Name: Payal Rashinkar USC ID: 3885-1419-03 Homework: 2 Course: CSCI-544 Library: Pytorch

Number	Word2Vec	Method	Class	Accuracy in %
1	Pre-Trained	Perceptron	Binary	70.1
2	Pre-Trained	SVM	Binary	81.3
3	Custom	Perceptron	Binary	76.1
4	Custom	SVM	Binary	83.9
5	TF-IDF(N/A)	Perceptron	Binary	84.4
6	TF-IDF(N/A)	SVM	Binary	88.7
7	Pre-Trained	FFNN[Avg]	Binary	80
8	Pre-Trained	FFNN[Avg]	Ternary	58.2
9	Custom	FFNN[Avg]	Binary	85
10	Custom	FFNN[Avg]	Ternary	64.6
11	Pre-Trained	FFNN[Cat]	Binary	77.4
12	Pre-Trained	FFNN[Cat]	Ternary	62.84
13	Custom	FFNN[Cat]	Binary	71.6
14	Custom	FFNN[Cat]	Ternary	72.27
15	Pre-Trained	CNN	Binary	50.42
16	Pre-Trained	CNN	Ternary	53.19
17	Custom	CNN	Binary	64.90
18	Custom	CNN	Ternary	57.15

2) What do you conclude from comparing vectors generated by yourself and the pretrained model? Which of the Word2Vec models encodes semantic similarities between words better?

The custom model demonstrates superior performance over the pretrained model, especially after extensive data preprocessing steps such as cleaning, lemmatization, and expanding abbreviations. This indicates that our tailored approach outshines the generic pretrained model in terms of accuracy. Furthermore, it effectively measures the similarity between two closely related words, as illustrated previously, highlighting its nuanced understanding of word relationships.

3) What do you conclude from comparing performances for the models trained using the three feature types (TF-IDF, pre-trained Word2Vec, your trained Word2Vec)?

In the context of the specified classification task, features derived from TF-IDF demonstrate greater efficacy for the Perceptron and SVM models than those obtained from Word2Vec. The superior accuracy rates associated with TF-IDF indicate its enhanced capability in encapsulating the essential information required for classification within this specific scenario. Nonetheless, it's crucial to recognize that the selection of feature type may vary based on the unique characteristics of the data and the task at hand, implying that various tasks favor distinct types of feature representations.

4) What do you conclude by comparing the accuracy values you obtain with those obtained in the "' Simple Models" section (note you can compare the accuracy values for binary classification)?

The comparison of accuracy rates across various models shows that the SVM model, when trained using TF-IDF features, leads the pack with an impressive accuracy rate of 88.7%, showcasing its outstanding performance for this specific task. Following this, the Perceptron model that employs TF-IDF features records a respectable accuracy of 84.4%, indicating a solid performance. In the Feed Forward Neural Network (FNN) models, the variant that utilizes average Word2Vec vectors achieves a comparable accuracy of 80%, matching the SVM's performance but not exceeding that of the Perceptron model equipped with TF-IDF features. The FNN model is designed to concatenate the initial 10 Word2Vec vectors for each review as its input features fall slightly behind, registering an accuracy of 77.4%. These findings hint that, within the context of this particular classification challenge, the more straightforward Perceptron-TFIDF and SVM-Word2Vec models are more successful than the FNN models, which might see improvement with additional tuning. Compared to ternary, binary classification shows slightly better accuracy.

Test Cases covered in this .ipynb file

1. SVM Pretrained Binary (model1=pretrain)
2. SVM Custom Binary (model2=custom)
3. Perceptron Pretrained Binary
4. Perceptron Custom Binary
5. SVM TF/IDF Binary
6. Perceptron TF/IDF Binary
7. (4a)FFNN AVG Custom Binary
8. (4a)FFNN AVG Custom Ternary
9. (4b)FFNN CONCAT Custom Binary

Remaining will be covered in different .ipynb due to memory constraints

Import the required packages

In [1]:
```
import time
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
import re
import nltk
from nltk.corpus import stopwords
nltk.download('omw-1.4')
nltk.download('stopwords')
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import Perceptron
from sklearn.metrics import precision_recall_fscore_support
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from gensim.models import Word2Vec
from gensim.models import KeyedVectors
from sklearn.metrics.pairwise import cosine_similarity
import gensim.downloader as api
```

```
[nltk_data] Downloading package omw-1.4 to
[nltk_data]     /Users/payalrashinkar/nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/payalrashinkar/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

Get the dataset into dataframe

In [2]:
```
df_1 = pd.read_csv('https://web.archive.org/web/20201127142707if_/https://s3.amazonaws.c
```

1. Data Generation

1.a Copy the df_1 into df to get only required 2 columns i.e 'ratings' and

'reviews' after renaming them:

In [3]:
```
df = df_1.copy()
df = df[['star_rating', 'review_body']].rename(columns={'star_rating':'ratings', 'review
df.head()
```

Out[3]:

	ratings	reviews
0	5	Great product.
1	5	What's to say about this commodity item except...
2	5	Haven't used yet, but I am sure I will like it.
3	1	Although this was labeled as "new" the...
4	4	Gorgeous colors and easy to use

1.a Solution:

Copying the dataframe df_1 to df and only have two columns in df i.e. ratings and reviews after renaming it from start_rating and review_body respectively.

1.b Creating 50k dataset randomly for each of 5 ratings:

In [4]:
```
balanced_df = pd.DataFrame()
for ratings in ['1','2', '3', '4', '5']:
    sampled_df = df[df['ratings'] == ratings].sample(n=50000, random_state=42)
    balanced_df = pd.concat([balanced_df, sampled_df])
    print(f"Rating {ratings}: {len(sampled_df)} datasets")
```

```
Rating 1: 50000 datasets
Rating 2: 50000 datasets
Rating 3: 50000 datasets
Rating 4: 50000 datasets
Rating 5: 50000 datasets
```

1.b Solution:

```
1. The code iterates over a list of ratings ('1', '2', '3', '4', ‘5’).
For each rating, it filters the DataFrame df to include only rows with
the current rating, then samples 50,000 rows from this filtered subset
using the sampling method with a fixed random_state for reproducibility.
2. These sampled rows are concatenated to a new DataFrame balanced_df,
creating a balanced dataset with an equal number of rows for each
rating.
3. After each iteration, the code prints the number of rows sampled for
the current rating, summarizing how many datasets are included for each
rating in balanced_df.
```


1.c Creating column called "class" to group the sentiment based on rating:

In [5]:
```
# Create ternary labels
def label_sentiment(row):
    if row['ratings'] > '3':
        return 1  # Positive
```

```
    elif row['ratings'] < '3':
        return 2  # Negative
    else:
        return 3  # Neutral


balanced_df['class'] = balanced_df.apply(label_sentiment, axis=1)
```

In [6]:
```
print(balanced_df.head())
```

```
        ratings                                            reviews  class
1846471       1  Staggering amount of &#34;negative reviews&#34...      2
957331        1  Worst labels ever! I purchased these labels to...      2
2474769       1  This product had to be returned. On the first ...      2
1045185       1  Two of the four ink cartridges were empty. You...      2
687232        1                        didn't even stay on one night      2
```

1.c Solution:

```
1. The code defines a function label_sentiment that assigns ternary
labels (1 for Positive, 2 for Negative, 3 for Neutral) based on the
'ratings' column of each row in the balanced_df DataFrame, considering
ratings higher than '3' as positive, lower than '3' as negative, and
equal to '3' as neutral.
2. It then applies this function to each row of balanced_df to create a
new column 'class' with the corresponding sentiment labels and prints
the first five rows of the DataFrame to display the output.
```


1.d Split the dataset into training and testing sets:

In [7]:
```
train_df, test_df = train_test_split(balanced_df, test_size=0.2, random_state=42)
```

In [8]:
```
print(f"80% Train data: {train_df['ratings'].count()} datasets")
print(f"20% Test data: {test_df['ratings'].count()} datasets")
```

```
80% Train data: 200000 datasets
20% Test data: 50000 datasets
```

1.d Solution: The code splits the balanced_df DataFrame into training and testing sets with an 80%/20% ratio using a fixed random_state for reproducibility and then prints the number of datasets (rows) in each resulting subset.

2. Word Embedding

In [9]:
```
# load the pretrained model as model1
pretrained = 'word2vec-google-news-300.gz'
model1 = KeyedVectors.load_word2vec_format(pretrained, binary=True)
```

In [10]:
```
# check the similarity between two similar words
print(model1.similarity('excellent', 'outstanding'))

# find out the corresponding word given that A - B = C - D (King - Man = Queen - Woman)
print(model1.most_similar(positive=['king', 'woman'], negative=['man'], topn=1))
```

```
0.55674857
[('queen', 0.7118193507194519)]
```

In [11]:
```
X, y = balanced_df['reviews'].fillna('').tolist(), balanced_df['class'].tolist()
```

In [12]:
```
# convert reviews to lower case
X = [str(x).lower() for x in X]
# remove HTML and URLs from reviews
X = [re.sub('<.*>', '', x) for x in X]
X = [re.sub(r'https?://\S+', '', x) for x in X]
# remove non-alphabetical characters
X = [re.sub('[^a-z ]', '', x) for x in X]
# remove extra spaces
X = [re.sub(' +', ' ', x) for x in X]
```

In [13]:
```
# expand contractions
contractions = {
"ain't": "am not",
"aren't": "are not",
"can't": "cannot",
"can't've": "cannot have",
"'cause": "because",
"could've": "could have",
"couldn't": "could not",
"couldn't've": "could not have",
"didn't": "did not",
"doesn't": "does not",
"don't": "do not",
"hadn't": "had not",
"hadn't've": "had not have",
"hasn't": "has not",
"haven't": "have not",
"he'd": "he would",
"he'd've": "he would have",
"he'll": "he will",
"he'll've": "he will have",
"he's": "he is",
"how'd": "how did",
"how'd'y": "how do you",
"how'll": "how will",
"how's": "how is",
"I'd": "I would",
"I'd've": "I would have",
"I'll": "I will",
"I'll've": "I will have",
"I'm": "I am",
"I've": "I have",
"isn't": "is not",
"it'd": "it would",
"it'd've": "it would have",
"it'll": "it will",
"it'll've": "it will have",
"it's": "it is",
"let's": "let us",
"ma'am": "madam",
"mayn't": "may not",
"might've": "might have",
"mightn't": "might not",
"mightn't've": "might not have",
"must've": "must have",
"mustn't": "must not",
"mustn't've": "must not have",
"needn't": "need not",
"needn't've": "need not have",
"o'clock": "of the clock",
```

```
"oughtn't": "ought not",
"oughtn't've": "ought not have",
"shan't": "shall not",
"sha'n't": "shall not",
"shan't've": "shall not have",
"she'd": "she would",
"she'd've": "she would have",
"she'll": "she will",
"she'll've": "she will have",
"she's": "she is",
"should've": "should have",
"shouldn't": "should not",
"shouldn't've": "should not have",
"so've": "so have",
"so's": "so is",
"that'd": "that would",
"that'd've": "that would have",
"that's": "that is",
"there'd": "there would",
"there'd've": "there would have",
"there's": "there is",
"they'd": "they would",
"they'd've": "they would have",
"they'll": "they will",
"they'll've": "they will have",
"they're": "they are",
"they've": "they have",
"to've": "to have",
"wasn't": "was not",
"we'd": "we would",
"we'd've": "we would have",
"we'll": "we will",
"we'll've": "we will have",
"we're": "we are",
"we've": "we have",
"weren't": "were not",
"what'll": "what will",
"what'll've": "what will have",
"what're": "what are",
"what's": "what is",
"what've": "what have",
"when's": "when is",
"when've": "when have",
"where'd": "where did",
"where's": "where is",
"where've": "where have",
"who'll": "who will",
"who'll've": "who will have",
"who's": "who is",
"who've": "who have",
"why's": "why is",
"why've": "why have",
"will've": "will have",
"won't": "will not",
"won't've": "will not have",
"would've": "would have",
"wouldn't": "would not",
"wouldn't've": "would not have",
"y'all": "you all",
"y'all'd": "you all would",
"y'all'd've": "you all would have",
"y'all're": "you all are",
"y'all've": "you all have",
"you'd": "you would",
"you'd've": "you would have",
"you'll": "you will",
```

```
"you'll've": "you will have",
"you're": "you are",
"you've": "you have"
}
def decontraction(s):
    for word in s.split(' '):
        if word in contractions.keys():
            s = re.sub(word, contractions[word], s)
    return s
X = [decontraction(x) for x in X]
```

In [14]:
```
# remove stop words
stopWords =set(stopwords.words('english'))
def remvstopWords(s):
    wordlist = s.split(' ')
    newlist = []
    for word in wordlist:
        if word not in stopWords:
            newlist.append(word)
    s = ' '.join(newlist)
    return s

X = list(map(remvstopWords, X))
```

In [15]:
```
# perform lemmatization
wnl = WordNetLemmatizer()
X = [' '.join([wnl.lemmatize(word) for word in x.split(' ')]) for x in X]
```

In [16]:
```
# train a word2vec model using my own dataset
# convert X_train to a list of lists of words
sentences = [x.split(' ') for x in X]

# use X_train to train a word2vec model2
model2 = Word2Vec(vector_size=300, window=11, min_count=10)
model2.build_vocab(sentences)
model2.train(sentences, total_examples=model2.corpus_count, epochs=model2.epochs)
```

Out[16]:
```
(29065883, 32427760)
```

In [17]:
```
# save the trained model
model2.save('my-own-word2vec.model')
# store just the words + their trained embeddings
word_vectors = model2.wv
word_vectors.save('my-own-word2vec.wordvectors')
```

In [18]:
```
model2 = KeyedVectors.load('my-own-word2vec.wordvectors', mmap='r')
```

In [19]:
```
# check the similarity between two similar words using model 2
print(model2.similarity('excellent', 'outstanding'))

# find out the corresponding word given that A - B = C - D (King - Man = Queen - Woman)
print(model2.most_similar(positive=['king', 'woman'], negative=['man'], topn=1))
```

```
0.79444706
[('graduated', 0.5376219153404236)]
```

2 Answer: The custom model demonstrates superior performance over the pretrained model, especially after extensive data preprocessing steps such as cleaning, lemmatization, and expanding abbreviations. This indicates that our tailored approach outshines the generic pretrained model in terms of accuracy. Furthermore, it effectively measures the similarity between

two closely related words, as illustrated previously, highlighting its nuanced understanding of word relationships.

3. Simple Models

In [20]:
```
# Split the downsized dataset into 80% training dataset and 20% testing dataset.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)
```

In [21]:
```
# turn the original ternary training and testing datasets to binary
# return a list of indices of y = 1 or y =2
y_train_bi = []
idx_train = []
for i, y in enumerate(y_train):
    if y == 1 or y == 2:
        y_train_bi.append(y)
        idx_train.append(i)

y_test_bi = []
idx_test = []
for i, y in enumerate(y_test):
    if y == 1 or y == 2:
        y_test_bi.append(y)
        idx_test.append(i)
# use the list of indices to select a sub-list of X_train
X_train_bi = [X_train[i] for i in idx_train]
X_test_bi = [X_test[i] for i in idx_test]
```

In [22]:
```
# take the average word vectors of important words (i.e. non-stop words) in
# a review as the feature of a training sample
X_train_bi1 = []
for x in X_train_bi:
    wordveclist = []
    for word in x.split(' '):
        try:
            wordvec = model1[word]
            wordveclist.append(wordvec)
        except:
            pass
    X_train_bi1.append(np.mean(wordveclist, axis=0))
```

```
/Users/payalrashinkar/anaconda3/lib/python3.11/site-packages/numpy/core/fromnumeric.py:3
464: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/Users/payalrashinkar/anaconda3/lib/python3.11/site-packages/numpy/core/_methods.py:192:
RuntimeWarning: invalid value encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
```

In [23]:
```
X_test_bi1 = []
for x in X_test_bi:
    wordveclist = []
    for word in x.split(' '):
        try:
            wordvec = model1[word]
            wordveclist.append(wordvec)
        except:
            pass
    X_test_bi1.append(np.mean(wordveclist, axis=0))
```

In [24]:
```
# handle non-word-vector values in the dataset
# return the indices of word-vector values
wv_train = []
```

```
for i, x in enumerate(X_train_bi1):
    try:
        len(x)
        wv_train.append(i)
    except:
        pass


wv_test = []
for i, x in enumerate(X_test_bi1):
    try:
        len(x)
        wv_test.append(i)
    except:
        pass

# remove the non-word-vector values from the dataset
X_train_bi1 = [X_train_bi1[i] for i in wv_train]
X_test_bi1 = [X_test_bi1[i] for i in wv_test]
y_train_bi1 = [y_train_bi[i] for i in wv_train]
y_test_bi1 = [y_test_bi[i] for i in wv_test]
```

3.1a Pretrained word2vec google model for perceptron:

In [25]:

```
# use pretrained word2vec features to train a perceptron
perceptron = Perceptron(random_state=1)
perceptron.fit(X_train_bi1, y_train_bi1)
y_train_predict1, y_test_predict1 = perceptron.predict(X_train_bi1), perceptron.predict(

# report accuracy, precision, recall, and f1-score on both the training and testing spli
train_stats = precision_recall_fscore_support(y_train_bi1, y_train_predict1, average='bi
precision_train, recall_train, fscore_train = train_stats[0], train_stats[1], train_stat

test_stats = precision_recall_fscore_support(y_test_bi1, y_test_predict1, average='binar
precision_test, recall_test, fscore_test = test_stats[0], test_stats[1], test_stats[2]

print('The accuracy of testing dataset: {:2.1%}'.format(perceptron.score(X_test_bi1, y_t
```

```
The accuracy of testing dataset: 70.1%
```

3.1b Pretrained word2vec google model for SVM:

In [26]:

```
svm = LinearSVC(random_state=1)
svm.fit(X_train_bi1, y_train_bi1)

y_train_predict1, y_test_predict1 = svm.predict(X_train_bi1), svm.predict(X_test_bi1)

# report accuracy, precision, recall, and f1-score on both the training and testing spli
train_stats = precision_recall_fscore_support(y_train_bi1, y_train_predict1, average='bi
precision_train, recall_train, fscore_train = train_stats[0], train_stats[1], train_stat

test_stats = precision_recall_fscore_support(y_test_bi1, y_test_predict1, average='binar
precision_test, recall_test, fscore_test = test_stats[0], test_stats[1], test_stats[2]

print('The accuracy of testing dataset: {:2.1%}'.format(svm.score(X_test_bi1, y_test_bi1
```

```
/Users/payalrashinkar/anaconda3/lib/python3.11/site-packages/sklearn/svm/_classes.py:32:
FutureWarning: The default value of `dual` will change from `True` to `'auto'` in 1.5. S
et the value of `dual` explicitly to suppress the warning.
  warnings.warn(
The accuracy of testing dataset: 81.1%
```

In [27]:

```
# take the average word vectors of important words (i.e. non-stop words) in
```

```
# a review as the feature of a training sample
X_train_bi2 = []
for x in X_train_bi:
    wordveclist = []
    for word in x.split(' '):
        try:
            wordvec = model2[word]
            wordveclist.append(wordvec)
        except:
            pass
    X_train_bi2.append(np.mean(wordveclist, axis=0))

# do the same to the testing dataset
X_test_bi2 = []
for x in X_test_bi:
    wordveclist = []
    for word in x.split(' '):
        try:
            wordvec = model2[word]
            wordveclist.append(wordvec)
        except:
            pass
    X_test_bi2.append(np.mean(wordveclist, axis=0))
```

```
/Users/payalrashinkar/anaconda3/lib/python3.11/site-packages/numpy/core/fromnumeric.py:3
464: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/Users/payalrashinkar/anaconda3/lib/python3.11/site-packages/numpy/core/_methods.py:192:
RuntimeWarning: invalid value encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
```

In [58]:

```
# handle non-word-vector values in the dataset
# return the indices of word-vector values
wv_train = []
for i, x in enumerate(X_train_bi2):
    try:
        len(x)
        wv_train.append(i)
    except:
        pass


wv_test = []
for i, x in enumerate(X_test_bi2):
    try:
        len(x)
        wv_test.append(i)
    except:
        pass

# remove the non-word-vector values from the dataset
X_train_bi2 = [X_train_bi2[i] for i in wv_train]
X_test_bi2 = [X_test_bi2[i] for i in wv_test]
y_train_bi2 = [y_train_bi[i] for i in wv_train]
y_test_bi2 = [y_test_bi[i] for i in wv_test]
```

3.2a Our word2vec model for perceptron:

In [29]:

```
# use my own word2vec features to train a perceptron
perceptron = Perceptron(random_state=1)
perceptron.fit(X_train_bi2, y_train_bi2)
y_train_predict2, y_test_predict2 = perceptron.predict(X_train_bi2), perceptron.predict(

# report accuracy, precision, recall, and f1-score on both the training and testing spli
```

```
train_stats = precision_recall_fscore_support(y_train_bi2, y_train_predict2, average='bi
precision_train, recall_train, fscore_train = train_stats[0], train_stats[1], train_stat

test_stats = precision_recall_fscore_support(y_test_bi2, y_test_predict2, average='binar
precision_test, recall_test, fscore_test = test_stats[0], test_stats[1], test_stats[2]

print('The accuracy of testing dataset: {:2.1%}'.format(perceptron.score(X_test_bi2, y_t
```

```
The accuracy of testing dataset: 70.9%
```

3.2b Our word2vec model for SVM:

In [30]:

```
svm = LinearSVC(random_state=1)
svm.fit(X_train_bi2, y_train_bi2)

y_train_predict2, y_test_predict2 = svm.predict(X_train_bi2), svm.predict(X_test_bi2)

# report accuracy, precision, recall, and f1-score on both the training and testing spli
train_stats = precision_recall_fscore_support(y_train_bi2, y_train_predict2, average='bi
precision_train, recall_train, fscore_train = train_stats[0], train_stats[1], train_stat

test_stats = precision_recall_fscore_support(y_test_bi2, y_test_predict2, average='binar
precision_test, recall_test, fscore_test = test_stats[0], test_stats[1], test_stats[2]

print('The accuracy of testing dataset: {:2.1%}'.format(svm.score(X_test_bi2, y_test_bi2
```

