Checking if the Sentiments has been balanced out:\n")
print("Sentiment  Count")
print(balanced_data['sentiment'].value_counts())

# Perform train-test split
train_data, test_data = train_test_split(balanced_data, test_size=0.2,␣
 ↪random_state=42)

# Define train and test datasets
train_dataset = AmazonReviewsDataset(train_data['review_body'].values,␣
 ↪train_data['sentiment'].values)
test_dataset = AmazonReviewsDataset(test_data['review_body'].values,␣
 ↪test_data['sentiment'].values)

# Define DataLoader
batch_size = 64
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)

# Print sizes of train and test datasets
print("\nPrinting the Test and the Training set data sizes")
print("Train dataset size:", len(train_dataset))
print("Test dataset size:", len(test_dataset))
```

```
/var/folders/xx/d1tzxzhj3fzbmswz4z7m_40w0000gn/T/ipykernel_31112/2411086817.py:6
: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  df['star_rating'] = pd.to_numeric(df['star_rating'], errors='coerce')
Checking if the Dataset has been balanced out:
```

```
Star_rating  Count
5    50000
1    50000
4    50000
2    50000
3    50000
Name: star_rating, dtype: int64
Checking if the Sentiments has been balanced out:

Sentiment  Count
1    100000
2    100000
3     50000
Name: sentiment, dtype: int64

Printing the Test and the Training set data sizes
Train dataset size: 200000
Test dataset size: 50000
```

0.6 Split the Training and Testing Data Set

[25]:
```
# Splitting the dataset into 80% training and 20% testing
X_train, X_test, y_train, y_test = train_test_split(balanced_data['review_body'],
                                                    balanced_data['sentiment'],
                                                    test_size=0.2,
                                                    random_state=42)
```

0.7 Clean the Data

Cleaning the data we have split.

1. Removing Contractions
2. Removing the unnecessary characters
3. Remving any HTML links
4. Converting to lower cases

[26]:
```
# Define a contraction map
CONTRACTION_MAP = {
    "won't": "will not",
    "can't": "cannot",
    "i'm": "i am",
    "you're": "you are",
    "he's": "he is",
    "she's": "she is",
    "it's": "it is",
    "that's": "that is",
```

```
    "we're": "we are",
    "they're": "they are",
    "isn't": "is not",
    "aren't": "are not",
    "haven't": "have not",
    "hasn't": "has not",
    "didn't": "did not",
    "doesn't": "does not",
    "don't": "do not",
    "wasn't": "was not",
    "weren't": "were not",
    "haven't": "have not",
    "hasn't": "has not",
    "won't've": "will not have",
    "can't've": "cannot have",
    "i'll": "i will",
    "you'll": "you will",
    "he'll": "he will",
    "she'll": "she will",
    "it'll": "it will",
    "that'll": "that will",
    "we'll": "we will",
    "they'll": "they will",
    "i'd": "i would",
    "you'd": "you would",
    "he'd": "he would",
    "she'd": "she would",
    "it'd": "it would",
    "that'd": "that would",
    "we'd": "we would",
    "they'd": "they would",
    "i've": "i have",
    "you've": "you have",
    "we've": "we have",
    "they've": "they have",
    "shouldn't": "should not",
    "couldn't": "could not",
    "wouldn't": "would not",
    "mightn't": "might not",
    "mustn't": "must not",
    "shan't": "shall not",
    "oughtn't": "ought not",
    "who's": "who is",
    "what's": "what is",
    "where's": "where is",
    "when's": "when is",
    "why's": "why is",
```

```
    "how's": "how is",
    "it's": "it is",
    "let's": "let us"
}

# Function to expand contractions
def expand_contractions(text):
    for contraction, expansion in CONTRACTION_MAP.items():
        text = re.sub(contraction, expansion, text)
    return text

# Preprocess the reviews
def preprocess_reviews(reviews):
    # Convert to lowercase and handle NaN values
    reviews = reviews.apply(lambda x: str(x).lower() if pd.notna(x) else '')

    # Remove HTML and URLs
    reviews = reviews.apply(lambda x: BeautifulSoup(x, 'html.parser').get_text())
    reviews = reviews.apply(lambda x: re.sub(r'http\S+', '', x))

    # Remove non-alphabetical characters (excluding single quote)
    reviews = reviews.apply(lambda x: re.sub(r'[^a-zA-Z\s\']', '', x))

    # Remove extra spaces
    reviews = reviews.apply(lambda x: re.sub(' +', ' ', x))

    # Perform contractions
    reviews = reviews.apply(expand_contractions)

    # Return the processed text of the review
    return reviews

# Preprocess the training set
X_train_preprocessed = preprocess_reviews(X_train)

# Print average length of reviews before and after cleaning
avg_length_before = X_train.apply(lambda x: len(str(x))).mean()
avg_length_after = X_train_preprocessed.apply(len).mean()
print("====================Printing the Average lenght of Reviews Before and␣
 ↪After Cleaning====================")
print(f"\nAverage Length of Reviews (Before Cleaning): {int(avg_length_before)}␣
 ↪characters")
print(f"Average Length of Reviews (After Cleaning): {int(avg_length_after)}␣
 ↪characters")
```

```
/var/folders/xx/d1tzxzhj3fzbmswz4z7m_40w0000gn/T/ipykernel_31112/3895173410.py:7
5: MarkupResemblesLocatorWarning: The input looks more like a filename than
```

```
markup. You may want to open this file and pass the filehandle into Beautiful
Soup.
  reviews = reviews.apply(lambda x: BeautifulSoup(x, 'html.parser').get_text())
/var/folders/xx/d1tzxzhj3fzbmswz4z7m_40w0000gn/T/ipykernel_31112/3895173410.py:7
5: MarkupResemblesLocatorWarning: The input looks more like a URL than markup.
You may want to use an HTTP client like requests to get the document behind the
URL, and feed that document to Beautiful Soup.
  reviews = reviews.apply(lambda x: BeautifulSoup(x, 'html.parser').get_text())

==================Printing the Average lenght of Reviews Before and After
Cleaning====================

Average Length of Reviews (Before Cleaning): 343 characters
Average Length of Reviews (After Cleaning): 326 characters
```

0.8 Pre-Process the Data

Using NLTK to remove the stop words that are unnecessary for sentiment classification.

[27]:
```
# Initialize NLTK's stopwords and WordNet lemmatizer
stop_words = set(stopwords.words('english'))
lemmatizer = WordNetLemmatizer()

# Function to remove stop words and perform lemmatization
def preprocess_nltk(review):
    if pd.notna(review):
        words = nltk.word_tokenize(str(review).lower())  # Convert to lowercase
        words = [lemmatizer.lemmatize(word) for word in words if word.isalpha()␣
 ↪and word not in stop_words]
        return ' '.join(words)
    else:
        return ''

# Preprocess the training set using NLTK
X_train_nltk_preprocessed = X_train_preprocessed.apply(preprocess_nltk)

# Print three sample reviews before and after NLTK preprocessing
sample_reviews_indices = X_train_preprocessed.sample(3).index

print("============ Printing Sample Reviews Before and After Pre-processing␣
 ↪=============")
for index in sample_reviews_indices:
    print(f"\nSample Review {index} Before Pre-processing:")
    print(X_train_preprocessed.loc[index])

    print(f"\nSample Review {index} After NLTK Pre-processing:")
    print(X_train_nltk_preprocessed.loc[index])
```

```
# Print average length of reviews before and after NLTK processing
avg_length_before_nltk = X_train_preprocessed.apply(len).mean()
avg_length_after_nltk = X_train_nltk_preprocessed.apply(len).mean()
print("\n=================Printing the Average lenght of Reviews Before and␣
 ↪After Pre-processing==================")
print(f"\nAverage Length of Reviews (Before NLTK Processing):␣
 ↪{int(avg_length_before_nltk)} characters")
print(f"Average Length of Reviews (After NLTK Processing):␣
 ↪{int(avg_length_after_nltk)} characters")
```

```
============ Printing Sample Reviews Before and After Pre-processing
=============

Sample Review 1958949 Before Pre-processing:
this is a good ink pad it seems to be wearing a little sooner than i expected
but it still inks everything so no complaints

Sample Review 1958949 After NLTK Pre-processing:
good ink pad seems wearing little sooner expected still ink everything complaint

Sample Review 983035 Before Pre-processing:
it is fine and will function well but i expected a little more pizazz for the
price just a plain black matt

Sample Review 983035 After NLTK Pre-processing:
fine function well expected little pizazz price plain black matt

Sample Review 2320561 Before Pre-processing:
this was a pretty cheap scale i wanted to weigh envelopes and packages at work
so i could just use stamps instead of waiting in line at the post office it does
the job and seems to weigh accurately as i have not had anything returned
without enough postage i am using it with batteries and have done so for about
months without having to change them although i only use it a couple times a
week so hopefully the batteries will last a good long while the power cord it
comes with is kind of short it would be nice if you could program it with
shipping charges so you could see the price for shipping rather than just the
weight but that is probably a feature that would add more cost to it every time
i turn it off it defaults to grams and i weigh in ounces so i wish it would stay
on my selection but that is only a minor inconvenience overall nothing fancy but
it is cheap and does the job of weighing stuff

Sample Review 2320561 After NLTK Pre-processing:
pretty cheap scale wanted weigh envelope package work could use stamp instead
waiting line post office job seems weigh accurately anything returned without
enough postage using battery done month without change although use couple time
week hopefully battery last good long power cord come kind short would nice
```

```
could program shipping charge could see price shipping rather weight probably
feature would add cost every time turn default gram weigh ounce wish would stay
selection minor inconvenience overall nothing fancy cheap job weighing stuff

=================Printing the Average lenght of Reviews Before and After Pre-
processing==================

