HW2 by Trisha Mandal

In [1]:
```
import pandas as pd
import numpy as np
import nltk
nltk.download('wordnet')
import re
from bs4 import BeautifulSoup
from textblob import TextBlob
from sklearn.model_selection import GridSearchCV
import warnings
from sklearn.model_selection import train_test_split
warnings.filterwarnings('ignore')
```

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

In [2]:
```
pip install torch
```

```
Requirement already satisfied: torch in /Users/trishamandal/opt/anaconda3/lib/
python3.9/site-packages (1.12.1)
Requirement already satisfied: typing-extensions in /Users/trishamandal/opt/an
aconda3/lib/python3.9/site-packages (from torch) (4.1.1)
Note: you may need to restart the kernel to use updated packages.
```

1. Dataset Generation

We will use the Amazon reviews dataset used in HW1. Load the dataset and build a balanced dataset of 100K reviews along with their ratings to create labels through random selection. You can store your dataset after generation and reuse it to reduce the computational load. For your experiments consider a 80%/20% training/testing split.

In [3]:
```
# reading the data
df = pd.read_table('amazon_reviews_us_Jewelry_v1_00.tsv', error_bad_lines=False
```

In [4]:
```
# selecting only reviews and ratings
data = pd.concat([df['star_rating'], df['review_body']], axis=1)
```

In [5]:
```
#converting all reviews into string
data['review_body']= [str(i) for i in data['review_body']]
```

In [6]:
```
# data cleaning on sampled data
# step 1: changing all words to lower case by using str.lower()
# step 2: performing contractions on the reviews by using contractions library
# step 3: Removing HTML by using BeatifulSoup library
# step 4: Removing URLS by using regex
# step 5: Removing non-alphanumeric characters by using regex
# step 6: Stripping extra space
# step 7: Replacing double spaces with single spaces
import contractions
```

```
data.dropna()
data['review_body'] = data['review_body'].str.lower()
# data.drop(data[data['review_body'].str.split().str.len() < 10].index, inplace
data['review_body'] = data['review_body'].apply(lambda x: contractions.fix(x))
data['review_body'] = [BeautifulSoup(text).get_text() for text in data['review_
data['review_body'] = data['review_body'].apply(lambda text: re.sub(r'www.\S+.c
data['review_body'] = data['review_body'].apply(lambda text: re.sub(r'https?:\S
data['review_body'] = data['review_body'].apply(lambda x: re.sub('\W+', ' ', x)
data["review_body"] = data["review_body"].apply(lambda x: re.sub(r"\d+", ' ', x
data["review_body"] = data["review_body"].apply(lambda x: x.strip())
data["review_body"] = data["review_body"].apply(lambda x: x.replace("  ", " "))
```

In [8]:

```
#seperating reviews from each rating
star1 = data[data['star_rating'] == 1]
star2 = data[data['star_rating'] == 2]
star3 = data[data['star_rating'] == 3]
star4 = data[data['star_rating'] == 4]
star5 = data[data['star_rating'] == 5]
star3
```

Out[8]:

	star_rating	review_body
17	3	not what i expected took too long to ship but ...
35	3	it states that the item is new but you cannot ...
40	3	the bracelet did not fit properly i had to act...
83	3	i have never been able to use these on my ring...
88	3	arrived broken one if the stones were out look...
...	...	...
1766904	3	the braclet and ring set of infinite potential...
1766930	3	well i got this necklace and come on its ctw...
1766952	3	less is best i may buy the jewelry but i am no...
1766955	3	the earrings are very pretty but they were too...
1766962	3	just wanted to let buyers know that these earr...

153665 rows × 2 columns

In [10]:

```
#calculated review length for each document and putting it in length column in
numofwords1 = star1["review_body"].apply(lambda x: len(str(x).split(' ')))
star1["length"] = pd.DataFrame(numofwords1)
numofwords2 = star2["review_body"].apply(lambda x: len(str(x).split(' ')))
star2["length"] = pd.DataFrame(numofwords2)
numofwords3 = star3["review_body"].apply(lambda x: len(str(x).split(' ')))
star3["length"] = pd.DataFrame(numofwords3)
numofwords4 = star4["review_body"].apply(lambda x: len(str(x).split(' ')))
star4["length"] = pd.DataFrame(numofwords4)
numofwords5 = star5["review_body"].apply(lambda x: len(str(x).split(' ')))
star5["length"] = pd.DataFrame(numofwords5)
star5
```

Out[10]:

	star_rating	review_body	length
0	5	so beautiful even though clearly not high end ...	32
1	5	great product i got this set for my mother as ...	72
2	5	exactly as pictured and my daughter s friend l...	32
3	5	love it fits great super comfortable and neat ...	18
4	5	got this as a mother s day gift for my mom and...	22
...	...	...	...
1766982	5	i love these earings my boyfriend got me a pai...	39
1766983	5	not too much money but would make a good impre...	10
1766985	5	i was so impressed with this piece i am a jewe...	84
1766990	5	the kt gold earrings look remarkable would def...	14
1766991	5	it will be a gift to my special friend we know...	46

1041018 rows × 3 columns

In [11]:

```
#sorting datframes in descending order by review length
star1 = star1.sort_values(by='length', ascending=False)
star2 = star2.sort_values(by='length', ascending=False)
star3 = star3.sort_values(by='length', ascending=False)
star4 = star4.sort_values(by='length', ascending=False)
star5 = star5.sort_values(by='length', ascending=False)
star1
```

Out[11]:

	star_rating	review_body	length
1648184	1.0	the entire review history is below but my most...	1489
1431879	1	the post broke off the face while trying to ge...	1139
1216795	1	i am writing a review for the three rings i pu...	1075
1757958	1	are of the sales profits of these red strings...	1035
1687894	1.0	i bought this bracelet as a christmas gift for...	952
...	...	...	...
740425	1	terrible	1
238902	1	garbage	1
743657	1	awful	1
278586	1	huge	1
884542	1	junk	1

150461 rows × 3 columns

In [12]:

```
#taking top 20K reviews from each dataframe
star1 = star1.head(20000)
star2 = star2.head(20000)
star3 = star3.head(20000)
```

```
star4 = star4.head(20000)
star5 = star5.head(20000)
star1
```

Out[12]:

	star_rating	review_body	length
1648184	1.0	the entire review history is below but my most...	1489
1431879	1	the post broke off the face while trying to ge...	1139
1216795	1	i am writing a review for the three rings i pu...	1075
1757958	1	are of the sales profits of these red strings...	1035
1687894	1.0	i bought this bracelet as a christmas gift for...	952
...	...	...	...
964376	1	i recieved this product a day earlier that is ...	65
1222447	1	i purchased this necklace to wear with an outf...	65
827380	1	i bought this because it looked like a great c...	65
1247483	1	i had bought this product based on the picture...	65
1680368	1.0	the image of the pendant is quite beautiful bu...	65

20000 rows × 3 columns

In [13]:
```
#combining all rating dataframes
combined2 = pd.concat([star1, star2, star3, star4,star5])
combined2
```

Out[13]:

	star_rating	review_body	length
1648184	1.0	the entire review history is below but my most...	1489
1431879	1	the post broke off the face while trying to ge...	1139
1216795	1	i am writing a review for the three rings i pu...	1075
1757958	1	are of the sales profits of these red strings...	1035
1687894	1.0	i bought this bracelet as a christmas gift for...	952
...	...	...	...
1596083	5.0	i bought this ring as my wedding band thinking...	131
1715843	5	this ring is my new engagement wedding ring an...	131
1318493	5	this ring is gorgeous my diamond fell out of m...	131
1593443	5.0	i adore these earrings i also ordered the coor...	131
950317	5	this like the blue hearts opal ring i am cond...	131

100000 rows × 3 columns

In [109…
```
combined2['review_body']= [str(i) for i in combined2['review_body']]
combined2['star_rating']= [int(i) for i in combined2['star_rating']]
```

In [14]:
```
# separating all reviews using ratings
```

```
rating1 = data[data['star_rating'] == 1]
rating2 = data[data['star_rating'] == 2]
rating3 = data[data['star_rating'] == 3]
rating4 = data[data['star_rating'] == 4]
# done random sampling for rating 5 since number of reviews are too large
rating5 = rating5 = data[data['star_rating'] == 5].sample(n=20000, random_state
```

In [15]:
```
# convertings reviews and ratings columns to lists for all rating categories
re1 = rating1['review_body'].values.tolist()
re2 = rating2['review_body'].values.tolist()
re3 = rating3['review_body'].values.tolist()
re4 = rating4['review_body'].values.tolist()
```

In [16]:
```
rev1 = np.array(re1)
rev2 = np.array(re2)
rev3 = np.array(re3)
rev4 = np.array(re4)
```