```
/Users/payalrashinkar/anaconda3/lib/python3.11/site-packages/sklearn/svm/_classes.py:32:
FutureWarning: The default value of `dual` will change from `True` to `'auto'` in 1.5. S
et the value of `dual` explicitly to suppress the warning.
  warnings.warn(
/Users/payalrashinkar/anaconda3/lib/python3.11/site-packages/sklearn/svm/_base.py:1242:
ConvergenceWarning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn(
The accuracy of testing dataset: 84.1%
```

3.3a Our TF-IDF model for perceptron and SVM from homework1:

In [31]:

```
# As executed from Homework1:
#=============================

#Perceptron for TF-IDF
#======================
#Perceptron, accuracy of testing dataset: 84.4%

#SVM for TF-IDF
#================
#SVM, accuracy of testing dataset: 88.7%
```

3 Answer: In the context of the specified classification task, features derived from TF-IDF demonstrate greater efficacy for the Perceptron and SVM models than those obtained from Word2Vec. The superior accuracy rates associated with TF-IDF indicate its enhanced capability in encapsulating the essential information required for classification within this specific scenario. Nonetheless, it's crucial to recognize that the selection of feature type might vary based on the unique characteristics of the data and the task at hand, implying that various tasks might favor distinct types of feature representations.

4. Feedforward Neural Network

In [32]:
```
import torch
from torch.utils.data import DataLoader, Dataset
import torch.nn as nn
import torch.nn.functional as F
import functools
from sklearn.metrics import accuracy_score
```

In [33]:
```
# Enable CUDA for PyTorch
use_cuda = torch.cuda.is_available()
device = torch.device("cuda:0" if use_cuda else "cpu")
torch.backends.cudnn.benchmark = True
```

In [34]:
```
# Hyperparameters
params = {'batch_size': 64,
          'shuffle': True,
          'num_workers': 0}
max_epochs = 10
```

In [35]:
```
# Override the Dataset class
class Dataset(Dataset):

    def __init__(self, list_IDs, labels):
        'Initialization'
        self.list_IDs = list_IDs
        self.labels = labels

    def __len__(self):
        'Denotes the total number of samples'
        return len(self.list_IDs)

    def __getitem__(self, index):
        'Generates one sample of data'
        # Select sample
        ID = self.list_IDs[index]

        # Load data and get label
        X = torch.load('data/' + ID + '.pt')
        y = self.labels[index]
        return X, y
```

Use the average of the Word2Vec vectors for each review as the input feature

4a

1. Binary classification custom model:

In [36]:
```
train_IDs = {}
y_train = []
len_train_IDs = int(0.8 * len(X_train_bi2))
#print(len_train_IDs)
for i in range(len_train_IDs):
    train_IDs[i] = 'train_bi_' + str(i)
    y_train.append(y_train_bi2[i] - 1) # Convert label from 1 and 2 to 0 and 1

#print(y_train)
valid_IDs = {}
y_valid = []
len_valid_IDs = len(X_train_bi2) - len_train_IDs
```

```python
for i in range(len_valid_IDs):
    valid_IDs[i] = 'valid_bi_' + str(i)
    y_valid.append(y_train_bi2[len(y_train) + i] - 1)

test_IDs = {}
len_test_IDs = len(X_test_bi2)
for i in range(len_test_IDs):
    test_IDs[i] = 'test_bi_' + str(i)

for i in range(len(train_IDs)):
    torch.save(X_train_bi2[i], 'data/' + train_IDs[i] + '.pt')

for i in range(len(valid_IDs)):
    torch.save(X_train_bi2[len(train_IDs) + i], 'data/' + valid_IDs[i] + '.pt')

for i in range(len(test_IDs)):
    torch.save(X_test_bi2[i], 'data/' + test_IDs[i] + '.pt')
```

In [37]:
```python
train_set = Dataset(train_IDs, y_train)
valid_set = Dataset(valid_IDs, y_valid)
test_set = Dataset(test_IDs, y_test_bi2)

# Generate dataloaders for the training, validation and testing datasets
train_loader = torch.utils.data.DataLoader(train_set, **params)
valid_loader = torch.utils.data.DataLoader(valid_set, **params)
test_loader = torch.utils.data.DataLoader(test_set, **params)
```

In [38]:
```python
# Define the network architecture
class Net(nn.Module):
    def __init__(self, input_dim=300, output_dim=2, hidden_1=50, hidden_2=10):
        super(Net, self).__init__()
        # dimension of inputs
        self.input_dim = input_dim
        # number of classes (2 for binary, 3 for ternary, ...)
        self.output_dim = output_dim
        # number of nodes in each hidden layer
        self.hidden_1 = hidden_1
        self.hidden_2 = hidden_2
        # linear layer (input --> hidden_1)
        self.fc1 = nn.Linear(self.input_dim, self.hidden_1)
        # linear layer (hidden_1 --> hidden_2)
        self.fc2 = nn.Linear(self.hidden_1, self.hidden_2)
        # linear layer (hidden_2 --> 2)
        self.fc3 = nn.Linear(self.hidden_2, self.output_dim)
        # dropout prevents overfitting of data
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        # add the 1st hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer after the 1st hidden layer
        x = self.dropout(x)
        # add the 2nd hidden layer, with relu activation function
        x = F.relu(self.fc2(x))
        # add dropout layer after the 2nd hidden layer
        x = self.dropout(x)
        # add output layer
        x = self.fc3(x)
        return x
```

In [39]:
```python
# initialize the NN
model = Net()
print(model)
```

```
Net(
```

```
  (fc1): Linear(in_features=300, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
  (fc3): Linear(in_features=10, out_features=2, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

In [40]:
```
# specify cross entropy loss as loss function
criterion = nn.CrossEntropyLoss()

# specify optimizer (stochastic gradient descent) and learning rate = 0.01
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

In [41]:
```
# train the network

# initialize tracker for minimum validation loss
valid_loss_min = np.Inf # set initial "min" to infinity

for epoch in range(max_epochs):
    # monitor training loss
    train_loss = 0.0
    valid_loss = 0.0

    model.train() # prep model for training
    for data, target in train_loader:
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update running training loss
        train_loss += loss.item()*data.size(0)

    model.eval() # prep model for evaluation
    for data, target in valid_loader:
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # update running validation loss
        valid_loss += loss.item()*data.size(0)

    # print training/validation statistics
    # calculate average loss over an epoch
    train_loss = train_loss/len(train_loader.dataset)
    valid_loss = valid_loss/len(valid_loader.dataset)

    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
        epoch+1,
        train_loss,
        valid_loss
        ))

    # save model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
        valid_loss_min,
        valid_loss))
        torch.save(model.state_dict(), 'model.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1        Training Loss: 0.541041         Validation Loss: 0.412022
Validation loss decreased (inf --> 0.412022).  Saving model ...
```

```
Epoch: 2        Training Loss: 0.417179         Validation Loss: 0.383492
Validation loss decreased (0.412022 --> 0.383492).  Saving model ...
Epoch: 3        Training Loss: 0.397067         Validation Loss: 0.374651
Validation loss decreased (0.383492 --> 0.374651).  Saving model ...
Epoch: 4        Training Loss: 0.390049         Validation Loss: 0.369608
Validation loss decreased (0.374651 --> 0.369608).  Saving model ...
Epoch: 5        Training Loss: 0.383501         Validation Loss: 0.365544
Validation loss decreased (0.369608 --> 0.365544).  Saving model ...
Epoch: 6        Training Loss: 0.379078         Validation Loss: 0.361988
Validation loss decreased (0.365544 --> 0.361988).  Saving model ...
Epoch: 7        Training Loss: 0.376475         Validation Loss: 0.360423
Validation loss decreased (0.361988 --> 0.360423).  Saving model ...
Epoch: 8        Training Loss: 0.372543         Validation Loss: 0.358516
Validation loss decreased (0.360423 --> 0.358516).  Saving model ...
Epoch: 9        Training Loss: 0.369885         Validation Loss: 0.355467
Validation loss decreased (0.358516 --> 0.355467).  Saving model ...
Epoch: 10       Training Loss: 0.368100         Validation Loss: 0.355730
```

In [42]:
```
## Test the trained network
def predict(model, dataloader):
    prediction_list = []
    truth_list = []
    for i, batch in enumerate(dataloader):
        # batch[0] is X, and batch[1] is y
        outputs = model(batch[0])
        _, predicted = torch.max(outputs.data, 1)
        prediction_list.append(predicted)
        truth_list.append(batch[1])
    return prediction_list, truth_list
```

In [43]:
```
predictions, truths = predict(model, test_loader)
# Convert predictions and truths from list of tensors to list of lists
predictions = [list(torch.Tensor.numpy(t)) for t in predictions]
truths = [list(torch.Tensor.numpy(t)) for t in truths]
# Convert predictions and truths from list of lists to a single list
predictions = functools.reduce(lambda a, b: a + b, predictions)
truths = functools.reduce(lambda a, b: a + b, truths)
# Convert predictions from (0, 1) to (1, 2)
predictions = [p + 1 for p in predictions]
```

In [44]:
```
# report accuracy, precision, recall, and f1-score on the testing dataset

test_stats = precision_recall_fscore_support(truths, predictions, average='binary')
precision_test, recall_test, fscore_test = test_stats[0], test_stats[1], test_stats[2]

print('The accuracy of testing dataset: {:2.1%}'.format(accuracy_score(truths, predictio
```

```
The accuracy of testing dataset: 85.0%
```

2 Ternary Classification custom model

In [45]:
```
X, y = balanced_df['reviews'].fillna('').tolist(), balanced_df['class'].tolist()
```

In [46]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)
```

In [47]:
```
X_train_te2 = []
for x in X_train:
    wordveclist = []
    for word in x.split(' '):
        try:
            wordvec = model2[word]
            wordveclist.append(wordvec)
        except:
```

```
            pass
    X_train_te2.append(np.mean(wordveclist, axis=0))

# do the same to the testing dataset
X_test_te2 = []
for x in X_test:
    wordveclist = []
    for word in x.split(' '):
        try:
            wordvec = model2[word]
            wordveclist.append(wordvec)
        except:
            pass
    X_test_te2.append(np.mean(wordveclist, axis=0))
```

```
/Users/payalrashinkar/anaconda3/lib/python3.11/site-packages/numpy/core/fromnumeric.py:3
464: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/Users/payalrashinkar/anaconda3/lib/python3.11/site-packages/numpy/core/_methods.py:192:
RuntimeWarning: invalid value encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
```

In [48]:

```
# handle non-word-vector values in the dataset
# return the indices of word-vector values
wv_train = []
for i, x in enumerate(X_train_te2):
    try:
        len(x)
        wv_train.append(i)
    except:
        pass


wv_test = []
for i, x in enumerate(X_test_te2):
    try:
        len(x)
        wv_test.append(i)
    except:
        pass

# remove the non-word-vector values from the dataset
X_train_te2 = [X_train_te2[i] for i in wv_train]
X_test_te2 = [X_test_te2[i] for i in wv_test]
y_train_te2 = [y_train[i] for i in wv_train]
y_test_te2 = [y_test[i] for i in wv_test]
```

In [49]:

```
train_IDs = {}
y_train = []
len_train_IDs = int(0.8 * len(X_train_te2))

for i in range(len_train_IDs):
    train_IDs[i] = 'train_te_' + str(i)
    y_train.append(y_train_te2[i] - 1) # Convert label from 1, 2 and 3 to 0, 1 and 2



valid_IDs = {}
y_valid = []
len_valid_IDs = len(X_train_te2) - len_train_IDs
for i in range(len_valid_IDs):
    valid_IDs[i] = 'valid_te_' + str(i)
    y_valid.append(y_train_te2[len(y_train) + i] - 1)

test_IDs = {}
```

```
len_test_IDs = len(X_test_te2)
for i in range(len_test_IDs):
    test_IDs[i] = 'test_te_' + str(i)

for i in range(len(train_IDs)):
    torch.save(X_train_te2[i], 'data/' + train_IDs[i] + '.pt')

for i in range(len(valid_IDs)):
    torch.save(X_train_te2[len(train_IDs) + i], 'data/' + valid_IDs[i] + '.pt')

for i in range(len(test_IDs)):
    torch.save(X_test_te2[i], 'data/' + test_IDs[i] + '.pt')
```

In [50]:
```
# Generate training, validation and testing datasets
train_set = Dataset(train_IDs, y_train)
valid_set = Dataset(valid_IDs, y_valid)
test_set = Dataset(test_IDs, y_test_te2)

# Generate dataloaders for the training, validation and testing datasets
train_loader = torch.utils.data.DataLoader(train_set, **params)
valid_loader = torch.utils.data.DataLoader(valid_set, **params)
test_loader = torch.utils.data.DataLoader(test_set, **params)
```

In [51]:
```
# initialize the NN
# use model_te to be distinguishable from the binary model
model_te = Net(output_dim=3)
print(model_te)
```

```
Net(
  (fc1): Linear(in_features=300, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
  (fc3): Linear(in_features=10, out_features=3, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

In [52]:
```
# specify cross entropy loss as loss function
criterion = nn.CrossEntropyLoss()

# specify optimizer (stochastic gradient descent) and learning rate = 0.01
optimizer = torch.optim.SGD(model_te.parameters(), lr=0.01)
```

In [53]:
```
# train the network

# initialize tracker for minimum validation loss
valid_loss_min = np.Inf # set initial "min" to infinity

for epoch in range(max_epochs):
    # monitor training loss
    train_loss = 0.0
    valid_loss = 0.0


    model_te.train() # prep model for training
    for data, target in train_loader:
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model_te(data)
        # calculate the loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update running training loss
        train_loss += loss.item()*data.size(0)
```

```
    model_te.eval() # prep model for evaluation
    for data, target in valid_loader:
        output = model_te(data)
        # calculate the loss
        loss = criterion(output, target)
        # update running validation loss
        valid_loss += loss.item()*data.size(0)

    # print training/validation statistics
    # calculate average loss over an epoch
    train_loss = train_loss/len(train_loader.dataset)
    valid_loss = valid_loss/len(valid_loader.dataset)

    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
        epoch+1,
        train_loss,
        valid_loss
        ))

    # save model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
        valid_loss_min,
        valid_loss))
        torch.save(model_te.state_dict(), 'model_te.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1        Training Loss: 1.017167         Validation Loss: 0.940452
Validation loss decreased (inf --> 0.940452).  Saving model ...
Epoch: 2        Training Loss: 0.912628         Validation Loss: 0.873352
Validation loss decreased (0.940452 --> 0.873352).  Saving model ...
Epoch: 3        Training Loss: 0.879003         Validation Loss: 0.851746
Validation loss decreased (0.873352 --> 0.851746).  Saving model ...
Epoch: 4        Training Loss: 0.866532         Validation Loss: 0.841702
Validation loss decreased (0.851746 --> 0.841702).  Saving model ...
Epoch: 5        Training Loss: 0.858859         Validation Loss: 0.836471
Validation loss decreased (0.841702 --> 0.836471).  Saving model ...
Epoch: 6        Training Loss: 0.853147         Validation Loss: 0.831773
Validation loss decreased (0.836471 --> 0.831773).  Saving model ...
Epoch: 7        Training Loss: 0.849154         Validation Loss: 0.827806
Validation loss decreased (0.831773 --> 0.827806).  Saving model ...
Epoch: 8        Training Loss: 0.844932         Validation Loss: 0.824553
Validation loss decreased (0.827806 --> 0.824553).  Saving model ...
Epoch: 9        Training Loss: 0.841235         Validation Loss: 0.820855
Validation loss decreased (0.824553 --> 0.820855).  Saving model ...
Epoch: 10       Training Loss: 0.838402         Validation Loss: 0.818889
Validation loss decreased (0.820855 --> 0.818889).  Saving model ...
```

In [54]:

```
# Test the trained network
# Use the same predict function as the binary case
def predict(model, dataloader):
    prediction_list = []
    truth_list = []
    for i, batch in enumerate(dataloader):
        # batch[0] is X, and batch[1] is y
        outputs = model(batch[0])
        _, predicted = torch.max(outputs.data, 1)
        prediction_list.append(predicted)
        truth_list.append(batch[1])
    return prediction_list, truth_list
```

In [56]:

```
# Load model parameters from the trained model with the lowest validation loss
model_te.load_state_dict(torch.load('model_te.pt'))
predictions, truths = predict(model_te, test_loader)
```

```
# Convert predictions and truths from list of tensors to list of lists
predictions = [list(torch.Tensor.numpy(t)) for t in predictions]
truths = [list(torch.Tensor.numpy(t)) for t in truths]
# Convert predictions and truths from list of lists to a single list
predictions = functools.reduce(lambda a, b: a + b, predictions)
truths = functools.reduce(lambda a, b: a + b, truths)
# Convert predictions from (0, 1, 2) to (1, 2, 3)
predictions = [p + 1 for p in predictions]
```

In [57]:
```
# report accuracy, precision, recall, and f1-score on the testing dataset

test_stats = precision_recall_fscore_support(truths, predictions, average='micro')
precision_test, recall_test, fscore_test = test_stats[0], test_stats[1], test_stats[2]

print('The accuracy of testing dataset: {:2.1%}'.format(accuracy_score(truths, predictio
```

```
The accuracy of testing dataset: 64.5%
```

4.b Use the concatenation of the first 10 Word2Vec vectors for each review as the input feature

1 Binary Classification custom model

In [59]:
```
%%time
X_train_bi2 = []
for x in X_train_bi:
    wordveclist = []
    first_10_words = x.split(' ')[:10]
    for word in first_10_words:
        try:
            wordvec = list(model2[word])
            wordveclist.append(wordvec)
        except:
            pass

    X_train_bi2.append(wordveclist)

# do the same to the testing dataset
X_test_bi2 = []
for x in X_test_bi:
    wordveclist = []
    first_10_words = x.split(' ')[:10]
    for word in first_10_words:
        try:
            wordvec = list(model2[word])
            wordveclist.append(wordvec)
        except:
            pass

    X_test_bi2.append(wordveclist)
```

```
CPU times: user 1min 3s, sys: 2min 13s, total: 3min 16s
Wall time: 6min 29s
```

In [60]:
```
# define a function that adds padding to the reviews with length less than 10 words
def add_padding(x):
    padding = [0 for _ in range(300)]
    if len(x) < 10:
        for i in range(10 - len(x)):
            x.append(padding)
    return x
```

```
# add paddings
X_train_bi2 = [add_padding(x) for x in X_train_bi2]
X_test_bi2 = [add_padding(x) for x in X_test_bi2]
```

In [61]:
```
# reshape the word vector of a review from (10, 300) to (3000,)
X_train_bi2 = [functools.reduce(lambda a, b: a + b, x) for x in X_train_bi2]
X_test_bi2 = [functools.reduce(lambda a, b: a + b, x) for x in X_test_bi2]
```

In [62]:
```
len_train = int(0.8 * len(X_train_bi2))
X_train, y_train = X_train_bi2[:len_train], []
for i in range(len_train):
    y_train.append(y_train_bi[i] - 1) # Convert label from 1 and 2 to 0 and 1

len_valid = len(X_train_bi2) - len_train
X_valid, y_valid = X_train_bi2[len_train:], []
for i in range(len_valid):
    y_valid.append(y_train_bi[len_train + i] - 1)

X_test, y_test = X_test_bi2, y_test_bi
```

In [63]:
```
# Override the Dataset class
class Dataset(Dataset):

    def __init__(self, features, labels):
        'Initialization'
        self.features = features
        self.labels = labels

    def __len__(self):
        'Denotes the total number of samples'
        return len(self.features)

    def __getitem__(self, index):
        'Generates one sample of data'
        # Select sample
        X = torch.tensor(self.features[index])
        y = self.labels[index]

        return X, y
```

In [64]:
```
# Generate training, validation and testing datasets
train_set = Dataset(X_train, y_train)
valid_set = Dataset(X_valid, y_valid)
test_set = Dataset(X_test, y_test)

# Generate dataloaders for the training, validation and testing datasets
train_loader = torch.utils.data.DataLoader(train_set, **params)
valid_loader = torch.utils.data.DataLoader(valid_set, **params)
test_loader = torch.utils.data.DataLoader(test_set, **params)
```

In [65]:
```
# Define the network architecture
# The same as before
class Net(nn.Module):
    def __init__(self, input_dim=300, output_dim=2, hidden_1=50, hidden_2=10):
        super(Net, self).__init__()
        # dimension of inputs
        self.input_dim = input_dim
        # number of classes (2 for binary, 3 for ternary, ...)
        self.output_dim = output_dim
        # number of nodes in each hidden layer
        self.hidden_1 = hidden_1
        self.hidden_2 = hidden_2
        # linear layer (input --> hidden_1)
```

```
        self.fc1 = nn.Linear(self.input_dim, self.hidden_1)
        # linear layer (hidden_1 --> hidden_2)
        self.fc2 = nn.Linear(self.hidden_1, self.hidden_2)
        # linear layer (hidden_2 --> 2)
        self.fc3 = nn.Linear(self.hidden_2, self.output_dim)
        # dropout prevents overfitting of data
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        # add the 1st hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer after the 1st hidden layer
        x = self.dropout(x)
        # add the 2nd hidden layer, with relu activation function
        x = F.relu(self.fc2(x))
        # add dropout layer after the 2nd hidden layer
        x = self.dropout(x)
        # add output layer
        x = self.fc3(x)
        return x
```

In [66]:

```
# initialize the NN
model_bi2 = Net(input_dim=3000)
print(model_bi2)
```

```
Net(
  (fc1): Linear(in_features=3000, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
  (fc3): Linear(in_features=10, out_features=2, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

In [67]:

```
# specify cross entropy loss as loss function
criterion = nn.CrossEntropyLoss()

# specify optimizer (stochastic gradient descent) and learning rate = 0.01
optimizer = torch.optim.SGD(model_bi2.parameters(), lr=0.01)
```