Average Length of Reviews (Before NLTK Processing): 326 characters
Average Length of Reviews (After NLTK Processing): 201 characters
```

2. Word Embedding

[28]:
```
!pip install gensim
```

0.9 Similarity Score using the Word2Vec Pretrained model

Since we have to use binary data for Simple Models(Perceptron and SVM), so we need to remove the neutral reviews with class 3 and keep only Class 1 and Class 2.

For Simple models classification, we need to change the class labels from 1 and 2 to 0 and 1, as the class labels should always start from '0'

[29]:
```
# Joining features and targets for training dataset
train_data = pd.concat([X_train_nltk_preprocessed, y_train], axis=1)
train_data_filtered = train_data[train_data['sentiment'] != 3]
X_train_binary = train_data_filtered['review_body']
y_train_binary = train_data_filtered['sentiment']

# Joining features and targets for testing dataset
test_data = pd.concat([X_test, y_test], axis=1)
test_data_filtered = test_data[test_data['sentiment'] != 3]
X_test_binary = test_data_filtered['review_body']
y_test_binary = test_data_filtered['sentiment']
```

Download the Pre-trained Model from Google.

[59]:
```
import gensim.downloader as api
wv = api.load('word2vec-google-news-300')
```

[60]:
```
# Function to extract Word2Vec features for a given sentence
def extract_word2vec_features(sentence, model, vector_size):
    word_vectors = []
    for word in sentence:
        if word in model.key_to_index:
            word_vectors.append(model.get_vector(word))
    if len(word_vectors) == 0:
        return np.zeros(vector_size)  # Return zero vector if no word vectors␣
↪found
```

```
    else:
        return np.mean(word_vectors, axis=0)  # Return average word vector

# Examples to check semantic similarities
example1 = wv.most_similar(positive=['king', 'woman'], negative=['man'], topn=1)
print("King - Man + Woman =", example1)

example2 = wv.most_similar(positive=['excellent', 'outstanding'], topn=1)
print("Excellent ~ Outstanding =", example2)

example3 = wv.most_similar(positive=['marriage', 'woman'], negative=['man'],␣
↪topn=1)
print("Marriage - Man + Woman =", example3)

example4 = wv.most_similar(positive=['good'], negative=['bad'], topn=1)
print("Good ~ Bad =", example4)
```

```
King - Man + Woman = [('queen', 0.7118192911148071)]
Excellent ~ Outstanding = [('oustanding', 0.750198483467102)]
Marriage - Man + Woman = [('marriages', 0.7118672728538513)]
Good ~ Bad = [('excellent', 0.46134573221206665)]
```

0.10 Similarity Score using the Word2Vec model with the custom model

[32]:
```
X_train_nltk_preprocessed_new = []

for e, k in tqdm(enumerate(X_train_nltk_preprocessed.to_list())):
    try:
        X_train_nltk_preprocessed_new.append(word_tokenize(k))
    except:
        pass
def get_doc_embedding(doc):
    words = doc.lower().split()
    return wv.get_mean_vector(words)
```

```
200000it [00:21, 9324.76it/s]
```

Customizing the Pre-trained model and making it custom trained with the data we have from the Reviews

[61]:
```
from gensim.models import Word2Vec
# Train the Word2Vec model
model_own = Word2Vec(X_train_nltk_preprocessed_new, vector_size=300, window=11,␣
↪min_count=10)
```

[62]:
```
# Function to extract Word2Vec features for a given sentence using the trained␣
↪model
```

```
def extract_word2vec_features_own(sentence, model, vector_size):
    word_vectors = []
    for word in sentence:
        if word in model.wv.key_to_index:
            word_vectors.append(model.wv.get_vector(word))
    if len(word_vectors) == 0:
        return np.zeros(vector_size)  # Return zero vector if no word vectors␣
 ↪found
    else:
        return np.mean(word_vectors, axis=0)  # Return average word vector

# Examples to check semantic similarities using your own model
if 'king' in model_own.wv.key_to_index and 'woman' in model_own.wv.key_to_index:
    example1_own = model_own.wv.most_similar(positive=['king', 'woman'],␣
 ↪negative=['man'], topn=1)
    print("King - Man + Woman (Own Model) =", example1_own)
else:
    print("'good' is not present in the vocabulary.")

if 'excellent' in model_own.wv.key_to_index and 'outstanding' in model_own.wv.
 ↪key_to_index:
    example2_own = model_own.wv.most_similar(positive=['excellent',␣
 ↪'outstanding'], topn=1)
    print("Excellent ~ Outstanding (Own Model) =", example2_own)
else:
    print("'excellent' and/or 'outstanding' are not present in the vocabulary.")

if 'marriage' in model_own.wv.key_to_index and 'woman' in model_own.wv.
 ↪key_to_index:
    example3_own = model_own.wv.most_similar(positive=['marriage', 'woman'],␣
 ↪negative=['man'], topn=1)
    print("Marriage - Man + Woman (Own Model) =", example3_own)
else:
    print("'marriage' and/or 'woman' are not present in the vocabulary.")

if 'good' in model_own.wv.key_to_index:
    example4_own = model_own.wv.most_similar(positive=['good'],␣
 ↪negative=['bad'], topn=1)
    print("Good ~ Bad (Own Model) =", example4_own)
else:
    print("'marriage' and/or 'woman' are not present in the vocabulary.")

# Compare semantic similarities between pretrained and own models
print("\nSemantic similarity (Pretrained Model):", example1)
print("Semantic similarity (Pretrained Model):", example2)
print("Semantic similarity (Pretrained Model):", example3)
```

```
print("Semantic similarity (Pretrained Model):", example4)
```

```
King - Man + Woman (Own Model) = [('romney', 0.5904972553253174)]
Excellent ~ Outstanding (Own Model) = [('superb', 0.8473477363586426)]
Marriage - Man + Woman (Own Model) = [('romney', 0.5809527039527893)]
Good ~ Bad (Own Model) = [('excellent', 0.49807488918304443)]

Semantic similarity (Pretrained Model): [('queen', 0.7118192911148071)]
Semantic similarity (Pretrained Model): [('oustanding', 0.750198483467102)]
Semantic similarity (Pretrained Model): [('marriages', 0.7118672728538513)]
Semantic similarity (Pretrained Model): [('excellent', 0.46134573221206665)]
```

0.11 What do you conclude from comparing vectors generated by yourself and the pretrained model? Which of the Word2Vec models seems to encode semantic similarities between words better?

Comparing vectors generated by pre-trained Word2Vec models and those trained on specific datasets reveals nuanced trade-offs. Pre-trained models, leveraging vast corpora, excel in capturing broad semantic similarities but may lack fine-grained domain specificity. Conversely, self-generated vectors, tailored to specific contexts, offer potential for domain-specific insights but require representative data and entail computational costs. Evaluating both models on relevant tasks like word similarity or downstream applications is crucial to determining which better encodes semantic relationships for specific use cases.

0.12 3. Simple Models

0.13 Extract the TF-IDF Features from Dataset and train Perceptron and SVM

Extracting the TF-IDF feature vectors from the initial dataset, and using that data set to train the Perceptron and SVM, and calculate the accuracy on the testing dataset

[35]:
```
# Initialize the TF-IDF vectorizer
tfidf_vectorizer = TfidfVectorizer(max_features=2000000)

# Fit and transform the training set
X_train_tfidf = tfidf_vectorizer.fit_transform(X_train_binary)

# Transform the test set
X_test_tfidf = tfidf_vectorizer.transform(X_test_binary.apply(preprocess_nltk))

# Print the shape of the TF-IDF matrices
print(f"\nShape of X_train_tfidf: {X_train_tfidf.shape}")
print(f"Shape of X_test_tfidf: {X_test_tfidf.shape}")
```

```
Shape of X_train_tfidf: (160200, 112187)
```

```
Shape of X_test_tfidf: (39800, 112187)
```

[36]:
```
perceptron = Perceptron(max_iter=1000)

perceptron.fit(X_train_tfidf, y_train_binary)
y_pred = perceptron.predict(X_test_tfidf)
accuracy_perceptron_tf_idf = accuracy_score(y_test_binary, y_pred)
print(f"Accuracy of the Single Layer Perceptron using TF-IDF Features:␣
↪{accuracy_perceptron_tf_idf}")
```

```
Accuracy of the Single Layer Perceptron using TF-IDF Features:
0.8129396984924623
```

[37]:
```
svm = LinearSVC()

svm.fit(X_train_tfidf, y_train_binary)
y_pred = svm.predict(X_test_tfidf)
accuracy_svm_tf_idf = accuracy_score(y_test_binary, y_pred)
print(f"Accuracy of the SVM using TF-IDF Features: {accuracy_svm_tf_idf}")
```

```
Accuracy of the SVM using TF-IDF Features: 0.8603015075376884
```

0.14 Extract the Word2Vec Pre-trained Features from Dataset and train Perceptron and SVM

Extracting the Word2Vec feature vectors on the Pre-trained model from the initial dataset, and using that data set to train the Perceptron and SVM, and calculate the accuracy on the testing dataset

[38]:
```
def get_doc_embedding(doc):
    doc_str = ' '.join(doc)  # Join tokens within the document list to form a␣
↪single string
    words = doc_str.lower().split()  # Split the string into words
    if not words:
        return np.zeros(300)  # Return a zero vector if no words are present
    else:
        return wv.get_mean_vector(words)  # Compute the mean vector using␣
↪Word2Vec model

X_train_w2v_binary = []
X_test_w2v_binary = []

# Extract Word2Vec features for training data
for e in tqdm(X_train_binary):
    X_train_w2v_binary.append(get_doc_embedding(e))

# Extract Word2Vec features for testing data
```

```
for e in tqdm(X_test_binary):
    X_test_w2v_binary.append(get_doc_embedding(e))

X_train_w2v_binary = np.array(X_train_w2v_binary)
X_test_w2v_binary = np.array(X_test_w2v_binary)

X_train_w2v_binary.shape, X_test_w2v_binary.shape
```

```
100%|âŰĹâŰĹâŰĹâ
```

[42]:
```
def get_doc_embedding(doc):
    doc_str = ' '.join(doc)  # Join tokens within the document list to form a␣
 ↪single string
    words = doc_str.lower().split()  # Split the string into words
    if not words:
        return np.zeros(300)  # Return a zero vector if no words are present
    else:
        return model_own.wv.get_mean_vector(words)  # Compute the mean vector␣
 ↪using Word2Vec model