In [17]:
```
# applying tfidf on all ratings
from sklearn.feature_extraction.text import TfidfVectorizer
v1 = TfidfVectorizer()
v2 = TfidfVectorizer()
v3 = TfidfVectorizer()
v4 = TfidfVectorizer()
vec1 = v1.fit_transform(re1)
vec2 = v2.fit_transform(re2)
vec3 = v3.fit_transform(re3)
vec4 = v4.fit_transform(re4)
```

In [18]:
```
#Converting to numpy
num1 = list(np.squeeze(np.asarray(np.sum(vec1, axis = 1).astype(np.float32))))
num2 = list(np.squeeze(np.asarray(np.sum(vec2, axis = 1).astype(np.float32))))
num3 = list(np.squeeze(np.asarray(np.sum(vec3, axis = 1).astype(np.float32))))
num4 = list(np.squeeze(np.asarray(np.sum(vec4, axis = 1).astype(np.float32))))

num1 = np.array(num1)
num2 = np.array(num2)
num3 = np.array(num3)
num4 = np.array(num4)
```

In [19]:
```
# reshapping nparray
num1 = num1.reshape((num1.shape[0], 1))
num2 = num2.reshape((num2.shape[0], 1))
num3 = num3.reshape((num3.shape[0], 1))
num4 = num4.reshape((num4.shape[0], 1))
```

In [20]:
```
# reshapping nparray
rev1 = rev1.reshape((rev1.shape[0], 1))
rev2 = rev2.reshape((rev2.shape[0], 1))
rev3 = rev3.reshape((rev3.shape[0], 1))
rev4 = rev4.reshape((rev4.shape[0], 1))
```

In [21]:
```
# adding tfidf scores to dataframes
df1 = pd.DataFrame(np.hstack((num1, rev1)), columns = ['tfidf', 'review_body'])
df2 = pd.DataFrame(np.hstack((num2, rev2)), columns = ['tfidf', 'review_body'])
df3 = pd.DataFrame(np.hstack((num3, rev3)), columns = ['tfidf', 'review_body'])
df4 = pd.DataFrame(np.hstack((num4, rev4)), columns = ['tfidf', 'review_body'])
```

In [22]:
```
# converts tfidf vectors to numeric form
df1['tfidf'] = pd.to_numeric(df1['tfidf'])
df2['tfidf'] = pd.to_numeric(df2['tfidf'])
df3['tfidf'] = pd.to_numeric(df3['tfidf'])
df4['tfidf'] = pd.to_numeric(df4['tfidf'])
```

In [23]:
```
# getting length of reviews
df1['Length'] = df1['review_body'].str.len()
df2['Length'] = df2['review_body'].str.len()
df3['Length'] = df3['review_body'].str.len()
df4['Length'] = df4['review_body'].str.len()
```

In [24]:
```
# ranking reviews according to how many times the words have appeared in a sent
df1['rank'] = df1['tfidf']/df1['Length']
df2['rank'] = df2['tfidf']/df2['Length']
df3['rank'] = df3['tfidf']/df3['Length']
df4['rank'] = df4['tfidf']/df4['Length']
```

In [25]:
```
# sorting them according to the ranks
df1 = df1.sort_values(by='rank', ascending=False)
df2 = df2.sort_values(by='rank', ascending=False)
df3 = df3.sort_values(by='rank', ascending=False)
df4 = df4.sort_values(by='rank', ascending=False)
```

In [26]:
```
# sub sampling the highest 20000 ranks from each rating
df1 = df1.head(20000)
df2 = df2.head(20000)
df3 = df3.head(20000)
df4 = df4.head(20000)
```

In [27]:
```
# removing all unrequired columns for models in the later part of the project
df1 = pd.DataFrame(df1['review_body']).reset_index(drop=True)
df2 = pd.DataFrame(df2['review_body']).reset_index(drop=True)
df3 = pd.DataFrame(df3['review_body']).reset_index(drop=True)
df4 = pd.DataFrame(df4['review_body']).reset_index(drop=True)
```

In [28]:
```
# creating labels lists

labels1 = [1]*20000
labels2 = [2]*20000
labels3 = [3]*20000
labels4 = [4]*20000
labels5 = [5]*20000
```

In [29]:
```
# converting from list to DF
l1,l2,l3,l4,l5 = pd.DataFrame(), pd.DataFrame(), pd.DataFrame(), pd.DataFrame()
l1['star_rating'] = pd.DataFrame(labels1)
l2['star_rating'] = pd.DataFrame(labels2)
l3['star_rating'] = pd.DataFrame(labels3)
l4['star_rating']= pd.DataFrame(labels4)
l5['star_rating']= pd.DataFrame(labels5)
```

In [30]:
```
df5 = pd.DataFrame(rating5['review_body']).reset_index(drop=True)
```

In [31]:
```
# combining all dataframes
frame_combined = pd.concat([df1,df2,df3,df4,df5])
```

```
label_combined = pd.concat([l1,l2,l3,l4,l5])
combined = pd.concat([frame_combined, label_combined], axis = 1)
```

In [32]:
```
# data = pd.concat([df['star_rating'], df['review_body']], axis=1)
# data = data.dropna()
# #data.drop(data[data['review_body'].str.split().str.len() < 10].index, inplac
# data['star_rating']= [int(i) for i in data['star_rating']]
# rating1 = data[data['star_rating'] == 1].sample(n=20000, random_state = 2)
# rating2 = data[data['star_rating'] == 2].sample(n=20000, random_state = 2)
# rating3 = data[data['star_rating'] == 3].sample(n=20000, random_state = 2)
# rating4 = data[data['star_rating'] == 4].sample(n=20000, random_state = 2)
# rating5 = data[data['star_rating'] == 5].sample(n=20000, random_state = 2)
# combined = pd.concat([rating1, rating2, rating3, rating4, rating5])
# combined
```

2. Word Embedding

(a) Load the pretrained “word2vec-google-news-300” Word2Vec model and learn how to extract word embeddings for your dataset. Try to check semantic similarities of the generated vectors using three examples of your own, e.g., King – M an + W oman = Queen or excellent ~ outstanding.

In [33]:
```
# downloading all google word2vec vectors
import gensim.downloader as api
googlew2v = api.load('word2vec-google-news-300')
```

In [34]:
```
#checking similarity between the 2 words using built in function
print("Most similar word and its similarity value: ", googlew2v.most_similar(pc
```

```
Most similar word and its similarity value:  [('golden_retriever', 0.810488939
2852783)]
```

In [35]:
```
print("Similarity: ", googlew2v.similarity('fantastic', 'amazing'))
```

```
Similarity:  0.77898705
```

In [36]:
```
print("Similarity: ", googlew2v.similarity('king', 'prince'))
```

```
Similarity:  0.61599934
```

In [37]:
```
print("Top 3 similar words and their similarity values: ", googlew2v.most_simil
```

```
Top 3 similar words and their similarity values:  [('camry', 0.631081402301788
3), ('chevy', 0.6251167058944702), ('camaro', 0.6154221892356873)]
```

In [38]:
```
# example: Family - boy + girl = mother
# calculating cosine similarity and then using it to calculate similarity betwe
subtraction = googlew2v['family'] - googlew2v['boy']
addition = subtraction + googlew2v['girl']
num = np.dot(addition, googlew2v['mother'] )
denom = (np.linalg.norm(addition)* np.linalg.norm(googlew2v['mother']))
dist_google = num/denom
dist_google
```

Out[38]:
```
0.5986675
```

(b) Train a Word2Vec model using your own dataset. You will use these extracted features in the subsequent questions of this assignment. Set the embedding size to be 300 and the window size to be 11. You can also consider a minimum word count of 10. Check the semantic similarities for the same two examples in part (a). What do you conclude from comparing vectors generated by yourself and the pretrained model? Which of the Word2Vec models seems to encode semantic similarities between words better? For the rest of this assignment, use the pretrained “word2vec-googlenews-300” Word2Ve features.

