# Importing Libraries

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
pd.options.mode.chained_assignment = None
```

In [2]:
```
import warnings
warnings.filterwarnings('ignore')
```

# Reading the data ¶

In [3]:
```
df = pd.read_csv("amazon_reviews_us_Office_Products_v1_00.tsv", sep='\t', on_bad_lines='skip')
df = df[['review_body', 'star_rating']].dropna().reset_index(drop=True)
df['star_rating'] = pd.to_numeric(df['star_rating'], errors='coerce')
```

In [4]:
```
# Reference - https://stackoverflow.com/questions/71758460/effect-of-pandas-dataframe-sample-with-frac-set-to-1

per_rating_instance = 50000

def balanced_dataset(group):
    return group.sample(per_rating_instance, random_state = 30, replace = True)

balanced_df = df.groupby('star_rating', group_keys=False).apply(balanced_dataset)
balanced_df = balanced_df.sample(frac = 1)
```

In [5]:
```
def sentiment_class(rating):
    if rating > 3:
        return 1
    elif rating <= 2:
        return 0
    else:
        return 2


balanced_df['sentiment'] = balanced_df['star_rating'].apply(sentiment_class)
balanced_df
```

Out[5]:

| | review_body | star_rating | sentiment |
|---|---|---|---|
| **1760505** | Good functionality but the battery failed afte... | 3.0 | 2 |
| **1530925** | have to manually change over to next month at ... | 3.0 | 2 |
| **1297405** | Does what expected for the price. If your look... | 3.0 | 2 |
| **1239759** | Well made and fits on book well but be careful... | 2.0 | 0 |
| **1058624** | It doesn't work well with my cutter...I have i... | 2.0 | 0 |
| **...** | ... | ... | ... |
| **759738** | This is the best $13 bucks I have spent in a l... | 5.0 | 1 |
| **2541069** | I've had many printers for my home transcripti... | 1.0 | 0 |
| **1524004** | THE PHONES TAKE A SECOND TO RING WHEN PLACING ... | 2.0 | 0 |
| **2555300** | Have owned for about 3 months now - Performing... | 5.0 | 1 |
| **855760** | this printer is not even 2.5 months old and th... | 2.0 | 0 |

250000 rows × 3 columns

In [6]:
```
sentiment_counts = balanced_df['sentiment'].value_counts()

print("Count of positive sentiments (Class 1):", sentiment_counts[1])
print("Count of negative sentiments (Class 2):", sentiment_counts[0])
print("Count of neutral sentiments (Class 3):", sentiment_counts[2])
```

```
Count of positive sentiments (Class 1): 100000
Count of negative sentiments (Class 2): 100000
Count of neutral sentiments (Class 3): 50000
```

# Data Cleaning

In [7]:
```
import re
from bs4 import BeautifulSoup

def expand_contractions(s):
    contraction_patterns = {
        r"won't": "will not",
        r"would't": "would not",
        r"could'nt": "could not",
        r"can't": "can not",
        r"n't": " not",
        r"\'re": " are",
        r"\'s": " is",
        r"\'ll": " will",
        r"\'t": " not",
        r"\'ve": " have",
        r"I've": "I have",
        r"I'm": "I am"
    }

    for pattern, replacement in contraction_patterns.items():
        s = re.sub(pattern, replacement, s)

    return s


balanced_df['cleaned_reviews'] = (
    balanced_df['review_body']
    .str.lower()
    .apply(lambda x: BeautifulSoup(x, 'html.parser').get_text())
    .apply(lambda x: re.sub(r'https?://\S+', '', x))
    .apply(expand_contractions)
    .apply(lambda x: re.sub(r'[^a-zA-Z\s]', '', x))
    .apply(lambda x: x.strip())
)

balanced_df
```

Out[7]:

| | review_body | star_rating | sentiment | cleaned_reviews |
|---|---|---|---|---|
| **1760505** | Good functionality but the battery failed afte... | 3.0 | 2 | good functionality but the battery failed afte... |
| **1530925** | have to manually change over to next month at ... | 3.0 | 2 | have to manually change over to next month at ... |
| **1297405** | Does what expected for the price. If your look... | 3.0 | 2 | does what expected for the price if your looki... |
| **1239759** | Well made and fits on book well but be careful... | 2.0 | 0 | well made and fits on book well but be careful... |
| **1058624** | It doesn't work well with my cutter...I have i... | 2.0 | 0 | it does not work well with my cutteri have inc... |
| **...** | ... | ... | ... | ... |
| **759738** | This is the best $13 bucks I have spent in a l... | 5.0 | 1 | this is the best bucks i have spent in a long... |
| **2541069** | I've had many printers for my home transcripti... | 1.0 | 0 | i have had many printers for my home transcrip... |
| **1524004** | THE PHONES TAKE A SECOND TO RING WHEN PLACING ... | 2.0 | 0 | the phones take a second to ring when placing ... |
| **2555300** | Have owned for about 3 months now - Performing... | 5.0 | 1 | have owned for about months now performing f... |
| **855760** | this printer is not even 2.5 months old and th... | 2.0 | 0 | this printer is not even months old and the p... |

250000 rows × 4 columns

# Data Pre-Processing

In [8]:
```
om nltk.corpus import stopwords
stom_stopwords_list = set(stopwords.words('english')) - {'not', 'no', 'nor', 'neither', 'but', 'however', 'although'}
lanced_df['cleaned_reviews'] = balanced_df['cleaned_reviews'].apply(lambda x: " ".join([word for word in x.split() if word
lanced_df
```

Out[8]:

| | review_body | star_rating | sentiment | cleaned_reviews |
|---|---|---|---|---|
| **1760505** | Good functionality but the battery failed afte... | 3.0 | 2 | good functionality but battery failed two week... |
| **1530925** | have to manually change over to next month at ... | 3.0 | 2 | manually change next month end every monthwont... |
| **1297405** | Does what expected for the price. If your look... | 3.0 | 2 | expected price looking home theatre quality pa... |
| **1239759** | Well made and fits on book well but be careful... | 2.0 | 0 | well made fits book well but careful photos cl... |
| **1058624** | It doesn't work well with my cutter...I have i... | 2.0 | 0 | not work well cutteri increased settings incre... |
| **...** | ... | ... | ... | ... |
| **759738** | This is the best $13 bucks I have spent in a l... | 5.0 | 1 | best bucks spent long time came promised cute ... |
| **2541069** | I've had many printers for my home transcripti... | 1.0 | 0 | many printers home transcription business far ... |
| **1524004** | THE PHONES TAKE A SECOND TO RING WHEN PLACING ... | 2.0 | 0 | phones take second ring placing call not total... |
| **2555300** | Have owned for about 3 months now - Performing... | 5.0 | 1 | owned months performing flawlessly think softw... |
| **855760** | this printer is not even 2.5 months old and th... | 2.0 | 0 | printer not even months old printer head jamme... |

250000 rows × 4 columns

In [9]:
```
import nltk
nltk.download('omw-1.4')

from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
balanced_df['cleaned_reviews']=balanced_df['cleaned_reviews'].apply(lambda x: " ".join([lemmatizer.lemmatize(w) for w in n
balanced_df
```

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

Out[9]:

| | review_body | star_rating | sentiment | cleaned_reviews |
|---|---|---|---|---|
| **1760505** | Good functionality but the battery failed afte... | 3.0 | 2 | good functionality but battery failed two week... |
| **1530925** | have to manually change over to next month at ... | 3.0 | 2 | manually change next month end every monthwont... |
| **1297405** | Does what expected for the price. If your look... | 3.0 | 2 | expected price looking home theatre quality pa... |
| **1239759** | Well made and fits on book well but be careful... | 2.0 | 0 | well made fit book well but careful photo clea... |
| **1058624** | It doesn't work well with my cutter...I have i... | 2.0 | 0 | not work well cutteri increased setting increa... |
| **...** | ... | ... | ... | ... |
| **759738** | This is the best $13 bucks I have spent in a l... | 5.0 | 1 | best buck spent long time came promised cute w... |
| **2541069** | I've had many printers for my home transcripti... | 1.0 | 0 | many printer home transcription business far w... |
| **1524004** | THE PHONES TAKE A SECOND TO RING WHEN PLACING ... | 2.0 | 0 | phone take second ring placing call not totall... |
| **2555300** | Have owned for about 3 months now - Performing... | 5.0 | 1 | owned month performing flawlessly think softwa... |
| **855760** | this printer is not even 2.5 months old and th... | 2.0 | 0 | printer not even month old printer head jammed... |

250000 rows × 4 columns

# 1. wv = word2vec-google-news-300

# 2. word2vec_model = my own model

## (a)

In [10]:
```
import gensim.downloader as api
wv = api.load('word2vec-google-news-300')
```

In [11]:
```
from nltk.tokenize import word_tokenize

df_review_text = balanced_df["cleaned_reviews"]
df_review_text = list(df_review_text)
df_review_text = [' '.join(text.split()) for text in df_review_text]

# Tokenize the text to words
df_review_text_tokenized = [word_tokenize(text) for text in df_review_text]
df_review_text_tokenized[1]
```

Out[11]:
```
['manually',
 'change',
 'next',
 'month',
 'end',
 'every',
 'monthwont',
 'automaticallydont',
 'know',
 'whyprints',
 'lightlydoes',
 'help',
 'keep',
 'track',
 'employee',
 'hour',
 'better']
```

In [12]:
```
similarity_score_1 = wv.most_similar(positive=['woman', 'king'], negative=['man'])[0][1]
print(f"Semantic similarity score for 'King - Man + Woman = Queen': {similarity_score_1}")
```