X_train_w2v_own_binary = []
X_test_w2v_own_binary = []

# Extract Word2Vec features for training data
for e in tqdm(X_train_binary):
    X_train_w2v_own_binary.append(get_doc_embedding(e))

# Extract Word2Vec features for testing data
for e in tqdm(X_test_binary):
    X_test_w2v_own_binary.append(get_doc_embedding(e))

X_train_w2v_own_binary = np.array(X_train_w2v_own_binary)
X_test_w2v_own_binary = np.array(X_test_w2v_own_binary)

X_train_w2v_own_binary.shape, X_test_w2v_own_binary.shape
```

```
100%|âŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâ
 ↪160200/160200 [01:28<00:00, 1804.61it/s]
100%|âŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâŰĺâ
 ↪39800/39800 [00:30<00:00, 1300.35it/s]
```

[42]:
```
((160200, 300), (39800, 300))
```

[43]:
```
perceptron = Perceptron(max_iter=1000)

perceptron.fit(X_train_w2v_own_binary, y_train_binary)
y_pred = perceptron.predict(X_test_w2v_own_binary)
accuracy_perceptron_w2v_own = accuracy_score(y_test_binary, y_pred)
print(f"Accuracy of the Single Layer Perceptron using Word2Vec Custom Features:␣
 ↪{accuracy_perceptron_w2v_own}")
```

```
Accuracy of the Single Layer Perceptron using Word2Vec Custom Features:
0.5012311557788944
```

[44]:
```
svm = LinearSVC()

svm.fit(X_train_w2v_own_binary, y_train_binary)
```

```
y_pred = svm.predict(X_test_w2v_own_binary)
accuracy_svm_w2v_own = accuracy_score(y_test_binary, y_pred)
print(f"Accuracy of the SVM using Word2Vec Custom Features:␣
 ↪{accuracy_svm_w2v_own}")
```

```
Accuracy of the SVM using Word2Vec Custom Features: 0.6076130653266332
```

0.16 4. Feed Forward Neural Networks

0.17 Multi Layer Perceptron with Binary Classification

Using the same binary dataset vectors that we have generated from the Word2Vec Pre-trained and Custom models that we have done in Question 3 to train the Multi-layer perceptron and evaluate the accuracy on the testing dataset.

Here as well we need to encode the classes to 0 and 1, as the class labels shall start from '0'

[45]:
```
# Convert labels to NumPy arrays
y_train_np = np.array(y_train_binary)
y_test_np = np.array(y_test_binary)

# Convert labels to binary format (0 for class 1, 1 for class 2)
y_train_binary = np.where(y_train_np == 1, 0, 1)
y_test_binary = np.where(y_test_np == 1, 0, 1)

# Define the MLP Model for binary classification
class BinaryMLP(nn.Module):
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size):
        super(BinaryMLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.fc3 = nn.Linear(hidden_size2, 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)
        return self.softmax(x)

# Training Loop
def train(model, train_loader, criterion, optimizer, num_epochs):
    model.train()
    for epoch in range(num_epochs):
        running_loss = 0.0
        for inputs, labels in train_loader:
```

```
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        print(f"Epoch {epoch+1}/{num_epochs}, Loss: {running_loss}")

# Evaluation
def evaluate(model, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in test_loader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    print(f"Accuracy: {correct/total}")
```

0.18 Extract the Word2Vec Custom Features and train a Multi Layer Perceptron with Binary Classification

[46]:
```
# Convert data to PyTorch tensors
train_dataset_own_binary = TensorDataset(torch.tensor(X_train_w2v_own_binary,␣
 ↪dtype=torch.float32), torch.tensor(y_train_binary, dtype=torch.long))
test_dataset_own_binary = TensorDataset(torch.tensor(X_test_w2v_own_binary,␣
 ↪dtype=torch.float32), torch.tensor(y_test_binary, dtype=torch.long))

# Create DataLoader
train_loader_own_binary = DataLoader(train_dataset_own_binary, batch_size=32,␣
 ↪shuffle=True)
test_loader_own_binary = DataLoader(test_dataset_own_binary, batch_size=32)

# Initialize the model
input_size = X_train_w2v_own_binary.shape[1]  # Size of input features
hidden_size1 = 50
hidden_size2 = 10
output_size = 2  # Binary classification (classes 1 and 2)
binary_model_own_binary = BinaryMLP(input_size, hidden_size1, hidden_size2,␣
 ↪output_size)

# Define Loss Function and Optimizer
criterion = nn.CrossEntropyLoss()
```

```
optimizer = optim.Adam(binary_model_own_binary.parameters(), lr=0.001)

# Train the binary classification model
num_epochs = 50
train(binary_model_own_binary, train_loader_own_binary, criterion, optimizer,␣
 ↪num_epochs)

# Evaluate the binary classification model
evaluate(binary_model_own_binary, test_loader_own_binary)
```

```
Epoch 1/50, Loss: 3323.425560414791
Epoch 2/50, Loss: 3242.8365510106087
Epoch 3/50, Loss: 3224.7675822377205
Epoch 4/50, Loss: 3214.373399734497
Epoch 5/50, Loss: 3209.4932764172554
Epoch 6/50, Loss: 3200.5022310614586
Epoch 7/50, Loss: 3193.3247091174126
Epoch 8/50, Loss: 3185.9956729114056
Epoch 9/50, Loss: 3181.4531664848328
Epoch 10/50, Loss: 3176.8937999606133
Epoch 11/50, Loss: 3175.9828359484673
Epoch 12/50, Loss: 3170.8038108944893
Epoch 13/50, Loss: 3168.285962700844
Epoch 14/50, Loss: 3164.8113036751747
Epoch 15/50, Loss: 3161.546014279127
Epoch 16/50, Loss: 3160.4918980002403
Epoch 17/50, Loss: 3157.501613199711
Epoch 18/50, Loss: 3157.6596927046776
Epoch 19/50, Loss: 3155.937397301197
Epoch 20/50, Loss: 3153.528224468231
Epoch 21/50, Loss: 3151.404807239771
Epoch 22/50, Loss: 3151.1297699809074
Epoch 23/50, Loss: 3149.1825039088726
Epoch 24/50, Loss: 3149.206249654293
Epoch 25/50, Loss: 3146.9454906880856
Epoch 26/50, Loss: 3145.9316977858543
Epoch 27/50, Loss: 3147.08208245039
Epoch 28/50, Loss: 3144.826644539833
Epoch 29/50, Loss: 3143.413956552744
Epoch 30/50, Loss: 3145.089313417673
Epoch 31/50, Loss: 3143.160977780819
Epoch 32/50, Loss: 3143.921137303114
Epoch 33/50, Loss: 3140.6391310095787
Epoch 34/50, Loss: 3141.163193643093
Epoch 35/50, Loss: 3139.8949775993824
Epoch 36/50, Loss: 3138.606913626194
Epoch 37/50, Loss: 3139.145885795355
```

```
Epoch 38/50, Loss: 3137.8794299662113
Epoch 39/50, Loss: 3138.6251110732555
Epoch 40/50, Loss: 3137.357636541128
Epoch 41/50, Loss: 3136.401563555002
Epoch 42/50, Loss: 3136.378142297268
Epoch 43/50, Loss: 3135.6460728645325
Epoch 44/50, Loss: 3134.1910824477673
Epoch 45/50, Loss: 3135.3988238573074
Epoch 46/50, Loss: 3134.54019472003
Epoch 47/50, Loss: 3133.9537192881107
Epoch 48/50, Loss: 3133.610229164362
Epoch 49/50, Loss: 3132.686781704426
Epoch 50/50, Loss: 3133.3254142701626
Accuracy: 0.6349748743718593
```

0.19 Extract the Word2Vec Pretrained Features and train a Multi Layer Perceptron with Binary Classification

[47]:
```
# Convert data to PyTorch tensors
train_dataset_pre_binary = TensorDataset(torch.tensor(X_train_w2v_binary,␣
 ↪dtype=torch.float32), torch.tensor(y_train_binary, dtype=torch.long))
test_dataset_pre_binary = TensorDataset(torch.tensor(X_test_w2v_binary,␣
 ↪dtype=torch.float32), torch.tensor(y_test_binary, dtype=torch.long))

# Create DataLoader
train_loader_pre_binary = DataLoader(train_dataset_pre_binary, batch_size=32,␣
 ↪shuffle=True)
test_loader_pre_binary = DataLoader(test_dataset_pre_binary, batch_size=32)

# Initialize the model
input_size = X_train_w2v_binary.shape[1]  # Size of input features
hidden_size1 = 50
hidden_size2 = 10
output_size = 2  # Binary classification (classes 1 and 2)
binary_model_pre_binary = BinaryMLP(input_size, hidden_size1, hidden_size2,␣
 ↪output_size)

# Define Loss Function and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(binary_model_pre_binary.parameters(), lr=0.001)

# Train the binary classification model
num_epochs = 50
train(binary_model_pre_binary, train_loader_pre_binary, criterion, optimizer,␣
 ↪num_epochs)
```

```
# Evaluate the binary classification model
evaluate(binary_model_pre_binary, test_loader_pre_binary)
```