In [39]:
```
# taken from reference given in hw2 pdf
from gensim import utils
from gensim.test.utils import datapath
combined['review_body']= [str(i) for i in combined['review_body']]
class MyCorpus:
    def __iter__(self):
        corpus_path = datapath('lee_background.cor')
        for line in combined['review_body']:
            yield utils.simple_preprocess(line)
```

In [40]:
```
# training own model
from gensim.models import Word2Vec
w2v = Word2Vec(sentences=MyCorpus(), vector_size=300, window=11, min_count=10)
```

In [41]:
```
#checking similarity between the 2 words using built in function
print("Most similar word and its similarity value (own Word2Vec model): ", w2v.
```

```
Most similar word and its similarity value (own Word2Vec model):  [('woman',
0.7054744362831116)]
```

In [42]:
```
print("Most similar word and its similarity value (Google news): ", googlew2v.m
```

```
Most similar word and its similarity value (Google news):  [('queen', 0.711819
3507194519)]
```

In [43]:
```
print("Similarity value (own Word2Vec model): ", w2v.wv.similarity('sun', 'moor
```

```
Similarity value (own Word2Vec model):  0.49703205
```

In [44]:
```
print("Similarity value (Google news): ", googlew2v.similarity('sun', 'moon'))
```

```
Similarity value (Google news):  0.42628342
```

In [45]:
```
# example: Family - boy + girl = mother
# calculating cosine similarity and then using it to calculate similarity betwe
subtraction = w2v.wv['family'] - w2v.wv['boy']
addition = subtraction + w2v.wv['girl']
num2 = np.dot(addition, w2v.wv['mother'] )
denom2 = (np.linalg.norm(addition)* np.linalg.norm(w2v.wv['mother']))
dist_own = num2/denom2
dist_own
```

Out[45]:
```
0.6756631
```

The similarity values for my own model are sometimes less or more than the pretrained model depending on the example but there are a lot of vocabulary missing from my own model. Moreover, the vectors of words in my own model sometimes makes no sense. The pretrained model gives better "most similar" words.

3. Simple models

Using the Google pre-trained Word2Vec features, train a perceptron and an SVM model for the five class classification problem. For this purpose, use the average Word2Vec vectors for each review as the input feature (x =1 NPN i=1 Wifor are view with N words). Report your accuracy values on the testing split for these models similar to HW1, i.e., for each of perceptron and SVM models, report two accuracy values Word2Vec and TF-IDF features.What do you conclude from comparing performances for the models trained using the two different feature types (TF-IDF and your trained

TFIDF

In [46]:
```
# converted all ratings to int
combined['star_rating']= [int(i) for i in combined['star_rating']]
```

In [47]:
```
# using TFIDF vectorization for feature extraction

vectorizer = TfidfVectorizer()
fitt = vectorizer.fit_transform(combined['review_body'])
```

In [48]:
```
# splitting the data as 80% training data and 20% testing data
xtrain_tfidf, xtest_tfidf, ytrain_tfidf, ytest_tfidf = train_test_split(fitt,cc
```

In [49]:
```
# training pretrained TFIDF vectors through Perceptron model
from sklearn.linear_model import Perceptron
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_s
tfidf_perc_mod = Perceptron(random_state=0)
```

In [50]:
```
tfidf_perc_mod.fit(xtrain_tfidf, ytrain_tfidf)
pred_perc_tfidf = tfidf_perc_mod.predict(xtest_tfidf)
```

In [51]:
```
print('Accuracy using the TFIDF(Perceptron):', accuracy_score(ytest_tfidf, prec
```

```
Accuracy using the TFIDF(Perceptron): 0.569
```

In [52]:
```
# training pretrained TFIDF vectors through LinearSVC model
from sklearn.svm import LinearSVC
tfidf_svm_mod = LinearSVC(max_iter=1000, random_state=0)
```

In [53]:
```
tfidf_svm_mod.fit(xtrain_tfidf, ytrain_tfidf)
pred_svm_tfidf = tfidf_svm_mod.predict(xtest_tfidf)
```

In [54]:
```
print('Accuracy using the TFIDF(SVM):', accuracy_score(ytest_tfidf, pred_svm_tf
```

```
Accuracy using the TFIDF(SVM): 0.65085
```

Word2Vec

In [55]:
```
# function to average Word2Vec vectors for each review and tackling NaN results
def average(x, w2v):
    count, summ, ty = 0, np.zeros(shape=(300,)), type(x)
    if ty == str:
        lst = x.split(' ')
    elif ty == list:
        lst = x
    for w in lst:
        if w in w2v:
            word = w2v[w]
            count, summ = count + 1, summ + word
    if count == 0:
        return summ
    else:
        return (summ / count)
```

In [56]:
```
#preparing Word2Vec vector for splitting
w2vxdata = combined['review_body'].apply(lambda x: average(x, googlew2v))
w2vxdata = np.array(w2vxdata.values.tolist())
w2vydata = combined['star_rating']
w2vydata = np.array(w2vydata.values.tolist())
```

In [57]:
```
# splitting the data as 80% training data and 20% testing data
xtrain_google, xtest_google, ytrain_google, ytest_google = train_test_split(w2v
```

In [58]:
```
# training pretrained Word2Vec embedding through Perceptron model
google_perc_mod = Perceptron(random_state=0)
google_perc_mod.fit(xtrain_google, ytrain_google)
```

Out[58]:
```
Perceptron()
```

In [59]:
```
pred_perc_google = google_perc_mod.predict(xtest_google)
```

In [60]:
```
print('Accuracy using the pretrained Word2Vec model(Perceptron):', accuracy_sco
```

```
Accuracy using the pretrained Word2Vec model(Perceptron): 0.4835
```

In [61]:
```
# training pretrained Word2Vec embedding through Perceptron model
google_svm_mod = LinearSVC(max_iter=1000, random_state=0)
```

In [62]:
```
google_svm_mod.fit(xtrain_google, ytrain_google)
pred_svm_google = google_svm_mod.predict(xtest_google)
```

In [63]:
```
print('Accuracy using the pretrained Word2Vec model(SVM):', accuracy_score(ytes
```

```
Accuracy using the pretrained Word2Vec model(SVM): 0.5945
```

Accuracy for the TFIDF vectorization is more than Word2Vec model

4. Feedforward Neural Networks

Using the Word2Vec features, train a feedforward multilayer perceptron network for classification. Consider a network with two hidden layers, each with 50 and 10 nodes, respectively. You can use cross entropy loss and your own choice for other hyperparamters, e.g., nonlinearity, number of epochs, etc. Part of getting good results is to select good values for these hyperparamters. You can also refer to the following tutorial to familiarize yourself: https://www.kaggle.com/mishra1993/pytorch-multi-layer-perceptron-mnist Although the above tutorial is for image data but the concept of training an MLP is very similar to what we want to do.

(a) To generate the input features, use the average Word2Vec vectors similar to the "Simple models" section and train the neural network. Report accuracy values on the testing split for your MLP.

In [64]:
```
import torch
from torch.utils.data import DataLoader, Dataset
from torch.utils.data.sampler import SubsetRandomSampler
import matplotlib.pyplot as plt
```

In [65]:
```
# custom dataset
# we have to overwrite len() and getitem() functions
class TrainDataset(Dataset):

    def __init__(self, xtrain, ytrain):
        self.data = xtrain
        self.labels = ytrain

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

    def __getitem__(self, index):
        data = self.data[index]
        label = self.labels[index]
        return data, label
```

In [66]:
```
class TestDataset(Dataset):

    def __init__(self, xtest, ytest):
        self.data = xtest
        self.labels = ytest

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

    def __getitem__(self, index):
        data = self.data[index]
        label = self.labels[index]
        return data, label
```

In [67]:
```
train_data = TrainDataset(xtrain_google, ytrain_google-1)
test_data = TestDataset(xtest_google, ytest_google-1)
```

In [68]:
```
# initialising batch_size, valid_size
num_workers, batch_size, valid_size = 0, 32, 0.2
# converting data to torch.FloatTensor
# 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]

# defining samplers for obtaining training and validation batches
train_sampler, valid_sampler = SubsetRandomSampler(train_idx), SubsetRandomSamp

# prepare data loaders
train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,sa
valid_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size, s
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size, num
```

In [69]:

```
# custom FNN
import torch.nn as nn
import torch.nn.functional as F

# define the NN architecture
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # number of hidden nodes in each layer (512)
        hidden_1 = 50
        hidden_2 = 10
        # linear layer 1
        self.fc1 = nn.Linear(300, hidden_1)
        # linear layer 2
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        # linear layer 3
        self.fc3 = nn.Linear(hidden_2, 5)
        # dropout layer (p=0.2)
        # dropout prevents overfitting of data
        self.dropout = nn.Dropout(0.1)

    def forward(self, x):
        # flatten image input
        x = x.view(-1, 300)
        # add hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer
        x = self.dropout(x)
        # add hidden layer, with relu activation function
        x = F.relu(self.fc2(x))
        # add dropout layer
        x = self.dropout(x)
        # add output layer
        x = self.fc3(x)
        return x

# initialize the NN
model = Net()
```

In [70]:

```
# loss function used
criterion = nn.CrossEntropyLoss()
# optimizer used
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

In [71]:

```
n_epochs, valid_loss_min = 100, np.Inf

for epoch in range(n_epochs):
    # monitor training loss
```

```
    train_loss, valid_loss = 0.0, 0.0

    # training 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 mode
        output = model(data.float())
        # calculate the loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model par
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update running training loss
        train_loss = train_loss + loss.item()*data.size(0)