```
Semantic similarity score for 'King - Man + Woman = Queen': 0.7118192911148071
```

In [13]:
```
similarity_score_2 = wv.similarity('outstanding', 'excellent')
print(f"Semantic similarity score for 'Outstanding ~ Excellent': {similarity_score_2}")
```

```
Semantic similarity score for 'Outstanding ~ Excellent': 0.556748628616333
```

In [14]:
```
wv.most_similar(positive=['king', 'woman'], negative=['man'], topn=1)
```

Out[14]:
```
[('queen', 0.7118192911148071)]
```

In [15]:
```
similar_words = wv.most_similar('excellent', topn=5)

print(f"Words most similar to 'excellent':")
for word, similarity in similar_words:
    print(f"{word}: {similarity}")
```

```
Words most similar to 'excellent':
terrific: 0.7409728169441223
superb: 0.7062716484069824
exceptional: 0.681470513343811
fantastic: 0.6802847385406494
good: 0.644292950630188
```

In [*]:
```
wv.most_similar(positive=['bill'], topn=1)
```

# (b)

In [16]:
```
from gensim.models import Word2Vec

tokenized_reviews = [review.split() for review in balanced_df['review_body']]

embedding_size = 300
window_size = 11
min_word_count = 10

word2vec_model = Word2Vec(
    sentences=tokenized_reviews,
    vector_size=embedding_size,
    window=window_size,
    min_count=min_word_count,
    workers=4
)
```

In [17]:
```
word2vec_model.wv.most_similar(positive=['king', 'woman'], negative=['man'], topn=1)
```

Out[17]:
```
[('$3,', 0.36969098448753357)]
```

In [18]:
```
similar_words = word2vec_model.wv.most_similar('excellent', topn=5)

print(f"Words most similar to 'excellent':")
for word, similarity in similar_words:
    print(f"{word}: {similarity}")
```

```
Words most similar to 'excellent':
outstanding: 0.8158041834831238
amazing: 0.7264388203620911
exceptional: 0.7083240747451782
awesome: 0.6833122372627258
incredible: 0.6757016181945801
```

In [*]:
```
my_model.wv.most_similar(positive=['bill'], topn=1)
```

## Remarks

- When it comes to recognising broader similarities and general word associations, the Google pretrained model performs better.
- Specifically, when computing "king + woman - man," the Google model yields the result "queen," whereas our own model proposes the result "," demonstrating the influence of the corresponding training datasets. Because it was trained on review data, the custom model has a tendency to give context-specific connections more weight.
- Likewise, when it comes to the word "excellent," our model offers "outstanding," whereas the Google model suggests "terrific," which reflects the contextual variations present in their training data.
- In conclusion, different datasets were used to train each model, which explains why the results of similarity checks sometimes differ. This illustrates how word usage and relationships can change.

# Simple Models

In [19]:
```
balanced_df["reviews_cleaned_and_tokenized"] = df_review_text_tokenized
balanced_df = balanced_df[balanced_df['sentiment'] != 2]
balanced_df
```

Out[19]:

| | review_body | star_rating | sentiment | cleaned_reviews | reviews_cleaned_and_tokenized |
|---|---|---|---|---|---|
| **1239759** | Well made and fits on book well but be careful... | 2.0 | 0 | well made fit book well but careful photo clea... | [well, made, fit, book, well, but, careful, ph... |
| **1058624** | It doesn't work well with my cutter...I have i... | 2.0 | 0 | not work well cutteri increased setting increa... | [not, work, well, cutteri, increased, setting,... |
| **722262** | Good Quality, | 4.0 | 1 | good quality | [good, quality] |
| **1945003** | I messed up on my ordering and did not get my ... | 5.0 | 1 | messed ordering not get color choice submitted... | [messed, ordering, not, get, color, choice, su... |
| **1955455** | Do not buy this piece of crap mine did not eve... | 1.0 | 0 | not buy piece crap mine not even power really ... | [not, buy, piece, crap, mine, not, even, power... |
| **...** | ... | ... | ... | ... | ... |
| **759738** | This is the best $13 bucks I have spent in a l... | 5.0 | 1 | best buck spent long time came promised cute w... | [best, buck, spent, long, time, came, promised... |
| **2541069** | I've had many printers for my home transcripti... | 1.0 | 0 | many printer home transcription business far w... | [many, printer, home, transcription, business,... |
| **1524004** | THE PHONES TAKE A SECOND TO RING WHEN PLACING ... | 2.0 | 0 | phone take second ring placing call not totall... | [phone, take, second, ring, placing, call, not... |
| **2555300** | Have owned for about 3 months now - Performing... | 5.0 | 1 | owned month performing flawlessly think softwa... | [owned, month, performing, flawlessly, think, ... |
| **855760** | this printer is not even 2.5 months old and th... | 2.0 | 0 | printer not even month old printer head jammed... | [printer, not, even, month, old, printer, head... |

200000 rows × 5 columns

In [20]:
```
def reviews_with_avg(df, model):
    vectorized_data_temp = []

    for review_body in df['review_body']:
        tokens = review_body.split()
        weights = [model[token] for token in tokens if token in model]

        average_weight = np.mean(weights, axis=0) if weights else np.zeros(300, dtype=float)

        vectorized_data_temp.append(average_weight)

    return np.array(vectorized_data_temp)
```

In [21]:
```
print("Accuracy for Perceptron Testing set (HW - 1): 86.98%")
print("Accuracy for SVM Testing set (HW - 1): 90.88%")
```

```
Accuracy for Perceptron Testing set (HW - 1): 86.98%
Accuracy for SVM Testing set (HW - 1): 90.88%
```

# Train - Test Split - Pretrained Model

In [22]:
```
from sklearn.model_selection import train_test_split

vectorized_data = reviews_with_avg(balanced_df, wv)

X_train, X_test, y_train, y_test = train_test_split(
    vectorized_data, balanced_df['sentiment'] ,test_size=0.2, random_state=42
)
```

# Perceptron

In [23]:
```
from sklearn.metrics import accuracy_score

from sklearn.linear_model import Perceptron

perceptron_model = Perceptron()
perceptron_model.fit(X_train, y_train)
perceptron_predictions = perceptron_model.predict(X_test)
perceptron_accuracy = accuracy_score(y_test, perceptron_predictions)

print("Perceptron Accuracy:", perceptron_accuracy)
```

```
Perceptron Accuracy: 0.574075
```

# SVM

In [24]:
```
from sklearn.svm import LinearSVC

linear_svc_model = LinearSVC()
linear_svc_model.fit(X_train, y_train)
linear_svc_predictions = linear_svc_model.predict(X_test)
linear_svc_accuracy = accuracy_score(y_test, linear_svc_predictions)

print("LinearSVC Accuracy:", linear_svc_accuracy)
```

```
LinearSVC Accuracy: 0.814
```

# Train - Test Split - own Model

In [25]:
```
from sklearn.model_selection import train_test_split

vectorized_data = reviews_with_avg(balanced_df, word2vec_model.wv)

X_train, X_test, y_train, y_test = train_test_split(
    vectorized_data, balanced_df['sentiment'] ,test_size=0.2, random_state=42
)
```

# Perceptron

In [26]:
```
from sklearn.linear_model import Perceptron

perceptron_model = Perceptron()
perceptron_model.fit(X_train, y_train)
perceptron_predictions = perceptron_model.predict(X_test)
perceptron_accuracy = accuracy_score(y_test, perceptron_predictions)

print("Perceptron Accuracy:", perceptron_accuracy)
```

```
Perceptron Accuracy: 0.72735
```

# SVM

In [27]:
```
from sklearn.svm import LinearSVC

linear_svc_model = LinearSVC()
linear_svc_model.fit(X_train, y_train)
linear_svc_predictions = linear_svc_model.predict(X_test)
linear_svc_accuracy = accuracy_score(y_test, linear_svc_predictions)

print("LinearSVC Accuracy:", linear_svc_accuracy)
```

```
LinearSVC Accuracy: 0.86505
```

## Remarks

### Accuracy:

- TF-IDF
  - Perceptron - 86.98% (from HW1)
  - SVM - 90.88% (from HW1)
- Google pre-trained model
  - Perceptron - 57.4%
  - SVM - 81.4%
- Own Trained model
  - Perceptron - 72.73%
  - SVM - 86.5%

## Conclusion

- From the above statistics, it is evident that TF-IDF performs better than Google's pre-trained model as well as our own trained model. The reason behind these statistics could be because TF-IDF takes a more intuitive approach, looking at how many times a word appears in general, in how many of the documents it appears, and how many times. As many reviews have similar words, hence TF-IDF performs better than word2vec models.
- When we compare both the word2vec models, it is observed that our custom-trained model outperforms Google's pre-trained model as it better understands similarities in our dataset compared to the Google model trained on the news dataset.

In sentiment classification tasks, TF-IDF is still quite effective, outperforming both custom-trained Word2Vec models and Google pre-trained models in terms of accuracy. Although the pre-trained Word2Vec model from Google is convenient and captures broad word associations, when applied to particular domains, its sentiment classification performance is not as good as that of TF-IDF-based methods and custom-trained models. It is important to customise models to the job and dataset at hand, as demonstrated by the fact that custom training a Word2Vec model on domain-specific data can result in increased sentiment classification accuracy. In general, the trade-off between generalizability, convenience, and task-specific performance requirements determines which model is best. For sentiment classification tasks, TF-IDF combined with conventional machine learning methods is still a dependable option. However, well-trained Word2Vec models

# FeedForward Neural Networks

In [28]:
```
import torch
from torch.utils.data import DataLoader, Dataset
from torch.utils.data.sampler import SubsetRandomSampler
import torch.nn as nn
import torch.nn.functional as F
```

In [29]:
```
class TrainData(Dataset):

    def __init__(self, vectorized_vector, class_labels):
        self.data = vectorized_vector
        self.class_labels = class_labels