```
Epoch 1/50, Loss: 3344.9393923282623
Epoch 2/50, Loss: 3267.1416442990303
Epoch 3/50, Loss: 3232.56443169713
Epoch 4/50, Loss: 3216.721622556448
Epoch 5/50, Loss: 3211.3986123502254
Epoch 6/50, Loss: 3207.7164747714996
Epoch 7/50, Loss: 3202.744212627411
Epoch 8/50, Loss: 3200.1807994246483
Epoch 9/50, Loss: 3194.121215969324
Epoch 10/50, Loss: 3189.0789400935173
Epoch 11/50, Loss: 3185.3685515522957
Epoch 12/50, Loss: 3179.5080044567585
Epoch 13/50, Loss: 3172.4399179518223
Epoch 14/50, Loss: 3169.3857010900974
Epoch 15/50, Loss: 3167.2382534742355
Epoch 16/50, Loss: 3163.475892752409
Epoch 17/50, Loss: 3164.04864063859
Epoch 18/50, Loss: 3159.970018863678
Epoch 19/50, Loss: 3158.055665344
Epoch 20/50, Loss: 3156.751287251711
Epoch 21/50, Loss: 3156.282939761877
Epoch 22/50, Loss: 3154.486574202776
Epoch 23/50, Loss: 3153.950118690729
Epoch 24/50, Loss: 3152.3372714221478
Epoch 25/50, Loss: 3151.794505864382
Epoch 26/50, Loss: 3150.168945878744
Epoch 27/50, Loss: 3149.924498349428
Epoch 28/50, Loss: 3148.044831752777
Epoch 29/50, Loss: 3148.335063278675
Epoch 30/50, Loss: 3147.457699537277
Epoch 31/50, Loss: 3145.4660854637623
Epoch 32/50, Loss: 3145.5864459574223
Epoch 33/50, Loss: 3144.6990844011307
Epoch 34/50, Loss: 3144.0857751965523
Epoch 35/50, Loss: 3144.998258292675
Epoch 36/50, Loss: 3143.953347861767
Epoch 37/50, Loss: 3143.556082755327
Epoch 38/50, Loss: 3143.4792056381702
Epoch 39/50, Loss: 3141.8969056606293
Epoch 40/50, Loss: 3141.6346136033535
Epoch 41/50, Loss: 3140.0632943212986
Epoch 42/50, Loss: 3140.9293306469917
Epoch 43/50, Loss: 3140.620075672865
Epoch 44/50, Loss: 3140.5128760635853
```

```
Epoch 45/50, Loss: 3137.908288091421
Epoch 46/50, Loss: 3138.1942223012447
Epoch 47/50, Loss: 3137.895871460438
Epoch 48/50, Loss: 3139.127146035433
Epoch 49/50, Loss: 3138.0721495449543
Epoch 50/50, Loss: 3136.5219447016716
Accuracy: 0.6264321608040201
```

0.20 Multi Layer Perceptron with Ternary Classification

Here we shall be using the original dataset, that has 3 classes 1, 2 and 3. then calculating the Word2Vec vectors for both the pre-trained and Custom model and using them to train the multi-layer perceptron and evaluating on the testing set.

Here in this case as well we need to encode the class labels from 1 to 3 to 0 to 2, as the class labels shall start from '0' only.

[48]:
```
def get_doc_embedding(doc):
    doc_str = ' '.join(doc)  # Join tokens within the document list to form a␣
 ↪single string
    words = doc_str.lower().split()  # Split the string into words
    if not words:
        return np.zeros(300)  # Return a zero vector if no words are present
    else:
        return model_own.wv.get_mean_vector(words)  # Compute the mean vector␣
 ↪using Word2Vec model

X_train_w2v_own = []
X_test_w2v_own = []

# Extract Word2Vec features for training data
for e in tqdm(X_train_nltk_preprocessed):
    X_train_w2v_own.append(get_doc_embedding(e))

# Extract Word2Vec features for testing data
for e in tqdm(X_test):
    X_test_w2v_own.append(get_doc_embedding(e))

X_train_w2v_own = np.array(X_train_w2v_own)
X_test_w2v_own = np.array(X_test_w2v_own)

X_train_w2v_own.shape, X_test_w2v_own.shape
```

```
100%|âŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹ| 
 ↪200000/200000 [01:46<00:00, 1873.31it/s]
100%|âŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹ| 
 ↪50000/50000 [00:39<00:00, 1254.70it/s]
```

[48]: ((200000, 300), (50000, 300))

[49]:
```
def get_doc_embedding(doc):
    doc_str = ' '.join(doc)  # Join tokens within the document list to form a␣
↪single string
    words = doc_str.lower().split()  # Split the string into words
    if not words:
        return np.zeros(300)  # Return a zero vector if no words are present
    else:
        return wv.get_mean_vector(words)  # Compute the mean vector using␣
↪Word2Vec model

X_train_w2v = []
X_test_w2v = []

# Extract Word2Vec features for training data
for e in tqdm(X_train_nltk_preprocessed):
    X_train_w2v.append(get_doc_embedding(e))

# Extract Word2Vec features for testing data
for e in tqdm(X_test):
    X_test_w2v.append(get_doc_embedding(e))

X_train_w2v = np.array(X_train_w2v)
X_test_w2v = np.array(X_test_w2v)

X_train_w2v.shape, X_test_w2v.shape
```

```
100%|âŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹ
↪200000/200000 [01:53<00:00, 1759.83it/s]
100%|âŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹâŰĹ
↪50000/50000 [00:43<00:00, 1142.95it/s]
```

[49]: ((200000, 300), (50000, 300))

[50]:
```
# Convert labels to NumPy arrays
y_train_np = np.array(y_train)
y_test_np = np.array(y_test)

# Convert labels to integers ranging from 0 to 2
y_train_encoded = [label - 1 for label in y_train_np]  # Assuming labels start␣
↪from 1
y_test_encoded = [label - 1 for label in y_test_np]  # Assuming labels start␣
↪from 1

# Define the MLP Model
class MLP(nn.Module):
```

```
    def __init__(self, input_size, hidden_size1, hidden_size2, output_size):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.fc3 = nn.Linear(hidden_size2, 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)
        return self.softmax(x)

# Training Loop
def train(model, train_loader, criterion, optimizer, num_epochs):
    model.train()
    for epoch in range(num_epochs):
        running_loss = 0.0
        for inputs, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        print(f"Epoch {epoch+1}/{num_epochs}, Loss: {running_loss}")

# Evaluation
def evaluate(model, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in test_loader:
            outputs = model(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    print(f"Accuracy: {correct/total}")
```

0.21 Extract the Word2Vec Pre-trained Features and train a Multi Layer Perceptron

[51]:
```
# Convert data to PyTorch tensors
train_dataset_pre = TensorDataset(torch.tensor(X_train_w2v, dtype=torch.
 ↪float32), torch.tensor(y_train_encoded, dtype=torch.long))
test_dataset_pre = TensorDataset(torch.tensor(X_test_w2v, dtype=torch.float32),␣
 ↪torch.tensor(y_test_encoded, dtype=torch.long))

# Create DataLoader
train_loader_pre = DataLoader(train_dataset_pre, batch_size=32, shuffle=True)
test_loader_pre = DataLoader(test_dataset_pre, batch_size=32)

# Initialize the model
input_size = X_train_w2v.shape[1]  # Size of input features
hidden_size1 = 50
hidden_size2 = 10
output_size = 3  # Assuming 3 classes for sentiment analysis
model_pre = MLP(input_size, hidden_size1, hidden_size2, output_size)

# Define Loss Function and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model_pre.parameters(), lr=0.001)

# Train the model
num_epochs = 50
train(model_pre, train_loader_pre, criterion, optimizer, num_epochs)

# Evaluate the model
evaluate(model_pre, test_loader_pre)
```

```
Epoch 1/50, Loss: 6505.194889605045
Epoch 2/50, Loss: 6407.516536474228
Epoch 3/50, Loss: 6372.225348472595
Epoch 4/50, Loss: 6355.613764286041
Epoch 5/50, Loss: 6347.46262139082
Epoch 6/50, Loss: 6340.601443588734
Epoch 7/50, Loss: 6331.0599210858345
Epoch 8/50, Loss: 6315.430656015873
Epoch 9/50, Loss: 6306.298391401768
Epoch 10/50, Loss: 6300.394243955612
Epoch 11/50, Loss: 6293.240439116955
Epoch 12/50, Loss: 6289.825047254562
Epoch 13/50, Loss: 6285.750542700291
Epoch 14/50, Loss: 6281.849463224411
Epoch 15/50, Loss: 6279.120568394661
Epoch 16/50, Loss: 6277.236977458
```

```
Epoch 17/50, Loss: 6274.526634931564
Epoch 18/50, Loss: 6273.958921611309
Epoch 19/50, Loss: 6268.318421125412
Epoch 20/50, Loss: 6268.669386148453
Epoch 21/50, Loss: 6265.7270964980125
Epoch 22/50, Loss: 6263.161739349365
Epoch 23/50, Loss: 6262.33771365881
Epoch 24/50, Loss: 6261.571010649204
Epoch 25/50, Loss: 6260.096761286259
Epoch 26/50, Loss: 6260.204388618469
Epoch 27/50, Loss: 6258.781262874603
Epoch 28/50, Loss: 6256.204732000828
Epoch 29/50, Loss: 6254.868431091309
Epoch 30/50, Loss: 6253.441541552544
Epoch 31/50, Loss: 6251.667254507542
Epoch 32/50, Loss: 6251.896208524704
Epoch 33/50, Loss: 6249.928118169308
Epoch 34/50, Loss: 6250.055064558983
Epoch 35/50, Loss: 6247.988093972206
Epoch 36/50, Loss: 6246.2828566432
Epoch 37/50, Loss: 6246.4094088077545
Epoch 38/50, Loss: 6245.59342867136
Epoch 39/50, Loss: 6243.528662264347
Epoch 40/50, Loss: 6243.890145599842
Epoch 41/50, Loss: 6244.921311080456
Epoch 42/50, Loss: 6241.981376111507
Epoch 43/50, Loss: 6242.319493114948
Epoch 44/50, Loss: 6241.5356143713
Epoch 45/50, Loss: 6239.235330283642
Epoch 46/50, Loss: 6240.063661873341
Epoch 47/50, Loss: 6238.990886032581
Epoch 48/50, Loss: 6238.533412337303
Epoch 49/50, Loss: 6237.538753211498
Epoch 50/50, Loss: 6237.690585613251
Accuracy: 0.5133
```

0.22 Extract the Word2Vec Custom Features and train a Multi Layer Perceptron

[52]:
```
# Convert data to PyTorch tensors
train_dataset_own = TensorDataset(torch.tensor(X_train_w2v_own, dtype=torch.
 ↪float32), torch.tensor(y_train_encoded, dtype=torch.long))
test_dataset_own = TensorDataset(torch.tensor(X_test_w2v_own, dtype=torch.
 ↪float32), torch.tensor(y_test_encoded, dtype=torch.long))