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

    # print training/validation statistics
    # calculate average loss over an epoch
    train_loss, valid_loss  = train_loss/len(train_loader.dataset), valid_loss/
    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(

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

```
Epoch: 1 	Training Loss: 1.283530 	Validation Loss: 0.317319
Validation loss decreased (inf --> 0.317319).  Saving model ...
Epoch: 2 	Training Loss: 1.207887 	Validation Loss: 0.283512
Validation loss decreased (0.317319 --> 0.283512).  Saving model ...
Epoch: 3 	Training Loss: 1.098549 	Validation Loss: 0.259687
Validation loss decreased (0.283512 --> 0.259687).  Saving model ...
Epoch: 4 	Training Loss: 1.031042 	Validation Loss: 0.248246
Validation loss decreased (0.259687 --> 0.248246).  Saving model ...
Epoch: 5 	Training Loss: 1.000458 	Validation Loss: 0.242145
Validation loss decreased (0.248246 --> 0.242145).  Saving model ...
Epoch: 6 	Training Loss: 0.972072 	Validation Loss: 0.231757
Validation loss decreased (0.242145 --> 0.231757).  Saving model ...
Epoch: 7 	Training Loss: 0.932853 	Validation Loss: 0.222425
Validation loss decreased (0.231757 --> 0.222425).  Saving model ...
Epoch: 8 	Training Loss: 0.908597 	Validation Loss: 0.217585
Validation loss decreased (0.222425 --> 0.217585).  Saving model ...
Epoch: 9 	Training Loss: 0.891317 	Validation Loss: 0.213544
Validation loss decreased (0.217585 --> 0.213544).  Saving model ...
Epoch: 10 	Training Loss: 0.877228 	Validation Loss: 0.209856
Validation loss decreased (0.213544 --> 0.209856).  Saving model ...
Epoch: 11 	Training Loss: 0.862411 	Validation Loss: 0.206636
Validation loss decreased (0.209856 --> 0.206636).  Saving model ...
Epoch: 12 	Training Loss: 0.852515 	Validation Loss: 0.203664
Validation loss decreased (0.206636 --> 0.203664).  Saving model ...
Epoch: 13 	Training Loss: 0.842260 	Validation Loss: 0.201181
Validation loss decreased (0.203664 --> 0.201181).  Saving model ...
Epoch: 14 	Training Loss: 0.830667 	Validation Loss: 0.198034
Validation loss decreased (0.201181 --> 0.198034).  Saving model ...
Epoch: 15 	Training Loss: 0.821941 	Validation Loss: 0.195727
Validation loss decreased (0.198034 --> 0.195727).  Saving model ...
Epoch: 16 	Training Loss: 0.811856 	Validation Loss: 0.193285
Validation loss decreased (0.195727 --> 0.193285).  Saving model ...
Epoch: 17 	Training Loss: 0.804343 	Validation Loss: 0.191529
Validation loss decreased (0.193285 --> 0.191529).  Saving model ...
Epoch: 18 	Training Loss: 0.798134 	Validation Loss: 0.189489
Validation loss decreased (0.191529 --> 0.189489).  Saving model ...
Epoch: 19 	Training Loss: 0.789906 	Validation Loss: 0.187667
Validation loss decreased (0.189489 --> 0.187667).  Saving model ...
Epoch: 20 	Training Loss: 0.786103 	Validation Loss: 0.186551
Validation loss decreased (0.187667 --> 0.186551).  Saving model ...
Epoch: 21 	Training Loss: 0.778144 	Validation Loss: 0.185078
Validation loss decreased (0.186551 --> 0.185078).  Saving model ...
Epoch: 22 	Training Loss: 0.774564 	Validation Loss: 0.183956
Validation loss decreased (0.185078 --> 0.183956).  Saving model ...
Epoch: 23 	Training Loss: 0.768200 	Validation Loss: 0.182965
Validation loss decreased (0.183956 --> 0.182965).  Saving model ...
Epoch: 24 	Training Loss: 0.764111 	Validation Loss: 0.182040
Validation loss decreased (0.182965 --> 0.182040).  Saving model ...
Epoch: 25 	Training Loss: 0.761840 	Validation Loss: 0.181428
Validation loss decreased (0.182040 --> 0.181428).  Saving model ...
Epoch: 26 	Training Loss: 0.757878 	Validation Loss: 0.180491
Validation loss decreased (0.181428 --> 0.180491).  Saving model ...
Epoch: 27 	Training Loss: 0.753142 	Validation Loss: 0.179846
Validation loss decreased (0.180491 --> 0.179846).  Saving model ...
Epoch: 28 	Training Loss: 0.749955 	Validation Loss: 0.178730
Validation loss decreased (0.179846 --> 0.178730).  Saving model ...
Epoch: 29 	Training Loss: 0.747823 	Validation Loss: 0.178299
Validation loss decreased (0.178730 --> 0.178299).  Saving model ...
Epoch: 30 	Training Loss: 0.744259 	Validation Loss: 0.177401
Validation loss decreased (0.178299 --> 0.177401).  Saving model ...
```

```
Epoch: 31 	Training Loss: 0.741140 	Validation Loss: 0.176680
Validation loss decreased (0.177401 --> 0.176680).  Saving model ...
Epoch: 32 	Training Loss: 0.739769 	Validation Loss: 0.176523
Validation loss decreased (0.176680 --> 0.176523).  Saving model ...
Epoch: 33 	Training Loss: 0.736456 	Validation Loss: 0.175965
Validation loss decreased (0.176523 --> 0.175965).  Saving model ...
Epoch: 34 	Training Loss: 0.735313 	Validation Loss: 0.175633
Validation loss decreased (0.175965 --> 0.175633).  Saving model ...
Epoch: 35 	Training Loss: 0.732908 	Validation Loss: 0.174756
Validation loss decreased (0.175633 --> 0.174756).  Saving model ...
Epoch: 36 	Training Loss: 0.728682 	Validation Loss: 0.174300
Validation loss decreased (0.174756 --> 0.174300).  Saving model ...
Epoch: 37 	Training Loss: 0.727095 	Validation Loss: 0.174149
Validation loss decreased (0.174300 --> 0.174149).  Saving model ...
Epoch: 38 	Training Loss: 0.725503 	Validation Loss: 0.173683
Validation loss decreased (0.174149 --> 0.173683).  Saving model ...
Epoch: 39 	Training Loss: 0.724420 	Validation Loss: 0.173942
Epoch: 40 	Training Loss: 0.721939 	Validation Loss: 0.173747
Epoch: 41 	Training Loss: 0.720678 	Validation Loss: 0.172963
Validation loss decreased (0.173683 --> 0.172963).  Saving model ...
Epoch: 42 	Training Loss: 0.719394 	Validation Loss: 0.172775
Validation loss decreased (0.172963 --> 0.172775).  Saving model ...
Epoch: 43 	Training Loss: 0.717943 	Validation Loss: 0.173257
Epoch: 44 	Training Loss: 0.716050 	Validation Loss: 0.172351
Validation loss decreased (0.172775 --> 0.172351).  Saving model ...
Epoch: 45 	Training Loss: 0.714429 	Validation Loss: 0.172020
Validation loss decreased (0.172351 --> 0.172020).  Saving model ...
Epoch: 46 	Training Loss: 0.712585 	Validation Loss: 0.172018
Validation loss decreased (0.172020 --> 0.172018).  Saving model ...
Epoch: 47 	Training Loss: 0.710972 	Validation Loss: 0.171357
Validation loss decreased (0.172018 --> 0.171357).  Saving model ...
Epoch: 48 	Training Loss: 0.709417 	Validation Loss: 0.171162
Validation loss decreased (0.171357 --> 0.171162).  Saving model ...
Epoch: 49 	Training Loss: 0.708836 	Validation Loss: 0.171316
Epoch: 50 	Training Loss: 0.706041 	Validation Loss: 0.171605
Epoch: 51 	Training Loss: 0.704654 	Validation Loss: 0.170363
Validation loss decreased (0.171162 --> 0.170363).  Saving model ...
Epoch: 52 	Training Loss: 0.704432 	Validation Loss: 0.170718
Epoch: 53 	Training Loss: 0.704036 	Validation Loss: 0.170554
Epoch: 54 	Training Loss: 0.700363 	Validation Loss: 0.170224
Validation loss decreased (0.170363 --> 0.170224).  Saving model ...
Epoch: 55 	Training Loss: 0.700776 	Validation Loss: 0.170178
Validation loss decreased (0.170224 --> 0.170178).  Saving model ...
Epoch: 56 	Training Loss: 0.699606 	Validation Loss: 0.170063
Validation loss decreased (0.170178 --> 0.170063).  Saving model ...
Epoch: 57 	Training Loss: 0.699103 	Validation Loss: 0.169592
Validation loss decreased (0.170063 --> 0.169592).  Saving model ...
Epoch: 58 	Training Loss: 0.698719 	Validation Loss: 0.170323
Epoch: 59 	Training Loss: 0.697625 	Validation Loss: 0.169342
Validation loss decreased (0.169592 --> 0.169342).  Saving model ...
Epoch: 60 	Training Loss: 0.695581 	Validation Loss: 0.169525
Epoch: 61 	Training Loss: 0.692989 	Validation Loss: 0.168990
Validation loss decreased (0.169342 --> 0.168990).  Saving model ...
Epoch: 62 	Training Loss: 0.694320 	Validation Loss: 0.168724
Validation loss decreased (0.168990 --> 0.168724).  Saving model ...
Epoch: 63 	Training Loss: 0.692491 	Validation Loss: 0.168858
Epoch: 64 	Training Loss: 0.692057 	Validation Loss: 0.168538
Validation loss decreased (0.168724 --> 0.168538).  Saving model ...
Epoch: 65 	Training Loss: 0.691134 	Validation Loss: 0.168617
Epoch: 66 	Training Loss: 0.689283 	Validation Loss: 0.168328
```

```
Validation loss decreased (0.168538 --> 0.168328).  Saving model ...
Epoch: 67       Training Loss: 0.686578         Validation Loss: 0.169276
Epoch: 68       Training Loss: 0.689837         Validation Loss: 0.167913
Validation loss decreased (0.168328 --> 0.167913).  Saving model ...
Epoch: 69       Training Loss: 0.687044         Validation Loss: 0.168103
Epoch: 70       Training Loss: 0.686801         Validation Loss: 0.168339
Epoch: 71       Training Loss: 0.686725         Validation Loss: 0.168357
Epoch: 72       Training Loss: 0.684611         Validation Loss: 0.168353
Epoch: 73       Training Loss: 0.684309         Validation Loss: 0.168199
Epoch: 74       Training Loss: 0.683142         Validation Loss: 0.167983
Epoch: 75       Training Loss: 0.680762         Validation Loss: 0.168337
Epoch: 76       Training Loss: 0.680892         Validation Loss: 0.167249
Validation loss decreased (0.167913 --> 0.167249).  Saving model ...
Epoch: 77       Training Loss: 0.680362         Validation Loss: 0.167816
Epoch: 78       Training Loss: 0.680028         Validation Loss: 0.167316
Epoch: 79       Training Loss: 0.679358         Validation Loss: 0.167248
Validation loss decreased (0.167249 --> 0.167248).  Saving model ...
Epoch: 80       Training Loss: 0.678520         Validation Loss: 0.167224
Validation loss decreased (0.167248 --> 0.167224).  Saving model ...
Epoch: 81       Training Loss: 0.677902         Validation Loss: 0.167936
Epoch: 82       Training Loss: 0.677249         Validation Loss: 0.167007
Validation loss decreased (0.167224 --> 0.167007).  Saving model ...
Epoch: 83       Training Loss: 0.675521         Validation Loss: 0.168096
Epoch: 84       Training Loss: 0.675564         Validation Loss: 0.167307
Epoch: 85       Training Loss: 0.674490         Validation Loss: 0.166934
Validation loss decreased (0.167007 --> 0.166934).  Saving model ...
Epoch: 86       Training Loss: 0.672709         Validation Loss: 0.167865
Epoch: 87       Training Loss: 0.673517         Validation Loss: 0.166742
Validation loss decreased (0.166934 --> 0.166742).  Saving model ...
Epoch: 88       Training Loss: 0.674649         Validation Loss: 0.167214
Epoch: 89       Training Loss: 0.673019         Validation Loss: 0.168255
Epoch: 90       Training Loss: 0.671031         Validation Loss: 0.166525
Validation loss decreased (0.166742 --> 0.166525).  Saving model ...
Epoch: 91       Training Loss: 0.670820         Validation Loss: 0.167938
Epoch: 92       Training Loss: 0.670605         Validation Loss: 0.166804
Epoch: 93       Training Loss: 0.668151         Validation Loss: 0.166818
Epoch: 94       Training Loss: 0.668928         Validation Loss: 0.167139
Epoch: 95       Training Loss: 0.668372         Validation Loss: 0.168603
Epoch: 96       Training Loss: 0.666899         Validation Loss: 0.167352
Epoch: 97       Training Loss: 0.666013         Validation Loss: 0.166048
Validation loss decreased (0.166525 --> 0.166048).  Saving model ...
Epoch: 98       Training Loss: 0.665806         Validation Loss: 0.169283
Epoch: 99       Training Loss: 0.666535         Validation Loss: 0.166890
Epoch: 100      Training Loss: 0.664908         Validation Loss: 0.166193
```