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

    def __getitem__(self, index):

        review_word_vector = torch.from_numpy(np.array(self.data[index]))
        label = self.class_labels.iloc[index]

        return review_word_vector, label
```

In [30]:
```
class TestData(Dataset):

    def __init__(self, vectorized_vector):
        self.data = vectorized_vector

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

    def __getitem__(self, index):
        review_word_vector = torch.from_numpy(np.array(self.data[index]))

        return review_word_vector
```

In [31]:
```
def predict(model, dataloader):
    prediction_list = []
    for i, batch in enumerate(dataloader):
        batch = batch.double()
        batch = batch.to(device)
        outputs = model(batch)
        _, predicted = torch.max(outputs.data, 1)
        prediction_list.append(predicted.cpu().tolist())
    return prediction_list
```

# Binary Classification - word2vec

In [32]:
```
# Reference - # https://www.kaggle.com/code/mishra1993/pytorch-multi-layer-perceptron-mnist/notebook

class MLP_Binary(nn.Module):
    def __init__(self):
        super(MLP_Binary, self).__init__()

        hidden_1 = 50
        hidden_2 = 10

        self.fc1 = nn.Linear(300, hidden_1)
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        self.fc3 = nn.Linear(hidden_2, 2)

        self.dropout = nn.Dropout(0.2)

        self.double()

    def forward(self, x):

        x = F.relu(self.fc1(x))
        x = self.dropout(x)

        x = F.relu(self.fc2(x))
        x = self.dropout(x)

        x = self.fc3(x)

        return x
```

In [33]:
```
vectorized_data = reviews_with_avg(balanced_df, word2vec_model.wv)
```

In [34]:
```
X_train,X_test,Y_train, Y_test = train_test_split(vectorized_data, balanced_df['sentiment'], test_size=0.2, random_state=6
```

In [35]:
```
train_data = TrainData(X_train, Y_train)
test_data = TestData(X_test)
```

In [36]:
```
batch_size = 100
valid_size = 0.2

num_train = len(train_data)
indices = list(range(num_train))

np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]

train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

train_loader = DataLoader(train_data, batch_size=batch_size, sampler=train_sampler)
valid_loader = DataLoader(train_data, batch_size=batch_size, sampler=valid_sampler)
```

In [37]:
```
model = MLP_Binary()
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)
print(model)
```

```
MLP_Binary(
  (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 [38]:
```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
```

In [39]:
```
n_epochs = 30

valid_loss_min = np.Inf # set initial "min" to infinity

for epoch in range(n_epochs):
    train_loss = 0.0
    valid_loss = 0.0

    ###################
    # train the model #
    ###################
    model.train() # prep model for training
    for data, target in train_loader:
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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)

    ######################
    # validate the model #
    ######################
    model.eval() # prep model for evaluation
    for data, target in valid_loader:
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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(), 'MLP_Binary_mymodel.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1        Training Loss: 0.285600         Validation Loss: 0.063508
Validation loss decreased (inf --> 0.063508).  Saving model ...
Epoch: 2        Training Loss: 0.264577         Validation Loss: 0.062597
Validation loss decreased (0.063508 --> 0.062597).  Saving model ...
Epoch: 3        Training Loss: 0.257028         Validation Loss: 0.063304
Epoch: 4        Training Loss: 0.253132         Validation Loss: 0.063864
Epoch: 5        Training Loss: 0.250784         Validation Loss: 0.061391
Validation loss decreased (0.062597 --> 0.061391).  Saving model ...
Epoch: 6        Training Loss: 0.247651         Validation Loss: 0.062296
Epoch: 7        Training Loss: 0.245637         Validation Loss: 0.061189
Validation loss decreased (0.061391 --> 0.061189).  Saving model ...
Epoch: 8        Training Loss: 0.244118         Validation Loss: 0.062321
Epoch: 9        Training Loss: 0.241912         Validation Loss: 0.060150
Validation loss decreased (0.061189 --> 0.060150).  Saving model ...
Epoch: 10       Training Loss: 0.239269         Validation Loss: 0.061174
Epoch: 11       Training Loss: 0.237144         Validation Loss: 0.059362
Validation loss decreased (0.060150 --> 0.059362).  Saving model ...
Epoch: 12       Training Loss: 0.236180         Validation Loss: 0.059558
Epoch: 13       Training Loss: 0.234409         Validation Loss: 0.059952
Epoch: 14       Training Loss: 0.234758         Validation Loss: 0.060748
Epoch: 15       Training Loss: 0.232323         Validation Loss: 0.059949
Epoch: 16       Training Loss: 0.231885         Validation Loss: 0.061179
Epoch: 17       Training Loss: 0.229426         Validation Loss: 0.060842
Epoch: 18       Training Loss: 0.231158         Validation Loss: 0.060456
Epoch: 19       Training Loss: 0.230034         Validation Loss: 0.060284
Epoch: 20       Training Loss: 0.228465         Validation Loss: 0.060367
Epoch: 21       Training Loss: 0.227588         Validation Loss: 0.060567
Epoch: 22       Training Loss: 0.227756         Validation Loss: 0.061303
Epoch: 23       Training Loss: 0.226074         Validation Loss: 0.060708
Epoch: 24       Training Loss: 0.224857         Validation Loss: 0.060702
Epoch: 25       Training Loss: 0.225154         Validation Loss: 0.059973
Epoch: 26       Training Loss: 0.224137         Validation Loss: 0.059588
Epoch: 27       Training Loss: 0.223631         Validation Loss: 0.060859
Epoch: 28       Training Loss: 0.221843         Validation Loss: 0.060066
Epoch: 29       Training Loss: 0.222890         Validation Loss: 0.059631
Epoch: 30       Training Loss: 0.222322         Validation Loss: 0.060571
```

In [40]:
```
model.load_state_dict(torch.load('MLP_Binary_mymodel.pt'))
```

Out[40]:
```
<All keys matched successfully>
```

In [41]:
```
test_loader = DataLoader(test_data, batch_size=1,)
```

In [42]:
```
predictions = predict(model,test_loader)
predictions = np.array(predictions)
```

In [43]:
```
test_accuracy_mlp_binary_mymodel_avg = accuracy_score(Y_test, predictions)
print("Accuracy for Binary Classification using my own model is : ", test_accuracy_mlp_binary_mymodel_avg)
```

```
Accuracy for Binary Classification using Average of word vectors using my own model is :  0.87535
```

# Binary Classification - own model

In [44]:
```
vectorized_data = reviews_with_avg(balanced_df, wv)
X_train,X_test,Y_train, Y_test = train_test_split(vectorized_data, balanced_df['sentiment'], test_size=0.2, random_state=6)
```

In [45]:
```
train_data = TrainData(X_train, Y_train)
test_data = TestData(X_test)
```

In [46]:
```
# how many samples per batch to load
batch_size = 100
# percentage of training set to use as validation
valid_size = 0.2

# obtain training indices that will be used for validation
num_train = len(train_data)
indices = list(range(num_train))

np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]

# define samplers for obtaining training and validation batches
train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

# prepare data loaders
train_loader = DataLoader(train_data, batch_size=batch_size, sampler=train_sampler)
valid_loader = DataLoader(train_data, batch_size=batch_size, sampler=valid_sampler)
```

In [47]:
```
# initialize the NN
model = MLP_Binary()
model.to(device)
print(model)
```

```
MLP_Binary(
  (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 [48]:
```
# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()

# specify optimizer (Adam) and learning rate = 0.005
optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
```

In [49]:
```
# number of epochs to train the model
n_epochs = 30

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

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

    ###################
    # train the model #
    ###################
    model.train() # prep model for training
    for data, target in train_loader:
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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)

    ######################
    # validate the model #
    ######################
    model.eval() # prep model for evaluation
    for data, target in valid_loader:
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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(), 'MLP_Binary_google.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1 	Training Loss: 0.363464 	Validation Loss: 0.082277
Validation loss decreased (inf --> 0.082277).  Saving model ...
Epoch: 2 	Training Loss: 0.337878 	Validation Loss: 0.079364
Validation loss decreased (0.082277 --> 0.079364).  Saving model ...
Epoch: 3 	Training Loss: 0.328984 	Validation Loss: 0.079037
Validation loss decreased (0.079364 --> 0.079037).  Saving model ...
Epoch: 4 	Training Loss: 0.323759 	Validation Loss: 0.077542
Validation loss decreased (0.079037 --> 0.077542).  Saving model ...
Epoch: 5 	Training Loss: 0.318987 	Validation Loss: 0.077722
Epoch: 6 	Training Loss: 0.315109 	Validation Loss: 0.076162
Validation loss decreased (0.077542 --> 0.076162).  Saving model ...
Epoch: 7 	Training Loss: 0.313955 	Validation Loss: 0.076640
Epoch: 8 	Training Loss: 0.310462 	Validation Loss: 0.077157
Epoch: 9 	Training Loss: 0.308338 	Validation Loss: 0.076562
Epoch: 10 	Training Loss: 0.306493 	Validation Loss: 0.077538
Epoch: 11 	Training Loss: 0.304987 	Validation Loss: 0.074158
Validation loss decreased (0.076162 --> 0.074158).  Saving model ...
Epoch: 12 	Training Loss: 0.302323 	Validation Loss: 0.075420
Epoch: 13 	Training Loss: 0.302457 	Validation Loss: 0.075806
Epoch: 14 	Training Loss: 0.300782 	Validation Loss: 0.074490
Epoch: 15 	Training Loss: 0.299250 	Validation Loss: 0.074503
Epoch: 16 	Training Loss: 0.297543 	Validation Loss: 0.074600
Epoch: 17 	Training Loss: 0.296495 	Validation Loss: 0.076006
Epoch: 18 	Training Loss: 0.296147 	Validation Loss: 0.075564
Epoch: 19 	Training Loss: 0.294374 	Validation Loss: 0.076568
Epoch: 20 	Training Loss: 0.293859 	Validation Loss: 0.074175
Epoch: 21 	Training Loss: 0.292235 	Validation Loss: 0.074701
Epoch: 22 	Training Loss: 0.291742 	Validation Loss: 0.077182
Epoch: 23 	Training Loss: 0.289880 	Validation Loss: 0.076123
Epoch: 24 	Training Loss: 0.289984 	Validation Loss: 0.074187
Epoch: 25 	Training Loss: 0.289892 	Validation Loss: 0.074317
Epoch: 26 	Training Loss: 0.288767 	Validation Loss: 0.074503
Epoch: 27 	Training Loss: 0.289220 	Validation Loss: 0.075493
Epoch: 28 	Training Loss: 0.287926 	Validation Loss: 0.074708
Epoch: 29 	Training Loss: 0.286787 	Validation Loss: 0.074444
Epoch: 30 	Training Loss: 0.286030 	Validation Loss: 0.074708
```

In [50]:

```
model.load_state_dict(torch.load('MLP_Binary_google.pt'))
```

Out[50]:

```
<All keys matched successfully>
```

In [51]:

```
test_loader = DataLoader(test_data, batch_size=1,)
```

In [52]:
```
predictions = predict(model,test_loader)
predictions = np.array(predictions)
```

In [53]:
```
test_accuracy_mlp_binary_google_avg = accuracy_score(Y_test, predictions)
```

In [54]:
```
print("Accuracy for Binary Classification using google pretrained model is : ", test_accuracy_mlp_binary_google_avg)
```

```
Accuracy for Binary Classification using Average of word vectors using google pretrained model is :  0.83665
```

# Ternary - own model

In [55]:
```
class MLP_Ternary(nn.Module):
    def __init__(self):
        super(MLP_Ternary, self).__init__()

        hidden_1 = 50
        hidden_2 = 10

        self.fc1 = nn.Linear(300, hidden_1)
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        self.fc3 = nn.Linear(hidden_2, 3)

        self.dropout = nn.Dropout(0.2)

        self.double()

    def forward(self, x):

        x = F.relu(self.fc1(x))
        x = self.dropout(x)

        x = F.relu(self.fc2(x))
        x = self.dropout(x)

        x = self.fc3(x)

        return x
```

In [56]:
```
vectorized_data = reviews_with_avg(balanced_df, word2vec_model.wv)
```

In [57]:
```
X_train,X_test,Y_train, Y_test = train_test_split(vectorized_data, balanced_df['sentiment'], test_size=0.2, random_state=6
```

In [58]:
```
train_data = TrainData(X_train, Y_train)
test_data = TestData(X_test)
```

In [59]:
```
batch_size = 100
valid_size = 0.2

num_train = len(train_data)
indices = list(range(num_train))

np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]

train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

train_loader = DataLoader(train_data, batch_size=batch_size, sampler=train_sampler)
valid_loader = DataLoader(train_data, batch_size=batch_size, sampler=valid_sampler)
```

In [60]:
```
model = MLP_Ternary()
model = model.to(device)
print(model)
```

```
MLP_Ternary(
  (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 [61]:
```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
```

In [62]:
```
n_epochs = 30

valid_loss_min = np.Inf # set initial "min" to infinity

for epoch in range(n_epochs):
    train_loss = 0.0
    valid_loss = 0.0

    ###################
    # train the model #
    ###################
    model.train() # prep model for training
    for data, target in train_loader:
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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)

    ######################
    # validate the model #
    ######################
    model.eval() # prep model for evaluation
    for data, target in valid_loader:
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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(), 'MLP_Ternary_mymodel.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1        Training Loss: 0.295901         Validation Loss: 0.063233
Validation loss decreased (inf --> 0.063233).  Saving model ...
Epoch: 2        Training Loss: 0.266155         Validation Loss: 0.062228
Validation loss decreased (0.063233 --> 0.062228).  Saving model ...
Epoch: 3        Training Loss: 0.259215         Validation Loss: 0.064130
Epoch: 4        Training Loss: 0.254243         Validation Loss: 0.060864
Validation loss decreased (0.062228 --> 0.060864).  Saving model ...
Epoch: 5        Training Loss: 0.251559         Validation Loss: 0.060385
Validation loss decreased (0.060864 --> 0.060385).  Saving model ...
Epoch: 6        Training Loss: 0.249566         Validation Loss: 0.063521
Epoch: 7        Training Loss: 0.245448         Validation Loss: 0.059431
Validation loss decreased (0.060385 --> 0.059431).  Saving model ...
Epoch: 8        Training Loss: 0.243550         Validation Loss: 0.059766
Epoch: 9        Training Loss: 0.243813         Validation Loss: 0.059017
Validation loss decreased (0.059431 --> 0.059017).  Saving model ...
Epoch: 10       Training Loss: 0.240469         Validation Loss: 0.060463
Epoch: 11       Training Loss: 0.238521         Validation Loss: 0.059306
Epoch: 12       Training Loss: 0.238224         Validation Loss: 0.059221
Epoch: 13       Training Loss: 0.236400         Validation Loss: 0.059984
Epoch: 14       Training Loss: 0.235027         Validation Loss: 0.059691
Epoch: 15       Training Loss: 0.233452         Validation Loss: 0.058858
Validation loss decreased (0.059017 --> 0.058858).  Saving model ...
Epoch: 16       Training Loss: 0.232831         Validation Loss: 0.059439
Epoch: 17       Training Loss: 0.232542         Validation Loss: 0.060281
Epoch: 18       Training Loss: 0.231464         Validation Loss: 0.059167
Epoch: 19       Training Loss: 0.230872         Validation Loss: 0.059100
Epoch: 20       Training Loss: 0.229101         Validation Loss: 0.058670
Validation loss decreased (0.058858 --> 0.058670).  Saving model ...
Epoch: 21       Training Loss: 0.228756         Validation Loss: 0.059383
Epoch: 22       Training Loss: 0.227179         Validation Loss: 0.059229
Epoch: 23       Training Loss: 0.227279         Validation Loss: 0.059556
Epoch: 24       Training Loss: 0.227299         Validation Loss: 0.058727
Epoch: 25       Training Loss: 0.226237         Validation Loss: 0.059695
Epoch: 26       Training Loss: 0.226207         Validation Loss: 0.059091
Epoch: 27       Training Loss: 0.224940         Validation Loss: 0.060454
Epoch: 28       Training Loss: 0.223619         Validation Loss: 0.060029
Epoch: 29       Training Loss: 0.222440         Validation Loss: 0.059677
Epoch: 30       Training Loss: 0.223500         Validation Loss: 0.059161
```

In [63]:

```
model.load_state_dict(torch.load('MLP_Ternary_mymodel.pt'))
```

Out[63]:

```
<All keys matched successfully>
```

In [64]:
```
test_loader = DataLoader(test_data, batch_size=1,)
```

In [65]:
```
predictions = predict(model,test_loader)
predictions = np.array(predictions)
```

In [66]:
```
test_accuracy_mlp_ternary_mymodel_avg = accuracy_score(Y_test, predictions)
```

In [67]:
```
print("Accuracy for Ternary Classification using my own model is : ", test_accuracy_mlp_ternary_mymodel_avg)
```

```
Accuracy for Ternary Classification using Average of word vectors using my own model is :  0.877425
```

# Ternary - pretrained model

In [68]:
```
vectorized_data = reviews_with_avg(balanced_df, wv)
```

In [69]:
```
X_train,X_test,Y_train, Y_test = train_test_split(vectorized_data, balanced_df['sentiment'], test_size=0.2, random_state=6
```

In [70]:
```
train_data = TrainData(X_train, Y_train)
test_data = TestData(X_test)
```

In [71]:
```
# how many samples per batch to load
batch_size = 100
# percentage of training set to use as validation
valid_size = 0.2

# obtain training indices that will be used for validation
num_train = len(train_data)
indices = list(range(num_train))

np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]

# define samplers for obtaining training and validation batches
train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

# prepare data loaders
train_loader = DataLoader(train_data, batch_size=batch_size, sampler=train_sampler)
valid_loader = DataLoader(train_data, batch_size=batch_size, sampler=valid_sampler)
```

In [72]:
```
# initialize the NN
model = MLP_Ternary()
model.to(device)
print(model)
```

```
MLP_Ternary(
  (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 [73]:
```
# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()

# specify optimizer (Adam) and learning rate = 0.005
optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
```

In [74]:
```
# number of epochs to train the model
n_epochs = 30

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

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

    ###################
    # train the model #
    ###################
    model.train() # prep model for training
    for data, target in train_loader:
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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)