# Create DataLoader
```

```
train_loader_own = DataLoader(train_dataset_own, batch_size=32, shuffle=True)
test_loader_own = DataLoader(test_dataset_own, batch_size=32)

# Initialize the model
input_size = X_train_w2v_own.shape[1]  # Size of input features
hidden_size1 = 50
hidden_size2 = 10
output_size = 3  # Assuming 3 classes for sentiment analysis
model_own = MLP(input_size, hidden_size1, hidden_size2, output_size)

# Define Loss Function and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model_own.parameters(), lr=0.001)

# Train the model
num_epochs = 50
train(model_own, train_loader_own, criterion, optimizer, num_epochs)

# Evaluate the model
evaluate(model_own, test_loader_own)
```

```
Epoch 1/50, Loss: 6498.780159831047
Epoch 2/50, Loss: 6400.97633677721
Epoch 3/50, Loss: 6376.639673233032
Epoch 4/50, Loss: 6358.732979893684
Epoch 5/50, Loss: 6347.87987112999
Epoch 6/50, Loss: 6341.1168175935745
Epoch 7/50, Loss: 6332.512089669704
Epoch 8/50, Loss: 6328.183353543282
Epoch 9/50, Loss: 6323.460064649582
Epoch 10/50, Loss: 6319.396956145763
Epoch 11/50, Loss: 6314.587701499462
Epoch 12/50, Loss: 6310.881038248539
Epoch 13/50, Loss: 6307.944660484791
Epoch 14/50, Loss: 6303.935755848885
Epoch 15/50, Loss: 6302.715979993343
Epoch 16/50, Loss: 6298.875631213188
Epoch 17/50, Loss: 6296.430803596973
Epoch 18/50, Loss: 6293.912975907326
Epoch 19/50, Loss: 6291.618820667267
Epoch 20/50, Loss: 6288.986372232437
Epoch 21/50, Loss: 6285.127147078514
Epoch 22/50, Loss: 6283.881210565567
Epoch 23/50, Loss: 6279.84032189846
Epoch 24/50, Loss: 6279.9689974188805
Epoch 25/50, Loss: 6277.729133725166
Epoch 26/50, Loss: 6276.328289449215
```

```
Epoch 27/50, Loss: 6273.352016687393
Epoch 28/50, Loss: 6270.606888830662
Epoch 29/50, Loss: 6268.878164708614
Epoch 30/50, Loss: 6268.279967546463
Epoch 31/50, Loss: 6266.174606144428
Epoch 32/50, Loss: 6265.104053735733
Epoch 33/50, Loss: 6264.6649406552315
Epoch 34/50, Loss: 6262.693824112415
Epoch 35/50, Loss: 6261.148292958736
Epoch 36/50, Loss: 6261.369768679142
Epoch 37/50, Loss: 6258.9298285245895
Epoch 38/50, Loss: 6256.8996948599815
Epoch 39/50, Loss: 6257.075434565544
Epoch 40/50, Loss: 6255.959533751011
Epoch 41/50, Loss: 6255.498541593552
Epoch 42/50, Loss: 6254.348329305649
Epoch 43/50, Loss: 6253.644511044025
Epoch 44/50, Loss: 6252.764105439186
Epoch 45/50, Loss: 6251.325192272663
Epoch 46/50, Loss: 6250.597810804844
Epoch 47/50, Loss: 6250.223139643669
Epoch 48/50, Loss: 6249.160574257374
Epoch 49/50, Loss: 6248.502880871296
Epoch 50/50, Loss: 6247.869339823723
Accuracy: 0.50938
```

Here, I am extracting the first 10 Word2Vec features, and using them for the ternary classification as well as using the binary features that we have extracted to train the binary Multi layer perceptron on the training dataset and evaluating their performance on the testing dataset.

0.23 Using the First 10 Word2Vec vectors on Ternary Classification for the Word2Vec Custom Model

[53]:
```python
# Concatenate the first 10 Word2Vec vectors for each review
X_train_concat_own = []
X_test_concat_own = []

for review in X_train_w2v_own:
    review_reshaped = review.reshape(1, -1)  # Reshape to ensure it's a 2D array
    concatenated_vector = np.concatenate(review_reshaped[:10], axis=0)
    X_train_concat_own.append(concatenated_vector)

for review in X_test_w2v_own:
    review_reshaped = review.reshape(1, -1)  # Reshape to ensure it's a 2D array
    concatenated_vector = np.concatenate(review_reshaped[:10], axis=0)
    X_test_concat_own.append(concatenated_vector)
```

```
# Convert the concatenated features into PyTorch tensors
X_train_concat_tensor_own = torch.tensor(X_train_concat_own, dtype=torch.float32)
X_test_concat_tensor_own = torch.tensor(X_test_concat_own, dtype=torch.float32)

# Convert labels to ternary format
y_train_tensor_own = torch.tensor(y_train_encoded, dtype=torch.long)
y_test_tensor_own = torch.tensor(y_test_encoded, dtype=torch.long)

# Create DataLoader
train_dataset_concat_own = TensorDataset(X_train_concat_tensor_own,␣
↪y_train_tensor_own)
test_dataset_concat_own = TensorDataset(X_test_concat_tensor_own,␣
↪y_test_tensor_own)

train_loader_concat_own = DataLoader(train_dataset_concat_own, batch_size=32,␣
↪shuffle=True)
test_loader_concat_own = DataLoader(test_dataset_concat_own, batch_size=32)

# Initialize the model
input_size = X_train_concat_tensor_own.shape[1]  # Size of input features
hidden_size1 = 50
hidden_size2 = 10
output_size = 3  # Assuming 3 classes for sentiment analysis
model_concat_own = MLP(input_size, hidden_size1, hidden_size2, output_size)

# Define Loss Function and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model_concat_own.parameters(), lr=0.001)

# Train the model
num_epochs = 50
train(model_concat_own, train_loader_concat_own, criterion, optimizer,␣
↪num_epochs)

# Evaluate the model
evaluate(model_concat_own, test_loader_concat_own)
```

```
/var/folders/xx/d1tzxzhj3fzbmswz4z7m_40w0000gn/T/ipykernel_31112/1352761604.py:1
6: UserWarning: Creating a tensor from a list of numpy.ndarrays is extremely
slow. Please consider converting the list to a single numpy.ndarray with
numpy.array() before converting to a tensor. (Triggered internally at /private/v
ar/folders/nz/j6p8yfhx1mv_0grj5xl4650h0000gp/T/abs_1aidzjezue/croot/pytorch_1687
856425340/work/torch/csrc/utils/tensor_new.cpp:248.)
  X_train_concat_tensor_own = torch.tensor(X_train_concat_own,
dtype=torch.float32)
```

```
Epoch 1/50, Loss: 6481.61340034008
Epoch 2/50, Loss: 6385.503250360489
```

```
Epoch 3/50, Loss: 6360.853358089924
Epoch 4/50, Loss: 6345.465584874153
Epoch 5/50, Loss: 6333.520486533642
Epoch 6/50, Loss: 6324.639396846294
Epoch 7/50, Loss: 6317.478264570236
Epoch 8/50, Loss: 6312.688789844513
Epoch 9/50, Loss: 6306.100179135799
Epoch 10/50, Loss: 6300.011463165283
Epoch 11/50, Loss: 6294.431249856949
Epoch 12/50, Loss: 6293.446542024612
Epoch 13/50, Loss: 6289.821278214455
Epoch 14/50, Loss: 6288.257537126541
Epoch 15/50, Loss: 6284.445706427097
Epoch 16/50, Loss: 6282.852754354477
Epoch 17/50, Loss: 6279.708474516869
Epoch 18/50, Loss: 6279.602629005909
Epoch 19/50, Loss: 6276.000140547752
Epoch 20/50, Loss: 6273.767419040203
Epoch 21/50, Loss: 6269.479419648647
Epoch 22/50, Loss: 6268.02747040987
Epoch 23/50, Loss: 6267.527205705643
Epoch 24/50, Loss: 6262.9584149718285
Epoch 25/50, Loss: 6262.649322628975
Epoch 26/50, Loss: 6258.973907291889
Epoch 27/50, Loss: 6258.186829328537
Epoch 28/50, Loss: 6256.603058755398
Epoch 29/50, Loss: 6256.090897679329
Epoch 30/50, Loss: 6253.6809985637665
Epoch 31/50, Loss: 6254.460513472557
Epoch 32/50, Loss: 6252.560277581215
Epoch 33/50, Loss: 6249.541831314564
Epoch 34/50, Loss: 6250.814956605434
Epoch 35/50, Loss: 6248.070578336716
Epoch 36/50, Loss: 6246.799599528313
Epoch 37/50, Loss: 6246.783444583416
Epoch 38/50, Loss: 6245.462326824665
Epoch 39/50, Loss: 6244.607045471668
Epoch 40/50, Loss: 6242.118771910667
Epoch 41/50, Loss: 6241.047801613808
Epoch 42/50, Loss: 6239.572349309921
Epoch 43/50, Loss: 6240.246558308601
Epoch 44/50, Loss: 6238.977270483971
Epoch 45/50, Loss: 6237.48008698225
Epoch 46/50, Loss: 6236.673461198807
Epoch 47/50, Loss: 6237.036950469017
Epoch 48/50, Loss: 6234.947586297989
Epoch 49/50, Loss: 6235.281324267387
Epoch 50/50, Loss: 6233.786520183086
```

```
Accuracy: 0.50792
```

0.24 Using the First 10 Word2Vec vectors on Ternary Classification for the Word2Vec Pre-Trained Model

[54]:
```
# Concatenate the first 10 Word2Vec vectors for each review
X_train_concat = []
X_test_concat = []

for review in X_train_w2v:
    review_reshaped = review.reshape(1, -1)  # Reshape to ensure it's a 2D array
    concatenated_vector = np.concatenate(review_reshaped[:10], axis=0)
    X_train_concat.append(concatenated_vector)

for review in X_test_w2v:
    review_reshaped = review.reshape(1, -1)  # Reshape to ensure it's a 2D array
    concatenated_vector = np.concatenate(review_reshaped[:10], axis=0)
    X_test_concat.append(concatenated_vector)