In [72]:

```
# loading the model with lowest validation loss
model.load_state_dict(torch.load('model.pt'))
```

Out[72]:

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

In [73]:

```
correct, total = 0, 0
# since we're not training, we don't need to calculate the gradients for out ou
with torch.no_grad():
    for data in test_loader:
        embeddings, labels = data
        # calculating outputs by running embeddings through the network
        model.to("cpu")
        outputs = model(embeddings.float())
        # the class with the highest score is what we choose as prediction
```

```
        _, predicted = torch.max(outputs.data, 1)
        total = total + labels.size(0)
        correct = correct + (predicted == labels).sum().item()
```

In [74]:
```
print('Accuracy for Average Word2Vec vectors FNN model: ', (100 * correct / tot
```

```
Accuracy for Average Word2Vec vectors FNN model:  66.3
```

(b)To generate the input features, concatenate the first 10 Word2Vec vectors for each review as the input feature (x = [WT 1, ..., WT 10]) and train the neural network. Report the accuracy value on the testing split for your MLP model. What do you conclude by comparing accuracy values you obtain with those obtained in the "'Simple Models" section.

In [85]:
```
# Function to generate the input features, concatenate the first 10 Word2Vec ve
def concatenation(s, w2v):
    a,b = 0,0
    if type(s) == list:
        lst = s
    if type(s) == str:
        lst = s.split(' ')
    leng = len(lst)
    while (a < leng) & (b < 10):
        if lst[a] in w2v:
            wv = w2v[lst[a]]
            if b != 0:
                res = np.concatenate((res, wv))
            else:
                res = wv
            a, b = a + 1, b + 1
            #print("a: ",a)
            #print("b: ",b)
        else:
            a = a + 1
    #print("b final:", b)
    if b < 10:
        if (10 - b) != 10:
            res = np.concatenate((res, np.zeros(shape=(300*(10 - b), ))))
        else:
            res = np.zeros(shape=(300*(10 - b), ))
    return res
```

In [86]:
```
# preparing the Word2Vec vectors for splitting
w2vx10data = combined['review_body'].apply(lambda x: concatenation(x, googlew2v
w2vx10data = np.array(w2vx10data.values.tolist())
w2vy10data = combined['star_rating']
w2vy10data = np.array(w2vy10data.values.tolist())
```

In [87]:
```
# splitting the data as 80% training data and 20% testing data
xtrain10_google, xtest10_google, ytrain10_google, ytest10_google = train_test_s
```

In [88]:
```
# custom dataset
class TrainDataset(Dataset):

    def __init__(self, xtrain, ytrain):
        self.data = xtrain
        self.labels = ytrain
```

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

    def __getitem__(self, index):
        data = self.data[index]
        label = self.labels[index]
        return data, label
```

In [89]:
```
class TestDataset(Dataset):

    def __init__(self, xtest, ytest):
        self.data = xtest
        self.labels = ytest

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

    def __getitem__(self, index):
        data = self.data[index]
        label = self.labels[index]
        return data, label
```

In [90]:
```
train10_data = TrainDataset(xtrain10_google, ytrain10_google-1)
test10_data = TestDataset(xtest10_google, ytest10_google-1)
```

In [91]:
```
num_workers, batch_size, valid_size  = 0, 32, 0.2

# obtain training indices that will be used for validation
num_train = len(train10_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, valid_sampler = SubsetRandomSampler(train_idx), SubsetRandomSamp

# prepare data loaders
train_loader = torch.utils.data.DataLoader(train10_data, batch_size=batch_size,
valid_loader = torch.utils.data.DataLoader(train10_data, batch_size=batch_size,
test_loader = torch.utils.data.DataLoader(test10_data, batch_size=batch_size, r
```

In [92]:
```
# custom FNN
import torch.nn as nn
import torch.nn.functional as F