    ######################
    # validate the model #
    ######################
    model.eval() # prep model for evaluation
    for data, target in valid_loader:
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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(), 'MLP_Ternary_google.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1        Training Loss: 0.380234         Validation Loss: 0.084654
Validation loss decreased (inf --> 0.084654).  Saving model ...
Epoch: 2        Training Loss: 0.342594         Validation Loss: 0.082508
Validation loss decreased (0.084654 --> 0.082508).  Saving model ...
Epoch: 3        Training Loss: 0.332205         Validation Loss: 0.077672
Validation loss decreased (0.082508 --> 0.077672).  Saving model ...
Epoch: 4        Training Loss: 0.324480         Validation Loss: 0.080993
Epoch: 5        Training Loss: 0.321196         Validation Loss: 0.077929
Epoch: 6        Training Loss: 0.316876         Validation Loss: 0.076181
Validation loss decreased (0.077672 --> 0.076181).  Saving model ...
Epoch: 7        Training Loss: 0.312398         Validation Loss: 0.075694
Validation loss decreased (0.076181 --> 0.075694).  Saving model ...
Epoch: 8        Training Loss: 0.309886         Validation Loss: 0.075743
Epoch: 9        Training Loss: 0.307123         Validation Loss: 0.074429
Validation loss decreased (0.075694 --> 0.074429).  Saving model ...
Epoch: 10       Training Loss: 0.304604         Validation Loss: 0.074656
Epoch: 11       Training Loss: 0.303756         Validation Loss: 0.074285
Validation loss decreased (0.074429 --> 0.074285).  Saving model ...
Epoch: 12       Training Loss: 0.301974         Validation Loss: 0.073920
Validation loss decreased (0.074285 --> 0.073920).  Saving model ...
Epoch: 13       Training Loss: 0.298224         Validation Loss: 0.074008
Epoch: 14       Training Loss: 0.297110         Validation Loss: 0.074360
Epoch: 15       Training Loss: 0.296762         Validation Loss: 0.074075
Epoch: 16       Training Loss: 0.295196         Validation Loss: 0.074237
Epoch: 17       Training Loss: 0.293805         Validation Loss: 0.074099
Epoch: 18       Training Loss: 0.291882         Validation Loss: 0.075948
Epoch: 19       Training Loss: 0.292420         Validation Loss: 0.074266
Epoch: 20       Training Loss: 0.289741         Validation Loss: 0.073046
Validation loss decreased (0.073920 --> 0.073046).  Saving model ...
Epoch: 21       Training Loss: 0.288459         Validation Loss: 0.074215
Epoch: 22       Training Loss: 0.289108         Validation Loss: 0.072982
Validation loss decreased (0.073046 --> 0.072982).  Saving model ...
Epoch: 23       Training Loss: 0.288162         Validation Loss: 0.072914
Validation loss decreased (0.072982 --> 0.072914).  Saving model ...
Epoch: 24       Training Loss: 0.285804         Validation Loss: 0.073034
Epoch: 25       Training Loss: 0.285797         Validation Loss: 0.073605
Epoch: 26       Training Loss: 0.285394         Validation Loss: 0.073531
Epoch: 27       Training Loss: 0.283337         Validation Loss: 0.073637
Epoch: 28       Training Loss: 0.284257         Validation Loss: 0.073004
Epoch: 29       Training Loss: 0.281964         Validation Loss: 0.073479
Epoch: 30       Training Loss: 0.282441         Validation Loss: 0.073828
```

In [75]:
```
model.load_state_dict(torch.load('MLP_Ternary_google.pt'))
```

Out[75]:
```
<All keys matched successfully>
```

In [76]:
```
test_loader = DataLoader(test_data, batch_size=1,)
```

In [77]:
```
predictions = predict(model,test_loader)
predictions = np.array(predictions)
```

In [78]:
```
test_accuracy_mlp_ternary_google_avg = accuracy_score(Y_test, predictions)
```

In [79]:
```
print("Accuracy for Ternary Classification using google pretrained model is : ", test_accuracy_mlp_ternary_google_avg)
```

```
Accuracy for Ternary Classification using Average of word vectors using google pretrained model is :  0.840125
```

# 4 - (b)

In [80]:
```
def process_reviews_concat(df, model):
    vectorized_data_temp = []

    for review_body in df['review_body']:
        tokens = review_body.split()
        weight = [model[x] for count, x in enumerate(tokens) if x in model and count < 10]

        while len(weight) < 10:
            weight.append(np.zeros(300, dtype=float))

        transposed_weight = np.transpose(weight)
        vectorized_data_temp.append(transposed_weight)

    return vectorized_data_temp
```

# Binary - Own Model

In [81]:
```
# Reference - # https://www.kaggle.com/code/mishra1993/pytorch-multi-layer-perceptron-mnist/notebook

# define the NN architecture
class MLP_Binary_Concat(nn.Module):
    def __init__(self):
        super(MLP_Binary_Concat, self).__init__()

        hidden_1 = 50
        hidden_2 = 10

        self.fc1 = nn.Linear(3000, hidden_1)
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        self.fc3 = nn.Linear(hidden_2, 2)

        self.dropout = nn.Dropout(0.2)

        self.double()

    def forward(self, x):

        # Flattening the input
        x = x.view(-1, 300*10)

        x = F.relu(self.fc1(x))
        x = self.dropout(x)

        x = F.relu(self.fc2(x))
        x = self.dropout(x)

        x = self.fc3(x)

        return x
```

In [82]:
```
vectorized_data = process_reviews_concat(balanced_df, word2vec_model.wv)
```

In [83]:
```
(_train,X_test,Y_train, Y_test = train_test_split(vectorized_data, balanced_df['sentiment'], test_size=0.2, random_state=6)
```

In [84]:
```
train_data = TrainData(X_train, Y_train)
test_data = TestData(X_test)
```

In [85]:
```
# how many samples per batch to load
batch_size = 100
# percentage of training set to use as validation
valid_size = 0.2

# obtain training indices that will be used for validation
num_train = len(train_data)
indices = list(range(num_train))

np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]

# define samplers for obtaining training and validation batches
train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

# prepare data loaders
train_loader = DataLoader(train_data, batch_size=batch_size, sampler=train_sampler)
valid_loader = DataLoader(train_data, batch_size=batch_size, sampler=valid_sampler)
```

In [86]:
```
# initialize the NN
model = MLP_Binary_Concat()
model.to(device)
print(model)
```

```
MLP_Binary_Concat(
  (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 [87]:
```
# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()

# specify optimizer (Adam) and learning rate = 0.005
optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
```

In [88]:
```
# number of epochs to train the model
n_epochs = 30

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

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

    ###################
    # train the model #
    ###################
    model.train() # prep model for training
    for data, target in train_loader:
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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)

    ######################
    # validate the model #
    ######################
    model.eval() # prep model for evaluation
    for data, target in valid_loader:
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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(), 'MLP_Binary_Concat_mymodel.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1 	Training Loss: 0.409312 	Validation Loss: 0.094115
Validation loss decreased (inf --> 0.094115).  Saving model ...
Epoch: 2 	Training Loss: 0.377101 	Validation Loss: 0.092929
Validation loss decreased (0.094115 --> 0.092929).  Saving model ...
Epoch: 3 	Training Loss: 0.364225 	Validation Loss: 0.091815
Validation loss decreased (0.092929 --> 0.091815).  Saving model ...
Epoch: 4 	Training Loss: 0.351826 	Validation Loss: 0.090682
Validation loss decreased (0.091815 --> 0.090682).  Saving model ...
Epoch: 5 	Training Loss: 0.340658 	Validation Loss: 0.091128
Epoch: 6 	Training Loss: 0.330302 	Validation Loss: 0.091655
Epoch: 7 	Training Loss: 0.319953 	Validation Loss: 0.095834
Epoch: 8 	Training Loss: 0.309850 	Validation Loss: 0.097236
Epoch: 9 	Training Loss: 0.301525 	Validation Loss: 0.099895
Epoch: 10 	Training Loss: 0.294042 	Validation Loss: 0.094316
Epoch: 11 	Training Loss: 0.286335 	Validation Loss: 0.107708
Epoch: 12 	Training Loss: 0.280964 	Validation Loss: 0.103803
Epoch: 13 	Training Loss: 0.274523 	Validation Loss: 0.111723
Epoch: 14 	Training Loss: 0.268987 	Validation Loss: 0.130959
Epoch: 15 	Training Loss: 0.266069 	Validation Loss: 0.102400
Epoch: 16 	Training Loss: 0.258337 	Validation Loss: 0.104593
Epoch: 17 	Training Loss: 0.254019 	Validation Loss: 0.111263
Epoch: 18 	Training Loss: 0.250439 	Validation Loss: 0.108565
Epoch: 19 	Training Loss: 0.244881 	Validation Loss: 0.105946
Epoch: 20 	Training Loss: 0.242535 	Validation Loss: 0.111095
Epoch: 21 	Training Loss: 0.238565 	Validation Loss: 0.113532
Epoch: 22 	Training Loss: 0.236469 	Validation Loss: 0.107305
Epoch: 23 	Training Loss: 0.235382 	Validation Loss: 0.129606
Epoch: 24 	Training Loss: 0.229807 	Validation Loss: 0.150340
Epoch: 25 	Training Loss: 0.227751 	Validation Loss: 0.118687
Epoch: 26 	Training Loss: 0.224175 	Validation Loss: 0.122983
Epoch: 27 	Training Loss: 0.222318 	Validation Loss: 0.130280
Epoch: 28 	Training Loss: 0.219710 	Validation Loss: 0.115139
Epoch: 29 	Training Loss: 0.218238 	Validation Loss: 0.144792
Epoch: 30 	Training Loss: 0.215092 	Validation Loss: 0.147040
```

In [89]:
```
model.load_state_dict(torch.load('MLP_Binary_Concat_mymodel.pt'))
```