# Convert the concatenated features into PyTorch tensors
X_train_concat_tensor = torch.tensor(X_train_concat, dtype=torch.float32)
X_test_concat_tensor = torch.tensor(X_test_concat, dtype=torch.float32)

# Convert labels to ternary format
y_train_tensor = torch.tensor(y_train_encoded, dtype=torch.long)
y_test_tensor = torch.tensor(y_test_encoded, dtype=torch.long)

# Create DataLoader
train_dataset_concat = TensorDataset(X_train_concat_tensor, y_train_tensor)
test_dataset_concat = TensorDataset(X_test_concat_tensor, y_test_tensor)

train_loader_concat = DataLoader(train_dataset_concat, batch_size=32,␣
↪shuffle=True)
test_loader_concat = DataLoader(test_dataset_concat, batch_size=32)

# Initialize the model
input_size = X_train_concat_tensor.shape[1]  # Size of input features
hidden_size1 = 50
hidden_size2 = 10
output_size = 3  # Assuming 3 classes for sentiment analysis
model_concat = MLP(input_size, hidden_size1, hidden_size2, output_size)

# Define Loss Function and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model_concat.parameters(), lr=0.001)
```

```
# Train the model
num_epochs = 50
train(model_concat, train_loader_concat, criterion, optimizer, num_epochs)

# Evaluate the model
evaluate(model_concat, test_loader_concat)
```

```
Epoch 1/50, Loss: 6501.003310024738
Epoch 2/50, Loss: 6411.406356275082
Epoch 3/50, Loss: 6379.237156510353
Epoch 4/50, Loss: 6357.855707705021
Epoch 5/50, Loss: 6346.558179080486
Epoch 6/50, Loss: 6336.867486059666
Epoch 7/50, Loss: 6328.332867145538
Epoch 8/50, Loss: 6320.53949290514
Epoch 9/50, Loss: 6312.367366075516
Epoch 10/50, Loss: 6304.602900922298
Epoch 11/50, Loss: 6297.362203478813
Epoch 12/50, Loss: 6291.952714383602
Epoch 13/50, Loss: 6288.066511452198
Epoch 14/50, Loss: 6285.03642898798
Epoch 15/50, Loss: 6282.7331283688545
Epoch 16/50, Loss: 6283.071981787682
Epoch 17/50, Loss: 6278.795620620251
Epoch 18/50, Loss: 6276.931710362434
Epoch 19/50, Loss: 6272.630007445812
Epoch 20/50, Loss: 6270.558665752411
Epoch 21/50, Loss: 6268.948526859283
Epoch 22/50, Loss: 6267.434275388718
Epoch 23/50, Loss: 6267.472657203674
Epoch 24/50, Loss: 6263.57393437624
Epoch 25/50, Loss: 6261.93185031414
Epoch 26/50, Loss: 6261.628469407558
Epoch 27/50, Loss: 6259.062380969524
Epoch 28/50, Loss: 6257.009374797344
Epoch 29/50, Loss: 6258.113060712814
Epoch 30/50, Loss: 6257.009604692459
Epoch 31/50, Loss: 6256.341547608376
Epoch 32/50, Loss: 6253.2241161465645
Epoch 33/50, Loss: 6252.160817086697
Epoch 34/50, Loss: 6251.296446084976
Epoch 35/50, Loss: 6249.434270620346
Epoch 36/50, Loss: 6248.6408215761185
Epoch 37/50, Loss: 6247.61165034771
Epoch 38/50, Loss: 6245.353277087212
Epoch 39/50, Loss: 6245.599706590176
Epoch 40/50, Loss: 6244.8497838974
```

```
Epoch 41/50, Loss: 6243.380714297295
Epoch 42/50, Loss: 6241.21665763855
Epoch 43/50, Loss: 6241.91231995821
Epoch 44/50, Loss: 6239.715795278549
Epoch 45/50, Loss: 6238.923026382923
Epoch 46/50, Loss: 6239.439045011997
Epoch 47/50, Loss: 6237.387090682983
Epoch 48/50, Loss: 6237.779659986496
Epoch 49/50, Loss: 6237.464747846127
Epoch 50/50, Loss: 6234.900174915791
Accuracy: 0.50782
```

0.25 Using the First 10 Word2Vec vectors on Binary Classification for the Word2Vec Pre-Trained Model

[55]:
```python
# Concatenate the first 10 Word2Vec vectors for each review
X_train_concat = []
X_test_concat = []

for review in X_train_w2v_binary:
    review_reshaped = review.reshape(1, -1)  # Reshape to ensure it's a 2D array
    concatenated_vector = np.concatenate(review_reshaped[:10], axis=0)
    X_train_concat.append(concatenated_vector)

for review in X_test_w2v_binary:
    review_reshaped = review.reshape(1, -1)  # Reshape to ensure it's a 2D array
    concatenated_vector = np.concatenate(review_reshaped[:10], axis=0)
    X_test_concat.append(concatenated_vector)

# Convert the concatenated features into PyTorch tensors
X_train_concat_tensor = torch.tensor(X_train_concat, dtype=torch.float32)
X_test_concat_tensor = torch.tensor(X_test_concat, dtype=torch.float32)

# Convert labels to ternary format
y_train_tensor = torch.tensor(y_train_binary, dtype=torch.long)
y_test_tensor = torch.tensor(y_test_binary, dtype=torch.long)

# Create DataLoader
train_dataset_concat = TensorDataset(X_train_concat_tensor, y_train_tensor)
test_dataset_concat = TensorDataset(X_test_concat_tensor, y_test_tensor)

train_loader_concat = DataLoader(train_dataset_concat, batch_size=32,␣
 ↪shuffle=True)
test_loader_concat = DataLoader(test_dataset_concat, batch_size=32)

# Initialize the model
```

```
input_size = X_train_concat_tensor.shape[1]  # Size of input features
hidden_size1 = 50
hidden_size2 = 10
output_size = 2  # Assuming 2 classes for sentiment analysis
model_concat = MLP(input_size, hidden_size1, hidden_size2, output_size)

# Define Loss Function and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model_concat.parameters(), lr=0.001)

# Train the model
num_epochs = 50
train(model_concat, train_loader_concat, criterion, optimizer, num_epochs)

# Evaluate the model
evaluate(model_concat, test_loader_concat)
```

```
Epoch 1/50, Loss: 3356.1845531463623
Epoch 2/50, Loss: 3271.8983495235443
Epoch 3/50, Loss: 3233.304957896471
Epoch 4/50, Loss: 3217.90689894557
Epoch 5/50, Loss: 3210.572159022093
Epoch 6/50, Loss: 3204.472539514303
Epoch 7/50, Loss: 3197.940410375595
Epoch 8/50, Loss: 3195.5977049171925
Epoch 9/50, Loss: 3192.0230244100094
Epoch 10/50, Loss: 3190.5923019349575
Epoch 11/50, Loss: 3189.697148323059
Epoch 12/50, Loss: 3187.129758745432
Epoch 13/50, Loss: 3184.9911658465862
Epoch 14/50, Loss: 3184.9546769559383
Epoch 15/50, Loss: 3184.019710868597
Epoch 16/50, Loss: 3183.9676348268986
Epoch 17/50, Loss: 3181.434725135565
Epoch 18/50, Loss: 3181.268100142479
Epoch 19/50, Loss: 3178.8603362739086
Epoch 20/50, Loss: 3176.5032584369183
Epoch 21/50, Loss: 3174.0020075440407
Epoch 22/50, Loss: 3172.442873120308
Epoch 23/50, Loss: 3170.2709589600563
Epoch 24/50, Loss: 3167.2356988191605
Epoch 25/50, Loss: 3163.2466747760773
Epoch 26/50, Loss: 3160.235849380493
Epoch 27/50, Loss: 3157.401012778282
Epoch 28/50, Loss: 3155.81391620636
Epoch 29/50, Loss: 3154.6678814291954
Epoch 30/50, Loss: 3153.270157933235
```

```
Epoch 31/50, Loss: 3152.0375125408173
Epoch 32/50, Loss: 3149.1826129853725
Epoch 33/50, Loss: 3149.0340922772884
Epoch 34/50, Loss: 3147.9201694130898
Epoch 35/50, Loss: 3146.5886808633804
Epoch 36/50, Loss: 3146.3337756097317
Epoch 37/50, Loss: 3145.937994837761
Epoch 38/50, Loss: 3144.23573538661
Epoch 39/50, Loss: 3143.859791278839
Epoch 40/50, Loss: 3143.041027188301
Epoch 41/50, Loss: 3143.0817979574203
Epoch 42/50, Loss: 3141.215103149414
Epoch 43/50, Loss: 3141.534513771534
Epoch 44/50, Loss: 3139.5992555618286
Epoch 45/50, Loss: 3138.9017379283905
Epoch 46/50, Loss: 3139.047589302063
Epoch 47/50, Loss: 3134.6020649671555
Epoch 48/50, Loss: 3134.8247643113136
Epoch 49/50, Loss: 3132.8643520772457
Epoch 50/50, Loss: 3132.6437705159187
Accuracy: 0.638467336683417
```

0.26 Using the First 10 Word2Vec vectors on Binary Classification for the Word2Vec Custom Model

[56]:
```python
# Concatenate the first 10 Word2Vec vectors for each review
X_train_concat = []
X_test_concat = []

for review in X_train_w2v_own_binary:
    review_reshaped = review.reshape(1, -1)  # Reshape to ensure it's a 2D array
    concatenated_vector = np.concatenate(review_reshaped[:10], axis=0)
    X_train_concat.append(concatenated_vector)

for review in X_test_w2v_own_binary:
    review_reshaped = review.reshape(1, -1)  # Reshape to ensure it's a 2D array
    concatenated_vector = np.concatenate(review_reshaped[:10], axis=0)
    X_test_concat.append(concatenated_vector)

# Convert the concatenated features into PyTorch tensors
X_train_concat_tensor = torch.tensor(X_train_concat, dtype=torch.float32)
X_test_concat_tensor = torch.tensor(X_test_concat, dtype=torch.float32)

# Convert labels to ternary format
y_train_tensor = torch.tensor(y_train_binary, dtype=torch.long)
y_test_tensor = torch.tensor(y_test_binary, dtype=torch.long)
```

```
# Create DataLoader
train_dataset_concat = TensorDataset(X_train_concat_tensor, y_train_tensor)
test_dataset_concat = TensorDataset(X_test_concat_tensor, y_test_tensor)

train_loader_concat = DataLoader(train_dataset_concat, batch_size=32,␣
 ↪shuffle=True)
test_loader_concat = DataLoader(test_dataset_concat, batch_size=32)

# Initialize the model
input_size = X_train_concat_tensor.shape[1]  # Size of input features
hidden_size1 = 50
hidden_size2 = 10
output_size = 2  # Assuming 2 classes for sentiment analysis
model_concat = MLP(input_size, hidden_size1, hidden_size2, output_size)

# Define Loss Function and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model_concat.parameters(), lr=0.001)

# Train the model
num_epochs = 50
train(model_concat, train_loader_concat, criterion, optimizer, num_epochs)

# Evaluate the model
evaluate(model_concat, test_loader_concat)
```

```
Epoch 1/50, Loss: 3314.3547974824905
Epoch 2/50, Loss: 3242.0267139971256
Epoch 3/50, Loss: 3224.446726948023
Epoch 4/50, Loss: 3211.0086663365364
Epoch 5/50, Loss: 3202.2065628170967
Epoch 6/50, Loss: 3195.6309146285057
Epoch 7/50, Loss: 3191.9821802079678
Epoch 8/50, Loss: 3187.1544785499573
Epoch 9/50, Loss: 3182.3843071758747
Epoch 10/50, Loss: 3175.2804859280586
Epoch 11/50, Loss: 3170.7927663326263
Epoch 12/50, Loss: 3166.9846815764904
Epoch 13/50, Loss: 3163.516598433256
Epoch 14/50, Loss: 3160.2638128995895
Epoch 15/50, Loss: 3157.704405605793
Epoch 16/50, Loss: 3154.710099965334
Epoch 17/50, Loss: 3152.761065542698
Epoch 18/50, Loss: 3149.1145381629467
Epoch 19/50, Loss: 3148.310146123171
Epoch 20/50, Loss: 3146.2868282794952
```

```
Epoch 21/50, Loss: 3143.970229446888
Epoch 22/50, Loss: 3143.9778038859367
Epoch 23/50, Loss: 3141.7837656736374
Epoch 24/50, Loss: 3141.3118644058704
Epoch 25/50, Loss: 3139.039125174284
Epoch 26/50, Loss: 3137.4842279553413
Epoch 27/50, Loss: 3137.0150220394135
Epoch 28/50, Loss: 3135.701799094677
Epoch 29/50, Loss: 3133.949146270752
Epoch 30/50, Loss: 3133.1597623229027
Epoch 31/50, Loss: 3132.5265150666237
Epoch 32/50, Loss: 3131.253515601158
Epoch 33/50, Loss: 3131.8348763287067
Epoch 34/50, Loss: 3131.656037300825
Epoch 35/50, Loss: 3128.4385494589806
Epoch 36/50, Loss: 3127.106327176094
Epoch 37/50, Loss: 3125.7150690853596
Epoch 38/50, Loss: 3126.908812582493
Epoch 39/50, Loss: 3124.813260704279
Epoch 40/50, Loss: 3124.156830430031
Epoch 41/50, Loss: 3124.5963036715984
Epoch 42/50, Loss: 3122.7433320581913
Epoch 43/50, Loss: 3121.6939577162266
Epoch 44/50, Loss: 3121.4175332188606
Epoch 45/50, Loss: 3120.565030038357
Epoch 46/50, Loss: 3121.519037991762
Epoch 47/50, Loss: 3118.65090867877
Epoch 48/50, Loss: 3118.485656142235
Epoch 49/50, Loss: 3118.358697682619
Epoch 50/50, Loss: 3117.938712745905
Accuracy: 0.6419346733668342
```

The comparison between MLPs and simpler models like perceptrons and Support Vector Machines (SVMs) highlights the trade-offs between model complexity and performance. While the MLPs have definetly over-performed the simple models in terms of the testing accuracy. However the training time between the MLPs and Simple Models is significantly higher, so that is a negative of the MLPs. Perceptrons and SVMs, while less complex, are easier to interpret and computationally more efficient.

0.27 5. Convolutional Neural Networks

0.28 CNN with Binary Classification using the Word2Vec Pre-Trained Vectors

Here I am splitting the dataset again from the initial data, and removing the sentiment values with class 3, as we are doing a binary classification. and encoding the class labels fro 1 and 2 to and 1 as the class labels have to start from '0'.

I am using a CNN, with the input size as 300 features from the Word2Vec feature vectors and the

output size to be 2 as we are working on a binary classification.

I am also printing the accuracy of the model on the test set.

[37]:
```
# Joining features and targets for training dataset
train_data = pd.concat([X_train_nltk_preprocessed, y_train], axis=1)
train_data_filtered = train_data[train_data['sentiment'] != 3]
X_train_binary = train_data_filtered['review_body']
y_train_binary = train_data_filtered['sentiment']

# Joining features and targets for testing dataset
test_data = pd.concat([X_test, y_test], axis=1)
test_data_filtered = test_data[test_data['sentiment'] != 3]
X_test_binary = test_data_filtered['review_body']
y_test_binary = test_data_filtered['sentiment']
```

[38]:
```
import gensim.downloader as api
wv = api.load('word2vec-google-news-300')
```

[39]:
```
# Preprocess text data
def preprocess_text(text, max_length=50):
    tokens = text.split()[:max_length]  # Limit maximum review length
    padded_tokens = tokens + ['<PAD>'] * (max_length - len(tokens))  # Pad␣
 ↪shorter reviews
    return padded_tokens

# Convert text data into Word2Vec vectors
def text_to_vectors(texts, wv, max_length=50):
    vectors = []
    for text in texts:
        tokens = preprocess_text(text, max_length)
        vector = [wv[word] if word in wv else np.zeros(wv.vector_size) for word␣
 ↪in tokens]
        vectors.append(vector)
    return np.array(vectors)

# Prepare data
X_train_vectors = text_to_vectors(X_train_binary, wv)
X_test_vectors = text_to_vectors(X_test_binary, wv)

# Convert labels to binary format (class 1 and class 2)
y_train_binary = np.where(y_train_binary == 1, 0, 1)
y_test_binary = np.where(y_test_binary == 1, 0, 1)

# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train_vectors, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train_binary, dtype=torch.long)
X_test_tensor = torch.tensor(X_test_vectors, dtype=torch.float32)
```

```
y_test_tensor = torch.tensor(y_test_binary, dtype=torch.long)

# Create DataLoader
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32)
```

[40]:
```
# Define the CNN model
class CNN(nn.Module):
    def __init__(self, input_size, output_size):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv1d(input_size, 50, kernel_size=3, padding=1)
        self.conv2 = nn.Conv1d(50, 10, kernel_size=3, padding=1)
        self.pool = nn.MaxPool1d(kernel_size=2, stride=2)
        self.fc = nn.Linear(10 * 12, output_size)  # Adjust the input size for␣
 ↪the linear layer

    def forward(self, x):
        x = self.pool(nn.functional.relu(self.conv1(x)))
        x = self.pool(nn.functional.relu(self.conv2(x)))
        x = x.view(-1, 10 * 12)  # Adjust the reshaping operation
        x = self.fc(x)
        return nn.functional.softmax(x, dim=1)

# Train the CNN model
def train_cnn(model, train_loader, criterion, optimizer, num_epochs):
    model.train()
    for epoch in range(num_epochs):
        running_loss = 0.0
        for inputs, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(inputs.permute(0, 2, 1))  # Permute input dimensions␣
 ↪for Conv1D
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        print(f"Epoch {epoch+1}/{num_epochs}, Loss: {running_loss}")

# Evaluation
def evaluate_cnn(model, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in test_loader:
```

```
            outputs = model(inputs.permute(0, 2, 1))  # Permute input dimensions␣
↪for Conv1D
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    print(f"Accuracy: {correct / total}")
```

[41]:
```
# Initialize the model
input_size = X_train_vectors.shape[2]  # Size of input features
output_size = 2  # Binary classification
model = CNN(input_size, output_size)

# Define Loss Function and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
# Train the model
train_cnn(model, train_loader, criterion, optimizer, num_epochs=10)

# Evaluate the model
evaluate_cnn(model, test_loader)
```

```
Epoch 1/10, Loss: 2385.407571732998
Epoch 2/10, Loss: 2231.0871135890484
Epoch 3/10, Loss: 2179.0149119198322
Epoch 4/10, Loss: 2136.2539499402046
Epoch 5/10, Loss: 2105.5084967315197
Epoch 6/10, Loss: 2078.3546420931816
Epoch 7/10, Loss: 2060.8316709697247
Epoch 8/10, Loss: 2042.662799268961
Epoch 9/10, Loss: 2027.9348596930504
Epoch 10/10, Loss: 2012.1750220358372
Accuracy: 0.7983668341708543
```

0.29 CNN with Binary Classification using the Word2Vec Custom Vectors

[42]:
```
# Joining features and targets for training dataset
train_data = pd.concat([X_train_nltk_preprocessed, y_train], axis=1)
train_data_filtered = train_data[train_data['sentiment'] != 3]
X_train_binary = train_data_filtered['review_body']
y_train_binary = train_data_filtered['sentiment']

# Joining features and targets for testing dataset
test_data = pd.concat([X_test, y_test], axis=1)
test_data_filtered = test_data[test_data['sentiment'] != 3]
X_test_binary = test_data_filtered['review_body']
y_test_binary = test_data_filtered['sentiment']
```

[43]:
```
from gensim.models import Word2Vec
# Train the Word2Vec model
model_own = Word2Vec(X_train_nltk_preprocessed_new, vector_size=300, window=11,␣
↪min_count=10)
```

[44]:
```
# Prepare data
X_train_vectors = text_to_vectors(X_train_binary, model_own.wv)
X_test_vectors = text_to_vectors(X_test_binary, model_own.wv)

# Convert labels to binary format (class 1 and class 2)
y_train_binary = np.where(y_train_binary == 1, 0, 1)
y_test_binary = np.where(y_test_binary == 1, 0, 1)

# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train_vectors, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train_binary, dtype=torch.long)
X_test_tensor = torch.tensor(X_test_vectors, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test_binary, dtype=torch.long)

# Create DataLoader
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32)
```

[45]:
```
# Initialize the model
input_size = X_train_tensor.shape[2]  # Size of input features
output_size = 2  # Binary classification
model = CNN(input_size, output_size)

# Define Loss Function and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
# Train the model
train_cnn(model, train_loader, criterion, optimizer, num_epochs=10)

# Evaluate the model
evaluate_cnn(model, test_loader)
```

```
Epoch 1/10, Loss: 2332.43872615695
Epoch 2/10, Loss: 2261.077203631401
Epoch 3/10, Loss: 2236.7446866333485
Epoch 4/10, Loss: 2215.3790468871593
Epoch 5/10, Loss: 2202.6864687800407
Epoch 6/10, Loss: 2187.498899549246
Epoch 7/10, Loss: 2177.059113651514
Epoch 8/10, Loss: 2169.3832735419273
```

```
Epoch 9/10, Loss: 2164.6578074991703
Epoch 10/10, Loss: 2163.1868684887886
Accuracy: 0.7752763819095477
```

0.30 CNN with Ternary Classification with Pre-Trained Word2Vec model

Here I am using the dataset again from the initial data, and encoding the class labels from 1, 2 and 3 to 0, 1 and 2 as the class labels have to start from '0'.

I am using a CNN, with the input size as 300 features from the Word2Vec feature vectors and the output size to be 3 as we are working on a Ternary classification.

I am also printing the accuracy of the model on the test set.

[16]:
```
# Joining features and targets for training dataset
train_data = pd.concat([X_train_nltk_preprocessed, y_train], axis=1)
X_train = train_data['review_body']
y_train = train_data['sentiment']

# Joining features and targets for testing dataset
test_data = pd.concat([X_test, y_test], axis=1)
X_test = test_data['review_body']
y_test = test_data['sentiment']
```

[20]:
```
import gensim.downloader as api
wv = api.load('word2vec-google-news-300')
```

[24]:
```
# Preprocess text data
def preprocess_text(text, max_length=50):
    tokens = text.split()[:max_length]  # Limit maximum review length
    padded_tokens = tokens + ['<PAD>'] * (max_length - len(tokens))  # Pad␣
 ↪shorter reviews
    return padded_tokens

# Convert text data into Word2Vec vectors
def text_to_vectors(texts, wv, max_length=50):
    vectors = []
    for text in texts:
        tokens = preprocess_text(text, max_length)
        vector = [wv[word] if word in wv else np.zeros(wv.vector_size) for word␣
 ↪in tokens]
        vectors.append(vector)
    return np.array(vectors)

# Prepare data
X_train_vectors = text_to_vectors(X_train, wv)
X_test_vectors = text_to_vectors(X_test, wv)
```

```
# Convert labels to ternary format (classes 1, 2, and 3)
y_train_ternary = np.array(y_train)  # Convert Pandas Series to NumPy array
y_test_ternary = np.array(y_test)  # Convert Pandas Series to NumPy array

# Convert labels to the range [0, 1, 2]
y_train_ternary = y_train_ternary - 1
y_test_ternary = y_test_ternary - 1

# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train_vectors, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train_ternary, dtype=torch.long)
X_test_tensor = torch.tensor(X_test_vectors, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test_ternary, dtype=torch.long)

# Create DataLoader
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32)
```

[16]:
```
# Define the CNN model for three classes
class CNN(nn.Module):
    def __init__(self, input_size, output_size):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv1d(input_size, 50, kernel_size=3, padding=1)
        self.conv2 = nn.Conv1d(50, 10, kernel_size=3, padding=1)
        self.pool = nn.MaxPool1d(kernel_size=2, stride=2)
        self.fc = nn.Linear(10 * 12, output_size)  # Adjust the input size for␣
↪the linear layer

    def forward(self, x):
        x = self.pool(nn.functional.relu(self.conv1(x)))
        x = self.pool(nn.functional.relu(self.conv2(x)))
        x = x.view(-1, 10 * 12)  # Adjust the reshaping operation
        x = self.fc(x)
        return nn.functional.softmax(x, dim=1)

# Train the CNN model for three classes
def train_cnn(model, train_loader, criterion, optimizer, num_epochs):
    model.train()
    for epoch in range(num_epochs):
        running_loss = 0.0
        for inputs, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(inputs.permute(0, 2, 1))  # Permute input dimensions␣
↪for Conv1D
            loss = criterion(outputs, labels)
```

```
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        print(f"Epoch {epoch+1}/{num_epochs}, Loss: {running_loss}")

# Evaluation for three classes
def evaluate_cnn(model, test_loader):
    model.eval()
    correct = 0
    total = 0
    with torch.no_grad():
        for inputs, labels in test_loader:
            outputs = model(inputs.permute(0, 2, 1))  # Permute input dimensions␣
 ↪for Conv1D
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
    print(f"Accuracy: {correct / total}")
```

[26]:
```
# Initialize the model for ternary classification
input_size = X_train_vectors.shape[2]  # Size of input features
output_size = 3  # Ternary classification
model = CNN(input_size, output_size)

# Define Loss Function and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Train the model
train_cnn(model, train_loader, criterion, optimizer, num_epochs=10)

# Evaluate the model
evaluate_cnn(model, test_loader)
```

```
Epoch 1/10, Loss: 5506.197615265846
Epoch 2/10, Loss: 5268.540997862816
Epoch 3/10, Loss: 5184.202219247818
Epoch 4/10, Loss: 5130.5243673324585
Epoch 5/10, Loss: 5083.251011490822
Epoch 6/10, Loss: 5039.066096842289
Epoch 7/10, Loss: 5000.43970990181
Epoch 8/10, Loss: 4974.9873413443565
Epoch 9/10, Loss: 4947.3433557748795
Epoch 10/10, Loss: 4927.723435997963
Accuracy: 0.63098
```

0.31 CNN with Ternary Classification using the Word2Vec Custom Model

[17]:
```
# Joining features and targets for training dataset
train_data = pd.concat([X_train_nltk_preprocessed, y_train], axis=1)
X_train = train_data['review_body']
y_train = train_data['sentiment']

# Joining features and targets for testing dataset
test_data = pd.concat([X_test, y_test], axis=1)
X_test = test_data['review_body']
y_test = test_data['sentiment']
```

[28]:
```
from gensim.models import Word2Vec
# Train the Word2Vec model
model_own = Word2Vec(X_train_nltk_preprocessed_new, vector_size=300, window=11,␣
↪min_count=10)
```

[18]:
```
# Preprocess text data
def preprocess_text(text, max_length=50):
    tokens = text.split()[:max_length]  # Limit maximum review length
    padded_tokens = tokens + ['<PAD>'] * (max_length - len(tokens))  # Pad␣
↪shorter reviews
    return padded_tokens

# Convert text data into Word2Vec vectors
def text_to_vectors(texts, wv, max_length=50):
    vectors = []
    for text in texts:
        tokens = preprocess_text(text, max_length)
        vector = [wv[word] if word in wv else np.zeros(wv.vector_size) for word␣
↪in tokens]
        vectors.append(vector)
    return np.array(vectors)

# Prepare data
X_train_vectors = text_to_vectors(X_train, model_own.wv)
X_test_vectors = text_to_vectors(X_test, model_own.wv)

# Convert labels to ternary format (classes 1, 2, and 3)
y_train_ternary = np.array(y_train)  # Convert Pandas Series to NumPy array
y_test_ternary = np.array(y_test)  # Convert Pandas Series to NumPy array

# Convert labels to the range [0, 1, 2]
y_train_ternary = y_train_ternary - 1
y_test_ternary = y_test_ternary - 1

# Convert data to PyTorch tensors
```

```
X_train_tensor = torch.tensor(X_train_vectors, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train_ternary, dtype=torch.long)
X_test_tensor = torch.tensor(X_test_vectors, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test_ternary, dtype=torch.long)

# Create DataLoader
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32)
```

[19]:
```
# Initialize the model for ternary classification
input_size = X_train_vectors.shape[2]  # Size of input features
output_size = 3  # Ternary classification
model = CNN(input_size, output_size)

# Define Loss Function and Optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Train the model
train_cnn(model, train_loader, criterion, optimizer, num_epochs=10)

# Evaluate the model
evaluate_cnn(model, test_loader)
```

```
Epoch 1/10, Loss: 5383.035078883171
Epoch 2/10, Loss: 5289.389633178711
Epoch 3/10, Loss: 5247.818126320839
Epoch 4/10, Loss: 5223.400608718395
Epoch 5/10, Loss: 5203.720783531666
Epoch 6/10, Loss: 5194.734705924988
Epoch 7/10, Loss: 5186.168330729008
Epoch 8/10, Loss: 5181.665784597397
Epoch 9/10, Loss: 5175.856912493706
Epoch 10/10, Loss: 5178.5594519376755
Accuracy: 0.62016
```

References:

1. PyTorch Official Documentation: https://pytorch.org/docs/stable/index.html
2. PyTorch Tutorials on GitHub: https://github.com/pytorch/tutorials
3. Deep Learning with PyTorch Book: https://www.manning.com/books/deep-learning-with-pytorch
4. PyTorch Lightning GitHub Repository: https://github.com/PyTorchLightning/pytorch-lightning

5. Stanford CS231n Course Website: http://cs231n.stanford.edu/

6. Fast.ai: https://www.fast.ai/