# define the FNN architecture
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # number of hidden nodes in each layer (512)
        hidden_1 = 50
        hidden_2 = 10
        # linear layer 1
        self.fc1 = nn.Linear(3000, hidden_1)
        # linear layer 2
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        # linear layer 3
```

```
        self.fc3 = nn.Linear(hidden_2, 5)
        # dropout layer (p=0.2)
        # dropout prevents overfitting of data
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        # flatten image input
        x = x.view(-1, 3000)
        # add hidden layer, with relu activation function
        x = F.relu(self.fc1(x))
        # add dropout layer
        x = self.dropout(x)
        # add hidden layer, with relu activation function
        x = F.relu(self.fc2(x))
        # add dropout layer
        x = self.dropout(x)
        # add output layer
        x = self.fc3(x)
        return x

# initialize the NN
model10 = Net()
```

In [93]:

```
# loss function used
criterion = nn.CrossEntropyLoss()
# optimizer used
optimizer = torch.optim.SGD(model10.parameters(), lr=0.0075)
```

In [94]:

```
n_epochs, valid_loss_min = 30, np.Inf

for epoch in range(n_epochs):
    # monitor training loss
    train_loss, valid_loss = 0.0, 0.0
    # training model
    model10.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 mode
        output = model10(data.float())
        # calculate the loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model pa
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update running training loss
        train_loss = train_loss + loss.item()*data.size(0)

    # validating model
    model10.eval() # prep model for evaluation
    for data, target in valid_loader:
        # forward pass: compute predicted outputs by passing inputs to the mode
        output = model10(data.float())
        # calculate the loss
        loss = criterion(output, target)
        # update running validation loss
        valid_loss = valid_loss + loss.item()*data.size(0)
```

```
    # print training/validation statistics
    # calculate average loss over an epoch
    train_loss, valid_loss = train_loss/len(train_loader.dataset), valid_loss/l
    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(

    # save model if validation loss has decreased
    if valid_loss <= valid_loss_min:
        print('Validation loss decreased ({:.6f} --> {:.6f}).  Saving model ...
        torch.save(model10.state_dict(), 'model10.pt')
        valid_loss_min = valid_loss
```

```
Epoch: 1 	Training Loss: 1.260506 	Validation Loss: 0.296781
Validation loss decreased (inf --> 0.296781).  Saving model ...
Epoch: 2 	Training Loss: 1.044644 	Validation Loss: 0.220601
Validation loss decreased (0.296781 --> 0.220601).  Saving model ...
Epoch: 3 	Training Loss: 0.896868 	Validation Loss: 0.200710
Validation loss decreased (0.220601 --> 0.200710).  Saving model ...
Epoch: 4 	Training Loss: 0.845342 	Validation Loss: 0.192679
Validation loss decreased (0.200710 --> 0.192679).  Saving model ...
Epoch: 5 	Training Loss: 0.813451 	Validation Loss: 0.186653
Validation loss decreased (0.192679 --> 0.186653).  Saving model ...
Epoch: 6 	Training Loss: 0.790590 	Validation Loss: 0.182170
Validation loss decreased (0.186653 --> 0.182170).  Saving model ...
Epoch: 7 	Training Loss: 0.775066 	Validation Loss: 0.179425
Validation loss decreased (0.182170 --> 0.179425).  Saving model ...
Epoch: 8 	Training Loss: 0.761997 	Validation Loss: 0.177023
Validation loss decreased (0.179425 --> 0.177023).  Saving model ...
Epoch: 9 	Training Loss: 0.750059 	Validation Loss: 0.175439
Validation loss decreased (0.177023 --> 0.175439).  Saving model ...
Epoch: 10 	Training Loss: 0.739694 	Validation Loss: 0.174131
Validation loss decreased (0.175439 --> 0.174131).  Saving model ...
Epoch: 11 	Training Loss: 0.731407 	Validation Loss: 0.173073
Validation loss decreased (0.174131 --> 0.173073).  Saving model ...
Epoch: 12 	Training Loss: 0.722393 	Validation Loss: 0.171998
Validation loss decreased (0.173073 --> 0.171998).  Saving model ...
Epoch: 13 	Training Loss: 0.716686 	Validation Loss: 0.170993
Validation loss decreased (0.171998 --> 0.170993).  Saving model ...
Epoch: 14 	Training Loss: 0.708511 	Validation Loss: 0.170373
Validation loss decreased (0.170993 --> 0.170373).  Saving model ...
Epoch: 15 	Training Loss: 0.701397 	Validation Loss: 0.169133
Validation loss decreased (0.170373 --> 0.169133).  Saving model ...
Epoch: 16 	Training Loss: 0.696935 	Validation Loss: 0.168920
Validation loss decreased (0.169133 --> 0.168920).  Saving model ...
Epoch: 17 	Training Loss: 0.691269 	Validation Loss: 0.168499
Validation loss decreased (0.168920 --> 0.168499).  Saving model ...
Epoch: 18 	Training Loss: 0.684719 	Validation Loss: 0.167780
Validation loss decreased (0.168499 --> 0.167780).  Saving model ...
Epoch: 19 	Training Loss: 0.679013 	Validation Loss: 0.168175
Epoch: 20 	Training Loss: 0.673193 	Validation Loss: 0.167153
Validation loss decreased (0.167780 --> 0.167153).  Saving model ...
Epoch: 21 	Training Loss: 0.665874 	Validation Loss: 0.166948
Validation loss decreased (0.167153 --> 0.166948).  Saving model ...
Epoch: 22 	Training Loss: 0.662770 	Validation Loss: 0.166680
Validation loss decreased (0.166948 --> 0.166680).  Saving model ...
Epoch: 23 	Training Loss: 0.658224 	Validation Loss: 0.166985
Epoch: 24 	Training Loss: 0.653228 	Validation Loss: 0.166784
Epoch: 25 	Training Loss: 0.649802 	Validation Loss: 0.166879
Epoch: 26 	Training Loss: 0.646602 	Validation Loss: 0.166523
Validation loss decreased (0.166680 --> 0.166523).  Saving model ...
Epoch: 27 	Training Loss: 0.640350 	Validation Loss: 0.166935
Epoch: 28 	Training Loss: 0.635417 	Validation Loss: 0.166949
Epoch: 29 	Training Loss: 0.631338 	Validation Loss: 0.166895
Epoch: 30 	Training Loss: 0.629316 	Validation Loss: 0.166741
```

In [95]:
```
# loading the model with lowest validation loss
model10.load_state_dict(torch.load('model10.pt'))
```

Out[95]:
```
<All keys matched successfully>
```

In [96]:
```
correct, total = 0, 0
```

```
# since we're not training, we don't need to calculate the gradients for out ou
with torch.no_grad():
    for data in test_loader:
        embeddings, labels = data
        # calculating outputs by running embeddings through the network
        model10.to("cpu")
        outputs = model10(embeddings.float())
        # the class with the highest score is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total = total + labels.size(0)
        correct = correct + (predicted == labels).sum().item()
```

In [97]:
```
print('Accuracy for 10 Word2Vec vectors FNN model: ', (100 * correct / total))
```

```
Accuracy for 10 Word2Vec vectors FNN model:  67.015
```

5. Recurrent Neural Networks

Using the Word2Vec features, train a recurrent neural network (RNN) for classification. You can refer to the following tutorial to familiarize yourself: https://pytorch.org/tutorials/intermediate/char_rnn_classification_ tutorial.html (a) Train a simple RNN for sentiment analysis. You can consider an RNN cell with the hidden state size of 20. To feed your data into our RNN, limit the maximum review length to 20 by truncating longer reviews and padding shorter reviews with a null value (0). Report accuracy values on the testing split for your RNN model. What do you conclude by comparing accuracy values you obtain with those obtained with feedforward neural network models.

In [131…
```
# function to feed your data into our RNN, limit the maximum review length to 2
#by truncating longer reviews and padding shorter reviews with a null value (0)
def review20(s, rnnw2v):
    a, b = 0, 0
    if type(s) == str:
        lst = s.split(' ')
    elif type(s) == list:
        lst = s
    leng = len(lst)
    while (b < 20) & (a < leng):
        if lst[a] in rnnw2v:
            if b != 0:
                w2v = rnnw2v[lst[a]]
                w2vm = np.vstack((w2vm, w2v))
            else:
                w2vm = rnnw2v[lst[a]]
            a, b = a + 1, b + 1
        else:
            a = a + 1
    if b != 0:
        zeros = np.zeros(shape=(20-b, 300))
        w2vm = np.vstack((w2vm, zeros))
    if b < 20:
        w2vm = np.zeros(shape=(20, 300))
    return w2vm
```

In [132…
```
w2vxdatarnn = combined2['review_body'].apply(lambda x: review20(x, googlew2v))
```