Out[89]: <All keys matched successfully>

In [90]:
```
test_loader = DataLoader(test_data, batch_size=1,)
```

In [91]:
```
predictions = predict(model,test_loader)
predictions = np.array(predictions)
```

In [92]:
```
test_accuracy_mlp_binary_concat_mymodel = accuracy_score(Y_test, predictions)
```

In [93]:
```
print("Accuracy for Binary Classification using my own model is : ", test_accuracy_mlp_binary_concat_mymodel)
```

```
Accuracy for Binary Classification using Concatination of word vectors using my own model is :  0.778775
```

# Binary - Pretrained Model

In [94]:
```
vectorized_data = process_reviews_concat(balanced_df, wv)
```

In [95]:
```
(_train,X_test,Y_train, Y_test = train_test_split(vectorized_data, balanced_df['sentiment'], test_size=0.2, random_state=6)
```

In [96]:
```
train_data = TrainData(X_train, Y_train)
test_data = TestData(X_test)
```

In [97]:
```
batch_size = 100
valid_size = 0.2

num_train = len(train_data)
indices = list(range(num_train))

np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]

train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

train_loader = DataLoader(train_data, batch_size=batch_size, sampler=train_sampler)
valid_loader = DataLoader(train_data, batch_size=batch_size, sampler=valid_sampler)
```

In [98]:
```
model = MLP_Binary_Concat()
model.to(device)
print(model)
```

```
MLP_Binary_Concat(
  (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 [99]:
```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.005)
```

In [100]:
```
n_epochs = 30

valid_loss_min = np.Inf # set initial "min" to infinity

for epoch in range(n_epochs):
    train_loss = 0.0
    valid_loss = 0.0

    ###################
    # train the model #
    ###################
    model.train() # prep model for training
    for data, target in train_loader:
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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)

    ######################
    # validate the model #
    ######################
    model.eval() # prep model for evaluation
    for data, target in valid_loader:
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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(), 'MLP_Binary_Concat_google.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1 	Training Loss: 0.425059 	Validation Loss: 0.099489
Validation loss decreased (inf --> 0.099489).  Saving model ...
Epoch: 2 	Training Loss: 0.386959 	Validation Loss: 0.095177
Validation loss decreased (0.099489 --> 0.095177).  Saving model ...
Epoch: 3 	Training Loss: 0.364787 	Validation Loss: 0.095734
Epoch: 4 	Training Loss: 0.346941 	Validation Loss: 0.095222
Epoch: 5 	Training Loss: 0.329516 	Validation Loss: 0.095523
Epoch: 6 	Training Loss: 0.313732 	Validation Loss: 0.098961
Epoch: 7 	Training Loss: 0.300009 	Validation Loss: 0.097193
Epoch: 8 	Training Loss: 0.287734 	Validation Loss: 0.103929
Epoch: 9 	Training Loss: 0.276946 	Validation Loss: 0.100603
Epoch: 10 	Training Loss: 0.266451 	Validation Loss: 0.102536
Epoch: 11 	Training Loss: 0.256706 	Validation Loss: 0.108202
Epoch: 12 	Training Loss: 0.248438 	Validation Loss: 0.108000
Epoch: 13 	Training Loss: 0.242723 	Validation Loss: 0.111269
Epoch: 14 	Training Loss: 0.236378 	Validation Loss: 0.114500
Epoch: 15 	Training Loss: 0.229530 	Validation Loss: 0.120218
Epoch: 16 	Training Loss: 0.224123 	Validation Loss: 0.115875
Epoch: 17 	Training Loss: 0.218441 	Validation Loss: 0.126638
Epoch: 18 	Training Loss: 0.212302 	Validation Loss: 0.128463
Epoch: 19 	Training Loss: 0.209506 	Validation Loss: 0.133925
Epoch: 20 	Training Loss: 0.205793 	Validation Loss: 0.123751
Epoch: 21 	Training Loss: 0.200899 	Validation Loss: 0.130127
Epoch: 22 	Training Loss: 0.197312 	Validation Loss: 0.142421
Epoch: 23 	Training Loss: 0.194047 	Validation Loss: 0.134388
Epoch: 24 	Training Loss: 0.191017 	Validation Loss: 0.144815
Epoch: 25 	Training Loss: 0.187543 	Validation Loss: 0.137182
Epoch: 26 	Training Loss: 0.185054 	Validation Loss: 0.152316
Epoch: 27 	Training Loss: 0.180837 	Validation Loss: 0.153299
Epoch: 28 	Training Loss: 0.178715 	Validation Loss: 0.155890
Epoch: 29 	Training Loss: 0.175134 	Validation Loss: 0.149053
Epoch: 30 	Training Loss: 0.173319 	Validation Loss: 0.147029
```

In [101]:

```
model.load_state_dict(torch.load('MLP_Binary_Concat_google.pt'))
```

Out[101]:

```
<All keys matched successfully>
```

In [102]:

```
test_loader = DataLoader(test_data, batch_size=1,)
```

In [103]:

```
predictions = predict(model,test_loader)
predictions = np.array(predictions)
```

In [104]:
```
test_accuracy_mlp_binary_concat_google = accuracy_score(Y_test, predictions)
```

In [105]:
```
print("Accuracy for Binary Classification using google pretrained model is : ", test_accuracy_mlp_binary_concat_google)
```

```
Accuracy for Binary Classification using Concatination of word vectors using google pretrained model is :  0.763775
```

# Ternary - Own Model

In [106]:
```
# Reference - # https://www.kaggle.com/code/mishra1993/pytorch-multi-layer-perceptron-mnist/notebook

class MLP_Ternary_Concat(nn.Module):
    def __init__(self):
        super(MLP_Ternary_Concat, self).__init__()

        hidden_1 = 50
        hidden_2 = 10

        self.fc1 = nn.Linear(3000, hidden_1)
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        self.fc3 = nn.Linear(hidden_2, 3)

        self.dropout = nn.Dropout(0.2)

        self.double()

    def forward(self, x):

        # Flatten the input
        x = x.view(-1, 300*10)

        x = F.relu(self.fc1(x))
        x = self.dropout(x)

        x = F.relu(self.fc2(x))
        x = self.dropout(x)

        x = self.fc3(x)

        return x
```

In [107]:
```
vectorized_data = process_reviews_concat(balanced_df, word2vec_model.wv)
```

In [108]:
```
X_train,X_test,Y_train, Y_test = train_test_split(vectorized_data, balanced_df['sentiment'], test_size=0.2, random_state=6)
```

In [109]:
```
train_data = TrainData(X_train, Y_train)
test_data = TestData(X_test)
```

In [110]:
```
batch_size = 100
valid_size = 0.2

num_train = len(train_data)
indices = list(range(num_train))

np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]

train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

train_loader = DataLoader(train_data, batch_size=batch_size, sampler=train_sampler)
valid_loader = DataLoader(train_data, batch_size=batch_size, sampler=valid_sampler)
```

In [111]:
```
model = MLP_Ternary_Concat()
model.to(device)
print(model)
```

```
MLP_Ternary_Concat(
  (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=3, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

In [112]:
```
criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=0.0005)
```

In [113]:
```
n_epochs = 30

valid_loss_min = np.Inf # set initial "min" to infinity

for epoch in range(n_epochs):
    train_loss = 0.0
    valid_loss = 0.0

    ###################
    # train the model #
    ###################
    model.train() # prep model for training
    for data, target in train_loader:
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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)

    ######################
    # validate the model #
    ######################
    model.eval() # prep model for evaluation
    for data, target in valid_loader:
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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(), 'MLP_Ternary_Concat_mymodel.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1        Training Loss: 0.422496         Validation Loss: 0.093401
Validation loss decreased (inf --> 0.093401).  Saving model ...
Epoch: 2        Training Loss: 0.371917         Validation Loss: 0.091199
Validation loss decreased (0.093401 --> 0.091199).  Saving model ...
Epoch: 3        Training Loss: 0.354878         Validation Loss: 0.090110
Validation loss decreased (0.091199 --> 0.090110).  Saving model ...
Epoch: 4        Training Loss: 0.339528         Validation Loss: 0.089294
Validation loss decreased (0.090110 --> 0.089294).  Saving model ...
Epoch: 5        Training Loss: 0.324450         Validation Loss: 0.089601
Epoch: 6        Training Loss: 0.310889         Validation Loss: 0.090195
Epoch: 7        Training Loss: 0.298505         Validation Loss: 0.091271
Epoch: 8        Training Loss: 0.287327         Validation Loss: 0.090961
Epoch: 9        Training Loss: 0.276259         Validation Loss: 0.093196
Epoch: 10       Training Loss: 0.266694         Validation Loss: 0.095290
Epoch: 11       Training Loss: 0.258870         Validation Loss: 0.096612
Epoch: 12       Training Loss: 0.250903         Validation Loss: 0.099236
Epoch: 13       Training Loss: 0.243471         Validation Loss: 0.098184
Epoch: 14       Training Loss: 0.236696         Validation Loss: 0.101838
Epoch: 15       Training Loss: 0.230995         Validation Loss: 0.103984
Epoch: 16       Training Loss: 0.225272         Validation Loss: 0.103902
Epoch: 17       Training Loss: 0.220367         Validation Loss: 0.107692
Epoch: 18       Training Loss: 0.216378         Validation Loss: 0.108545
Epoch: 19       Training Loss: 0.211765         Validation Loss: 0.109629
Epoch: 20       Training Loss: 0.208114         Validation Loss: 0.109748
Epoch: 21       Training Loss: 0.203710         Validation Loss: 0.112270
Epoch: 22       Training Loss: 0.200265         Validation Loss: 0.113160
Epoch: 23       Training Loss: 0.197210         Validation Loss: 0.112561
Epoch: 24       Training Loss: 0.194849         Validation Loss: 0.113596
Epoch: 25       Training Loss: 0.189076         Validation Loss: 0.118454
Epoch: 26       Training Loss: 0.186329         Validation Loss: 0.118568
Epoch: 27       Training Loss: 0.185629         Validation Loss: 0.124068
Epoch: 28       Training Loss: 0.182019         Validation Loss: 0.121708
Epoch: 29       Training Loss: 0.179862         Validation Loss: 0.123078
Epoch: 30       Training Loss: 0.176742         Validation Loss: 0.123932
```

In [114]:
```
model.load_state_dict(torch.load('MLP_Ternary_Concat_mymodel.pt'))
```

Out[114]:
```
<All keys matched successfully>
```

In [115]:
```
test_loader = DataLoader(test_data, batch_size=1,)
```

In [116]:
```
predictions = predict(model,test_loader)
predictions = np.array(predictions)
```

In [117]:
```
test_accuracy_mlp_ternary_concat_mymodel = accuracy_score(Y_test, predictions)
```

In [118]:
```
print("Accuracy for Ternary Classification using my own model is : ", test_accuracy_mlp_ternary_concat_mymodel)
```

```
Accuracy for Ternary Classification using Concatination of word vectors using my own model is :  0.78555
```

# Ternary - Pretrained Model

In [119]:
```
vectorized_data = process_reviews_concat(balanced_df, wv)
```

In [120]:
```
X_train,X_test,Y_train, Y_test = train_test_split(vectorized_data, balanced_df['sentiment'], test_size=0.2, random_state=6
```

In [121]:
```
train_data = TrainData(X_train, Y_train)
test_data = TestData(X_test)
```

In [122]:
```
batch_size = 100
valid_size = 0.2

num_train = len(train_data)
indices = list(range(num_train))

np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]

train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

train_loader = DataLoader(train_data, batch_size=batch_size, sampler=train_sampler)
valid_loader = DataLoader(train_data, batch_size=batch_size, sampler=valid_sampler)
```

In [123]:
```
model = MLP_Ternary_Concat()
model.to(device)
print(model)
```

```
MLP_Ternary_Concat(
  (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=3, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

In [124]:
```
criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=0.0005)
```

In [125]:
```
n_epochs = 30

valid_loss_min = np.Inf # set initial "min" to infinity

for epoch in range(n_epochs):
    train_loss = 0.0
    valid_loss = 0.0

    ###################
    # train the model #
    ###################
    model.train() # prep model for training
    for data, target in train_loader:
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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)

    ######################
    # validate the model #
    ######################
    model.eval() # prep model for evaluation
    for data, target in valid_loader:
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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(), 'MLP_Ternary_Concat_google.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1 	Training Loss: 0.480743 	Validation Loss: 0.102648
Validation loss decreased (inf --> 0.102648).  Saving model ...
Epoch: 2 	Training Loss: 0.411164 	Validation Loss: 0.099158
Validation loss decreased (0.102648 --> 0.099158).  Saving model ...
Epoch: 3 	Training Loss: 0.391295 	Validation Loss: 0.097040
Validation loss decreased (0.099158 --> 0.097040).  Saving model ...
Epoch: 4 	Training Loss: 0.374791 	Validation Loss: 0.096086
Validation loss decreased (0.097040 --> 0.096086).  Saving model ...
Epoch: 5 	Training Loss: 0.358587 	Validation Loss: 0.095068
Validation loss decreased (0.096086 --> 0.095068).  Saving model ...
Epoch: 6 	Training Loss: 0.342018 	Validation Loss: 0.095115
Epoch: 7 	Training Loss: 0.326420 	Validation Loss: 0.096210
Epoch: 8 	Training Loss: 0.310864 	Validation Loss: 0.098553
Epoch: 9 	Training Loss: 0.296857 	Validation Loss: 0.099090
Epoch: 10 	Training Loss: 0.283113 	Validation Loss: 0.101912
Epoch: 11 	Training Loss: 0.270438 	Validation Loss: 0.101886
Epoch: 12 	Training Loss: 0.260146 	Validation Loss: 0.105631
Epoch: 13 	Training Loss: 0.247249 	Validation Loss: 0.110794
Epoch: 14 	Training Loss: 0.240550 	Validation Loss: 0.110461
Epoch: 15 	Training Loss: 0.232287 	Validation Loss: 0.112188
Epoch: 16 	Training Loss: 0.224764 	Validation Loss: 0.114597
Epoch: 17 	Training Loss: 0.218822 	Validation Loss: 0.118844
Epoch: 18 	Training Loss: 0.213637 	Validation Loss: 0.120605
Epoch: 19 	Training Loss: 0.206993 	Validation Loss: 0.122645
Epoch: 20 	Training Loss: 0.201280 	Validation Loss: 0.124325
Epoch: 21 	Training Loss: 0.198071 	Validation Loss: 0.126706
Epoch: 22 	Training Loss: 0.194069 	Validation Loss: 0.129744
Epoch: 23 	Training Loss: 0.188238 	Validation Loss: 0.135864
Epoch: 24 	Training Loss: 0.184261 	Validation Loss: 0.132794
Epoch: 25 	Training Loss: 0.182887 	Validation Loss: 0.130917
Epoch: 26 	Training Loss: 0.177701 	Validation Loss: 0.135917
Epoch: 27 	Training Loss: 0.174405 	Validation Loss: 0.135689
Epoch: 28 	Training Loss: 0.171109 	Validation Loss: 0.139083
Epoch: 29 	Training Loss: 0.168801 	Validation Loss: 0.136244
Epoch: 30 	Training Loss: 0.166410 	Validation Loss: 0.141464
```

In [126]:
```
model.load_state_dict(torch.load('MLP_Ternary_Concat_google.pt'))
```

Out[126]:
```
<All keys matched successfully>
```

In [127]:
```
test_loader = DataLoader(test_data, batch_size=1,)
```

In [128]:
```
predictions = predict(model,test_loader)
predictions = np.array(predictions)
```

In [129]:
```
test_accuracy_mlp_ternary_concat_google = accuracy_score(Y_test, predictions)
```

In [130]:
```
print("Accuracy for Ternary Classification using google pretrained model is : ", test_accuracy_mlp_ternary_concat_google)
```

```
Accuracy for Ternary Classification using Concatination of word vectors using google pretrained model is :  0.766375
```

## Remarks

### Accuracy:

- Part(a):
  - Binary Classification using Own Model - 87.5%
  - Binary Classification using Google Pretrained Model - 83.6%
  - Ternary Classification using Own Model - 87.7%
  - Ternary Classification using Google Pretrained Model - 84%
- Part(b):
  - Binary Classification using Own Model - 77.8%
  - Binary Classification using Google Pretrained Model - 76.3%
  - Ternary Classification using Own Model - 78.5%
  - Ternary Classification using Google Pretrained Model - 76.6%

### Conclusion

As anticipated, Feedforward Neural Networks (FFN) outperform Simple models in binary classification. This is because FFN's multiple layers enable it to capture the semantic meaning of individual reviews much more effectively than Simple models. This is evident when we compare the binary classification values for FFN and Simple models. Another finding is that the SVM model outperforms the Part (b) feedforward neural networks because in Part (b), we only take into account the word vectors for the first 10 words, which decreased accuracy because the first 10 words are insufficient to convey the whole emotion of a review. Another common finding is that, because binary classification requires fewer meta parameters and is hence simpler to train, it performs better than ternary classification.

# CNN - own model

In [*]:
```
vectorized_data = process_reviews_cnn(balanced_df, word2vec_model.wv)
```

In [*]:
```
X_train,X_test,Y_train, Y_test = train_test_split(vectorized_data, balanced_df['sentiment'], test_size=0.2, random_state=6
```

In [131]:
```
def process_reviews_cnn(df, model):
    def vectorize_review(review):
        tokens = review.split()[:50]
        weight = [model[token] for token in tokens if token in model]
        weight += [np.zeros(300, dtype=float)] * (50 - len(weight))
        return weight

    df['vectorized_data'] = df['review_body'].apply(vectorize_review)
    return df['vectorized_data'].tolist()
```

In [132]:
```
# Reference - # https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html

# define the NN architecture
class CNN_Binary(nn.Module):
    def __init__(self):
        super(CNN_Binary, self).__init__()

        out_channels_1 = 50
        out_channels_2 = 10

        self.conv1 = nn.Conv2d(in_channels=1, out_channels=out_channels_1, kernel_size=5)
        self.pool = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(in_channels=out_channels_1, out_channels=out_channels_2, kernel_size=5)
        self.fc1 = nn.Linear(out_channels_2 * 9 * 72, 100)
        self.fc2 = nn.Linear(100, 2)
        self.dropout1 = nn.Dropout(0.3)
        self.dropout2 = nn.Dropout(0.4)

        self.double()

    def forward(self, x):

        # Adding new dimension as Conv2d requires 4D tensors
        x = x.unsqueeze(1)

        x = self.conv1(x)
        x = self.pool(F.relu(x))
        x = self.dropout1(x)

        x = self.conv2(x)
        x = self.pool(F.relu(x))

        # flatten all dimensions except batch
        x = torch.flatten(x, 1)

        x = F.relu(self.fc1(x))
        x = self.dropout2(x)

        x = self.fc2(x)

        return x
```

In [133]:
```
vectorized_data = process_reviews_cnn(balanced_df, word2vec_model.wv)
```

In [134]:
```
X_train,X_test,Y_train, Y_test = train_test_split(vectorized_data, balanced_df['sentiment'], test_size=0.2, random_state=6
```

In [135]:
```
train_data = TrainData(X_train, Y_train)
test_data = TestData(X_test)
```

In [136]:
```
batch_size = 100
valid_size = 0.2

num_train = len(train_data)
indices = list(range(num_train))

np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]

train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

train_loader = DataLoader(train_data, batch_size=batch_size, sampler=train_sampler)
valid_loader = DataLoader(train_data, batch_size=batch_size, sampler=valid_sampler)
```

In [137]:
```
model = CNN_Binary()
model.to(device)
print(model)
```

```
CNN_Binary(
  (conv1): Conv2d(1, 50, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(50, 10, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=6480, out_features=100, bias=True)
  (fc2): Linear(in_features=100, out_features=2, bias=True)
  (dropout1): Dropout(p=0.3, inplace=False)
  (dropout2): Dropout(p=0.4, inplace=False)
)
```

In [138]:
```
criterion = nn.CrossEntropyLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=0.0005)
```

In [*]:
```
n_epochs = 15

valid_loss_min = np.Inf # set initial "min" to infinity

for epoch in range(n_epochs):
    train_loss = 0.0
    valid_loss = 0.0

    ###################
    # train the model #
    ###################
    model.train() # prep model for training
    for data, target in train_loader:
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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)

    ######################
    # validate the model #
    ######################
    model.eval() # prep model for evaluation
    for data, target in valid_loader:
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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(), 'CNN_Binary_mymodel.pt')
        valid_loss_min = valid_loss
```

In [*]:
```
model.load_state_dict(torch.load('CNN_Binary_mymodel.pt'))
```

In [*]:
```
test_loader = DataLoader(test_data, batch_size=1,)
```

In [*]:
```
predictions = predict(model,test_loader)
predictions = np.array(predictions)
```

In [*]:
```
test_accuracy_cnn_binary_mymodel = accuracy_score(Y_test, predictions)
```

In [*]:
```
print("Accuracy for Binary Classification using CNN with my own model is : ", test_accuracy_cnn_binary_mymodel)
```

# CNN - Pretrained model

In [*]:
```
vectorized_data = process_reviews_cnn(balanced_df, wv)
```

In [*]:
```
X_train,X_test,Y_train, Y_test = train_test_split(vectorized_data, balanced_df['sentiment'], test_size=0.2, random_state=6
```

In [*]:
```
train_data = TrainData(X_train, Y_train)
test_data = TestData(X_test)
```

In [*]:
```
batch_size = 100
valid_size = 0.2

num_train = len(train_data)
indices = list(range(num_train))

np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]

train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

train_loader = DataLoader(train_data, batch_size=batch_size, sampler=train_sampler)
valid_loader = DataLoader(train_data, batch_size=batch_size, sampler=valid_sampler)
```

# CNN - Binary

In [*]:
```
model = CNN_Binary()
model.to(device)
print(model)
```

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

optimizer = torch.optim.Adam(model.parameters(), lr=0.0005)
```

In [*]:
```
n_epochs = 15

valid_loss_min = np.Inf # set initial "min" to infinity

for epoch in range(n_epochs):
    train_loss = 0.0
    valid_loss = 0.0

    ###################
    # train the model #
    ###################
    model.train() # prep model for training
    for data, target in train_loader:
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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)

    ######################
    # validate the model #
    ######################
    model.eval() # prep model for evaluation
    for data, target in valid_loader:
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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(), 'CNN_Binary_google.pt')
        valid_loss_min = valid_loss
```

In [*]:
```
model.load_state_dict(torch.load('CNN_Binary_google.pt'))
```

In [*]:
```
test_loader = DataLoader(test_data, batch_size=1,)
```

In [*]:
```
predictions = predict(model,test_loader)
predictions = np.array(predictions)
```

In [*]:
```
test_accuracy_cnn_binary_google = accuracy_score(Y_test, predictions)
```

In [*]:
```
print("Accuracy for Binary Classification using CNN with google pretrained model is : ", test_accuracy_cnn_binary_google)
```

# CNN

In [*]:
```
# Reference - # https://www.kaggle.com/code/mishra1993/pytorch-multi-layer-perceptron-mnist/notebook

class CNN_Ternary(nn.Module):
    def __init__(self):
        super(CNN_Ternary, self).__init__()

        out_channels_1 = 50
        out_channels_2 = 10

        self.conv1 = nn.Conv2d(in_channels=1, out_channels=out_channels_1, kernel_size=5)
        self.pool = nn.MaxPool2d(2)
        self.conv2 = nn.Conv2d(in_channels=out_channels_1, out_channels=out_channels_2, kernel_size=5)
        self.fc1 = nn.Linear(out_channels_2 * 9 * 72, 100)
        self.fc2 = nn.Linear(100, 3)
        self.dropout1 = nn.Dropout(0.3)
        self.dropout2 = nn.Dropout(0.4)

        self.double()

    def forward(self, x):

        # Adding new dimension for Conv2d as it requires 4D tensors
        x = x.unsqueeze(1)

        x = self.conv1(x)
        x = self.pool(F.relu(x))
        x = self.dropout1(x)

        x = self.conv2(x)
        x = self.pool(F.relu(x))

        # flatten all dimensions except batch
        x = torch.flatten(x, 1)

        x = F.relu(self.fc1(x))
        x = self.dropout2(x)

        x = self.fc2(x)

        return x
```

# CNN -

In [*]:
```
vectorized_data = process_reviews_cnn(balanced_df, word2vec_model.wv)
```

In [*]:
```
X_train,X_test,Y_train, Y_test = train_test_split(vectorized_data, balanced_df['sentiment'], test_size=0.2, random_state=6)
```

In [*]:
```
train_data = TrainData(X_train, Y_train)
test_data = TestData(X_test)
```

In [*]:
```
batch_size = 100
valid_size = 0.2

num_train = len(train_data)
indices = list(range(num_train))

np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]

train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

train_loader = DataLoader(train_data, batch_size=batch_size, sampler=train_sampler)
valid_loader = DataLoader(train_data, batch_size=batch_size, sampler=valid_sampler)
```

In [*]:
```
model = CNN_Ternary()
model = model.to(device)
print(model)
```

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

optimizer = torch.optim.Adam(model.parameters(), lr=0.0005)
```

In [*]:
```
n_epochs = 15

valid_loss_min = np.Inf # set initial "min" to infinity

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

    ###################
    # train the model #
    ###################
    model.train() # prep model for training
    for data, target in train_loader:
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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)

    ######################
    # validate the model #
    ######################
    model.eval() # prep model for evaluation
    for data, target in valid_loader:
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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(), 'CNN_Ternary_mymodel.pt')
        valid_loss_min = valid_loss
```

In [*]:
```
model.load_state_dict(torch.load('CNN_Ternary_mymodel.pt'))
```

In [*]:
```
test_loader = DataLoader(test_data, batch_size=1,)
```

In [*]:
```
predictions = predict(model,test_loader)
predictions = np.array(predictions)
```

In [*]:
```
test_accuracy_cnn_ternary_mymodel = accuracy_score(Y_test, predictions)
```

In [*]:
```
print("Accuracy for Ternary Classification using CNN with my own model is : ", test_accuracy_cnn_ternary_mymodel)
```

In [ ]:

In [*]:
```
vectorized_data = process_reviews_cnn(balanced_df, wv)
```

In [*]:
```
X_train,X_test,Y_train, Y_test = train_test_split(vectorized_data, balanced_df['sentiment'], test_size=0.2, random_state=6
```

In [*]:
```
train_data = TrainData(X_train, Y_train)
test_data = TestData(X_test)
```

In [*]:
```
batch_size = 100
valid_size = 0.2

num_train = len(train_data)
indices = list(range(num_train))

np.random.shuffle(indices)
split = int(np.floor(valid_size * num_train))
train_idx, valid_idx = indices[split:], indices[:split]

train_sampler = SubsetRandomSampler(train_idx)
valid_sampler = SubsetRandomSampler(valid_idx)

train_loader = DataLoader(train_data, batch_size=batch_size, sampler=train_sampler)
valid_loader = DataLoader(train_data, batch_size=batch_size, sampler=valid_sampler)
```

In [*]:
```
model = CNN_Ternary()
model.to(device)
print(model)
```

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

optimizer = torch.optim.Adam(model.parameters(), lr=0.0005)
```

In [*]:
```
n_epochs = 15

valid_loss_min = np.Inf # set initial "min" to infinity

for epoch in range(n_epochs):
    train_loss = 0.0
    valid_loss = 0.0

    ###################
    # train the model #
    ###################
    model.train() # prep model for training
    for data, target in train_loader:
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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)

    ######################
    # validate the model #
    ######################
    model.eval() # prep model for evaluation
    for data, target in valid_loader:
        # forward pass: compute predicted outputs by passing inputs to the model
        data = data.to(device)
        output = model(data)
        # calculate the loss
        target = target.to(device)
        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(), 'CNN_Ternary_google.pt')
        valid_loss_min = valid_loss
```

In [*]:
```
model.load_state_dict(torch.load('CNN_Ternary_google.pt'))
```

In [*]:
```
test_loader = DataLoader(test_data, batch_size=1,)
```

In [*]:
```
predictions = predict(model,test_loader)
predictions = np.array(predictions)
```

In [*]:
```
test_accuracy_cnn_ternary_google = accuracy_score(Y_test, predictions)
```

In [*]:
```
print("Accuracy for Ternary Classification using CNN with google pretrained model is : ", test_accuracy_cnn_ternary_google)
```

In [ ]:
```

```