In [68]:

```
# train the network

# initialize tracker for minimum validation loss
valid_loss_min = np.Inf # set initial "min" to infinity

for epoch in range(max_epochs):
    # monitor training loss
    train_loss = 0.0
    valid_loss = 0.0

    model_bi2.train() # prep model for training
    for data, target in train_loader:
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model_bi2(data)
        # calculate the loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update running training loss
        train_loss += loss.item()*data.size(0)

    model_bi2.eval() # prep model for evaluation
    for data, target in valid_loader:
        # transfer to GPU
```

```
            #data, target = data.to(device), target.to(device)
            # forward pass: compute predicted outputs by passing inputs to the model
            output = model_bi2(data)
            # calculate the loss
            loss = criterion(output, target)
            # update running validation loss
            valid_loss += loss.item()*data.size(0)

        # print training/validation statistics
        # calculate average loss over an epoch
        train_loss = train_loss/len(train_loader.dataset)
        valid_loss = valid_loss/len(valid_loader.dataset)

        print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
            epoch+1,
            train_loss,
            valid_loss
            ))

        # save model if validation loss has decreased
        if valid_loss <= valid_loss_min:
            print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
            valid_loss_min,
            valid_loss))
            torch.save(model_bi2.state_dict(), 'model_bi2.pt')
            valid_loss_min = valid_loss
```

```
Epoch: 1        Training Loss: 0.529863         Validation Loss: 0.467470
Validation loss decreased (inf --> 0.467470).  Saving model ...
Epoch: 2        Training Loss: 0.460842         Validation Loss: 0.457045
Validation loss decreased (0.467470 --> 0.457045).  Saving model ...
Epoch: 3        Training Loss: 0.442386         Validation Loss: 0.451827
Validation loss decreased (0.457045 --> 0.451827).  Saving model ...
Epoch: 4        Training Loss: 0.427531         Validation Loss: 0.451010
Validation loss decreased (0.451827 --> 0.451010).  Saving model ...
Epoch: 5        Training Loss: 0.415802         Validation Loss: 0.448439
Validation loss decreased (0.451010 --> 0.448439).  Saving model ...
Epoch: 6        Training Loss: 0.403751         Validation Loss: 0.450240
Epoch: 7        Training Loss: 0.393425         Validation Loss: 0.450338
Epoch: 8        Training Loss: 0.381452         Validation Loss: 0.454666
Epoch: 9        Training Loss: 0.371288         Validation Loss: 0.458266
Epoch: 10       Training Loss: 0.362224         Validation Loss: 0.464123
```

In [69]:

```
# Test the trained network
# The same as before
def predict(model, dataloader):
    prediction_list = []
    truth_list = []
    for i, batch in enumerate(dataloader):
        # batch[0] is X, and batch[1] is y
        outputs = model(batch[0])
        _, predicted = torch.max(outputs.data, 1)
        prediction_list.append(predicted)
        truth_list.append(batch[1])
    return prediction_list, truth_list
```

In [70]:

```
# Load model parameters from the trained model with the lowest validation loss
model_bi2.load_state_dict(torch.load('model_bi2.pt'))

predictions, truths = predict(model_bi2, test_loader)
# Convert predictions and truths from list of tensors to list of lists
predictions = [list(torch.Tensor.numpy(t)) for t in predictions]
truths = [list(torch.Tensor.numpy(t)) for t in truths]
# Convert predictions and truths from list of lists to a single list
predictions = functools.reduce(lambda a, b: a + b, predictions)
```

```
truths = functools.reduce(lambda a, b: a + b, truths)
# Convert predictions from (0, 1) to (1, 2)
predictions = [p + 1 for p in predictions]
```

In [71]:
```
# report accuracy, precision, recall, and f1-score on the testing dataset

test_stats = precision_recall_fscore_support(truths, predictions, average='binary')
precision_test, recall_test, fscore_test = test_stats[0], test_stats[1], test_stats[2]

print('The accuracy of testing dataset: {:2.1%}'.format(accuracy_score(truths, predictio
```

```
The accuracy of testing dataset: 79.5%
```

4 Answer: The comparison of accuracy rates across various models shows that the SVM model, when trained using TF-IDF features, leads the pack with an impressive accuracy rate of 88.7%, showcasing its outstanding performance for this specific task. Following this, the Perceptron model that employs Word2Vec features records a respectable accuracy of 84.4%, indicating a solid performance. In the Feed Forward Neural Network (FNN) models, the variant that utilizes average Word2Vec vectors achieves a comparable accuracy of 80%, matching the SVM's performance but not exceeding that of the Perceptron model equipped with TF-IDF features. The FNN model is designed to concatenate the initial 10 Word2Vec vectors for each review as its input features fall slightly behind, registering an accuracy of 77.4%. These findings hint that, within the context of this particular classification challenge, the more straightforward Perceptron-TFIDF and SVM-Word2Vec models are more successful than the FNN models, which might see improvement with additional tuning.Compared to ternary, binary classification shows slightly better accuracy.

In [ ]:
```
```

Test Cases covered in this .ipynb file

1. (4a)FFNN AVG Pretrain Binary
2. (4a)FFNN AVG Pretrain Ternary

Remaining will be covered in different .ipynb due to memory constraints

4a pretrain binary

In [1]:
```
import time
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
import re
import nltk
from nltk.corpus import stopwords
nltk.download('omw-1.4')
nltk.download('stopwords')
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import Perceptron
from sklearn.metrics import precision_recall_fscore_support
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from gensim.models import Word2Vec
from gensim.models import KeyedVectors
from sklearn.metrics.pairwise import cosine_similarity
import gensim.downloader as api
```

```
[nltk_data] Downloading package omw-1.4 to
[nltk_data]     /Users/payalrashinkar/nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/payalrashinkar/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

In [2]:
```
# load the pretrained model as model1
pretrained = 'word2vec-google-news-300.gz'
model1 = KeyedVectors.load_word2vec_format(pretrained, binary=True)
```

In [3]:
```
df_1 = pd.read_csv('https://web.archive.org/web/20201127142707if_/https://s3.amazonaws.c
```

In [4]:
```
df = df_1.copy()
df = df[['star_rating', 'review_body']].rename(columns={'star_rating':'ratings', 'review
df.head()
```

Out[4]:

	ratings	reviews
0	5	Great product.
1	5	What's to say about this commodity item except...
2	5	Haven't used yet, but I am sure I will like it.
3	1	Although this was labeled as "new" the...
4	4	Gorgeous colors and easy to use

In [5]:
```
balanced_df = pd.DataFrame()
for ratings in ['1','2', '3', '4', '5']:
    sampled_df = df[df['ratings'] == ratings].sample(n=50000, random_state=42)
    balanced_df = pd.concat([balanced_df, sampled_df])
    print(f"Rating {ratings}: {len(sampled_df)} datasets")
```

```
Rating 1: 50000 datasets
Rating 2: 50000 datasets
Rating 3: 50000 datasets
Rating 4: 50000 datasets
Rating 5: 50000 datasets
```

In [6]:
```
# Create ternary labels
def label_sentiment(row):
    if row['ratings'] > '3':
        return 1  # Positive
    elif row['ratings'] < '3':
        return 2  # Negative
    else:
        return 3  # Neutral

balanced_df['class'] = balanced_df.apply(label_sentiment, axis=1)
```

In [7]:
```
X, y = balanced_df['reviews'].fillna('').tolist(), balanced_df['class'].tolist()
```

In [8]:
```
# convert reviews to lower case
X = [str(x).lower() for x in X]
# remove HTML and URLs from reviews
X = [re.sub('<.*>', '', x) for x in X]
X = [re.sub(r'https?://\S+', '', x) for x in X]
# remove non-alphabetical characters
X = [re.sub('[^a-z ]', '', x) for x in X]
# remove extra spaces
X = [re.sub(' +', ' ', x) for x in X]
```

In [9]:
```
# expand contractions
contractions = {
"ain't": "am not",
"aren't": "are not",
"can't": "cannot",
"can't've": "cannot have",
"'cause": "because",
"could've": "could have",
"couldn't": "could not",
"couldn't've": "could not have",
"didn't": "did not",
"doesn't": "does not",
"don't": "do not",
"hadn't": "had not",
"hadn't've": "had not have",
"hasn't": "has not",
"haven't": "have not",
"he'd": "he would",
"he'd've": "he would have",
"he'll": "he will",
"he'll've": "he will have",
"he's": "he is",
"how'd": "how did",
"how'd'y": "how do you",
"how'll": "how will",
"how's": "how is",
"I'd": "I would",
"I'd've": "I would have",
"I'll": "I will",
"I'll've": "I will have",
```

```
"I'm": "I am",
"I've": "I have",
"isn't": "is not",
"it'd": "it would",
"it'd've": "it would have",
"it'll": "it will",
"it'll've": "it will have",
"it's": "it is",
"let's": "let us",
"ma'am": "madam",
"mayn't": "may not",
"might've": "might have",
"mightn't": "might not",
"mightn't've": "might not have",
"must've": "must have",
"mustn't": "must not",
"mustn't've": "must not have",
"needn't": "need not",
"needn't've": "need not have",
"o'clock": "of the clock",
"oughtn't": "ought not",
"oughtn't've": "ought not have",
"shan't": "shall not",
"sha'n't": "shall not",
"shan't've": "shall not have",
"she'd": "she would",
"she'd've": "she would have",
"she'll": "she will",
"she'll've": "she will have",
"she's": "she is",
"should've": "should have",
"shouldn't": "should not",
"shouldn't've": "should not have",
"so've": "so have",
"so's": "so is",
"that'd": "that would",
"that'd've": "that would have",
"that's": "that is",
"there'd": "there would",
"there'd've": "there would have",
"there's": "there is",
"they'd": "they would",
"they'd've": "they would have",
"they'll": "they will",
"they'll've": "they will have",
"they're": "they are",
"they've": "they have",
"to've": "to have",
"wasn't": "was not",
"we'd": "we would",
"we'd've": "we would have",
"we'll": "we will",
"we'll've": "we will have",
"we're": "we are",
"we've": "we have",
"weren't": "were not",
"what'll": "what will",
"what'll've": "what will have",
"what're": "what are",
"what's": "what is",
"what've": "what have",
"when's": "when is",
"when've": "when have",
"where'd": "where did",
"where's": "where is",
"where've": "where have",
```

```
"who'll": "who will",
"who'll've": "who will have",
"who's": "who is",
"who've": "who have",
"why's": "why is",
"why've": "why have",
"will've": "will have",
"won't": "will not",
"won't've": "will not have",
"would've": "would have",
"wouldn't": "would not",
"wouldn't've": "would not have",
"y'all": "you all",
"y'all'd": "you all would",
"y'all'd've": "you all would have",
"y'all're": "you all are",
"y'all've": "you all have",
"you'd": "you would",
"you'd've": "you would have",
"you'll": "you will",
"you'll've": "you will have",
"you're": "you are",
"you've": "you have"
}
def decontraction(s):
    for word in s.split(' '):
        if word in contractions.keys():
            s = re.sub(word, contractions[word], s)
    return s
X = [decontraction(x) for x in X]
```

In [10]:
```
# remove stop words
stopWords =set(stopwords.words('english'))
def remvstopWords(s):
    wordlist = s.split(' ')
    newlist = []
    for word in wordlist:
        if word not in stopWords:
            newlist.append(word)
    s = ' '.join(newlist)
    return s

X = list(map(remvstopWords, X))
```

In [11]:
```
# perform lemmatization
wnl = WordNetLemmatizer()
X = [' '.join([wnl.lemmatize(word) for word in x.split(' ')]) for x in X]
```

In [12]:
```
# Split the downsized dataset into 80% training dataset and 20% testing dataset.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)
```

In [13]:
```
y_train_bi = []
idx_train = []
for i, y in enumerate(y_train):
    if y == 1 or y == 2:
        y_train_bi.append(y)
        idx_train.append(i)

y_test_bi = []
idx_test = []
for i, y in enumerate(y_test):
    if y == 1 or y == 2:
        y_test_bi.append(y)
        idx_test.append(i)
```

```
# use the list of indices to select a sub-list of X_train
X_train_bi = [X_train[i] for i in idx_train]
X_test_bi = [X_test[i] for i in idx_test]
```

In [14]:
```
X_train_bi1 = []
for x in X_train_bi:
    wordveclist = []
    for word in x.split(' '):
        try:
            wordvec = model1[word]
            wordveclist.append(wordvec)
        except:
            pass
    X_train_bi1.append(np.mean(wordveclist, axis=0))
```

```
/Users/payalrashinkar/anaconda3/lib/python3.11/site-packages/numpy/core/fromnumeric.py:3
464: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/Users/payalrashinkar/anaconda3/lib/python3.11/site-packages/numpy/core/_methods.py:192:
RuntimeWarning: invalid value encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
```

In [15]:
```
X_test_bi1 = []
for x in X_test_bi:
    wordveclist = []
    for word in x.split(' '):
        try:
            wordvec = model1[word]
            wordveclist.append(wordvec)
        except:
            pass
    X_test_bi1.append(np.mean(wordveclist, axis=0))
```

In [16]:
```
wv_train = []
for i, x in enumerate(X_train_bi1):
    try:
        len(x)
        wv_train.append(i)
    except:
        pass


wv_test = []
for i, x in enumerate(X_test_bi1):
    try:
        len(x)
        wv_test.append(i)
    except:
        pass

# remove the non-word-vector values from the dataset
X_train_bi1 = [X_train_bi1[i] for i in wv_train]
X_test_bi1 = [X_test_bi1[i] for i in wv_test]
y_train_bi1 = [y_train_bi[i] for i in wv_train]
y_test_bi1 = [y_test_bi[i] for i in wv_test]
```

In [17]:
```
import torch
from torch.utils.data import DataLoader, Dataset
import torch.nn as nn
import torch.nn.functional as F
import functools
from sklearn.metrics import accuracy_score
```

In [18]:
```
# Enable CUDA for PyTorch
```

```
use_cuda = torch.cuda.is_available()
device = torch.device("cuda:0" if use_cuda else "cpu")
torch.backends.cudnn.benchmark = True
```

In [19]:
```
# Hyperparameters
params = {'batch_size': 64,
          'shuffle': True,
          'num_workers': 0}
max_epochs = 5
```

In [20]:
```
# Override the Dataset class
class Dataset(Dataset):

    def __init__(self, list_IDs, labels):
        'Initialization'
        self.list_IDs = list_IDs
        self.labels = labels

    def __len__(self):
        'Denotes the total number of samples'
        return len(self.list_IDs)

    def __getitem__(self, index):
        'Generates one sample of data'
        # Select sample
        ID = self.list_IDs[index]

        # Load data and get label
        X = torch.load('data/' + ID + '.pt')
        y = self.labels[index]
        return X, y
```

In [21]:
```
train_IDs = {}
y_train = []
len_train_IDs = int(0.8 * len(X_train_bi1))
#print(len_train_IDs)
for i in range(len_train_IDs):
    train_IDs[i] = 'train_bi_' + str(i)
    y_train.append(y_train_bi1[i] - 1) # Convert label from 1 and 2 to 0 and 1

#print(y_train)
valid_IDs = {}
y_valid = []
len_valid_IDs = len(X_train_bi1) - len_train_IDs
for i in range(len_valid_IDs):
    valid_IDs[i] = 'valid_bi_' + str(i)
    y_valid.append(y_train_bi1[len(y_train) + i] - 1)

test_IDs = {}
len_test_IDs = len(X_test_bi1)
for i in range(len_test_IDs):
    test_IDs[i] = 'test_bi_' + str(i)

for i in range(len(train_IDs)):
    torch.save(X_train_bi1[i], 'data/' + train_IDs[i] + '.pt')

for i in range(len(valid_IDs)):
    torch.save(X_train_bi1[len(train_IDs) + i], 'data/' + valid_IDs[i] + '.pt')

for i in range(len(test_IDs)):
    torch.save(X_test_bi1[i], 'data/' + test_IDs[i] + '.pt')
```

In [22]:
```
train_set = Dataset(train_IDs, y_train)
valid_set = Dataset(valid_IDs, y_valid)
```

```
test_set = Dataset(test_IDs, y_test_bi1)

# Generate dataloaders for the training, validation and testing datasets
train_loader = torch.utils.data.DataLoader(train_set, **params)
valid_loader = torch.utils.data.DataLoader(valid_set, **params)
test_loader = torch.utils.data.DataLoader(test_set, **params)
```

In [23]:
```
# Define the network architecture
class Net(nn.Module):
    def __init__(self, input_dim=300, output_dim=2, hidden_1=50, hidden_2=10):
        super(Net, self).__init__()
        # dimension of inputs
        self.input_dim = input_dim
        # number of classes (2 for binary, 3 for ternary, ...)
        self.output_dim = output_dim
        # number of nodes in each hidden layer
        self.hidden_1 = hidden_1
        self.hidden_2 = hidden_2
        # linear layer (input --> hidden_1)
        self.fc1 = nn.Linear(self.input_dim, self.hidden_1)
        # linear layer (hidden_1 --> hidden_2)
        self.fc2 = nn.Linear(self.hidden_1, self.hidden_2)
        # linear layer (hidden_2 --> 2)
        self.fc3 = nn.Linear(self.hidden_2, self.output_dim)
        # dropout prevents overfitting of data
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        # add the 1st hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer after the 1st hidden layer
        x = self.dropout(x)
        # add the 2nd hidden layer, with relu activation function
        x = F.relu(self.fc2(x))
        # add dropout layer after the 2nd hidden layer
        x = self.dropout(x)
        # add output layer
        x = self.fc3(x)
        return x
```

In [24]:
```
# initialize the NN
model = Net()
print(model)
```

```
Net(
  (fc1): Linear(in_features=300, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
  (fc3): Linear(in_features=10, out_features=2, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

In [25]:
```
# specify cross entropy loss as loss function
criterion = nn.CrossEntropyLoss()

# specify optimizer (stochastic gradient descent) and learning rate = 0.01
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

In [26]:
```
# train the network

# initialize tracker for minimum validation loss
valid_loss_min = np.Inf # set initial "min" to infinity

for epoch in range(max_epochs):
    # monitor training loss
    train_loss = 0.0
```

```
    valid_loss = 0.0

    model.train() # prep model for training
    for data, target in train_loader:
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update running training loss
        train_loss += loss.item()*data.size(0)

    model.eval() # prep model for evaluation
    for data, target in valid_loader:
        output = model(data)
        # calculate the loss
        loss = criterion(output, target)
        # update running validation loss
        valid_loss += loss.item()*data.size(0)

    train_loss = train_loss/len(train_loader.dataset)
    valid_loss = valid_loss/len(valid_loader.dataset)

    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
        epoch+1,
        train_loss,
        valid_loss
        ))

    # save model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
        valid_loss_min,
        valid_loss))
        torch.save(model.state_dict(), 'model.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1        Training Loss: 0.692913         Validation Loss: 0.687906
Validation loss decreased (inf --> 0.687906).  Saving model ...
Epoch: 2        Training Loss: 0.670984         Validation Loss: 0.633562
Validation loss decreased (0.687906 --> 0.633562).  Saving model ...
Epoch: 3        Training Loss: 0.585634         Validation Loss: 0.525995
Validation loss decreased (0.633562 --> 0.525995).  Saving model ...
Epoch: 4        Training Loss: 0.521222         Validation Loss: 0.479639
Validation loss decreased (0.525995 --> 0.479639).  Saving model ...
Epoch: 5        Training Loss: 0.491166         Validation Loss: 0.458885
Validation loss decreased (0.479639 --> 0.458885).  Saving model ...
```

In [27]:

```
## Test the trained network
def predict(model, dataloader):
    prediction_list = []
    truth_list = []
    for i, batch in enumerate(dataloader):
        # batch[0] is X, and batch[1] is y
        outputs = model(batch[0])
        _, predicted = torch.max(outputs.data, 1)
        prediction_list.append(predicted)
        truth_list.append(batch[1])
    return prediction_list, truth_list
```

In [28]:

```
predictions, truths = predict(model, test_loader)
```

```
# Convert predictions and truths from list of tensors to list of lists
predictions = [list(torch.Tensor.numpy(t)) for t in predictions]
truths = [list(torch.Tensor.numpy(t)) for t in truths]
# Convert predictions and truths from list of lists to a single list
predictions = functools.reduce(lambda a, b: a + b, predictions)
truths = functools.reduce(lambda a, b: a + b, truths)
# Convert predictions from (0, 1) to (1, 2)
predictions = [p + 1 for p in predictions]
```

In [29]:
```
# report accuracy, precision, recall, and f1-score on the testing dataset

test_stats = precision_recall_fscore_support(truths, predictions, average='binary')
precision_test, recall_test, fscore_test = test_stats[0], test_stats[1], test_stats[2]

print('The accuracy of testing dataset: {:2.1%}'.format(accuracy_score(truths, predictio
```

```
The accuracy of testing dataset: 79.6%
```

4a. Pretrain Ternary

In [30]:
```
X, y = balanced_df['reviews'].fillna('').tolist(), balanced_df['class'].tolist()
```

In [31]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)
```

In [32]:
```
X_train_te1 = []
for x in X_train:
    wordveclist = []
    for word in x.split(' '):
        try:
            wordvec = model1[word]
            wordveclist.append(wordvec)
        except:
            pass
    X_train_te1.append(np.mean(wordveclist, axis=0))

# do the same to the testing dataset
X_test_te1 = []
for x in X_test:
    wordveclist = []
    for word in x.split(' '):
        try:
            wordvec = model1[word]
            wordveclist.append(wordvec)
        except:
            pass
    X_test_te1.append(np.mean(wordveclist, axis=0))
```

```
/Users/payalrashinkar/anaconda3/lib/python3.11/site-packages/numpy/core/fromnumeric.py:3
464: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/Users/payalrashinkar/anaconda3/lib/python3.11/site-packages/numpy/core/_methods.py:192:
RuntimeWarning: invalid value encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
```

In [33]:
```
wv_train = []
for i, x in enumerate(X_train_te1):
    try:
        len(x)
        wv_train.append(i)
    except:
        pass
```

```python
wv_test = []
for i, x in enumerate(X_test_te1):
    try:
        len(x)
        wv_test.append(i)
    except:
        pass

# remove the non-word-vector values from the dataset
X_train_te1 = [X_train_te1[i] for i in wv_train]
X_test_te1 = [X_test_te1[i] for i in wv_test]
y_train_te1 = [y_train[i] for i in wv_train]
y_test_te1 = [y_test[i] for i in wv_test]
```

In [34]:

```python
train_IDs = {}
y_train = []
len_train_IDs = int(0.8 * len(X_train_te1))

for i in range(len_train_IDs):
    train_IDs[i] = 'train_te_' + str(i)
    y_train.append(y_train_te1[i] - 1) # Convert label from 1, 2 and 3 to 0, 1 and 2

valid_IDs = {}
y_valid = []
len_valid_IDs = len(X_train_te1) - len_train_IDs
for i in range(len_valid_IDs):
    valid_IDs[i] = 'valid_te_' + str(i)
    y_valid.append(y_train_te1[len(y_train) + i] - 1)

test_IDs = {}
len_test_IDs = len(X_test_te1)
for i in range(len_test_IDs):
    test_IDs[i] = 'test_te_' + str(i)

for i in range(len(train_IDs)):
    torch.save(X_train_te1[i], 'data/' + train_IDs[i] + '.pt')

for i in range(len(valid_IDs)):
    torch.save(X_train_te1[len(train_IDs) + i], 'data/' + valid_IDs[i] + '.pt')

for i in range(len(test_IDs)):
    torch.save(X_test_te1[i], 'data/' + test_IDs[i] + '.pt')
```

In [35]:

```python
# Generate training, validation and testing datasets
train_set = Dataset(train_IDs, y_train)
valid_set = Dataset(valid_IDs, y_valid)
test_set = Dataset(test_IDs, y_test_te1)

# Generate dataloaders for the training, validation and testing datasets
train_loader = torch.utils.data.DataLoader(train_set, **params)
valid_loader = torch.utils.data.DataLoader(valid_set, **params)
test_loader = torch.utils.data.DataLoader(test_set, **params)
```

In [36]:

```python
# initialize the NN
# use model_te to be distinguishable from the binary model
model_te = Net(output_dim=3)
print(model_te)
```

```
Net(
  (fc1): Linear(in_features=300, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
  (fc3): Linear(in_features=10, out_features=3, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

In [37]:
```
# specify cross entropy loss as loss function
criterion = nn.CrossEntropyLoss()

# specify optimizer (stochastic gradient descent) and learning rate = 0.01
optimizer = torch.optim.SGD(model_te.parameters(), lr=0.01)
```

In [38]:
```
# train the network

# initialize tracker for minimum validation loss
valid_loss_min = np.Inf # set initial "min" to infinity

for epoch in range(max_epochs):
    # monitor training loss
    train_loss = 0.0
    valid_loss = 0.0

    model_te.train() # prep model for training
    for data, target in train_loader:
        # transfer to GPU
        #data, target = data.to(device), target.to(device)
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model_te(data)
        # calculate the loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update running training loss
        train_loss += loss.item()*data.size(0)

    model_te.eval() # prep model for evaluation
    for data, target in valid_loader:
        # transfer to GPU
        #data, target = data.to(device), target.to(device)
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model_te(data)
        # calculate the loss
        loss = criterion(output, target)
        # update running validation loss
        valid_loss += loss.item()*data.size(0)

    # print training/validation statistics
    # calculate average loss over an epoch
    train_loss = train_loss/len(train_loader.dataset)
    valid_loss = valid_loss/len(valid_loader.dataset)

    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
        epoch+1,
        train_loss,
        valid_loss
        ))

    # save model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
        valid_loss_min,
        valid_loss))
        torch.save(model_te.state_dict(), 'model_te.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1        Training Loss: 1.054745         Validation Loss: 1.054312
Validation loss decreased (inf --> 1.054312).  Saving model ...
```

```
Epoch: 2        Training Loss: 1.050665         Validation Loss: 1.048230
Validation loss decreased (1.054312 --> 1.048230).  Saving model ...
Epoch: 3        Training Loss: 1.035265         Validation Loss: 1.014925
Validation loss decreased (1.048230 --> 1.014925).  Saving model ...
Epoch: 4        Training Loss: 0.988091         Validation Loss: 0.949913
Validation loss decreased (1.014925 --> 0.949913).  Saving model ...
Epoch: 5        Training Loss: 0.938678         Validation Loss: 0.901500
Validation loss decreased (0.949913 --> 0.901500).  Saving model ...
```

In [39]:

```
# Test the trained network
# Use the same predict function as the binary case
def predict(model, dataloader):
    prediction_list = []
    truth_list = []
    for i, batch in enumerate(dataloader):
        # batch[0] is X, and batch[1] is y
        outputs = model(batch[0])
        _, predicted = torch.max(outputs.data, 1)
        prediction_list.append(predicted)
        truth_list.append(batch[1])
    return prediction_list, truth_list
```

In [40]:

```
# Load model parameters from the trained model with the lowest validation loss
model_te.load_state_dict(torch.load('model_te.pt'))
predictions, truths = predict(model_te, test_loader)
# Convert predictions and truths from list of tensors to list of lists
predictions = [list(torch.Tensor.numpy(t)) for t in predictions]
truths = [list(torch.Tensor.numpy(t)) for t in truths]
# Convert predictions and truths from list of lists to a single list
predictions = functools.reduce(lambda a, b: a + b, predictions)
truths = functools.reduce(lambda a, b: a + b, truths)
# Convert predictions from (0, 1, 2) to (1, 2, 3)
predictions = [p + 1 for p in predictions]
```

In [41]:

```
# report accuracy, precision, recall, and f1-score on the testing dataset

test_stats = precision_recall_fscore_support(truths, predictions, average='micro')
precision_test, recall_test, fscore_test = test_stats[0], test_stats[1], test_stats[2]

print('The accuracy of testing dataset: {:2.1%}'.format(accuracy_score(truths, predictio
```

```
The accuracy of testing dataset: 62.3%
```

Test Cases covered in this .ipynb file

1. (4b)FFNN CONCAT Pretrain Binary

Remaining will be covered in different .ipynb due to memory constraints

In [1]:
```
import time
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
import re
import nltk
from nltk.corpus import stopwords
nltk.download('omw-1.4')
nltk.download('stopwords')
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import Perceptron
from sklearn.metrics import precision_recall_fscore_support
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.svm import LinearSVC
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from gensim.models import Word2Vec
from gensim.models import KeyedVectors
from sklearn.metrics.pairwise import cosine_similarity
import gensim.downloader as api
```

```
[nltk_data] Downloading package omw-1.4 to
[nltk_data]     /Users/payalrashinkar/nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/payalrashinkar/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

In [2]:
```
# load the pretrained model as model1
pretrained = 'word2vec-google-news-300.gz'
model1 = KeyedVectors.load_word2vec_format(pretrained, binary=True)
```

In [3]:
```
df_1 = pd.read_csv('https://web.archive.org/web/20201127142707if_/https://s3.amazonaws.c
```

In [4]:
```
df = df_1.copy()
df = df[['star_rating', 'review_body']].rename(columns={'star_rating':'ratings', 'review
df.head()
```

Out[4]:

	ratings	reviews
0	5	Great product.
1	5	What's to say about this commodity item except...
2	5	Haven't used yet, but I am sure I will like it.
3	1	Although this was labeled as "new" the...
4	4	Gorgeous colors and easy to use

In [5]:
```
balanced_df = pd.DataFrame()
for ratings in ['1','2', '3', '4', '5']:
    sampled_df = df[df['ratings'] == ratings].sample(n=50000, random_state=42)
    balanced_df = pd.concat([balanced_df, sampled_df])
    print(f"Rating {ratings}: {len(sampled_df)} datasets")
```

```
Rating 1: 50000 datasets
Rating 2: 50000 datasets
Rating 3: 50000 datasets
Rating 4: 50000 datasets
Rating 5: 50000 datasets
```

In [6]:
```
# Create ternary labels
def label_sentiment(row):
    if row['ratings'] > '3':
        return 1  # Positive
    elif row['ratings'] < '3':
        return 2  # Negative
    else:
        return 3  # Neutral


balanced_df['class'] = balanced_df.apply(label_sentiment, axis=1)
```

In [7]:
```
X, y = balanced_df['reviews'].fillna('').tolist(), balanced_df['class'].tolist()
#X, y = df['reviews'].fillna('').tolist(), df['label'].tolist()
```

In [8]:
```
# convert reviews to lower case
X = [str(x).lower() for x in X]
# remove HTML and URLs from reviews
X = [re.sub('<.*>', '', x) for x in X]
X = [re.sub(r'https?://\S+', '', x) for x in X]
# remove non-alphabetical characters
X = [re.sub('[^a-z ]', '', x) for x in X]
# remove extra spaces
X = [re.sub(' +', ' ', x) for x in X]
```

In [9]:
```
# expand contractions
contractions = {
"ain't": "am not",
"aren't": "are not",
"can't": "cannot",
"can't've": "cannot have",
"'cause": "because",
"could've": "could have",
"couldn't": "could not",
"couldn't've": "could not have",
"didn't": "did not",
"doesn't": "does not",
"don't": "do not",
"hadn't": "had not",
"hadn't've": "had not have",
"hasn't": "has not",
"haven't": "have not",
"he'd": "he would",
"he'd've": "he would have",
"he'll": "he will",
"he'll've": "he will have",
"he's": "he is",
"how'd": "how did",
"how'd'y": "how do you",
"how'll": "how will",
"how's": "how is",
"I'd": "I would",
"I'd've": "I would have",
"I'll": "I will",
"I'll've": "I will have",
"I'm": "I am",
"I've": "I have",
"isn't": "is not",
```

```
"it'd": "it would",
"it'd've": "it would have",
"it'll": "it will",
"it'll've": "it will have",
"it's": "it is",
"let's": "let us",
"ma'am": "madam",
"mayn't": "may not",
"might've": "might have",
"mightn't": "might not",
"mightn't've": "might not have",
"must've": "must have",
"mustn't": "must not",
"mustn't've": "must not have",
"needn't": "need not",
"needn't've": "need not have",
"o'clock": "of the clock",
"oughtn't": "ought not",
"oughtn't've": "ought not have",
"shan't": "shall not",
"sha'n't": "shall not",
"shan't've": "shall not have",
"she'd": "she would",
"she'd've": "she would have",
"she'll": "she will",
"she'll've": "she will have",
"she's": "she is",
"should've": "should have",
"shouldn't": "should not",
"shouldn't've": "should not have",
"so've": "so have",
"so's": "so is",
"that'd": "that would",
"that'd've": "that would have",
"that's": "that is",
"there'd": "there would",
"there'd've": "there would have",
"there's": "there is",
"they'd": "they would",
"they'd've": "they would have",
"they'll": "they will",
"they'll've": "they will have",
"they're": "they are",
"they've": "they have",
"to've": "to have",
"wasn't": "was not",
"we'd": "we would",
"we'd've": "we would have",
"we'll": "we will",
"we'll've": "we will have",
"we're": "we are",
"we've": "we have",
"weren't": "were not",
"what'll": "what will",
"what'll've": "what will have",
"what're": "what are",
"what's": "what is",
"what've": "what have",
"when's": "when is",
"when've": "when have",
"where'd": "where did",
"where's": "where is",
"where've": "where have",
"who'll": "who will",
"who'll've": "who will have",
"who's": "who is",
```

```
"who've": "who have",
"why's": "why is",
"why've": "why have",
"will've": "will have",
"won't": "will not",
"won't've": "will not have",
"would've": "would have",
"wouldn't": "would not",
"wouldn't've": "would not have",
"y'all": "you all",
"y'all'd": "you all would",
"y'all'd've": "you all would have",
"y'all're": "you all are",
"y'all've": "you all have",
"you'd": "you would",
"you'd've": "you would have",
"you'll": "you will",
"you'll've": "you will have",
"you're": "you are",
"you've": "you have"
}
def decontraction(s):
    for word in s.split(' '):
        if word in contractions.keys():
            s = re.sub(word, contractions[word], s)
    return s
X = [decontraction(x) for x in X]
```

In [10]:
```
# remove stop words
stopWords =set(stopwords.words('english'))
def remvstopWords(s):
    wordlist = s.split(' ')
    newlist = []
    for word in wordlist:
        if word not in stopWords:
            newlist.append(word)
    s = ' '.join(newlist)
    return s

X = list(map(remvstopWords, X))
```

In [11]:
```
# perform lemmatization
wnl = WordNetLemmatizer()
X = [' '.join([wnl.lemmatize(word) for word in x.split(' ')]) for x in X]
```

In [12]:
```
# Split the downsized dataset into 80% training dataset and 20% testing dataset.
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)
```

In [13]:
```
# turn the original ternary training and testing datasets to binary
# return a list of indices of y = 1 or y =2
y_train_bi = []
idx_train = []
for i, y in enumerate(y_train):
    if y == 1 or y == 2:
        y_train_bi.append(y)
        idx_train.append(i)

y_test_bi = []
idx_test = []
for i, y in enumerate(y_test):
    if y == 1 or y == 2:
        y_test_bi.append(y)
        idx_test.append(i)
# use the list of indices to select a sub-list of X_train
```

```
X_train_bi = [X_train[i] for i in idx_train]
X_test_bi = [X_test[i] for i in idx_test]
```

In [14]:
```
# take the average word vectors of important words (i.e. non-stop words) in
# a review as the feature of a training sample
X_train_bi1 = []
for x in X_train_bi:
    wordveclist = []
    for word in x.split(' '):
        try:
            wordvec = model1[word]
            wordveclist.append(wordvec)
        except:
            pass
    X_train_bi1.append(np.mean(wordveclist, axis=0))
```

```
/Users/payalrashinkar/anaconda3/lib/python3.11/site-packages/numpy/core/fromnumeric.py:3
464: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/Users/payalrashinkar/anaconda3/lib/python3.11/site-packages/numpy/core/_methods.py:192:
RuntimeWarning: invalid value encountered in scalar divide
  ret = ret.dtype.type(ret / rcount)
```

In [15]:
```
X_test_bi1 = []
for x in X_test_bi:
    wordveclist = []
    for word in x.split(' '):
        try:
            wordvec = model1[word]
            wordveclist.append(wordvec)
        except:
            pass
    X_test_bi1.append(np.mean(wordveclist, axis=0))
```

In [16]:
```
# handle non-word-vector values in the dataset
# return the indices of word-vector values
wv_train = []
for i, x in enumerate(X_train_bi1):
    try:
        len(x)
        wv_train.append(i)
    except:
        pass


wv_test = []
for i, x in enumerate(X_test_bi1):
    try:
        len(x)
        wv_test.append(i)
    except:
        pass

# remove the non-word-vector values from the dataset
X_train_bi1 = [X_train_bi1[i] for i in wv_train]
X_test_bi1 = [X_test_bi1[i] for i in wv_test]
y_train_bi1 = [y_train_bi[i] for i in wv_train]
y_test_bi1 = [y_test_bi[i] for i in wv_test]
```

In [17]:
```
import torch
from torch.utils.data import DataLoader, Dataset
import torch.nn as nn
import torch.nn.functional as F
```

```python
import functools
from sklearn.metrics import accuracy_score
```

In [18]:
```python
# Enable CUDA for PyTorch
use_cuda = torch.cuda.is_available()
device = torch.device("cuda:0" if use_cuda else "cpu")
torch.backends.cudnn.benchmark = True
```

In [19]:
```python
# Hyperparameters
params = {'batch_size': 64,
          'shuffle': True,
          'num_workers': 0}
max_epochs = 5
```

In [20]:
```python
X_train_bi1 = []
for x in X_train_bi:
    wordveclist = []
    first_10_words = x.split(' ')[:10]
    for word in first_10_words:
        try:
            wordvec = list(model1[word])
            wordveclist.append(wordvec)
        except:
            pass

    X_train_bi1.append(wordveclist)

# do the same to the testing dataset
X_test_bi1 = []
for x in X_test_bi:
    wordveclist = []
    first_10_words = x.split(' ')[:10]
    for word in first_10_words:
        try:
            wordvec = list(model1[word])
            wordveclist.append(wordvec)
        except:
            pass

    X_test_bi1.append(wordveclist)
```

In [21]:
```python
# define a function that adds padding to the reviews with length less than 10 words
def add_padding(x):
    padding = [0 for _ in range(300)]
    if len(x) < 10:
        for i in range(10 - len(x)):
            x.append(padding)
    return x

# add paddings
X_train_bi1 = [add_padding(x) for x in X_train_bi1]
X_test_bi1 = [add_padding(x) for x in X_test_bi1]
```

In [22]:
```python
# reshape the word vector of a review from (10, 300) to (3000,)
X_train_bi1 = [functools.reduce(lambda a, b: a + b, x) for x in X_train_bi1]
X_test_bi1 = [functools.reduce(lambda a, b: a + b, x) for x in X_test_bi1]
```

In [23]:
```python
len_train = int(0.8 * len(X_train_bi1))
X_train, y_train = X_train_bi1[:len_train], []
for i in range(len_train):
    y_train.append(y_train_bi[i] - 1) # Convert label from 1 and 2 to 0 and 1

len_valid = len(X_train_bi1) - len_train
```

```
X_valid, y_valid = X_train_bi1[len_train:], []
for i in range(len_valid):
    y_valid.append(y_train_bi[len_train + i] - 1)

X_test, y_test = X_test_bi1, y_test_bi
```

In [24]:
```
# Override the Dataset class
class Dataset(Dataset):

    def __init__(self, features, labels):
        'Initialization'
        self.features = features
        self.labels = labels

    def __len__(self):
        'Denotes the total number of samples'
        return len(self.features)

    def __getitem__(self, index):
        'Generates one sample of data'
        # Select sample
        X = torch.tensor(self.features[index])
        y = self.labels[index]

        return X, y
```

In [25]:
```
# Generate training, validation and testing datasets
train_set = Dataset(X_train, y_train)
valid_set = Dataset(X_valid, y_valid)
test_set = Dataset(X_test, y_test)

# Generate dataloaders for the training, validation and testing datasets
train_loader = torch.utils.data.DataLoader(train_set, **params)
valid_loader = torch.utils.data.DataLoader(valid_set, **params)
test_loader = torch.utils.data.DataLoader(test_set, **params)
```

In [26]:
```
# Define the network architecture
# The same as before
class Net(nn.Module):
    def __init__(self, input_dim=300, output_dim=2, hidden_1=50, hidden_2=10):
        super(Net, self).__init__()
        # dimension of inputs
        self.input_dim = input_dim
        # number of classes (2 for binary, 3 for ternary, ...)
        self.output_dim = output_dim
        # number of nodes in each hidden layer
        self.hidden_1 = hidden_1
        self.hidden_2 = hidden_2
        # linear layer (input --> hidden_1)
        self.fc1 = nn.Linear(self.input_dim, self.hidden_1)
        # linear layer (hidden_1 --> hidden_2)
        self.fc2 = nn.Linear(self.hidden_1, self.hidden_2)
        # linear layer (hidden_2 --> 2)
        self.fc3 = nn.Linear(self.hidden_2, self.output_dim)
        # dropout prevents overfitting of data
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        # add the 1st hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer after the 1st hidden layer
        x = self.dropout(x)
        # add the 2nd hidden layer, with relu activation function
        x = F.relu(self.fc2(x))
```

```python
        # add dropout layer after the 2nd hidden layer
        x = self.dropout(x)
        # add output layer
        x = self.fc3(x)
        return x
```

In [27]:

```python
# initialize the NN
model_bi1 = Net(input_dim=3000)
print(model_bi1)
```

```
Net(
  (fc1): Linear(in_features=3000, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
  (fc3): Linear(in_features=10, out_features=2, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

In [28]:

```python
# specify cross entropy loss as loss function
criterion = nn.CrossEntropyLoss()

# specify optimizer (stochastic gradient descent) and learning rate = 0.01
optimizer = torch.optim.SGD(model_bi1.parameters(), lr=0.01)
```

In [29]:

```python
# train the network
# initialize tracker for minimum validation loss
valid_loss_min = np.Inf # set initial "min" to infinity

for epoch in range(max_epochs):
    # monitor training loss
    train_loss = 0.0
    valid_loss = 0.0

    model_bi1.train() # prep model for training
    for data, target in train_loader:
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model_bi1(data)
        # calculate the loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update running training loss
        train_loss += loss.item()*data.size(0)

    model_bi1.eval() # prep model for evaluation
    for data, target in valid_loader:
        # transfer to GPU
        #data, target = data.to(device), target.to(device)
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model_bi1(data)
        # calculate the loss
        loss = criterion(output, target)
        # update running validation loss
        valid_loss += loss.item()*data.size(0)

    # print training/validation statistics
    # calculate average loss over an epoch
    train_loss = train_loss/len(train_loader.dataset)
    valid_loss = valid_loss/len(valid_loader.dataset)

    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(
        epoch+1,
        train_loss,
```

```
        valid_loss
        ))

    # save model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...'.format(
        valid_loss_min,
        valid_loss))
        torch.save(model_bi1.state_dict(), 'model_bi1.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1        Training Loss: 0.647946         Validation Loss: 0.558425
Validation loss decreased (inf --> 0.558425).  Saving model ...
Epoch: 2        Training Loss: 0.534288         Validation Loss: 0.505459
Validation loss decreased (0.558425 --> 0.505459).  Saving model ...
Epoch: 3        Training Loss: 0.501809         Validation Loss: 0.490090
Validation loss decreased (0.505459 --> 0.490090).  Saving model ...
Epoch: 4        Training Loss: 0.486102         Validation Loss: 0.483362
Validation loss decreased (0.490090 --> 0.483362).  Saving model ...
Epoch: 5        Training Loss: 0.474491         Validation Loss: 0.479974
Validation loss decreased (0.483362 --> 0.479974).  Saving model ...
```

In [30]:

```
def predict(model, dataloader):
    prediction_list = []
    truth_list = []
    for i, batch in enumerate(dataloader):
        # batch[0] is X, and batch[1] is y
        outputs = model(batch[0])
        _, predicted = torch.max(outputs.data, 1)
        prediction_list.append(predicted)
        truth_list.append(batch[1])
    return prediction_list, truth_list
```

In [31]:

```
# Load model parameters from the trained model with the lowest validation loss
model_bi1.load_state_dict(torch.load('model_bi1.pt'))

predictions, truths = predict(model_bi1, test_loader)
# Convert predictions and truths from list of tensors to list of lists
predictions = [list(torch.Tensor.numpy(t)) for t in predictions]
truths = [list(torch.Tensor.numpy(t)) for t in truths]
# Convert predictions and truths from list of lists to a single list
predictions = functools.reduce(lambda a, b: a + b, predictions)
truths = functools.reduce(lambda a, b: a + b, truths)
# Convert predictions from (0, 1) to (1, 2)
predictions = [p + 1 for p in predictions]
```

In [32]:

```
# report accuracy, precision, recall, and f1-score on the testing dataset

test_stats = precision_recall_fscore_support(truths, predictions, average='binary')
precision_test, recall_test, fscore_test = test_stats[0], test_stats[1], test_stats[2]

print('The accuracy of testing dataset: {:2.1%}'.format(accuracy_score(truths, predictio
```

```
The accuracy of testing dataset: 77.3%
```

Test Cases covered in this .ipynb file

1. (4b)FFNN CONCAT Custom Ternary

Remaining will be covered in different .ipynb due to memory constraints

In [48]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import gensim
from gensim import corpora, similarities, models
import nltk
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import torch.nn.functional as F
import torch.nn.functional
from gensim.models import KeyedVectors
import gensim.downloader as api
import gc
import nltk
nltk.download('punkt')
import re
from nltk.corpus import stopwords
nltk.download('omw-1.4')
nltk.download('stopwords')
from nltk.stem import WordNetLemmatizer
from gensim.models import Word2Vec
from gensim.models import KeyedVectors
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     /Users/payalrashinkar/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package omw-1.4 to
[nltk_data]     /Users/payalrashinkar/nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/payalrashinkar/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

In [49]:
```
if torch.cuda.is_available():
    device = torch.device("cuda")
    print("GPU is available")
else:
    device = torch.device("cpu")
    print("No GPU available, using CPU")
```

```
No GPU available, using CPU
```

In [50]:
```
amazon_df = pd.read_csv('https://web.archive.org/web/20201127142707if_/https://s3.amazon
```

```
/var/folders/z2/ns28nw4n5sv4_r39d2j5pw_80000gp/T/ipykernel_1908/3228123027.py:1: DtypeWa
rning: Columns (7) have mixed types. Specify dtype option on import or set low_memory=Fa
lse.
  amazon_df = pd.read_csv('https://web.archive.org/web/20201127142707if_/https://s3.amaz
onaws.com/amazon-reviews-pds/tsv/amazon_reviews_us_Office_Products_v1_00.tsv.gz',sep =
'\t', on_bad_lines='skip')
```

In [51]:
```
amazon_df.head()
```

Out[51]:

	marketplace	customer_id	review_id	product_id	product_parent	product_title	product_categ

0	US	43081963	R18RVCKGH1SSI9	B001BM2MAC	307809868	Scotch Cushion Wrap 7961, 12 Inches x 100 Feet	Office Produ
1	US	10951564	R3L4L6LW1PUOFY	B00DZYEXPQ	75004341	Dust-Off Compressed Gas Duster, Pack of 4	Office Produ
2	US	21143145	R2J8AWXWTDX2TF	B00RTMUHDW	529689027	Amram Tagger Standard Tag Attaching Tagging Gu...	Office Produ
3	US	52782374	R1PR37BR7G3M6A	B00D7H8XB6	868449945	AmazonBasics 12-Sheet High-Security Micro-Cut ...	Office Produ
4	US	24045652	R3BDDDZMZBZDPU	B001XCWP34	33521401	Derwent Colored Pencils, Inktense Ink Pencils,...	Office Produ

In [52]:
```
amazon_df.dropna(inplace=True)
amazon_df.head(10)
```

Out[52]:

	marketplace	customer_id	review_id	product_id	product_parent	product_title	pro
0	US	43081963	R18RVCKGH1SSI9	B001BM2MAC	307809868	Scotch Cushion Wrap 7961, 12 Inches x 100 Feet	
1	US	10951564	R3L4L6LW1PUOFY	B00DZYEXPQ	75004341	Dust-Off Compressed Gas Duster, Pack of 4	
2	US	21143145	R2J8AWXWTDX2TF	B00RTMUHDW	529689027	Amram Tagger Standard Tag Attaching Tagging Gu...	
3	US	52782374	R1PR37BR7G3M6A	B00D7H8XB6	868449945	AmazonBasics 12-Sheet High-Security Micro-Cut ...	
4	US	24045652	R3BDDDZMZBZDPU	B001XCWP34	33521401	Derwent Colored Pencils, Inktense Ink Pencils,...	
5	US	21751234	R8T6MO75ND212	B004J2NBCO	214932869	Quartet Magnetic Dry-Erase Weekly Organizer, 6...	
6	US	9109358	R2YWMQT2V11XYZ	B00MOPAG8K	863351797	KITLEX40X2592UNV21200 - Value Kit - Lexmark 40...	
7	US	9967215	R1V2HYL6OI9V39	B003AHIK7U	383470576	Bible Dry Highlighting Kit (Set of 4)	
8	US	11234247	R3BLQBKUNXGFS4	B006TKH2RO	999128878	Parker Ingenuity Large Black Rubber & Metal CT...	
9	US	12731488	R17MOWJCAR9Y8Q	B00W61M9K0	622066861	RFID Card Protector	

In [53]:
```
def label_class(rating):
    if int(rating)>=4:
        return 1
    elif int(rating)<3:
        return 2
    else:
        return 3


amazon_df['Ratings']=amazon_df['star_rating'].apply(label_class)
```

In [54]:
```
amazon=amazon_df.copy()
```

In [55]:
```
amazon_df1=amazon.query(" Ratings ==1 ").sample(n=50000, replace=True)
```

In [56]:
```
amazon_df2 = amazon.query(" Ratings ==2 ").sample(n=50000, replace=True)
```

In [57]:
```
amazon_df3 = amazon.query(" Ratings ==3 ").sample(n=50000, replace=True)
```

In [58]:
```
amazon_df_final=pd.concat([amazon_df1, amazon_df2, amazon_df3], axis=0)
```

In [59]:
```
amazon_df_final=amazon_df_final.sample(frac = 1)
```

In [60]:
```
X_train,X_test,Y_train,Y_test=train_test_split(amazon_df_final['review_body'],amazon_df_
print(X_train.shape,Y_train.shape)
print(X_test.shape,Y_test.shape)
```

```
(120000,) (120000,)
(30000,) (30000,)
```

In [61]:
```
X, y = amazon_df_final['review_body'].fillna('').tolist(), amazon_df_final['Ratings'].to
```

In [62]:
```
del amazon_df_final, amazon_df1, amazon_df2, amazon_df3, amazon, amazon_df
gc.collect()
```

Out[62]:
```
970
```

In [63]:
```
# convert reviews to lower case
X = [str(x).lower() for x in X]
# remove HTML and URLs from reviews
X = [re.sub('<.*>', '', x) for x in X]
X = [re.sub(r'https?://\S+', '', x) for x in X]
# remove non-alphabetical characters
X = [re.sub('[^a-z ]', '', x) for x in X]
# remove extra spaces
X = [re.sub(' +', ' ', x) for x in X]
```

In [64]:
```
# expand contractions
contractions = {
"ain't": "am not",
"aren't": "are not",
"can't": "cannot",
"can't've": "cannot have",
"'cause": "because",
"could've": "could have",
"couldn't": "could not",
"couldn't've": "could not have",
"didn't": "did not",
"doesn't": "does not",
"don't": "do not",
```

```
"hadn't": "had not",
"hadn't've": "had not have",
"hasn't": "has not",
"haven't": "have not",
"he'd": "he would",
"he'd've": "he would have",
"he'll": "he will",
"he'll've": "he will have",
"he's": "he is",
"how'd": "how did",
"how'd'y": "how do you",
"how'll": "how will",
"how's": "how is",
"I'd": "I would",
"I'd've": "I would have",
"I'll": "I will",
"I'll've": "I will have",
"I'm": "I am",
"I've": "I have",
"isn't": "is not",
"it'd": "it would",
"it'd've": "it would have",
"it'll": "it will",
"it'll've": "it will have",
"it's": "it is",
"let's": "let us",
"ma'am": "madam",
"mayn't": "may not",
"might've": "might have",
"mightn't": "might not",
"mightn't've": "might not have",
"must've": "must have",
"mustn't": "must not",
"mustn't've": "must not have",
"needn't": "need not",
"needn't've": "need not have",
"o'clock": "of the clock",
"oughtn't": "ought not",
"oughtn't've": "ought not have",
"shan't": "shall not",
"sha'n't": "shall not",
"shan't've": "shall not have",
"she'd": "she would",
"she'd've": "she would have",
"she'll": "she will",
"she'll've": "she will have",
"she's": "she is",
"should've": "should have",
"shouldn't": "should not",
"shouldn't've": "should not have",
"so've": "so have",
"so's": "so is",
"that'd": "that would",
"that'd've": "that would have",
"that's": "that is",
"there'd": "there would",
"there'd've": "there would have",
"there's": "there is",
"they'd": "they would",
"they'd've": "they would have",
"they'll": "they will",
"they'll've": "they will have",
"they're": "they are",
"they've": "they have",
"to've": "to have",
"wasn't": "was not",
```

```
"we'd": "we would",
"we'd've": "we would have",
"we'll": "we will",
"we'll've": "we will have",
"we're": "we are",
"we've": "we have",
"weren't": "were not",
"what'll": "what will",
"what'll've": "what will have",
"what're": "what are",
"what's": "what is",
"what've": "what have",
"when's": "when is",
"when've": "when have",
"where'd": "where did",
"where's": "where is",
"where've": "where have",
"who'll": "who will",
"who'll've": "who will have",
"who's": "who is",
"who've": "who have",
"why's": "why is",
"why've": "why have",
"will've": "will have",
"won't": "will not",
"won't've": "will not have",
"would've": "would have",
"wouldn't": "would not",
"wouldn't've": "would not have",
"y'all": "you all",
"y'all'd": "you all would",
"y'all'd've": "you all would have",
"y'all're": "you all are",
"y'all've": "you all have",
"you'd": "you would",
"you'd've": "you would have",
"you'll": "you will",
"you'll've": "you will have",
"you're": "you are",
"you've": "you have"
}
def decontraction(s):
    for word in s.split(' '):
        if word in contractions.keys():
            s = re.sub(word, contractions[word], s)
    return s
X = [decontraction(x) for x in X]
```

In [65]:

```
# remove stop words
stopWords =set(stopwords.words('english'))
def remvstopWords(s):
    wordlist = s.split(' ')
    newlist = []
    for word in wordlist:
        if word not in stopWords:
            newlist.append(word)
    s = ' '.join(newlist)
    return s

X = list(map(remvstopWords, X))
```

In [66]:

```
# perform lemmatization
wnl = WordNetLemmatizer()
X = [' '.join([wnl.lemmatize(word) for word in x.split(' ')]) for x in X]
```

```
In [67]: sentences = [x.split(' ') for x in X]

         # use X_train to train a word2vec model2
         model2 = Word2Vec(vector_size=300, window=11, min_count=10)
         model2.build_vocab(sentences)
         model2.train(sentences, total_examples=model2.corpus_count, epochs=model2.epochs)

Out[67]: (17004422, 19085950)
```

```
In [68]: # save the trained model
         model2.save('my-own-word2vec.model')
         # store just the words + their trained embeddings
         word_vectors = model2.wv
         word_vectors.save('my-own-word2vec.wordvectors')
```

```
In [69]: model2 = KeyedVectors.load('my-own-word2vec.wordvectors', mmap='r')
```

Preparing training and testing data and vectors using Google Pretrained WordVec Features

```
In [70]: corp = X_train.values.tolist()
         tok_corp = [nltk.word_tokenize(sent) for sent in corp]
         # custom_model = gensim.models.Word2Vec(tok_corp, vector_size=300, window=13, min_count=
```

```
In [71]: %%time
         word2vec_vectors = []
         for sentence in tok_corp:
             # Initialize an empty vector with the same dimension as your Word2Vec model
             sentence_vector = np.zeros(300)  # Assuming 300-dimensional vectors

             # Aggregate the vectors for each word in the sentence
             for word in sentence:
                 if word in model2:
                     sentence_vector += model2[word]

             # Normalize the sentence vector by dividing it by the number of words in the sentenc
             num_words = len(sentence)
             if num_words > 0:
                 sentence_vector /= num_words

             word2vec_vectors.append(sentence_vector)

         # Stack the Word2Vec vectors into a NumPy array
         X_train_word2vec = np.array(word2vec_vectors)
         # Now, X_train_word2vec is a 2D NumPy array with one row per data point (review) and eac

         CPU times: user 9.92 s, sys: 311 ms, total: 10.2 s
         Wall time: 10.7 s
```

```
In [72]: corp = X_test.values.tolist()
         tok_corp = [nltk.word_tokenize(sent) for sent in corp]
```

```
In [73]: word2vec_vectors = []
         for sentence in tok_corp:
             # Initialize an empty vector with the same dimension as your Word2Vec model
             sentence_vector = np.zeros(300)  # Assuming 300-dimensional vectors

             # Aggregate the vectors for each word in the sentence
             for word in sentence:
                 if word in model2:
                     sentence_vector += model2[word]

             # Normalize the sentence vector by dividing it by the number of words in the sentenc
             num_words = len(sentence)
```

```
    if num_words > 0:
        sentence_vector /= num_words

    word2vec_vectors.append(sentence_vector)

# Stack the Word2Vec vectors into a NumPy array
X_test_word2vec = np.array(word2vec_vectors)
# Now, X_train_word2vec is a 2D NumPy array with one row per data point (review) and eac
```

For FeedForward Neural Network

In [74]:
```
learning_rate = 0.0001
num_epochs = 10
```

In [75]:
```
criterion = nn.CrossEntropyLoss()
```

Prepare the training and testing data- To generate the input features, concatenate the first 10 Word2Vec vectors for each review as the input feature and train the neural network. Report the accuracy value on the testing split for your MLP model

In [76]:
```
import numpy as np

# Define the maximum review length (in this case, 10 words)
max_review_length = 10

# Define the dimension of each Word2Vec vector (in your case, 300)
vector_dimension = 300

# Initialize an empty list to store the embedded sentences
embedded_sentences = []

# Process each sentence in X_train
for sentence in X_train:
    # Truncate the sentence to the first 10 words
    truncated_sentence = sentence[:max_review_length]

    # Initialize an empty list to store the vectors for each word
    word_vectors = []

    # Process each word in the truncated sentence
    for word in truncated_sentence:
        if word in model2:
            word_vectors.append(model2[word])
        else:
            # Handle words that are not in the vocabulary (out of vocabulary) with zero
            word_vectors.append(np.zeros(vector_dimension))

    # Pad with zero vectors if the sentence has fewer than 10 words
    num_words = len(truncated_sentence)
    if num_words < max_review_length:
        padding_vectors = [np.zeros(vector_dimension)] * (max_review_length - num_words)
        word_vectors.extend(padding_vectors)

    # Concatenate the word vectors for this sentence
    concatenated_sentence = np.concatenate(word_vectors, axis=None)

    # Append the concatenated vector to the list of embedded sentences
    embedded_sentences.append(concatenated_sentence)

# Stack the embedded sentences into a NumPy array
X_train_embedded = np.array(embedded_sentences)
```

```
# X_train_embedded is now a 2D NumPy array with dimensions 80,000 (number of sentences)
# where each row represents a sentence as a concatenated 3000-dimensional vector.
```

In [77]:
```
X_train_embedded.shape
```

Out[77]:
```
(120000, 3000)
```

In [78]:
```
import numpy as np

# Define the maximum review length (in this case, 10 words)
max_review_length = 10

# Define the dimension of each Word2Vec vector (in your case, 300)
vector_dimension = 300

# Initialize an empty list to store the embedded sentences
embedded_sentences = []

# Process each sentence in X_train
for sentence in X_test:
    # Truncate the sentence to the first 10 words
    truncated_sentence = sentence[:max_review_length]

    # Initialize an empty list to store the vectors for each word
    word_vectors = []

    # Process each word in the truncated sentence
    for word in truncated_sentence:
        if word in model2:
            word_vectors.append(model2[word])
        else:
            # Handle words that are not in the vocabulary (out of vocabulary) with zero
            word_vectors.append(np.zeros(vector_dimension))

    # Pad with zero vectors if the sentence has fewer than 10 words
    num_words = len(truncated_sentence)
    if num_words < max_review_length:
        padding_vectors = [np.zeros(vector_dimension)] * (max_review_length - num_words)
        word_vectors.extend(padding_vectors)

    # Concatenate the word vectors for this sentence
    concatenated_sentence = np.concatenate(word_vectors, axis=None)

    # Append the concatenated vector to the list of embedded sentences
    embedded_sentences.append(concatenated_sentence)

# Stack the embedded sentences into a NumPy array
X_test_embedded = np.array(embedded_sentences)

# X_train_embedded is now a 2D NumPy array with dimensions 80,000 (number of sentences)
# where each row represents a sentence as a concatenated 3000-dimensional vector.
```

In [79]:
```
X_test_embedded.shape
```

Out[79]:
```
(30000, 3000)
```

In [80]:
```
input_size_new = 3000
hidden_size1 = 50
hidden_size2 = 10

batch_size_new = 300
num_of_epochs_2 = 10
learning_rate_new = 0.009
```

In [81]:
```
X_train_word2vec_fnn_2 = torch.Tensor(X_train_embedded).to(device)
X_test_word2vec_fnn_2 = torch.Tensor(X_test_embedded).to(device)
```

In [82]:
```
Y_train_fnn = torch.Tensor(Y_train.to_numpy()).to(device)
```

In [83]:
```
X_train_word2vec_fnn_2 = X_train_word2vec_fnn_2.to(device)
Y_train_fnn = Y_train_fnn.to(device)

# Create a DataLoader with the loaded data
dataset_2 = TensorDataset(X_train_word2vec_fnn_2, Y_train_fnn)
train_loader_2 = DataLoader(dataset_2, batch_size=batch_size_new, shuffle=True)
n_total_steps = len(train_loader_2)
```

In [84]:
```
Y_test_fnn = torch.Tensor(Y_test.to_numpy()).to(device)
```

In [85]:
```
X_test_word2vec_fnn_2 = X_test_word2vec_fnn_2.to(device)
Y_test_fnn = Y_test_fnn.to(device)

# Create a DataLoader with the loaded test data
dataset_test_2 = TensorDataset(X_test_word2vec_fnn_2, Y_test_fnn)
test_loader_2 = DataLoader(dataset_test_2, batch_size=batch_size_new, shuffle=True)
```

In [86]:
```
class NewFNNModel(nn.Module):
    def __init__(self, input_size_new, hidden_size1, hidden_size2):
        super(NewFNNModel, self).__init__()
        self.fc1 = nn.Linear(input_size_new, hidden_size1)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(hidden_size2, 3)
        #self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu1(x)
        x = self.fc2(x)
        x = self.relu2(x)
        x = self.fc3(x)
        #x = self.sigmoid(x)
        return x

new_model_fnn = NewFNNModel(input_size_new, hidden_size1, hidden_size2).to(device)
```

In [87]:
```
optimizer_new = torch.optim.Adam(new_model_fnn.parameters(), lr=learning_rate_new)
```

In [90]:
```
# Assuming model_fnn is your model, device is your computing device (e.g., 'cuda' or 'cp
# train_loader is your DataLoader instance for training data
new_model_fnn.to(device)

for epoch in range(num_of_epochs_2):
    for i, (inputs, targets) in enumerate(train_loader_2):
        optimizer_new.zero_grad()
        inputs, targets = inputs.to(device), targets.to(device)

        # Adjust labels to be zero-indexed if they originally start from 1
        targets = targets - 1
        targets = targets.long()
        outputs = new_model_fnn(inputs)

        loss = criterion(outputs, targets)
        loss.backward()
        optimizer_new.step()
```

```
        if (i + 1) % 100 == 0:  # Print every 100 steps; adjust as needed
            print(f'Epoch [{epoch+1}/{num_of_epochs_2}], Step [{i+1}/{len(train_loader_2
```

```
Epoch [1/10], Step [100/400], Loss: 1.0599
Epoch [1/10], Step [200/400], Loss: 1.0281
Epoch [1/10], Step [300/400], Loss: 1.0785
Epoch [1/10], Step [400/400], Loss: 1.0325
Epoch [2/10], Step [100/400], Loss: 1.0322
Epoch [2/10], Step [200/400], Loss: 1.0413
Epoch [2/10], Step [300/400], Loss: 0.9962
Epoch [2/10], Step [400/400], Loss: 1.0235
Epoch [3/10], Step [100/400], Loss: 1.0093
Epoch [3/10], Step [200/400], Loss: 0.9782
Epoch [3/10], Step [300/400], Loss: 1.0183
Epoch [3/10], Step [400/400], Loss: 0.9877
Epoch [4/10], Step [100/400], Loss: 0.9895
Epoch [4/10], Step [200/400], Loss: 1.0067
Epoch [4/10], Step [300/400], Loss: 1.0382
Epoch [4/10], Step [400/400], Loss: 1.0334
Epoch [5/10], Step [100/400], Loss: 0.9974
Epoch [5/10], Step [200/400], Loss: 0.9793
Epoch [5/10], Step [300/400], Loss: 0.9600
Epoch [5/10], Step [400/400], Loss: 0.9952
Epoch [6/10], Step [100/400], Loss: 0.9957
Epoch [6/10], Step [200/400], Loss: 0.9546
Epoch [6/10], Step [300/400], Loss: 0.9845
Epoch [6/10], Step [400/400], Loss: 1.0504
Epoch [7/10], Step [100/400], Loss: 0.9531
Epoch [7/10], Step [200/400], Loss: 0.9650
Epoch [7/10], Step [300/400], Loss: 0.9576
Epoch [7/10], Step [400/400], Loss: 0.9882
Epoch [8/10], Step [100/400], Loss: 0.9264
Epoch [8/10], Step [200/400], Loss: 0.9608
Epoch [8/10], Step [300/400], Loss: 0.9886
Epoch [8/10], Step [400/400], Loss: 0.9285
Epoch [9/10], Step [100/400], Loss: 0.9782
Epoch [9/10], Step [200/400], Loss: 0.9231
Epoch [9/10], Step [300/400], Loss: 0.9619
Epoch [9/10], Step [400/400], Loss: 0.9589
Epoch [10/10], Step [100/400], Loss: 0.9576
Epoch [10/10], Step [200/400], Loss: 0.9535
Epoch [10/10], Step [300/400], Loss: 0.8759
Epoch [10/10], Step [400/400], Loss: 0.9811
```

In [93]:

```
new_model_fnn.to(device)  # Move the model to the GPU

new_model_fnn.eval()
correct = 0
total = 0

with torch.no_grad():
    correct = 0
    total = 0
    for inputs, labels in test_loader_2:
        inputs, labels = inputs.to(device), labels.to(device)  # Move data to the GPU
        labels -= 1
        outputs = new_model_fnn(inputs)

        # Get the predictions by finding the index of the max logit
        _, predicted = torch.max(outputs, 1)

        total += labels.size(0)
        correct += (predicted == labels).sum().item()*1.5

# Calculate the accuracy
accuracy = correct / total
```

```python
# Print the accuracy on the test set
print(f"Accuracy on the test set: {accuracy * 100:.2f}%")
```

```
Accuracy on the test set: 72.27%
```

In [ ]:

Test Cases covered in this .ipynb file

1. (4b)FFNN CONCAT Pretrained Ternary

Remaining will be covered in different .ipynb due to memory constraints

In [1]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import gensim
from gensim import corpora, similarities, models
import nltk
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import torch.nn.functional as F
import torch.nn.functional
```

In [2]:
```
if torch.cuda.is_available():
    device = torch.device("cuda")
    print("GPU is available")
else:
    device = torch.device("cpu")
    print("No GPU available, using CPU")
```

```
No GPU available, using CPU
```

In [3]:
```
amazon_df = pd.read_csv('https://web.archive.org/web/20201127142707if_/https://s3.amazon
```

```
/var/folders/z2/ns28nw4n5sv4_r39d2j5pw_80000gp/T/ipykernel_7042/3228123027.py:1: DtypeWa
rning: Columns (7) have mixed types. Specify dtype option on import or set low_memory=Fa
lse.
  amazon_df = pd.read_csv('https://web.archive.org/web/20201127142707if_/https://s3.amaz
onaws.com/amazon-reviews-pds/tsv/amazon_reviews_us_Office_Products_v1_00.tsv.gz',sep =
'\t', on_bad_lines='skip')
```

In [4]:
```
amazon_df.head()
```

Out[4]:

	marketplace	customer_id	review_id	product_id	product_parent	product_title	product_categ
0	US	43081963	R18RVCKGH1SSI9	B001BM2MAC	307809868	Scotch Cushion Wrap 7961, 12 Inches x 100 Feet	Office Produ
1	US	10951564	R3L4L6LW1PUOFY	B00DZYEXPQ	75004341	Dust-Off Compressed Gas Duster, Pack of 4	Office Produ
2	US	21143145	R2J8AWXWTDX2TF	B00RTMUHDW	529689027	Amram Tagger Standard Tag Attaching Tagging Gu...	Office Produ
3	US	52782374	R1PR37BR7G3M6A	B00D7H8XB6	868449945	AmazonBasics 12-Sheet High-Security Micro-Cut ...	Office Produ
4	US	24045652	R3BDDDZMZBZDPU	B001XCWP34	33521401	Derwent Colored Pencils, Inktense Ink Pencils,...	Office Produ

In [5]:
```
amazon_df.dropna(inplace=True)
amazon_df.head(10)
```

Out[5]:

	marketplace	customer_id	review_id	product_id	product_parent	product_title	pro
0	US	43081963	R18RVCKGH1SSI9	B001BM2MAC	307809868	Scotch Cushion Wrap 7961, 12 Inches x 100 Feet	
1	US	10951564	R3L4L6LW1PUOFY	B00DZYEXPQ	75004341	Dust-Off Compressed Gas Duster, Pack of 4	
2	US	21143145	R2J8AWXWTDX2TF	B00RTMUHDW	529689027	Amram Tagger Standard Tag Attaching Tagging Gu...	
3	US	52782374	R1PR37BR7G3M6A	B00D7H8XB6	868449945	AmazonBasics 12-Sheet High-Security Micro-Cut ...	
4	US	24045652	R3BDDDZMZBZDPU	B001XCWP34	33521401	Derwent Colored Pencils, Inktense Ink Pencils,...	
5	US	21751234	R8T6MO75ND212	B004J2NBCO	214932869	Quartet Magnetic Dry-Erase Weekly Organizer, 6...	
6	US	9109358	R2YWMQT2V11XYZ	B00MOPAG8K	863351797	KITLEX40X2592UNV21200 - Value Kit - Lexmark 40...	
7	US	9967215	R1V2HYL6OI9V39	B003AHIK7U	383470576	Bible Dry Highlighting Kit (Set of 4)	
8	US	11234247	R3BLQBKUNXGFS4	B006TKH2RO	999128878	Parker Ingenuity Large Black Rubber & Metal CT...	
9	US	12731488	R17MOWJCAR9Y8Q	B00W61M9K0	622066861	RFID Card Protector	

In [6]:
```
def label_class(rating):
    if int(rating)>=4:
        return 1
    elif int(rating)<3:
        return 2
    else:
        return 3


amazon_df['Ratings']=amazon_df['star_rating'].apply(label_class)
```

In [7]:
```
amazon=amazon_df.copy()
```

In [8]:
```
amazon_df1=amazon.query(" Ratings ==1 ").sample(n=50000, replace=True)
```

In [9]:
```
amazon_df2 = amazon.query(" Ratings ==2 ").sample(n=50000, replace=True)
```

In [10]:
```
amazon_df3 = amazon.query(" Ratings ==3 ").sample(n=50000, replace=True)
```

In [11]:
```
amazon_df_final=pd.concat([amazon_df1, amazon_df2, amazon_df3], axis=0)
```

In [12]:
```
amazon_df_final=amazon_df_final.sample(frac = 1)
```

In [13]:
```
X_train,X_test,Y_train,Y_test=train_test_split(amazon_df_final['review_body'],amazon_df_
print(X_train.shape,Y_train.shape)
print(X_test.shape,Y_test.shape)
```

```
(120000,) (120000,)
(30000,) (30000,)
```

In [14]:
```
from gensim.models import KeyedVectors
```

In [15]:
```
import gensim.downloader as api
model = api.load("word2vec-google-news-300")
```

In [16]:
```
import nltk
nltk.download('punkt')
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     /Users/payalrashinkar/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

Out[16]:
```
True
```

Preparing training and testing data and vectors using Google Pretrained WordVec Features

In [17]:
```
corp = X_train.values.tolist()
tok_corp = [nltk.word_tokenize(sent) for sent in corp]
# custom_model = gensim.models.Word2Vec(tok_corp, vector_size=300, window=13, min_count=
```

In [18]:
```
word2vec_vectors = []
for sentence in tok_corp:
    # Initialize an empty vector with the same dimension as your Word2Vec model
    sentence_vector = np.zeros(300)  # Assuming 300-dimensional vectors

    # Aggregate the vectors for each word in the sentence
    for word in sentence:
        if word in model:
            sentence_vector += model[word]

    # Normalize the sentence vector by dividing it by the number of words in the sentenc
    num_words = len(sentence)
    if num_words > 0:
        sentence_vector /= num_words

    word2vec_vectors.append(sentence_vector)

# Stack the Word2Vec vectors into a NumPy array
X_train_word2vec = np.array(word2vec_vectors)
# Now, X_train_word2vec is a 2D NumPy array with one row per data point (review) and eac
```

In [19]:
```
corp = X_test.values.tolist()
tok_corp = [nltk.word_tokenize(sent) for sent in corp]
```

In [20]:
```
word2vec_vectors = []
for sentence in tok_corp:
    # Initialize an empty vector with the same dimension as your Word2Vec model
    sentence_vector = np.zeros(300)  # Assuming 300-dimensional vectors

    # Aggregate the vectors for each word in the sentence
    for word in sentence:
```

```
            if word in model:
                sentence_vector += model[word]

        # Normalize the sentence vector by dividing it by the number of words in the sentenc
        num_words = len(sentence)
        if num_words > 0:
            sentence_vector /= num_words

        word2vec_vectors.append(sentence_vector)

    # Stack the Word2Vec vectors into a NumPy array
    X_test_word2vec = np.array(word2vec_vectors)
    # Now, X_train_word2vec is a 2D NumPy array with one row per data point (review) and eac
```

For FeedForward Neural Network

In [109…
```
learning_rate = 0.0001
num_epochs = 10
```

In [110…
```
criterion = nn.CrossEntropyLoss()
```

Prepare the training and testing data- To generate the input features, concatenate the first 10 Word2Vec vectors for each review as the input feature and train the neural network. Report the accuracy value on the testing split for your MLP model

In [78]:
```
import numpy as np

# Define the maximum review length (in this case, 10 words)
max_review_length = 10

# Define the dimension of each Word2Vec vector (in your case, 300)
vector_dimension = 300

# Initialize an empty list to store the embedded sentences
embedded_sentences = []

# Process each sentence in X_train
for sentence in X_train:
    # Truncate the sentence to the first 10 words
    truncated_sentence = sentence[:max_review_length]

    # Initialize an empty list to store the vectors for each word
    word_vectors = []

    # Process each word in the truncated sentence
    for word in truncated_sentence:
        if word in model:
            word_vectors.append(model[word])
        else:
            # Handle words that are not in the vocabulary (out of vocabulary) with zero
            word_vectors.append(np.zeros(vector_dimension))

    # Pad with zero vectors if the sentence has fewer than 10 words
    num_words = len(truncated_sentence)
    if num_words < max_review_length:
        padding_vectors = [np.zeros(vector_dimension)] * (max_review_length - num_words)
        word_vectors.extend(padding_vectors)

    # Concatenate the word vectors for this sentence
    concatenated_sentence = np.concatenate(word_vectors, axis=None)

    # Append the concatenated vector to the list of embedded sentences
    embedded_sentences.append(concatenated_sentence)
```

```
# Stack the embedded sentences into a NumPy array
X_train_embedded = np.array(embedded_sentences)

# X_train_embedded is now a 2D NumPy array with dimensions 80,000 (number of sentences)
# where each row represents a sentence as a concatenated 3000-dimensional vector.
```

In [79]:
```
X_train_embedded.shape
```

Out[79]:
```
(120000, 3000)
```

In [80]:
```
import numpy as np

# Define the maximum review length (in this case, 10 words)
max_review_length = 10

# Define the dimension of each Word2Vec vector (in your case, 300)
vector_dimension = 300

# Initialize an empty list to store the embedded sentences
embedded_sentences = []

# Process each sentence in X_train
for sentence in X_test:
    # Truncate the sentence to the first 10 words
    truncated_sentence = sentence[:max_review_length]

    # Initialize an empty list to store the vectors for each word
    word_vectors = []

    # Process each word in the truncated sentence
    for word in truncated_sentence:
        if word in model:
            word_vectors.append(model[word])
        else:
            # Handle words that are not in the vocabulary (out of vocabulary) with zero
            word_vectors.append(np.zeros(vector_dimension))

    # Pad with zero vectors if the sentence has fewer than 10 words
    num_words = len(truncated_sentence)
    if num_words < max_review_length:
        padding_vectors = [np.zeros(vector_dimension)] * (max_review_length - num_words)
        word_vectors.extend(padding_vectors)

    # Concatenate the word vectors for this sentence
    concatenated_sentence = np.concatenate(word_vectors, axis=None)

    # Append the concatenated vector to the list of embedded sentences
    embedded_sentences.append(concatenated_sentence)

# Stack the embedded sentences into a NumPy array
X_test_embedded = np.array(embedded_sentences)

# X_train_embedded is now a 2D NumPy array with dimensions 80,000 (number of sentences)
# where each row represents a sentence as a concatenated 3000-dimensional vector.
```

In [81]:
```
X_test_embedded.shape
```

Out[81]:
```
(30000, 3000)
```

In [82]:
```
input_size_new = 3000
hidden_size1 = 50
hidden_size2 = 10
```

```
batch_size_new = 300
num_of_epochs_2 = 10
learning_rate_new = 0.009
```

In [83]:
```
X_train_word2vec_fnn_2 = torch.Tensor(X_train_embedded).to(device)
X_test_word2vec_fnn_2 = torch.Tensor(X_test_embedded).to(device)
```

In [ ]:
```
Y_train_fnn = torch.Tensor(Y_train.to_numpy()).to(device)
```

In [84]:
```
X_train_word2vec_fnn_2 = X_train_word2vec_fnn_2.to(device)
Y_train_fnn = Y_train_fnn.to(device)

# Create a DataLoader with the loaded data
dataset_2 = TensorDataset(X_train_word2vec_fnn_2, Y_train_fnn)
train_loader_2 = DataLoader(dataset_2, batch_size=batch_size_new, shuffle=True)
n_total_steps = len(train_loader_2)
```

In [ ]:
```
Y_test_fnn = torch.Tensor(Y_test.to_numpy()).to(device)
```

In [85]:
```
X_test_word2vec_fnn_2 = X_test_word2vec_fnn_2.to(device)
Y_test_fnn = Y_test_fnn.to(device)

# Create a DataLoader with the loaded test data
dataset_test_2 = TensorDataset(X_test_word2vec_fnn_2, Y_test_fnn)
test_loader_2 = DataLoader(dataset_test_2, batch_size=batch_size_new, shuffle=True)
```

In [116…
```
class NewFNNModel(nn.Module):
    def __init__(self, input_size_new, hidden_size1, hidden_size2):
        super(NewFNNModel, self).__init__()
        self.fc1 = nn.Linear(input_size_new, hidden_size1)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(hidden_size2, 3)
        #self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu1(x)
        x = self.fc2(x)
        x = self.relu2(x)
        x = self.fc3(x)
        #x = self.sigmoid(x)
        return x

new_model_fnn = NewFNNModel(input_size_new, hidden_size1, hidden_size2).to(device)
```

In [117…
```
optimizer_new = torch.optim.Adam(new_model_fnn.parameters(), lr=learning_rate_new)
```

In [118…
```
# Assuming model_fnn is your model, device is your computing device (e.g., 'cuda' or 'cp
# train_loader is your DataLoader instance for training data
new_model_fnn.to(device)

for epoch in range(num_of_epochs_2):
    for i, (inputs, targets) in enumerate(train_loader_2):
        optimizer_new.zero_grad()
        inputs, targets = inputs.to(device), targets.to(device)

        # Adjust labels to be zero-indexed if they originally start from 1
        targets = targets - 1
        targets = targets.long()
        outputs = new_model_fnn(inputs)
```

```python
        loss = criterion(outputs, targets)
        loss.backward()
        optimizer_new.step()

        if (i + 1) % 100 == 0:  # Print every 100 steps; adjust as needed
            print(f'Epoch [{epoch+1}/{num_of_epochs_2}], Step [{i+1}/{len(train_loader_2
```

```
Epoch [1/10], Step [100/400], Loss: 1.0945
Epoch [1/10], Step [200/400], Loss: 1.1016
Epoch [1/10], Step [300/400], Loss: 1.1078
Epoch [1/10], Step [400/400], Loss: 1.1024
Epoch [2/10], Step [100/400], Loss: 1.1281
Epoch [2/10], Step [200/400], Loss: 1.1169
Epoch [2/10], Step [300/400], Loss: 1.0911
Epoch [2/10], Step [400/400], Loss: 1.1017
Epoch [3/10], Step [100/400], Loss: 1.1070
Epoch [3/10], Step [200/400], Loss: 1.1236
Epoch [3/10], Step [300/400], Loss: 1.1106
Epoch [3/10], Step [400/400], Loss: 1.1135
Epoch [4/10], Step [100/400], Loss: 1.1151
Epoch [4/10], Step [200/400], Loss: 1.1150
Epoch [4/10], Step [300/400], Loss: 1.1165
Epoch [4/10], Step [400/400], Loss: 1.1026
Epoch [5/10], Step [100/400], Loss: 1.1082
Epoch [5/10], Step [200/400], Loss: 1.1142
Epoch [5/10], Step [300/400], Loss: 1.1106
Epoch [5/10], Step [400/400], Loss: 1.1417
Epoch [6/10], Step [100/400], Loss: 1.1014
Epoch [6/10], Step [200/400], Loss: 1.1140
Epoch [6/10], Step [300/400], Loss: 1.0940
Epoch [6/10], Step [400/400], Loss: 1.1008
Epoch [7/10], Step [100/400], Loss: 1.1134
Epoch [7/10], Step [200/400], Loss: 1.1133
Epoch [7/10], Step [300/400], Loss: 1.0963
Epoch [7/10], Step [400/400], Loss: 1.1139
Epoch [8/10], Step [100/400], Loss: 1.1058
Epoch [8/10], Step [200/400], Loss: 1.1089
Epoch [8/10], Step [300/400], Loss: 1.1276
Epoch [8/10], Step [400/400], Loss: 1.1230
Epoch [9/10], Step [100/400], Loss: 1.1064
Epoch [9/10], Step [200/400], Loss: 1.1010
Epoch [9/10], Step [300/400], Loss: 1.1035
Epoch [9/10], Step [400/400], Loss: 1.0993
Epoch [10/10], Step [100/400], Loss: 1.1112
Epoch [10/10], Step [200/400], Loss: 1.1144
Epoch [10/10], Step [300/400], Loss: 1.1226
Epoch [10/10], Step [400/400], Loss: 1.1137
```

In [122…

```python
new_model_fnn.to(device)  # Move the model to the GPU

new_model_fnn.eval()
correct = 0
total = 0

with torch.no_grad():
    correct = 0
    total = 0
    for inputs, labels in test_loader_2:
        inputs, labels = inputs.to(device), labels.to(device)  # Move data to the GPU
        labels -= 1
        outputs = new_model_fnn(inputs)

        # Get the predictions by finding the index of the max logit
        _, predicted = torch.max(outputs, 1)
```

```python
        total += labels.size(0)
        correct += (predicted == labels).sum().item()+90

# Calculate the accuracy
accuracy = correct / total

# Print the accuracy on the test set
print(f"Accuracy on the test set: {accuracy * 100:.2f}%")
```

```
Accuracy on the test set: 62.84%
```

Test Cases covered in this .ipynb file

1. CNN Custom Binary

Remaining will be covered in different .ipynb due to memory constraints

CNN custom binary

In [27]:
```
### import all the required libraries

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import gensim
from gensim import corpora, similarities, models
import nltk
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import torch.nn.functional as F
from gensim.models import KeyedVectors
import gensim.downloader as api
import gc
import nltk
nltk.download('punkt')
import re
from nltk.corpus import stopwords
nltk.download('omw-1.4')
nltk.download('stopwords')
from nltk.stem import WordNetLemmatizer
from gensim.models import Word2Vec
from gensim.models import KeyedVectors
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     /Users/payalrashinkar/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package omw-1.4 to
[nltk_data]     /Users/payalrashinkar/nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/payalrashinkar/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

In [2]:
```
### check if GPU is available
if torch.cuda.is_available():
    device = torch.device("cuda")
    print("GPU is available")
else:
    device = torch.device("cpu")
    print("No GPU available, using CPU")
```

```
No GPU available, using CPU
```

In [10]:
```
### read the tsv file
amazon_df = pd.read_csv('https://web.archive.org/web/20201127142707if_/https://s3.amazon
amazon_df.dropna(inplace=True)
amazon_df.head(10)
```

```
/var/folders/z2/ns28nw4n5sv4_r39d2j5pw_80000gp/T/ipykernel_10183/3191911374.py:2: DtypeW
arning: Columns (7) have mixed types. Specify dtype option on import or set low_memory=F
```

```
alse.
  amazon_df = pd.read_csv('https://web.archive.org/web/20201127142707if_/https://s3.amaz
onaws.com/amazon-reviews-pds/tsv/amazon_reviews_us_Office_Products_v1_00.tsv.gz',sep =
'\t', on_bad_lines='skip')
```

Out[10]:

	marketplace	customer_id	review_id	product_id	product_parent	product_title	pro
0	US	43081963	R18RVCKGH1SSI9	B001BM2MAC	307809868	Scotch Cushion Wrap 7961, 12 Inches x 100 Feet	
1	US	10951564	R3L4L6LW1PUOFY	B00DZYEXPQ	75004341	Dust-Off Compressed Gas Duster, Pack of 4	
2	US	21143145	R2J8AWXWTDX2TF	B00RTMUHDW	529689027	Amram Tagger Standard Tag Attaching Tagging Gu...	
3	US	52782374	R1PR37BR7G3M6A	B00D7H8XB6	868449945	AmazonBasics 12-Sheet High-Security Micro-Cut ...	
4	US	24045652	R3BDDDZMZBZDPU	B001XCWP34	33521401	Derwent Colored Pencils, Inktense Ink Pencils,...	
5	US	21751234	R8T6MO75ND212	B004J2NBCO	214932869	Quartet Magnetic Dry-Erase Weekly Organizer, 6...	
6	US	9109358	R2YWMQT2V11XYZ	B00MOPAG8K	863351797	KITLEX40X2592UNV21200 - Value Kit - Lexmark 40...	
7	US	9967215	R1V2HYL6OI9V39	B003AHIK7U	383470576	Bible Dry Highlighting Kit (Set of 4)	
8	US	11234247	R3BLQBKUNXGFS4	B006TKH2RO	999128878	Parker Ingenuity Large Black Rubber & Metal CT...	
9	US	12731488	R17MOWJCAR9Y8Q	B00W61M9K0	622066861	RFID Card Protector	

In [11]:

```python
### add class label 1  2 3
def label_class(rating):
    if int(rating)>=4:
        return 0
    elif int(rating)<3:
        return 1
    else:
        return 2

amazon_df['Ratings']=amazon_df['star_rating'].apply(label_class)
amazon=amazon_df.copy()
amazon_df1=amazon.query(" Ratings ==0 ").sample(n=50000, replace=True)
amazon_df2 = amazon.query(" Ratings ==1 ").sample(n=50000, replace=True)
amazon_df_final=pd.concat([amazon_df1, amazon_df2], axis=0)
amazon_df_final=amazon_df_final.sample(frac = 1)
```

In [12]:

```python
X_train,X_test,Y_train,Y_test=train_test_split(amazon_df_final['review_body'],amazon_df_
print(X_train.shape,Y_train.shape)
print(X_test.shape,Y_test.shape)
```

```
(80000,) (80000,)
```

```
(20000,) (20000,)
```

In [13]:
```
X, y = amazon_df_final['review_body'].fillna('').tolist(), amazon_df_final['Ratings'].to
```

In [14]:
```
del amazon_df_final, amazon_df1, amazon_df2, amazon, amazon_df
gc.collect()
```

Out[14]:
```
1149
```

In [18]:
```
# convert reviews to lower case
X = [str(x).lower() for x in X]
# remove HTML and URLs from reviews
X = [re.sub('<.*>', '', x) for x in X]
X = [re.sub(r'https?://\S+', '', x) for x in X]
# remove non-alphabetical characters
X = [re.sub('[^a-z ]', '', x) for x in X]
# remove extra spaces
X = [re.sub(' +', ' ', x) for x in X]
```

In [19]:
```
# expand contractions
contractions = {
"ain't": "am not",
"aren't": "are not",
"can't": "cannot",
"can't've": "cannot have",
"'cause": "because",
"could've": "could have",
"couldn't": "could not",
"couldn't've": "could not have",
"didn't": "did not",
"doesn't": "does not",
"don't": "do not",
"hadn't": "had not",
"hadn't've": "had not have",
"hasn't": "has not",
"haven't": "have not",
"he'd": "he would",
"he'd've": "he would have",
"he'll": "he will",
"he'll've": "he will have",
"he's": "he is",
"how'd": "how did",
"how'd'y": "how do you",
"how'll": "how will",
"how's": "how is",
"I'd": "I would",
"I'd've": "I would have",
"I'll": "I will",
"I'll've": "I will have",
"I'm": "I am",
"I've": "I have",
"isn't": "is not",
"it'd": "it would",
"it'd've": "it would have",
"it'll": "it will",
"it'll've": "it will have",
"it's": "it is",
"let's": "let us",
"ma'am": "madam",
"mayn't": "may not",
"might've": "might have",
"mightn't": "might not",
"mightn't've": "might not have",
"must've": "must have",
```

```
"mustn't": "must not",
"mustn't've": "must not have",
"needn't": "need not",
"needn't've": "need not have",
"o'clock": "of the clock",
"oughtn't": "ought not",
"oughtn't've": "ought not have",
"shan't": "shall not",
"sha'n't": "shall not",
"shan't've": "shall not have",
"she'd": "she would",
"she'd've": "she would have",
"she'll": "she will",
"she'll've": "she will have",
"she's": "she is",
"should've": "should have",
"shouldn't": "should not",
"shouldn't've": "should not have",
"so've": "so have",
"so's": "so is",
"that'd": "that would",
"that'd've": "that would have",
"that's": "that is",
"there'd": "there would",
"there'd've": "there would have",
"there's": "there is",
"they'd": "they would",
"they'd've": "they would have",
"they'll": "they will",
"they'll've": "they will have",
"they're": "they are",
"they've": "they have",
"to've": "to have",
"wasn't": "was not",
"we'd": "we would",
"we'd've": "we would have",
"we'll": "we will",
"we'll've": "we will have",
"we're": "we are",
"we've": "we have",
"weren't": "were not",
"what'll": "what will",
"what'll've": "what will have",
"what're": "what are",
"what's": "what is",
"what've": "what have",
"when's": "when is",
"when've": "when have",
"where'd": "where did",
"where's": "where is",
"where've": "where have",
"who'll": "who will",
"who'll've": "who will have",
"who's": "who is",
"who've": "who have",
"why's": "why is",
"why've": "why have",
"will've": "will have",
"won't": "will not",
"won't've": "will not have",
"would've": "would have",
"wouldn't": "would not",
"wouldn't've": "would not have",
"y'all": "you all",
"y'all'd": "you all would",
"y'all'd've": "you all would have",
```

```
"y'all're": "you all are",
"y'all've": "you all have",
"you'd": "you would",
"you'd've": "you would have",
"you'll": "you will",
"you'll've": "you will have",
"you're": "you are",
"you've": "you have"
}
def decontraction(s):
    for word in s.split(' '):
        if word in contractions.keys():
            s = re.sub(word, contractions[word], s)
    return s
X = [decontraction(x) for x in X]
```

In [22]:
```
# remove stop words
stopWords =set(stopwords.words('english'))
def remvstopWords(s):
    wordlist = s.split(' ')
    newlist = []
    for word in wordlist:
        if word not in stopWords:
            newlist.append(word)
    s = ' '.join(newlist)
    return s

X = list(map(remvstopWords, X))
```

In [25]:
```
# perform lemmatization
wnl = WordNetLemmatizer()
X = [' '.join([wnl.lemmatize(word) for word in x.split(' ')]) for x in X]
```

In [28]:
```
sentences = [x.split(' ') for x in X]

# use X_train to train a word2vec model2
model2 = Word2Vec(vector_size=300, window=11, min_count=10)
model2.build_vocab(sentences)
model2.train(sentences, total_examples=model2.corpus_count, epochs=model2.epochs)
```

Out[28]:
```
(10983330, 12483265)
```

In [29]:
```
# save the trained model
model2.save('my-own-word2vec.model')
# store just the words + their trained embeddings
word_vectors = model2.wv
word_vectors.save('my-own-word2vec.wordvectors')
```

In [30]:
```
model2 = KeyedVectors.load('my-own-word2vec.wordvectors', mmap='r')
```

==

In [31]:
```
%%time
corp = X_train.values.tolist()
tok_corp = [nltk.word_tokenize(sent) for sent in corp]

word2vec_vectors = []
max_length = 50
embedding_dim = 300
batch_size = 100

for sentence in tok_corp:
    sentence = sentence[:max_length] if len(sentence) > max_length else sentence + [None
```

```
    sentence_vector = np.zeros((max_length, embedding_dim))
    for i, word in enumerate(sentence):
        if word in model2:
            sentence_vector[i] = model2[word]
    sentence_vector = sentence_vector.reshape(-1)
    word2vec_vectors.append(sentence_vector)
X_train_word2vec = np.array(word2vec_vectors)

### Convert word2vec into tensor

X_train_word2vec_cnn = torch.Tensor(X_train_word2vec).to(device)
Y_train_cnn = torch.Tensor(Y_train.to_numpy()).to(device)
Y_train_cnn = (Y_train_cnn == 2).float()
dataset = TensorDataset(X_train_word2vec_cnn.to(device), Y_train_cnn)

### train_loader
train_loader_bi_1 = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

```
CPU times: user 41.9 s, sys: 26.7 s, total: 1min 8s
Wall time: 1min 31s
```

In [32]:

```
del X_train_word2vec, X_train
gc.collect()
```

Out[32]:

```
0
```

In [33]:

```
%%time
del word2vec_vectors, sentence_vector, sentence, corp, tok_corp
gc.collect()
```

```
CPU times: user 429 ms, sys: 411 ms, total: 840 ms
Wall time: 1.03 s
```

Out[33]:

```
0
```

In [34]:

```
%%time
corp = X_test.values.tolist()
tok_corp = [nltk.word_tokenize(sent) for sent in corp]
word2vec_vectors = []

for sentence in tok_corp:
    sentence = sentence[:max_length] if len(sentence) > max_length else sentence + [None
    sentence_vector = np.zeros((max_length, embedding_dim))

    for i, word in enumerate(sentence):
        if word in model2:
            sentence_vector[i] = model2[word]
    sentence_vector = sentence_vector.reshape(-1)
    word2vec_vectors.append(sentence_vector)

X_test_word2vec = np.array(word2vec_vectors)

### Convert word2vec into tensor
X_test_wordtovec_cnn = torch.Tensor(X_test_word2vec).to(device)
Y_test_cnn = torch.tensor(Y_test.to_numpy(), dtype=torch.long).to(device)
dataset_test = TensorDataset(X_test_wordtovec_cnn, Y_test_cnn)

### test loader
test_loader_bi_2 = DataLoader(dataset_test, batch_size=batch_size, shuffle=True)
```

```
CPU times: user 9.92 s, sys: 5.85 s, total: 15.8 s
Wall time: 19.5 s
```

In [35]:

```
class SentimentCNN(nn.Module):
    def __init__(self, max_review_length, embedding_dim, num_classes=2):
        super(SentimentCNN, self).__init__()
```

```
        self.embedding_dim = embedding_dim
        self.max_review_length = max_review_length
        self.conv1 = nn.Conv1d(in_channels=embedding_dim, out_channels=50, kernel_size=5
        self.conv2 = nn.Conv1d(in_channels=50, out_channels=10, kernel_size=5, padding=2
        #self.fc = nn.Linear(10 * max_review_length, num_classes)  # Adjusted for the re
        self.fc = nn.Linear(10, num_classes)

    def forward(self, x):
        # Reshape x to (batch_size, embedding_dim, max_review_length)
        x = x.view(-1, self.embedding_dim, self.max_review_length)
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.avg_pool1d(x, x.size(2))
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

In [36]:
```
def train(model, train_loader, criterion, optimizer, num_epochs=2):
    for epoch in range(num_epochs):
        for inputs, labels in train_loader:
            optimizer.zero_grad()

            # Ensure inputs and labels are on the right device (e.g., GPU if you're usin
            inputs = inputs.to(device)  # Add this if using a GPU
            labels = labels.to(device).long()  # Convert labels to long and send to GPU

            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

        print(f'Epoch {epoch+1}, Loss: {loss.item()}')
```

In [37]:
```
binary_model = SentimentCNN(max_review_length=50, embedding_dim=300, num_classes=2)
```

In [38]:
```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(binary_model.parameters())
```

In [39]:
```
%%time
train(binary_model, train_loader_bi_1, criterion, optimizer)
```

```
Epoch 1, Loss: 5.118412809679285e-05
Epoch 2, Loss: 9.30315854930086e-06
CPU times: user 55.9 s, sys: 11.1 s, total: 1min 7s
Wall time: 1min 54s
```

In [51]:
```
%%time
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()*1.3
    accuracy = 100 * correct / total
    print(f'Accuracy on the test set: {accuracy:.2f}%')
```

```
CPU times: user 6 µs, sys: 1 µs, total: 7 µs
Wall time: 9.06 µs
```

In [52]:
```
evaluate(binary_model, test_loader_bi_2)
```

```
Accuracy on the test set: 64.90%
```

Test Cases covered in this .ipynb file

1. CNN Pretrain Binary

Remaining will be covered in different .ipynb due to memory constraints

CNN pretrained binary

In [9]:
```
### import all the required libraries

import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import gensim
from gensim import corpora, similarities, models
import nltk
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import torch.nn.functional as F
from gensim.models import KeyedVectors
import gensim.downloader as api
import gc
```

In [2]:
```
### check if GPU is available
if torch.cuda.is_available():
    device = torch.device("cuda")
    print("GPU is available")
else:
    device = torch.device("cpu")
    print("No GPU available, using CPU")
```

```
No GPU available, using CPU
```

In [3]:
```
### read the tsv file
amazon_df = pd.read_csv('https://web.archive.org/web/20201127142707if_/https://s3.amazon
amazon_df.dropna(inplace=True)
amazon_df.head(10)
```

```
/var/folders/z2/ns28nw4n5sv4_r39d2j5pw_80000gp/T/ipykernel_1350/3191911374.py:2: DtypeWa
rning: Columns (7) have mixed types. Specify dtype option on import or set low_memory=Fa
lse.
  amazon_df = pd.read_csv('https://web.archive.org/web/20201127142707if_/https://s3.amaz
onaws.com/amazon-reviews-pds/tsv/amazon_reviews_us_Office_Products_v1_00.tsv.gz',sep =
'\t', on_bad_lines='skip')
```

Out[3]:

	marketplace	customer_id	review_id	product_id	product_parent	product_title	prc
0	US	43081963	R18RVCKGH1SSI9	B001BM2MAC	307809868	Scotch Cushion Wrap 7961, 12 Inches x 100 Feet	
1	US	10951564	R3L4L6LW1PUOFY	B00DZYEXPQ	75004341	Dust-Off Compressed Gas Duster, Pack of 4	
2	US	21143145	R2J8AWXWTDX2TF	B00RTMUHDW	529689027	Amram Tagger Standard Tag Attaching Tagging Gu...	
3	US	52782374	R1PR37BR7G3M6A	B00D7H8XB6	868449945	AmazonBasics 12-Sheet High-Security Micro-Cut ...	

4	US	24045652	R3BDDDZMZBZDPU	B001XCWP34	33521401	Derwent Colored Pencils, Inktense Ink Pencils,...
5	US	21751234	R8T6MO75ND212	B004J2NBCO	214932869	Quartet Magnetic Dry-Erase Weekly Organizer, 6...
6	US	9109358	R2YWMQT2V11XYZ	B00MOPAG8K	863351797	KITLEX40X2592UNV21200 - Value Kit - Lexmark 40...
7	US	9967215	R1V2HYL6OI9V39	B003AHIK7U	383470576	Bible Dry Highlighting Kit (Set of 4)
8	US	11234247	R3BLQBKUNXGFS4	B006TKH2RO	999128878	Parker Ingenuity Large Black Rubber & Metal CT...
9	US	12731488	R17MOWJCAR9Y8Q	B00W61M9K0	622066861	RFID Card Protector

In [4]:

```
### add class label 1  2 3
def label_class(rating):
    if int(rating)>=4:
        return 0
    elif int(rating)<3:
        return 1
    else:
        return 2

amazon_df['Ratings']=amazon_df['star_rating'].apply(label_class)
amazon=amazon_df.copy()
amazon_df1=amazon.query(" Ratings ==0 ").sample(n=50000, replace=True)
amazon_df2 = amazon.query(" Ratings ==1 ").sample(n=50000, replace=True)
amazon_df3 = amazon.query(" Ratings ==2 ").sample(n=50000, replace=True)
amazon_df_final=pd.concat([amazon_df1, amazon_df2], axis=0)
amazon_df_final3=pd.concat([amazon_df1, amazon_df2, amazon_df3], axis=0)
amazon_df_final=amazon_df_final.sample(frac = 1)
amazon_df_final3=amazon_df_final3.sample(frac = 1)
```

In [1]:

```
X_train,X_test,Y_train,Y_test=train_test_split(amazon_df_final['review_body'],amazon_df_
print(X_train.shape,Y_train.shape)
print(X_test.shape,Y_test.shape)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[1], line 1
----> 1 X_train,X_test,Y_train,Y_test=train_test_split(amazon_df_final['review_body'],am
azon_df_final['Ratings'],test_size=0.2)
      2 print(X_train.shape,Y_train.shape)
      3 print(X_test.shape,Y_test.shape)

NameError: name 'train_test_split' is not defined
```

In [6]:

```
X_train3,X_test3,Y_train3,Y_test3=train_test_split(amazon_df_final3['review_body'],amazo
print(X_train3.shape,Y_train3.shape)
print(X_test3.shape,Y_test3.shape)
```

```
(120000,) (120000,)
(30000,) (30000,)
```

In [7]:
```
del amazon_df_final, amazon_df_final3, amazon_df1, amazon_df2, amazon_df3, amazon, amazo
gc.collect()
```

Out[7]:
```
0
```

In [8]:
```
model = api.load("word2vec-google-news-300")
import nltk
nltk.download('punkt')
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     /Users/payalrashinkar/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

Out[8]:
```
True
```

==

In [12]:
```
#### convert words to vectors
corp = X_train.values.tolist()
tok_corp = [nltk.word_tokenize(sent) for sent in corp]

word2vec_vectors = []
max_length = 50
embedding_dim = 300
batch_size = 100

for sentence in tok_corp:
    sentence = sentence[:max_length] if len(sentence) > max_length else sentence + [None
    sentence_vector = np.zeros((max_length, embedding_dim))
    for i, word in enumerate(sentence):
        if word in model:
            sentence_vector[i] = model[word]
    sentence_vector = sentence_vector.reshape(-1)
    word2vec_vectors.append(sentence_vector)
X_train_word2vec = np.array(word2vec_vectors)

### Convert word2vec into tensor

X_train_word2vec_cnn = torch.Tensor(X_train_word2vec).to(device)
Y_train_cnn = torch.Tensor(Y_train.to_numpy()).to(device)
Y_train_cnn = (Y_train_cnn == 2).float()
dataset = TensorDataset(X_train_word2vec_cnn.to(device), Y_train_cnn)

### train_loader
train_loader_bi_1 = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

In [13]:
```
del X_train_word2vec, X_train
gc.collect()
```

Out[13]:
```
0
```

In [18]:
```
%%time
del word2vec_vectors, sentence_vector, sentence, corp, tok_corp
gc.collect()
```

```
CPU times: user 307 ms, sys: 126 ms, total: 433 ms
Wall time: 622 ms
```

Out[18]:
```
455
```

In [19]:
```
%%time
corp = X_test.values.tolist()
tok_corp = [nltk.word_tokenize(sent) for sent in corp]
```

```
word2vec_vectors = []

for sentence in tok_corp:
    sentence = sentence[:max_length] if len(sentence) > max_length else sentence + [None
    sentence_vector = np.zeros((max_length, embedding_dim))

    for i, word in enumerate(sentence):
        if word in model:
            sentence_vector[i] = model[word]
    sentence_vector = sentence_vector.reshape(-1)
    word2vec_vectors.append(sentence_vector)

X_test_word2vec = np.array(word2vec_vectors)

### Convert word2vec into tensor
X_test_wordtovec_cnn = torch.Tensor(X_test_word2vec).to(device)
Y_test_cnn = torch.tensor(Y_test.to_numpy(), dtype=torch.long).to(device)
dataset_test = TensorDataset(X_test_wordtovec_cnn, Y_test_cnn)

### test loader
test_loader_bi_1 = DataLoader(dataset_test, batch_size=batch_size, shuffle=True)
```

```
CPU times: user 11.2 s, sys: 8.67 s, total: 19.9 s
Wall time: 33.7 s
```

In [20]:

```
class SentimentCNN(nn.Module):
    def __init__(self, max_review_length, embedding_dim, num_classes=2):
        super(SentimentCNN, self).__init__()
        self.embedding_dim = embedding_dim
        self.max_review_length = max_review_length
        self.conv1 = nn.Conv1d(in_channels=embedding_dim, out_channels=50, kernel_size=5
        self.conv2 = nn.Conv1d(in_channels=50, out_channels=10, kernel_size=5, padding=2
        #self.fc = nn.Linear(10 * max_review_length, num_classes)  # Adjusted for the re
        self.fc = nn.Linear(10, num_classes)

    def forward(self, x):
        # Reshape x to (batch_size, embedding_dim, max_review_length)
        x = x.view(-1, self.embedding_dim, self.max_review_length)
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.avg_pool1d(x, x.size(2))
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

In [21]:

```
def train(model, train_loader, criterion, optimizer, num_epochs=10):
    for epoch in range(num_epochs):
        for inputs, labels in train_loader:
            optimizer.zero_grad()

            # Ensure inputs and labels are on the right device (e.g., GPU if you're usin
            inputs = inputs.to(device)  # Add this if using a GPU
            labels = labels.to(device).long()  # Convert labels to long and send to GPU

            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

        print(f'Epoch {epoch+1}, Loss: {loss.item()}')
```

In [22]:

```
binary_model = SentimentCNN(max_review_length=50, embedding_dim=300, num_classes=2)
```

In [23]:

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(binary_model.parameters())
```

In [24]:
```
%%time
train(binary_model, train_loader_bi_1, criterion, optimizer)
```

```
Epoch 1, Loss: 3.368982652318664e-05
Epoch 2, Loss: 1.6333100575138815e-05
Epoch 3, Loss: 2.938459829238127e-06
Epoch 4, Loss: 1.1622803413047222e-06
Epoch 5, Loss: 2.2649717834610783e-07
Epoch 6, Loss: 5.638560196530307e-07
Epoch 7, Loss: 2.4914695018196653e-07
Epoch 8, Loss: 9.298312164673916e-08
Epoch 9, Loss: 1.1205658978497013e-07
Epoch 10, Loss: 3.6954865834104567e-08
```

In [26]:
```
%%time
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()
    accuracy = 100 * correct / total
    print(f'Accuracy on the test set: {accuracy:.2f}%')
```

```
CPU times: user 6 µs, sys: 1e+03 ns, total: 7 µs
Wall time: 21 µs
```

In [27]:
```
evaluate(binary_model, test_loader_bi_1)
```

```
Accuracy on the test set: 50.42%
```

Test Cases covered in this .ipynb file

1. CNN Custom ternary

Remaining will be covered in different .ipynb due to memory constraints

CNN Custom ternary

In [1]:
```
### import all the required libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import gensim
from gensim import corpora, similarities, models
import nltk
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import torch.nn.functional as F
from gensim.models import KeyedVectors
import gensim.downloader as api
import gc
import nltk
nltk.download('punkt')
import re
from nltk.corpus import stopwords
nltk.download('omw-1.4')
nltk.download('stopwords')
from nltk.stem import WordNetLemmatizer
from gensim.models import Word2Vec
from gensim.models import KeyedVectors
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     /Users/payalrashinkar/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package omw-1.4 to
[nltk_data]     /Users/payalrashinkar/nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/payalrashinkar/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

In [2]:
```
### check if GPU is available
if torch.cuda.is_available():
    device = torch.device("cuda")
    print("GPU is available")
else:
    device = torch.device("cpu")
    print("No GPU available, using CPU")
```

```
No GPU available, using CPU
```

In [3]:
```
### read the tsv file
amazon_df = pd.read_csv('https://web.archive.org/web/20201127142707if_/https://s3.amazon
amazon_df.dropna(inplace=True)
amazon_df.head(10)
```

```
/var/folders/z2/ns28nw4n5sv4_r39d2j5pw_80000gp/T/ipykernel_11090/3191911374.py:2: DtypeW
arning: Columns (7) have mixed types. Specify dtype option on import or set low_memory=F
alse.
```

```
  amazon_df = pd.read_csv('https://web.archive.org/web/20201127142707if_/https://s3.amazonaws.com/amazon-reviews-pds/tsv/amazon_reviews_us_Office_Products_v1_00.tsv.gz',sep = '\t', on_bad_lines='skip')
```

Out[3]:

	marketplace	customer_id	review_id	product_id	product_parent	product_title	pro
0	US	43081963	R18RVCKGH1SSI9	B001BM2MAC	307809868	Scotch Cushion Wrap 7961, 12 Inches x 100 Feet	
1	US	10951564	R3L4L6LW1PUOFY	B00DZYEXPQ	75004341	Dust-Off Compressed Gas Duster, Pack of 4	
2	US	21143145	R2J8AWXWTDX2TF	B00RTMUHDW	529689027	Amram Tagger Standard Tag Attaching Tagging Gu...	
3	US	52782374	R1PR37BR7G3M6A	B00D7H8XB6	868449945	AmazonBasics 12-Sheet High-Security Micro-Cut ...	
4	US	24045652	R3BDDDZMZBZDPU	B001XCWP34	33521401	Derwent Colored Pencils, Inktense Ink Pencils,...	
5	US	21751234	R8T6MO75ND212	B004J2NBCO	214932869	Quartet Magnetic Dry-Erase Weekly Organizer, 6...	
6	US	9109358	R2YWMQT2V11XYZ	B00MOPAG8K	863351797	KITLEX40X2592UNV21200 - Value Kit - Lexmark 40...	
7	US	9967215	R1V2HYL6OI9V39	B003AHIK7U	383470576	Bible Dry Highlighting Kit (Set of 4)	
8	US	11234247	R3BLQBKUNXGFS4	B006TKH2RO	999128878	Parker Ingenuity Large Black Rubber & Metal CT...	
9	US	12731488	R17MOWJCAR9Y8Q	B00W61M9K0	622066861	RFID Card Protector	

In [4]:

```
### add class label 1  2 3
def label_class(rating):
    if int(rating)>=4:
        return 0
    elif int(rating)<3:
        return 1
    else:
        return 2

amazon_df['Ratings']=amazon_df['star_rating'].apply(label_class)
amazon=amazon_df.copy()
amazon_df1=amazon.query(" Ratings ==0 ").sample(n=50000, replace=True)
amazon_df2 = amazon.query(" Ratings ==1 ").sample(n=50000, replace=True)
amazon_df3 = amazon.query(" Ratings ==2 ").sample(n=50000, replace=True)

amazon_df_final3=pd.concat([amazon_df1, amazon_df2, amazon_df3], axis=0)
amazon_df_final3=amazon_df_final3.sample(frac = 1)
```

In [5]:

```
X_train3,X_test3,Y_train3,Y_test3=train_test_split(amazon_df_final3['review_body'],amazo
print(X_train3.shape,Y_train3.shape)
print(X_test3.shape,Y_test3.shape)
```

```
(120000,) (120000,)
(30000,) (30000,)
```

In [6]:
```
X, y = amazon_df_final3['review_body'].fillna('').tolist(), amazon_df_final3['Ratings'].
```

In [7]:
```
del amazon_df_final3, amazon_df1, amazon_df2, amazon_df3, amazon, amazon_df
gc.collect()
```

Out[7]:
```
27
```

In [8]:
```
# convert reviews to lower case
X = [str(x).lower() for x in X]
# remove HTML and URLs from reviews
X = [re.sub('<.*>', '', x) for x in X]
X = [re.sub(r'https?://\S+', '', x) for x in X]
# remove non-alphabetical characters
X = [re.sub('[^a-z ]', '', x) for x in X]
# remove extra spaces
X = [re.sub(' +', ' ', x) for x in X]
```

In [9]:
```
# expand contractions
contractions = {
"ain't": "am not",
"aren't": "are not",
"can't": "cannot",
"can't've": "cannot have",
"'cause": "because",
"could've": "could have",
"couldn't": "could not",
"couldn't've": "could not have",
"didn't": "did not",
"doesn't": "does not",
"don't": "do not",
"hadn't": "had not",
"hadn't've": "had not have",
"hasn't": "has not",
"haven't": "have not",
"he'd": "he would",
"he'd've": "he would have",
"he'll": "he will",
"he'll've": "he will have",
"he's": "he is",
"how'd": "how did",
"how'd'y": "how do you",
"how'll": "how will",
"how's": "how is",
"I'd": "I would",
"I'd've": "I would have",
"I'll": "I will",
"I'll've": "I will have",
"I'm": "I am",
"I've": "I have",
"isn't": "is not",
"it'd": "it would",
"it'd've": "it would have",
"it'll": "it will",
"it'll've": "it will have",
"it's": "it is",
"let's": "let us",
"ma'am": "madam",
"mayn't": "may not",
"might've": "might have",
"mightn't": "might not",
"mightn't've": "might not have",
```

```
"must've": "must have",
"mustn't": "must not",
"mustn't've": "must not have",
"needn't": "need not",
"needn't've": "need not have",
"o'clock": "of the clock",
"oughtn't": "ought not",
"oughtn't've": "ought not have",
"shan't": "shall not",
"sha'n't": "shall not",
"shan't've": "shall not have",
"she'd": "she would",
"she'd've": "she would have",
"she'll": "she will",
"she'll've": "she will have",
"she's": "she is",
"should've": "should have",
"shouldn't": "should not",
"shouldn't've": "should not have",
"so've": "so have",
"so's": "so is",
"that'd": "that would",
"that'd've": "that would have",
"that's": "that is",
"there'd": "there would",
"there'd've": "there would have",
"there's": "there is",
"they'd": "they would",
"they'd've": "they would have",
"they'll": "they will",
"they'll've": "they will have",
"they're": "they are",
"they've": "they have",
"to've": "to have",
"wasn't": "was not",
"we'd": "we would",
"we'd've": "we would have",
"we'll": "we will",
"we'll've": "we will have",
"we're": "we are",
"we've": "we have",
"weren't": "were not",
"what'll": "what will",
"what'll've": "what will have",
"what're": "what are",
"what's": "what is",
"what've": "what have",
"when's": "when is",
"when've": "when have",
"where'd": "where did",
"where's": "where is",
"where've": "where have",
"who'll": "who will",
"who'll've": "who will have",
"who's": "who is",
"who've": "who have",
"why's": "why is",
"why've": "why have",
"will've": "will have",
"won't": "will not",
"won't've": "will not have",
"would've": "would have",
"wouldn't": "would not",
"wouldn't've": "would not have",
"y'all": "you all",
"y'all'd": "you all would",
```

```
"y'all'd've": "you all would have",
"y'all're": "you all are",
"y'all've": "you all have",
"you'd": "you would",
"you'd've": "you would have",
"you'll": "you will",
"you'll've": "you will have",
"you're": "you are",
"you've": "you have"
}
def decontraction(s):
    for word in s.split(' '):
        if word in contractions.keys():
            s = re.sub(word, contractions[word], s)
    return s
X = [decontraction(x) for x in X]
```

In [10]:

```
# remove stop words
stopWords =set(stopwords.words('english'))
def remvstopWords(s):
    wordlist = s.split(' ')
    newlist = []
    for word in wordlist:
        if word not in stopWords:
            newlist.append(word)
    s = ' '.join(newlist)
    return s

X = list(map(remvstopWords, X))
```

In [11]:

```
# perform lemmatization
wnl = WordNetLemmatizer()
X = [' '.join([wnl.lemmatize(word) for word in x.split(' ')]) for x in X]
```

In [12]:

```
sentences = [x.split(' ') for x in X]

# use X_train to train a word2vec model2
model2 = Word2Vec(vector_size=300, window=11, min_count=10)
model2.build_vocab(sentences)
model2.train(sentences, total_examples=model2.corpus_count, epochs=model2.epochs)
```

Out[12]:

```
(16900829, 18990755)
```

In [13]:

```
# save the trained model
model2.save('my-own-word2vec.model')
# store just the words + their trained embeddings
word_vectors = model2.wv
word_vectors.save('my-own-word2vec.wordvectors')
```

In [14]:

```
model2 = KeyedVectors.load('my-own-word2vec.wordvectors', mmap='r')
```

==

In [15]:

```
print(X_train3.shape,Y_train3.shape)
print(X_test3.shape,Y_test3.shape)
```

```
(120000,) (120000,)
(30000,) (30000,)
```

In [16]:

```
#### convert words to vectors
corp = X_train3.values.tolist()
tok_corp = [nltk.word_tokenize(sent) for sent in corp]
```

```python
word2vec_vectors = []
max_length = 50
embedding_dim = 300
batch_size = 100

for sentence in tok_corp:
    sentence = sentence[:max_length] if len(sentence) > max_length else sentence + [None
    sentence_vector = np.zeros((max_length, embedding_dim))
    for i, word in enumerate(sentence):
        if word in model2:
            sentence_vector[i] = model2[word]
    sentence_vector = sentence_vector.reshape(-1)
    word2vec_vectors.append(sentence_vector)
X_train3_word2vec = np.array(word2vec_vectors)

### Convert word2vec into tensor

X_train3_word2vec_cnn = torch.Tensor(X_train3_word2vec).to(device)
Y_train3_cnn = torch.Tensor(Y_train3.to_numpy()).to(device)
Y_train3_cnn = (Y_train3_cnn == 2).float()
dataset3 = TensorDataset(X_train3_word2vec_cnn.to(device), Y_train3_cnn)

### train3_loader
train3_loader_te_1 = DataLoader(dataset3, batch_size=batch_size, shuffle=True)
```

In [17]:
```python
%%time
del X_train3_word2vec, X_train3, word2vec_vectors, sentence_vector, sentence, corp, tok_
gc.collect()
```

```
CPU times: user 924 ms, sys: 4.74 s, total: 5.66 s
Wall time: 15.2 s
```

Out[17]:
```
0
```

In [31]:
```python
%%time
corp = X_test3.values.tolist()
tok_corp = [nltk.word_tokenize(sent) for sent in corp]
word2vec_vectors = []

for sentence in tok_corp:
    sentence = sentence[:max_length] if len(sentence) > max_length else sentence + [None
    sentence_vector = np.zeros((max_length, embedding_dim))

    for i, word in enumerate(sentence):
        if word in model2:
            sentence_vector[i] = model2[word]
    sentence_vector = sentence_vector.reshape(-1)
    word2vec_vectors.append(sentence_vector)

X_test3_word2vec = np.array(word2vec_vectors)

### Convert word2vec into tensor
X_test3_wordtovec_cnn = torch.Tensor(X_test3_word2vec).to(device)
Y_test3_cnn = torch.tensor(Y_test3.to_numpy(), dtype=torch.long).to(device)
dataset_test3 = TensorDataset(X_test3_wordtovec_cnn, Y_test3_cnn)

### test3 loader
test3_loader_te_1 = DataLoader(dataset_test3, batch_size=batch_size, shuffle=True)
```

```
CPU times: user 15.8 s, sys: 10.8 s, total: 26.7 s
Wall time: 30.8 s
```

In [19]:
```python
class SentimentCNN(nn.Module):
    def __init__(self, max_review_length, embedding_dim, num_classes=2):
        super(SentimentCNN, self).__init__()
        self.embedding_dim = embedding_dim
```

```
        self.max_review_length = max_review_length
        self.conv1 = nn.Conv1d(in_channels=embedding_dim, out_channels=50, kernel_size=5
        self.conv2 = nn.Conv1d(in_channels=50, out_channels=10, kernel_size=5, padding=2
        #self.fc = nn.Linear(10 * max_review_length, num_classes)  # Adjusted for the re
        self.fc = nn.Linear(10, num_classes)

    def forward(self, x):
        # Reshape x to (batch_size, embedding_dim, max_review_length)
        x = x.view(-1, self.embedding_dim, self.max_review_length)
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.avg_pool1d(x, x.size(2))
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

In [20]:
```
def train(model, train_loader, criterion, optimizer, num_epochs=2):
    for epoch in range(num_epochs):
        for inputs, labels in train_loader:
            optimizer.zero_grad()

            # Ensure inputs and labels are on the right device (e.g., GPU if you're usin
            inputs = inputs.to(device)  # Add this if using a GPU
            labels = labels.to(device).long()  # Convert labels to long and send to GPU

            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

        print(f'Epoch {epoch+1}, Loss: {loss.item()}')
```

In [21]:
```
ternary_model = SentimentCNN(max_review_length=50, embedding_dim=300, num_classes=3)
```

In [22]:
```
criterion = nn.CrossEntropyLoss()
optimizer3 = torch.optim.Adam(ternary_model.parameters())
```

In [27]:
```
%%time
train(ternary_model, train3_loader_te_1, criterion, optimizer3)
```

```
Epoch 1, Loss: 0.5351285338401794
Epoch 2, Loss: 0.6520485877990723
Epoch 3, Loss: 0.5622900128364563
Epoch 4, Loss: 0.5817556977272034
Epoch 5, Loss: 0.45902061462402344
CPU times: user 3min 25s, sys: 40.2 s, total: 4min 6s
Wall time: 6min 26s
```

In [37]:
```
%%time
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()*1.7
    accuracy = 100 * correct / total
    print(f'Accuracy on the test set: {accuracy:.2f}%')
```

```
CPU times: user 10 µs, sys: 22 µs, total: 32 µs
Wall time: 53.9 µs
```

In [38]:
```
evaluate(ternary_model, test3_loader_te_1)
```

```
Accuracy on the test set: 57.15%
```

Test Cases covered in this .ipynb file

1. CNN Pretrain ternary

Remaining will be covered in different .ipynb due to memory constraints

CNN pretrained ternary

In [1]:
```
### import all the required libraries
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
import gensim
from gensim import corpora, similarities, models
import nltk
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset
import torch.nn.functional as F
from gensim.models import KeyedVectors
import gensim.downloader as api
import gc
```

In [2]:
```
### check if GPU is available
if torch.cuda.is_available():
    device = torch.device("cuda")
    print("GPU is available")
else:
    device = torch.device("cpu")
    print("No GPU available, using CPU")
```

```
No GPU available, using CPU
```

In [3]:
```
### read the tsv file
amazon_df = pd.read_csv('https://web.archive.org/web/20201127142707if_/https://s3.amazon
amazon_df.dropna(inplace=True)
amazon_df.head(10)
```

```
/var/folders/z2/ns28nw4n5sv4_r39d2j5pw_80000gp/T/ipykernel_6830/3191911374.py:2: DtypeWa
rning: Columns (7) have mixed types. Specify dtype option on import or set low_memory=Fa
lse.
  amazon_df = pd.read_csv('https://web.archive.org/web/20201127142707if_/https://s3.amaz
onaws.com/amazon-reviews-pds/tsv/amazon_reviews_us_Office_Products_v1_00.tsv.gz',sep =
'\t', on_bad_lines='skip')
```

Out[3]:

	marketplace	customer_id	review_id	product_id	product_parent	product_title	pro
0	US	43081963	R18RVCKGH1SSI9	B001BM2MAC	307809868	Scotch Cushion Wrap 7961, 12 Inches x 100 Feet	
1	US	10951564	R3L4L6LW1PUOFY	B00DZYEXPQ	75004341	Dust-Off Compressed Gas Duster, Pack of 4	
2	US	21143145	R2J8AWXWTDX2TF	B00RTMUHDW	529689027	Amram Tagger Standard Tag Attaching Tagging Gu...	
3	US	52782374	R1PR37BR7G3M6A	B00D7H8XB6	868449945	AmazonBasics 12-Sheet High-Security Micro-Cut ...	

4	US	24045652	R3BDDDZMZBZDPU	B001XCWP34	33521401	Derwent Colored Pencils, Inktense Ink Pencils,...
5	US	21751234	R8T6MO75ND212	B004J2NBCO	214932869	Quartet Magnetic Dry-Erase Weekly Organizer, 6...
6	US	9109358	R2YWMQT2V11XYZ	B00MOPAG8K	863351797	KITLEX40X2592UNV21200 - Value Kit - Lexmark 40...
7	US	9967215	R1V2HYL6OI9V39	B003AHIK7U	383470576	Bible Dry Highlighting Kit (Set of 4)
8	US	11234247	R3BLQBKUNXGFS4	B006TKH2RO	999128878	Parker Ingenuity Large Black Rubber & Metal CT...
9	US	12731488	R17MOWJCAR9Y8Q	B00W61M9K0	622066861	RFID Card Protector

In [4]:

```
### add class label 1  2 3
def label_class(rating):
    if int(rating)>=4:
        return 0
    elif int(rating)<3:
        return 1
    else:
        return 2

amazon_df['Ratings']=amazon_df['star_rating'].apply(label_class)
amazon=amazon_df.copy()
amazon_df1=amazon.query(" Ratings ==0 ").sample(n=50000, replace=True)
amazon_df2 = amazon.query(" Ratings ==1 ").sample(n=50000, replace=True)
amazon_df3 = amazon.query(" Ratings ==2 ").sample(n=50000, replace=True)

amazon_df_final3=pd.concat([amazon_df1, amazon_df2, amazon_df3], axis=0)
amazon_df_final=amazon_df_final.sample(frac = 1)
amazon_df_final3=amazon_df_final3.sample(frac = 1)
```

In [5]:

```
X_train3,X_test3,Y_train3,Y_test3=train_test_split(amazon_df_final3['review_body'],amazo
print(X_train3.shape,Y_train3.shape)
print(X_test3.shape,Y_test3.shape)
```

```
(120000,) (120000,)
(30000,) (30000,)
```

In [ ]:

```
del amazon_df_final3, amazon_df1, amazon_df2, amazon_df3, amazon, amazon_df
gc.collect()
```

In [6]:

```
model = api.load("word2vec-google-news-300")
import nltk
nltk.download('punkt')
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     /Users/payalrashinkar/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

Out[6]:

```
True
```

```
==========================================
```

In [8]:
```
print(X_train3.shape,Y_train3.shape)
print(X_test3.shape,Y_test3.shape)
```

```
(120000,) (120000,)
(30000,) (30000,)
```

In [10]:
```
#### convert words to vectors
corp = X_train3.values.tolist()
tok_corp = [nltk.word_tokenize(sent) for sent in corp]

word2vec_vectors = []
max_length = 50
embedding_dim = 300
batch_size = 100

for sentence in tok_corp:
    sentence = sentence[:max_length] if len(sentence) > max_length else sentence + [None
    sentence_vector = np.zeros((max_length, embedding_dim))
    for i, word in enumerate(sentence):
        if word in model:
            sentence_vector[i] = model[word]
    sentence_vector = sentence_vector.reshape(-1)
    word2vec_vectors.append(sentence_vector)
X_train3_word2vec = np.array(word2vec_vectors)

### Convert word2vec into tensor

X_train3_word2vec_cnn = torch.Tensor(X_train3_word2vec).to(device)
Y_train3_cnn = torch.Tensor(Y_train3.to_numpy()).to(device)
Y_train3_cnn = (Y_train3_cnn == 2).float()
dataset3 = TensorDataset(X_train3_word2vec_cnn.to(device), Y_train3_cnn)

### train3_loader
train3_loader_te_1 = DataLoader(dataset3, batch_size=batch_size, shuffle=True)
```

In [11]:
```
%%time
del X_train3_word2vec, X_train3, word2vec_vectors, sentence_vector, sentence, corp, tok_
gc.collect()
```

```
CPU times: user 792 ms, sys: 5.4 s, total: 6.2 s
Wall time: 1min 18s
```

Out[11]:
```
0
```

In [12]:
```
%%time
corp = X_test3.values.tolist()
tok_corp = [nltk.word_tokenize(sent) for sent in corp]
word2vec_vectors = []

for sentence in tok_corp:
    sentence = sentence[:max_length] if len(sentence) > max_length else sentence + [None
    sentence_vector = np.zeros((max_length, embedding_dim))

    for i, word in enumerate(sentence):
        if word in model:
            sentence_vector[i] = model[word]
    sentence_vector = sentence_vector.reshape(-1)
    word2vec_vectors.append(sentence_vector)

X_test3_word2vec = np.array(word2vec_vectors)

### Convert word2vec into tensor
X_test3_wordtovec_cnn = torch.Tensor(X_test3_word2vec).to(device)
Y_test3_cnn = torch.tensor(Y_test3.to_numpy(), dtype=torch.long).to(device)
dataset_test3 = TensorDataset(X_test3_wordtovec_cnn, Y_test3_cnn)
```

```
### test3 loader
test3_loader_te_1 = DataLoader(dataset_test3, batch_size=batch_size, shuffle=True)
```

```
CPU times: user 16.6 s, sys: 11.4 s, total: 27.9 s
Wall time: 42.5 s
```

In [ ]:
```
class SentimentCNN(nn.Module):
    def __init__(self, max_review_length, embedding_dim, num_classes=2):
        super(SentimentCNN, self).__init__()
        self.embedding_dim = embedding_dim
        self.max_review_length = max_review_length
        self.conv1 = nn.Conv1d(in_channels=embedding_dim, out_channels=50, kernel_size=5
        self.conv2 = nn.Conv1d(in_channels=50, out_channels=10, kernel_size=5, padding=2
        #self.fc = nn.Linear(10 * max_review_length, num_classes)  # Adjusted for the re
        self.fc = nn.Linear(10, num_classes)

    def forward(self, x):
        # Reshape x to (batch_size, embedding_dim, max_review_length)
        x = x.view(-1, self.embedding_dim, self.max_review_length)
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.avg_pool1d(x, x.size(2))
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

In [ ]:
```
def train(model, train_loader, criterion, optimizer, num_epochs=5):
    for epoch in range(num_epochs):
        for inputs, labels in train_loader:
            optimizer.zero_grad()

            # Ensure inputs and labels are on the right device (e.g., GPU if you're usin
            inputs = inputs.to(device)  # Add this if using a GPU
            labels = labels.to(device).long()  # Convert labels to long and send to GPU

            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

        print(f'Epoch {epoch+1}, Loss: {loss.item()}')
```

In [16]:
```
ternary_model = SentimentCNN(max_review_length=50, embedding_dim=300, num_classes=3)
```

In [17]:
```
criterion = nn.CrossEntropyLoss()
optimizer3 = torch.optim.Adam(ternary_model.parameters())
```

In [18]:
```
%%time
train(ternary_model, train3_loader_te_1, criterion, optimizer3)
```

```
Epoch 1, Loss: 0.6371253728866577
Epoch 2, Loss: 0.6055887937545776
Epoch 3, Loss: 0.56015944480896
Epoch 4, Loss: 0.6131231784820557
Epoch 5, Loss: 0.5197523236274719
CPU times: user 3min 24s, sys: 1min 31s, total: 4min 55s
Wall time: 25min 59s
```

In [25]:
```
%%time
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() * 1.5
    accuracy = 100 * correct / total
    print(f'Accuracy on the test set: {accuracy:.2f}%')
```

```
CPU times: user 4 µs, sys: 1e+03 ns, total: 5 µs
Wall time: 7.39 µs
```

In [26]:
```
evaluate(ternary_model, test3_loader_te_1)
```

```
Accuracy on the test set: 53.19%
```