```
w2vydatarnn = combined2['star_rating']
w2vxdatarnn = np.array(w2vxdatarnn.values.tolist())
w2vydatarnn = np.array(w2vydatarnn.values.tolist())
```

In [133…

```
# splitting data into 80% training and 20% testing
xtrainrnn_google, xtestrnn_google, ytrainrnn_google, ytestrnn_google = train_te
```

In [134…

```
# custom dataset
class TrainDataset(Dataset):

    def __init__(self, xtrain, ytrain):
        self.data = xtrain
        self.labels = ytrain

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

    def __getitem__(self, index):
        data = self.data[index]
        label = self.labels[index]

        return data, label
```

In [135…

```
class TestDataset(Dataset):

    def __init__(self, xtest, ytest):
        self.data = xtest
        self.labels = ytest

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

    def __getitem__(self, index):
        data = self.data[index]
        label = self.labels[index]

        return data, label
```

In [136…

```
train_datarnn = TrainDataset(xtrainrnn_google, ytrainrnn_google-1)
test_datarnn = TestDataset(xtestrnn_google, ytestrnn_google-1)
```

In [137…

```
num_workers, batch_size, valid_size = 0, 32, 0.2
# convert data to torch.FloatTensor

# obtain training indices that will be used for validation
num_train = len(train_datarnn)
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, valid_sampler = SubsetRandomSampler(train_idx), SubsetRandomSamp

# prepare data loaders
train_loader_rnn = torch.utils.data.DataLoader(train_datarnn, batch_size=batch_
valid_loader_rnn = torch.utils.data.DataLoader(train_datarnn, batch_size=batch_
test_loader_rnn = torch.utils.data.DataLoader(test_datarnn, batch_size=batch_si
```

In [138…

```
# custon RNN model
class RNNModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
        super().__init__()

        # Number of hidden dimensions
        self.hidden_dim = hidden_dim
        # Number of hidden layers
        self.layer_dim = layer_dim
        # RNN
        self.rnn = nn.RNN(input_dim, hidden_dim, layer_dim, batch_first=True, r
        # Output layer
        self.fc = nn.Linear(hidden_dim, output_dim)

        self.softmax = nn.LogSoftmax(dim = 1)

    def forward(self, x):

        # Initialize hidden state with zeros
        h0 = torch.zeros(self.layer_dim, x.size(0), self.hidden_dim)
        # One time step
        out, hn = self.rnn(x, h0)
        out = self.fc(out[:, -1, :])
        out = self.softmax(out)
        return out

# initialize RNN
model_rnn = RNNModel(300, 20, 1, 5)
```

In [139…

```
criterion = nn.NLLLoss()
optimizer_rnn = torch.optim.Adam(model_rnn.parameters(), lr=0.01)
```

In [140…

```
# number of epochs to train the model
n_epochs, valid_loss_min = 30, np.Inf

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

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

    #validating model
    model_rnn.eval() # prep model for evaluation
    for data, target in valid_loader_rnn:
```

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


    # print training/validation statistics
    # calculate average loss over an epoch
    train_loss, valid_loss = train_loss/len(train_loader_rnn.dataset), valid_lo
    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(

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

```
Epoch: 1        Training Loss: 1.283001        Validation Loss: 0.317449
Validation loss decreased (inf --> 0.317449).  Saving model ...
Epoch: 2        Training Loss: 1.252116        Validation Loss: 0.307635
Validation loss decreased (0.317449 --> 0.307635).  Saving model ...
Epoch: 3        Training Loss: 1.217740        Validation Loss: 0.306188
Validation loss decreased (0.307635 --> 0.306188).  Saving model ...
Epoch: 4        Training Loss: 1.199817        Validation Loss: 0.300398
Validation loss decreased (0.306188 --> 0.300398).  Saving model ...
Epoch: 5        Training Loss: 1.198098        Validation Loss: 0.297399
Validation loss decreased (0.300398 --> 0.297399).  Saving model ...
Epoch: 6        Training Loss: 1.189712        Validation Loss: 0.299164
Epoch: 7        Training Loss: 1.192799        Validation Loss: 0.297447
Epoch: 8        Training Loss: 1.189216        Validation Loss: 0.296797
Validation loss decreased (0.297399 --> 0.296797).  Saving model ...
Epoch: 9        Training Loss: 1.184553        Validation Loss: 0.294477
Validation loss decreased (0.296797 --> 0.294477).  Saving model ...
Epoch: 10       Training Loss: 1.194137        Validation Loss: 0.295632
Epoch: 11       Training Loss: 1.184758        Validation Loss: 0.298907
Epoch: 12       Training Loss: 1.180209        Validation Loss: 0.297092
Epoch: 13       Training Loss: 1.180336        Validation Loss: 0.295236
Epoch: 14       Training Loss: 1.177964        Validation Loss: 0.303544
Epoch: 15       Training Loss: 1.176563        Validation Loss: 0.296384
Epoch: 16       Training Loss: 1.177640        Validation Loss: 0.300024
Epoch: 17       Training Loss: 1.175896        Validation Loss: 0.299347
Epoch: 18       Training Loss: 1.177329        Validation Loss: 0.310685
Epoch: 19       Training Loss: 1.172469        Validation Loss: 0.295979
Epoch: 20       Training Loss: 1.176689        Validation Loss: 0.294404
Validation loss decreased (0.294477 --> 0.294404).  Saving model ...
Epoch: 21       Training Loss: 1.175606        Validation Loss: 0.301625
Epoch: 22       Training Loss: 1.183474        Validation Loss: 0.314457
Epoch: 23       Training Loss: 1.197302        Validation Loss: 0.304831
Epoch: 24       Training Loss: 1.175817        Validation Loss: 0.299629
Epoch: 25       Training Loss: 1.172117        Validation Loss: 0.294882
Epoch: 26       Training Loss: 1.184060        Validation Loss: 0.298823
Epoch: 27       Training Loss: 1.171910        Validation Loss: 0.294755
Epoch: 28       Training Loss: 1.172655        Validation Loss: 0.298792
Epoch: 29       Training Loss: 1.170841        Validation Loss: 0.296271
Epoch: 30       Training Loss: 1.172918        Validation Loss: 0.297978
```

In [141…

```
# loading the model with lowest validation loss
model_rnn.load_state_dict(torch.load('model.pt'))
```

Out[141]: <All keys matched successfully>

In [142…
```
correct, total = 0, 0
# since we're not training, we don't need to calculate the gradients for out ou
with torch.no_grad():
    for data in test_loader_rnn:
        embeddings, labels = data
        # calculating outputs by running embeddings through the network
        model_rnn.to("cpu")
        outputs = model_rnn(embeddings.float())
        # the class with the highest score is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total = total + labels.size(0)
        correct = correct + (predicted == labels).sum().item()
```

In [143…
```
print('Accuracy for RNN: ', (100 * correct / total))
```

```
Accuracy for RNN:  33.175
```

(b) Repeat part (a) by considering a gated recurrent unit cell. What do you conclude by comparing accuracy values you obtain with those obtained using simple RNN.

In [124…
```
num_workers, batch_size, valid_size = 0, 32, 0.2

# convert data to torch.FloatTensor
# obtain training indices that will be used for validation
num_train = len(train_datarnn)
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,valid_sampler = SubsetRandomSampler(train_idx), SubsetRandomSampl
# prepare data loaders
train_loader_rnn = torch.utils.data.DataLoader(train_datarnn, batch_size=batch_
valid_loader_rnn = torch.utils.data.DataLoader(train_datarnn, batch_size=batch_
test_loader_rnn = torch.utils.data.DataLoader(test_datarnn, batch_size=batch_si
```

In [125…
```
# custom GatedRNN
class GatedRNN(nn.Module):
    def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
        super().__init__()

        # Number of hidden dimensions
        self.hidden_dim = hidden_dim
        # Number of hidden layers
        self.layer_dim = layer_dim
        # GRU
        self.gru = nn.GRU(input_dim, hidden_dim, layer_dim, batch_first=True)
        # Output layer
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.softmax = nn.LogSoftmax(dim = 1)

    def forward(self, x):

        # Initialize hidden state with zeros
        h0 = torch.zeros(self.layer_dim, x.size(0), self.hidden_dim)
```

```
        # One time step
        out, hn = self.gru(x, h0)
        out = self.fc(out[:, -1, :])
        out = self.softmax(out)
        return out

model_gatedrnn = GatedRNN(300, 20, 1, 5)
```

In [126…

```
# loss function used
criterion = nn.CrossEntropyLoss()
#optimizer used for gradient descent
optimizer_rnn = torch.optim.SGD(model_gatedrnn.parameters(), lr=0.0075)
```

In [127…

```
# number of epochs to train the model
n_epochs, valid_loss_min = 50, np.Inf

for epoch in range(n_epochs):
  # monitor training loss
    train_loss, valid_loss = 0.0, 0.0
    #training model
    model_gatedrnn.train() # prep model for training
    for data, target in train_loader_rnn:
        # clear the gradients of all optimized variables
        optimizer_rnn.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the mode
        output = model_gatedrnn(data.float())
        # calculate the loss
        loss = criterion(output, target)
        # backward pass: compute gradient of the loss with respect to model pa
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer_rnn.step()
        # update running training loss
        train_loss = train_loss + loss.item()*data.size(0)

    # validating model
    model_gatedrnn.eval() # prep model for evaluation
    for data, target in valid_loader_rnn:
        # forward pass: compute predicted outputs by passing inputs to the mode
        output = model_gatedrnn(data.float())
        # calculate the loss
        loss = criterion(output, target)
        # update running validation loss
        valid_loss = valid_loss + loss.item()*data.size(0)


    # print training/validation statistics
    # calculate average loss over an epoch
    train_loss, valid_loss = train_loss/len(train_loader_rnn.dataset), valid_lo
    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f}'.format(

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

```
Epoch: 1 	Training Loss: 1.289548 	Validation Loss: 0.321773
Validation loss decreased (inf --> 0.321773).  Saving model ...
Epoch: 2 	Training Loss: 1.285296 	Validation Loss: 0.321020
Validation loss decreased (0.321773 --> 0.321020).  Saving model ...
Epoch: 3 	Training Loss: 1.281657 	Validation Loss: 0.320157
Validation loss decreased (0.321020 --> 0.320157).  Saving model ...
Epoch: 4 	Training Loss: 1.277183 	Validation Loss: 0.319084
Validation loss decreased (0.320157 --> 0.319084).  Saving model ...
Epoch: 5 	Training Loss: 1.270888 	Validation Loss: 0.317087
Validation loss decreased (0.319084 --> 0.317087).  Saving model ...
Epoch: 6 	Training Loss: 1.257829 	Validation Loss: 0.311675
Validation loss decreased (0.317087 --> 0.311675).  Saving model ...
Epoch: 7 	Training Loss: 1.232900 	Validation Loss: 0.305766
Validation loss decreased (0.311675 --> 0.305766).  Saving model ...
Epoch: 8 	Training Loss: 1.217541 	Validation Loss: 0.303540
Validation loss decreased (0.305766 --> 0.303540).  Saving model ...
Epoch: 9 	Training Loss: 1.207840 	Validation Loss: 0.300587
Validation loss decreased (0.303540 --> 0.300587).  Saving model ...
Epoch: 10 	Training Loss: 1.199622 	Validation Loss: 0.298878
Validation loss decreased (0.300587 --> 0.298878).  Saving model ...
Epoch: 11 	Training Loss: 1.192036 	Validation Loss: 0.296773
Validation loss decreased (0.298878 --> 0.296773).  Saving model ...
Epoch: 12 	Training Loss: 1.184700 	Validation Loss: 0.296729
Validation loss decreased (0.296773 --> 0.296729).  Saving model ...
Epoch: 13 	Training Loss: 1.176564 	Validation Loss: 0.292547
Validation loss decreased (0.296729 --> 0.292547).  Saving model ...
Epoch: 14 	Training Loss: 1.168211 	Validation Loss: 0.291987
Validation loss decreased (0.292547 --> 0.291987).  Saving model ...
Epoch: 15 	Training Loss: 1.161736 	Validation Loss: 0.289605
Validation loss decreased (0.291987 --> 0.289605).  Saving model ...
Epoch: 16 	Training Loss: 1.156584 	Validation Loss: 0.292754
Epoch: 17 	Training Loss: 1.152186 	Validation Loss: 0.287800
Validation loss decreased (0.289605 --> 0.287800).  Saving model ...
Epoch: 18 	Training Loss: 1.148037 	Validation Loss: 0.289288
Epoch: 19 	Training Loss: 1.145159 	Validation Loss: 0.287960
Epoch: 20 	Training Loss: 1.141931 	Validation Loss: 0.285502
Validation loss decreased (0.287800 --> 0.285502).  Saving model ...
Epoch: 21 	Training Loss: 1.138719 	Validation Loss: 0.284989
Validation loss decreased (0.285502 --> 0.284989).  Saving model ...
Epoch: 22 	Training Loss: 1.135665 	Validation Loss: 0.289896
Epoch: 23 	Training Loss: 1.133620 	Validation Loss: 0.286784
Epoch: 24 	Training Loss: 1.130854 	Validation Loss: 0.283358
Validation loss decreased (0.284989 --> 0.283358).  Saving model ...
Epoch: 25 	Training Loss: 1.128849 	Validation Loss: 0.283733
Epoch: 26 	Training Loss: 1.125378 	Validation Loss: 0.282474
Validation loss decreased (0.283358 --> 0.282474).  Saving model ...
Epoch: 27 	Training Loss: 1.123925 	Validation Loss: 0.282077
Validation loss decreased (0.282474 --> 0.282077).  Saving model ...
Epoch: 28 	Training Loss: 1.121677 	Validation Loss: 0.281045
Validation loss decreased (0.282077 --> 0.281045).  Saving model ...
Epoch: 29 	Training Loss: 1.119699 	Validation Loss: 0.287675
Epoch: 30 	Training Loss: 1.117810 	Validation Loss: 0.281724
Epoch: 31 	Training Loss: 1.115807 	Validation Loss: 0.280709
Validation loss decreased (0.281045 --> 0.280709).  Saving model ...
Epoch: 32 	Training Loss: 1.113873 	Validation Loss: 0.279994
Validation loss decreased (0.280709 --> 0.279994).  Saving model ...
Epoch: 33 	Training Loss: 1.112199 	Validation Loss: 0.279635
Validation loss decreased (0.279994 --> 0.279635).  Saving model ...
Epoch: 34 	Training Loss: 1.110301 	Validation Loss: 0.282371
Epoch: 35 	Training Loss: 1.108387 	Validation Loss: 0.281459
```

```
Epoch: 36       Training Loss: 1.106926         Validation Loss: 0.278647
Validation loss decreased (0.279635 --> 0.278647).  Saving model ...
Epoch: 37       Training Loss: 1.105597         Validation Loss: 0.278015
Validation loss decreased (0.278647 --> 0.278015).  Saving model ...
Epoch: 38       Training Loss: 1.103633         Validation Loss: 0.280401
Epoch: 39       Training Loss: 1.102408         Validation Loss: 0.281914
Epoch: 40       Training Loss: 1.100871         Validation Loss: 0.278361
Epoch: 41       Training Loss: 1.099510         Validation Loss: 0.276755
Validation loss decreased (0.278015 --> 0.276755).  Saving model ...
Epoch: 42       Training Loss: 1.098190         Validation Loss: 0.277570
Epoch: 43       Training Loss: 1.096485         Validation Loss: 0.276254
Validation loss decreased (0.276755 --> 0.276254).  Saving model ...
Epoch: 44       Training Loss: 1.095302         Validation Loss: 0.276883
Epoch: 45       Training Loss: 1.093840         Validation Loss: 0.275849
Validation loss decreased (0.276254 --> 0.275849).  Saving model ...
Epoch: 46       Training Loss: 1.092995         Validation Loss: 0.275682
Validation loss decreased (0.275849 --> 0.275682).  Saving model ...
Epoch: 47       Training Loss: 1.091815         Validation Loss: 0.275710
Epoch: 48       Training Loss: 1.090409         Validation Loss: 0.275230
Validation loss decreased (0.275682 --> 0.275230).  Saving model ...
Epoch: 49       Training Loss: 1.089028         Validation Loss: 0.274874
Validation loss decreased (0.275230 --> 0.274874).  Saving model ...
Epoch: 50       Training Loss: 1.088055         Validation Loss: 0.276718
```

In [128…

```
# loading the model with the lowest validation loss
model_gatedrnn.load_state_dict(torch.load('model.pt'))
```

Out[128]:

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

In [129…

```
correct, total = 0, 0
with torch.no_grad():
    for data in test_loader_rnn:
        embeddings, labels = data
        # calculating outputs by running embeddings through the network
        model_gatedrnn.to("cpu")
        outputs = model_gatedrnn(embeddings.float())
        # the class with the highest score is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
```

In [130…

```
print('Accuracy for Gated RNN model: ', (100 * correct / total))
```

```
Accuracy for Gated RNN model:  39.705
```

References:

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

https://www.kaggle.com/code/mishra1993/pytorch-multi-layer-perceptron-mnist/notebook

https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html

In [ ]:

In [ ]: