In [1]:
```
import pandas as pd
import gensim
from gensim.models import KeyedVectors
import numpy as np
import re
import contractions
import nltk
import numpy as np
from gensim.models import Word2Vec
from nltk.tokenize import word_tokenize
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import Perceptron
from sklearn.metrics import classification_report
from sklearn.svm import LinearSVC
import torch
import gensim.downloader as api
import copy
import torch.nn as nn
import torch.nn.functional as F
from sklearn.preprocessing import OneHotEncoder
import warnings
warnings.filterwarnings('ignore')
```

In [2]:
```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

1. Dataset Generation

The data given is read with the help of pandas read_csv using the compression gzip as data is compressed. Further I have used '\t' as seperator to format data.

In [3]:
```
data = pd.read_csv('data.tsv',sep='\t',on_bad_lines='skip', header=0, quotecha
```

Keep Reviews and Ratings

As mentioned below cell gets only 'review_body' and 'star_rating' columns from the whole data. Both the columns are merged and stored further

In [4]:
```
getData = data[['review_body','star_rating']]
getData['review_body']
```

Out[4]:
```
0                            Love this, excellent sun block!!
1          The great thing about this cream is that it do...
2          Great Product, I'm 65 years old and this is al...
3          I use them as shower caps & conditioning caps....
4          This is my go-to daily sunblock. It leaves no ...
                                 ...
5094302    After watching my Dad struggle with his scisso...
5094303    Like most sound machines, the sounds choices a...
5094304    I bought this product because it indicated 30 ...
5094305    We have used Oral-B products for 15 years; thi...
5094306    I love this toothbrush. It's easy to use, and ...
Name: review_body, Length: 5094307, dtype: object
```

Next the cell check for the NaN values for both the columns

Review body has 400 NaN values and Star rating has 10 NaN values

As the NaN values are less compared to the total dataset rows, I therefore dropped the NaN values from both the columns

In [5]:
```
print(getData.isnull().sum())
getData=getData.dropna()
print(getData.isnull().sum())
```

```
review_body    400
star_rating     10
dtype: int64
review_body    0
star_rating    0
dtype: int64
```

We form three classes and select 20000 reviews randomly from each class.

label_class funtion helps to to add designated class to star_rating. In this a extra column is created with name class which stores the value of class with respect to star_rating

Star_rating: 1, 2 Class: 1

Star_rating: 3 Class: 2

Star_rating: 4, 5 Class: 3

In [6]:
```
pd.options.mode.chained_assignment = None
def labelClass(rating):
    if rating == "1" :
          return 1
    if rating == "2" :
          return 1
    if rating == "3" :
          return 2
    if rating == "4":
          return 3
    if rating  == "5":
          return 3
getData['class'] = getData['star_rating'].map(labelClass)
```

Ramdom 20000 data rows from all three columns(review_body, star_rating, class) gets selected

In [7]:
```
classOne = getData.loc[getData['class'] == 1].sample(n=20000)
classTwo = getData.loc[getData['class'] == 2].sample(n=20000)
classThree = getData.loc[getData['class'] == 3].sample(n=20000)
```

The 20000 randomly slected data from all the three classes are further merged into one data providing the total rows of 60000

In [8]:
```
getRandomData = pd.concat([classOne, classTwo, classThree])
getRandomData
```

Out[8]:

	review_body	star_rating	class
633312	As a female, I'm a a fan of male razors for a ...	1	1
3990	This is NOT Natural & Not chemical Free. Don'...	1	1
2158261	not happy small model ,ill buy at bed and bath...	1	1
1584493	Way too much vanilla for me. Even the wife sai...	2	1
5051429	I have had two ER411 Nose and Ear Hair Groomer...	2	1
...	...	...	...
1257980	Love this stuff! Makes my face super smooth. S...	5	3
3130340	I had some baby perfume in a bottle for my dau...	5	3
1442853	People are saying that this shampoo leaves a f...	5	3
4836156	Started using twice a day and washing once a d...	4	3
577826	Great retinol product but more expensive than ...	5	3

60000 rows × 3 columns

various data cleaning strategies are are applied to clean and process the review data

1) all the reviews are converted to lower case

2) HTML and URL's are removed from the reviews

3) Last contractions are performed on the review's this will change won't → will not, I'm → i am and so on with the use of contraction library

4) Extra spaces are removed

5) Non alphabetical characters are also removed

In [9]:
```
getRandomData['review_body'] = getRandomData['review_body'].astype(str)

getRandomData['review_body'] = getRandomData['review_body'].apply(str.lower)
getRandomData['review_body'] = getRandomData['review_body'].apply(lambda x: re
getRandomData['review_body'] = getRandomData['review_body'].apply(lambda x: re
getRandomData['review_body'] = getRandomData['review_body'].apply(lambda x: re
```

In [10]:
```
def contra(text):
    words = []
    for word in text.split():
        words.append(contractions.fix(word))
    word_text = ' '.join(words)
    return word_text
getRandomData['review_body'] = getRandomData['review_body'].apply(contra)
getRandomData['review_body'] = getRandomData['review_body'].apply(lambda x: re
```

2. Word Embedding (25 points)

(a)

loading word2vec-google-news-300

In [11]:
```
model = api.load('word2vec-google-news-300')
```

pretrained examples

In [12]:
```
model.similarity("food", "rice")
```

Out[12]:
```
0.48333582
```

In [13]:
```
model.similarity("judge", "tedious")
```

Out[13]:
```
0.012386799
```

In [14]:
```
model.most_similar(positive=['polish', 'easily'], negative=['messy'])
```

Out[14]:
```
[('polished', 0.43561863899230957),
 ('effortlessly', 0.3926153779029846),
 ('readily', 0.37599194049835205),
 ('instantly', 0.3723205626010895),
 ('inexpensively', 0.35900598764419556),
 ('quickness_explosiveness', 0.3580486476421356),
 ('polishes', 0.3550904393196106),
 ('quickly', 0.3545926809310913),
 ('swf_file', 0.3500930070877075),
 ('skils', 0.3500587046146393)]
```

(b)

trianing word2vec model on our dataset

In [15]:
```
def getSplitwords(review):
    return review.split(' ')
splitSentence = getRandomData['review_body'].apply(getSplitwords)
```

In [16]:
```
trainModel = Word2Vec(splitSentence, vector_size=300, window=13, min_count=9)
```

example on our trained model

In [17]:
```
trainModel.wv.similarity("food", "rice")
```

Out[17]:
```
0.43362138
```

In [18]:
```
trainModel.wv.similarity("judge", "tedious")
```

Out[18]:
```
0.26440674
```

In [19]:
```
trainModel.wv.most_similar(positive=['polish', 'easily'], negative=['messy'])
```

Out[19]:
```
[('nail', 0.7342321276664734),
 ('nails', 0.6688219308853149),
 ('chip', 0.6668947339057922),
 ('polishes', 0.6637318730354309),
 ('file', 0.6354458332061768),
 ('chipping', 0.6278954148292542),
 ('coat', 0.6158057451248169),
 ('opi', 0.5834757089614868),
 ('manicure', 0.5627020597457886),
 ('acrylic', 0.5450944900512695)]
```

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

From the above examples taken it seems that the vectors generated by ourself (second part) gives better results compare to pretrained model

3. Simple models

TF-IDF

In [20]:
```
X = getRandomData['review_body']
# y = getRandomData['class']
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(X)
```

In [21]:
```
X_train, X_test, y_train, y_test = train_test_split(X, getRandomData['class'],
```

using perceptron model with TF-IDF vectors

In [22]:
```
tfiDfPer = Perceptron(tol=1e-3, random_state=0)
tfiDfPer.fit(X_train, y_train)
y_pred = tfiDfPer.predict(X_test)
perceptronCR = classification_report(y_test, y_pred, output_dict = True)
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           1       0.64      0.65      0.64      3969
           2       0.54      0.53      0.53      3986
           3       0.71      0.72      0.72      4045

    accuracy                           0.63     12000
   macro avg       0.63      0.63      0.63     12000
weighted avg       0.63      0.63      0.63     12000
```

In [23]:
```
print('classification report of Perceptron - TF-IDF')
print('Class        Precision           Recall             F1-score')
print('1'+"      "+ str(perceptronCR['1']['precision'])+",   "+str(perceptronC
print('2'+"      "+ str(perceptronCR['2']['precision'])+",   "+str(perceptronC
print('3'+"      "+ str(perceptronCR['3']['precision'])+",   "+str(perceptronC
print('average' +" "+str((perceptronCR['1']['precision']+perceptronCR['2']['pr
print('Final accuracy', perceptronCR['accuracy'])
```

```
classification report of Perceptron - TF-IDF
Class        Precision           Recall             F1-score
1      0.6431441486690106,   0.6452506928697405,   0.6441956986542574
2      0.538972655251725,   0.5291018564977421,   0.5339916445119636
3      0.7123020706455542,   0.722867737948084,   0.7175460122699386
average 0.6314729581887631, 0.6324067624385221, 0.63191111847872
Final accuracy 0.6328333333333334
```

using SVM model with TF-IDF vectors

In [24]:
```
tfiDf_SVM = LinearSVC(random_state = 0)
tfiDf_SVM.fit(X_train,y_train)
y_pred = tfiDf_SVM.predict(X_test)
print(classification_report(y_test, y_pred))
tfidfsvmacc = classification_report(y_test, y_pred, output_dict = True)
print('Final accuracy', tfidfsvmacc['accuracy'])
```

```
              precision    recall  f1-score   support

           1       0.71      0.73      0.72      3969
           2       0.61      0.59      0.60      3986
           3       0.77      0.78      0.77      4045

    accuracy                           0.70     12000
   macro avg       0.70      0.70      0.70     12000
weighted avg       0.70      0.70      0.70     12000

Final accuracy 0.6976666666666667
```

Word2Vec

In [25]:
```
embeddings = []
sentence = []

for review in getRandomData['review_body']:
    words = review.split(' ')
    for word in words:
        try:
            embedding = trainModel.wv[word]
        except KeyError:
            embedding = np.zeros(300)
        sentence.append(embedding)

    sentence = np.array(sentence)
    sentence_embedding = np.mean(sentence, axis=0)
    embeddings.append(sentence_embedding)
    sentence = []
data = np.array(embeddings)
```

In [26]:
```
X_w2v_train, X_w2v_test, y_w2v_train, y_w2v_test = train_test_split(data, getR
```

using perceptron model with word2vec

In [27]:
```
w2vPer = Perceptron(tol=1e-3, random_state=0)
w2vPer.fit(X_w2v_train, y_w2v_train)
y_pred = w2vPer.predict(X_w2v_test)
perceptronCR = classification_report(y_w2v_test, y_pred, output_dict = True)
print(classification_report(y_w2v_test, y_pred))
w2vpacc = classification_report(y_w2v_test, y_pred, output_dict = True)
print('Final accuracy', w2vpacc['accuracy'])
```

```
              precision    recall  f1-score   support

           1       0.78      0.13      0.22      3963
           2       0.41      0.89      0.56      4014
           3       0.77      0.49      0.60      4023

    accuracy                           0.50     12000
   macro avg       0.65      0.50      0.46     12000
weighted avg       0.65      0.50      0.46     12000

Final accuracy 0.504
```

using SVM model with word2vec

In [28]:
```
w2v_SVM = LinearSVC(random_state = 0)
w2v_SVM.fit(X_w2v_train,y_w2v_train)
y_pred = w2v_SVM.predict(X_w2v_test)
print(classification_report(y_w2v_test, y_pred))
w2vsvmacc = classification_report(y_w2v_test, y_pred, output_dict = True)
print('Final accuracy', w2vsvmacc['accuracy'])
```

```
              precision    recall  f1-score   support

           1       0.66      0.72      0.69      3963
           2       0.62      0.56      0.59      4014
           3       0.74      0.74      0.74      4023

    accuracy                           0.67     12000
   macro avg       0.67      0.67      0.67     12000
weighted avg       0.67      0.67      0.67     12000

Final accuracy 0.6746666666666666
```

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

Both the Perceptron and SVM models perform better with TF-IDF than with Word2Vec embeddings. This may be due to the fact that the Word2Vec model used in this study was trained on Google News articles, which may not include words commonly used in reviews. Additionally, the Word2Vec embeddings may not accurately represent the importance of certain words in our specific corpus, whereas TF-IDF is able to reflect the importance of words within our corpus.

4. Feedforward Neural Networks

creating one hot encoding on class labels

In [29]:
```
y = copy.deepcopy(getRandomData["class"])
ohe = OneHotEncoder(sparse = False)
ohe.fit(np.asarray([[1],[2],[3]]))
yohe = ohe.transform(y.values.reshape((60000,1)))

word2VecSet = set(trainModel.wv.key_to_index.keys())
```

(a)

In [30]:
```
embeddings = []
sentence = []

for review in getRandomData['review_body']:
    words = review.split(' ')
    for word in words:
        try:
            embedding = trainModel.wv[word]
        except KeyError:
            embedding = np.zeros(300)
        sentence.append(embedding)

    sentence = np.array(sentence)
    sentence_embedding = np.mean(sentence, axis=0)
    embeddings.append(sentence_embedding)
    sentence = []
data = np.array(embeddings)

Xw2v_train, Xw2v_test, Yw2v_train, Yw2v_test = train_test_split(data, yohe, te
```

converting the NumPy arrays Xw2v_train and Yw2v_train into PyTorch tensors and creating a dataset and data loader for efficient batching of training data during model training.

In [31]:
```
Xw2v_train = torch.FloatTensor(Xw2v_train).to(device)
Yw2v_train = torch.from_numpy(Yw2v_train).to(device)
train = torch.utils.data.TensorDataset(Xw2v_train, Yw2v_train)
trainLoader = torch.utils.data.DataLoader(train, batch_size=32, num_workers=0,
```

converting the NumPy arrays Xw2v_test and Yw2v_test into PyTorch tensors and creating a dataset and data loader

In [32]:
```
Xw2v_test = torch.FloatTensor(Xw2v_test).to(device)
Yw2v_test = torch.from_numpy(Yw2v_test).to(device)
test = torch.utils.data.TensorDataset(Xw2v_test, Yw2v_test)
testLoader = torch.utils.data.DataLoader(test, batch_size=32, num_workers=0, s
```

defines a Feedforward Neural Network (FNN) model using PyTorch's neural network module, where the model has three linear layers and ReLU activation functions. It then instantiates the model and assigns it to a variable named fnnModel.

In [33]:
```
import torch.nn as nn
import torch.nn.functional as F
class FNN(nn.Module):
    def __init__(self):
        super(FNN, self).__init__()
        self.fc1 = nn.Linear(300, 100)
        self.fc2 = nn.Linear(100, 10)
        self.fc3 = nn.Linear(10, 3)

    def forward(self, x):
        x = x.view(-1, 300)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        return x
fnnModel = FNN()
```

In [34]:
```
fnnModel = fnnModel.to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(fnnModel.parameters(), lr=0.0001)
print(fnnModel)
```

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

defines a function named runModel that trains and evaluates a PyTorch model for a certain number of epochs using the specified optimizer, loss function, and training and testing data loaders. It also saves the model's state dictionary if the validation loss improves.

In [35]:
```
def runModel(model, optimizer, criterion, trainLoader, testLoader):
    epochs = 50
    validationLossMin = np.Inf

    for epoch in range(epochs):
        trainingLoss = 0.0
        validationLoss = 0.0

        model.train()
        for data, target in trainLoader:
            optimizer.zero_grad()
            getOutput = model(data)
            getOutput = getOutput.to(device)
            getLoss = criterion(getOutput, target)
            getLoss.backward()
            optimizer.step()
            trainingLoss += getLoss.item()*data.size(0)

        model.eval()
        with torch.no_grad():
            for data, target in testLoader:
                getOutput = model(data)
                getOutput = getOutput.to(device)
                getLoss = criterion(getOutput, target)
                validationLoss += getLoss.item()*data.size(0)

        trainingLoss = trainingLoss/len(trainLoader.dataset)
        validationLoss = validationLoss/len(testLoader.dataset)
        print('Epoch:', epoch+1)
        print('Training loss', trainingLoss)
        print('Validation loss:', validationLoss)
        print('_____________________________________________________')

        if validationLoss <= validationLossMin:
            torch.save(model.state_dict(), 'model.pt')
            validationLossMin = validationLoss
```

In [36]:
```
runModel(fnnModel, optimizer, criterion, trainLoader, testLoader)
```

```
Epoch: 1
Training loss 0.9117901773699463
Validation loss: 0.8165627371261363
_________________________________________________________________
_________________________________
Epoch: 2
Training loss 0.7940597255324368
Validation loss: 0.7854675148968235
_________________________________________________________________
_________________________________
Epoch: 3
Training loss 0.7732538627692338
Validation loss: 0.770903786336421
_________________________________________________________________
_________________________________
Epoch: 4
Training loss 0.7627920664017317
Validation loss: 0.7646510052934755
_________________________________________________________________
```

load efficient model

In [37]:
```
fnnModel.load_state_dict(torch.load('model.pt'))
```

Out[37]:
```
<All keys matched successfully>
```

running model on test data

In [38]:
```
getPred = []
getLabel = []

fnnModel.eval()
with torch.no_grad():
    for data, target in testLoader:
        output = fnnModel(data)

        getLabel.append(target.cpu().detach().numpy())

        _, predicted = torch.max(output, 1)
        getPred.append(predicted.cpu().detach().numpy())

predictions = np.array(getPred)

finalLabels = []
for batch in getLabel:
    t = []
    for b in batch:
        t.append(np.argmax(b))
    finalLabels.append(t)
np.shape(finalLabels)

finalLabels = np.array(finalLabels)
predictions = list(predictions)

finalLabelss = []
for f in finalLabels:
    for d in f:
        finalLabelss.append(d)

predictionss = []
for f in predictions:
    for d in f:
        predictionss.append(d)

print(classification_report(finalLabelss, predictionss, target_names = ["1","2
fnnacc = classification_report(finalLabelss, predictionss, target_names = ["1"
print('Final accuracy', fnnacc['accuracy'])
```

```
              precision    recall  f1-score   support

           1       0.69      0.70      0.70      3963
           2       0.61      0.63      0.62      4014
           3       0.78      0.74      0.76      4023

    accuracy                           0.69     12000
   macro avg       0.69      0.69      0.69     12000
weighted avg       0.70      0.69      0.69     12000

Final accuracy 0.6928333333333333
```

##

(b)

a convolutional feedforward neural network with three fully connected layers. The input size to the network is 3000, and the activation function used in each layer is ReLU.

In [39]:
```
import torch.nn as nn
import torch.nn.functional as F
class FNNCon(nn.Module):
    def __init__(self):
        super(FNNCon, self).__init__()
        self.fc1 = nn.Linear(3000, 100)
        self.fc2 = nn.Linear(100, 10)
        self.fc3 = nn.Linear(10, 3)

    def forward(self, x):
        x = x.view(-1, 3000)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        return x
```

list of feature vectors for text data using word embeddings from a trained Word2Vec model. Each feature vector has a length of 3000 and contains the concatenated embeddings of the first 10 words in the text.

In [40]:
```
X = []
for sntnc in getRandomData['review_body']:
    wrds = sntnc.split(' ')
    tmp = np.array([trainModel.wv[w] for w in wrds[:10] if w in word2VecSet])
    if len(tmp) == 0:
        tmp = np.zeros((1,300))
    wrds = np.concatenate(tmp, axis = 0)
    if len(wrds)<3000:
        wrds = np.concatenate([wrds, np.zeros(3000-len(wrds))])
    X.append(wrds)
```

In [41]:
```
Xw2v_train, Xw2v_test, Yw2v_train, Yw2v_test = train_test_split(X, yohe, test_
```

converting the NumPy arrays Xw2v_train and Yw2v_train into PyTorch tensors and creating a dataset and data loader for efficient batching of training data during model training.

In [42]:
```
Xw2v_train = torch.FloatTensor(Xw2v_train).to(device)
Yw2v_train = torch.from_numpy(Yw2v_train).to(device)
train = torch.utils.data.TensorDataset(Xw2v_train, Yw2v_train)
trainLoader = torch.utils.data.DataLoader(train, batch_size=256, num_workers=0
```

converting the NumPy arrays Xw2v_test and Yw2v_test into PyTorch tensors and creating a dataset and data loader

In [43]:
```
Xw2v_test = torch.FloatTensor(Xw2v_test).to(device)
Yw2v_test = torch.from_numpy(Yw2v_test).to(device)
test = torch.utils.data.TensorDataset(Xw2v_test, Yw2v_test)
testLoader = torch.utils.data.DataLoader(test, batch_size=256, num_workers=0,
```

In [44]:
```
fnnconModel = FNNCon().to(device)
criterion = nn.CrossEntropyLoss().to(device)
optimizer = torch.optim.Adam(fnnconModel.parameters(), lr=0.01)
```

In [45]:
```
runModel(fnnconModel, optimizer, criterion, trainLoader, testLoader)
```

```
Training loss 0.9592153315918407
Validation loss: 1.0273787855710534
---------------------------------------------------------------------------
Epoch: 10
Training loss 0.9526289717454373
Validation loss: 1.0308136152181275
---------------------------------------------------------------------------
Epoch: 11
Training loss 0.9387644932328403
Validation loss: 1.0405589837586793
---------------------------------------------------------------------------
Epoch: 12
Training loss 0.927632860221642
Validation loss: 1.0466046424630324
---------------------------------------------------------------------------
Epoch: 13
```

load efficient model

In [46]:
```
getPred = []
getLabel = []

fnnconModel.eval()
with torch.no_grad():
    for data, target in testLoader:
        output = fnnconModel(data)

        getLabel.append(target.cpu().detach().numpy())

        _, predicted = torch.max(output, 1)
        getPred.append(predicted.cpu().detach().numpy())

predictions = np.array(getPred)

finalLabels = []
for batch in getLabel:
    t = []
    for b in batch:
        t.append(np.argmax(b))
    finalLabels.append(t)

finalLabels = np.array(finalLabels)
predictions = list(predictions)

finalLabelss = []
for f in finalLabels:
    for d in f:
        finalLabelss.append(d)

predictionss = []
for f in predictions:
    for d in f:
        predictionss.append(d)

print(classification_report(finalLabelss, predictionss, target_names = ["1","2
fnnconacc = classification_report(finalLabelss, predictionss, target_names = [
print('Final accuracy', fnnconacc['accuracy'])
```

```
              precision    recall  f1-score   support

           1       0.50      0.37      0.42      3963
           2       0.41      0.60      0.49      4014
           3       0.68      0.55      0.61      4023

    accuracy                           0.51     12000
   macro avg       0.53      0.51      0.51     12000
weighted avg       0.53      0.51      0.51     12000

Final accuracy 0.5068333333333334
```

Q) What do you conclude by comparing accuracy values you obtain with those obtained in the "'Simple Models" section.

The accuracy scores obtained by taking the mean outperform those generated by concatenation. One reason for this could be that taking the mean helps to capture the overall sentiment of the entire review, whereas the contribution of the first 10 words of a sentence may not necessarily be a strong indicator of sentiment and thus less useful for classification purposes.

Also TF-IDF SVM performs slightly better than FNN model part (a) as reviews can often contain specific keywords and phrases that can be easily captured by the TF-IDF SVM model

##

5. Recurrent Neural Networks

(a)

an RNN neural network model with a single RNN layer, followed by a fully connected layer for classification. The RNN layer takes a sequence of input vectors with 300 features, produces a sequence of hidden states with 20 features, and the final hidden state is fed to the fully connected layer to produce the output.

In [47]:
```
import torch.nn as nn
import torch.nn.functional as F
class RNN(nn.Module):
    def __init__(self):
        super(RNN, self).__init__()
        self.rnn = nn.RNN(300, 20, batch_first = True, nonlinearity='relu')
        self.fc = nn.Linear(20, 3)

    def forward(self, x):
        x, hidden = self.rnn(x)
        x = self.fc(x[:,-1,:])
        return x
```

a list of 20-dimensional word embeddings for each sentence in the 'review_body'. If a sentence has fewer than 20 words, it pads the embedding vector with zeros.

In [48]:
```
X = []
for sntnc in getRandomData['review_body']:
    wrds = sntnc.split(' ')
    if len(wrds)<20:
        t = np.array([trainModel.wv[w] for w in wrds if w in word2VecSet])
    else:
        t = np.array([trainModel.wv[w] for w in wrds[:20] if w in word2VecSet]
    if len(t) == 0:
        wrds = np.zeros((20,300))
    elif len(t)<20:
        wrds = np.concatenate([t,np.zeros((20-len(t), 300))])
    else:
        wrds = t
    X.append(wrds)
```

In [49]:
```
Xw2v_train, Xw2v_test, Yw2v_train, Yw2v_test = train_test_split(X, yohe, test_
```

converting the NumPy arrays Xw2v_train and Yw2v_train into PyTorch tensors and creating a dataset and data loader for efficient batching of training data during model training.

In [50]:
```
Xw2v_train = torch.FloatTensor(Xw2v_train).to(device)
Yw2v_train = torch.from_numpy(Yw2v_train).to(device)
train = torch.utils.data.TensorDataset(Xw2v_train, Yw2v_train)
trainLoader = torch.utils.data.DataLoader(train, batch_size=32, num_workers=0,
```

converting the NumPy arrays Xw2v_test and Yw2v_test into PyTorch tensors and creating a dataset and data loader

In [51]:
```
Xw2v_test = torch.FloatTensor(Xw2v_test).to(device)
Yw2v_test = torch.from_numpy(Yw2v_test).to(device)
test = torch.utils.data.TensorDataset(Xw2v_test, Yw2v_test)
testLoader = torch.utils.data.DataLoader(test, batch_size=32, num_workers=0, s
```

In [52]:
```
rnnModel = RNN().to(device)
criterion = nn.CrossEntropyLoss().to(device)
optimizer = torch.optim.Adam(rnnModel.parameters(), lr=0.001)
```

In [53]:
```
runModel(rnnModel, optimizer, criterion, trainLoader, testLoader)
```

```
_________________________________________________________________
Epoch: 4
Training loss 0.8382157579871922
Validation loss: 0.8506305748424391
_________________________________________________________________
_________________________________________________________________
Epoch: 5
Training loss 0.8271501441389467
Validation loss: 0.8412165344179806
_________________________________________________________________
_________________________________________________________________
Epoch: 6
Training loss 0.8194644876614843
Validation loss: 0.8378830165569767
_________________________________________________________________
_________________________________________________________________
Epoch: 7
Training loss 0.8153055359042519
Validation loss: 0.8287114498485583
```

load efficient model

In [54]:
```
getPred = []
getLabels = []

rnnModel.eval()
with torch.no_grad():
    for data, target in testLoader:
        output = rnnModel(data)

        getLabels.append(target.cpu().detach().numpy())
        _, predicted = torch.max(output, 1)
        getPred.append(predicted.cpu().detach().numpy())

predictions = np.array(getPred)

finalLabels = []
for batch in getLabels:
    t = []
    for b in batch:
        t.append(np.argmax(b))
    finalLabels.append(t)


finalLabels = np.array(finalLabels)
predictions = list(predictions)

finalLabelss = []
for f in finalLabels:
    for d in f:
        finalLabelss.append(d)

predictionss = []
for f in predictions:
    for d in f:
        predictionss.append(d)

print(classification_report(finalLabelss, predictionss, target_names = ["1","2
rnnacc = classification_report(finalLabelss, predictionss, target_names = ["1"
print('Final accuracy', rnnacc['accuracy'])
```

```
              precision    recall  f1-score   support

           1       0.61      0.69      0.64      3963
           2       0.59      0.43      0.49      4014
           3       0.67      0.77      0.72      4023

    accuracy                           0.63     12000
   macro avg       0.62      0.63      0.62     12000
weighted avg       0.62      0.63      0.62     12000

Final accuracy 0.6274166666666666
```

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

Feedforward neural network model part (a) has higher accuracy compare to RNN model, this can be because of the reason,

1) FNN models are generally faster to train and require fewer computational resources compared to RNN models.

2) Amazon reviews are often relatively short in length, and the context of the review can be inferred from a small window of words. FNN models are particularly good at capturing local features, as they can detect patterns in the input data regardless of their position in the sequence. In contrast, RNN models require contextual information from the entire sequence to make predictions, which may be less effective for short reviews.

##

(b)

This is a PyTorch implementation of a GRU model

In [55]:
```
class GRU(nn.Module):
    def __init__(self):
        super(GRU, self).__init__()
        self.rnn = nn.GRU(300, 20, batch_first = True)
        self.fc = nn.Linear(20, 3)

    def forward(self, x):
        x = x.view(-1, 20,300)
        x, hidden = self.rnn(x)
        x = self.fc(x[:,-1,:])
        return x
```

In [56]:
```
gruModel = GRU().to(device)
criterion = nn.CrossEntropyLoss().to(device)
optimizer = torch.optim.Adam(gruModel.parameters(), lr=0.001)
```

In [57]:
```
runModel(gruModel, optimizer, criterion, trainLoader, testLoader)
```

```
_________________________________________________________________
Epoch: 30
Training loss 0.677948063762063
Validation loss: 0.7923839490424919
_________________________________________________________________
Epoch: 31
Training loss 0.6783095939920556
Validation loss: 0.8008777759385606
_________________________________________________________________
Epoch: 32
Training loss 0.6737102895490049
Validation loss: 0.7962363455869879
_________________________________________________________________
Epoch: 33
Training loss 0.6764701716813821
Validation loss: 0.7985212057894872
```

load efficient model

In [58]:
```
getPred = []
getLabels = []

gruModel.eval()
with torch.no_grad():
    for data, target in testLoader:
        output = gruModel(data)

        getLabels.append(target.cpu().detach().numpy())
        _, predicted = torch.max(output, 1)
        getPred.append(predicted.cpu().detach().numpy())

predictions = np.array(getPred)

finalLabels = []
for batch in getLabels:
    t = []
    for b in batch:
        t.append(np.argmax(b))
    finalLabels.append(t)

finalLabels = np.array(finalLabels)
predictions = list(predictions)

finalLabelss = []
for f in finalLabels:
    for d in f:
        finalLabelss.append(d)

predictionss = []
for f in predictions:
    for d in f:
        predictionss.append(d)

print(classification_report(finalLabelss, predictionss, target_names = ["1","2
gruacc = classification_report(finalLabelss, predictionss, target_names = ["1"
print('Final accuracy', gruacc['accuracy'])
```

```
              precision    recall  f1-score   support

           1       0.64      0.62      0.63      3963
           2       0.55      0.57      0.56      4014
           3       0.72      0.72      0.72      4023

    accuracy                           0.64     12000
   macro avg       0.64      0.64      0.64     12000
weighted avg       0.64      0.64      0.64     12000

Final accuracy 0.6363333333333333

################################################################################
```

(c)

This is a PyTorch implementation of a LSTM model

In [59]:
```
class LSTM(nn.Module):
    def __init__(self):
        super(LSTM, self).__init__()
        self.lstm = nn.LSTM(300, 20, batch_first=True)
        self.fc = nn.Linear(20, 3)

    def forward(self, x):
        x = x.view(-1, 20, 300)
        x, (hidden, cell) = self.lstm(x)
        x = x[:, -1, :]
        x = self.fc(x)
        x = F.softmax(x, dim=1)
        return x
```

In [60]:
```
lstmModel = LSTM().to(device)
criterion = nn.CrossEntropyLoss().to(device)
optimizer = torch.optim.Adam(lstmModel.parameters(), lr=0.001)
```

In [61]:
```
runModel(lstmModel, optimizer, criterion, trainLoader, testLoader)
```

```
Epoch: 28
Training loss 0.8629405195191503
Validation loss: 0.9137267689903578
_________________________________________________________________
_________________________________
Epoch: 29
Training loss 0.8622919616413613
Validation loss: 0.9163985406508048
_________________________________________________________________
_________________________________
Epoch: 30
Training loss 0.860788869065543
Validation loss: 0.9111189749439558
_________________________________________________________________
_________________________________
Epoch: 31
Training loss 0.8557872375051181
Validation loss: 0.9106509808947643
_________________________________________________________________
```

load efficient model

In [62]:
```
getPred = []
getLabels = []

lstmModel.eval()
with torch.no_grad():
    for data, target in testLoader:
        output = lstmModel(data)

        getLabels.append(target.cpu().detach().numpy())
        _, predicted = torch.max(output, 1)
        getPred.append(predicted.cpu().detach().numpy())

predictions = np.array(getPred)

finalLabels = []
for batch in getLabels:
    t = []
    for b in batch:
        t.append(np.argmax(b))
    finalLabels.append(t)


finalLabels = np.array(finalLabels)
predictions = list(predictions)

finalLabelss = []
for f in finalLabels:
    for d in f:
        finalLabelss.append(d)

predictionss = []
for f in predictions:
    for d in f:
        predictionss.append(d)

print(classification_report(finalLabelss, predictionss, target_names = ["1","2
lstmacc = classification_report(finalLabelss, predictionss, target_names = ["1
print('Final accuracy', lstmacc['accuracy'])
```

```
              precision    recall  f1-score   support

           1       0.63      0.62      0.63      3963
           2       0.56      0.55      0.55      4014
           3       0.70      0.73      0.72      4023

    accuracy                           0.63     12000
   macro avg       0.63      0.63      0.63     12000
weighted avg       0.63      0.63      0.63     12000

Final accuracy 0.63375
```

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

The accuracy scores achieved with GRU are slightly greater to those achieved with RNN and LSTM. This is due to the gates present in GRU, which effectively address the issue of vanishing gradients during backpropagation, allowing for more effective loss propagation.

In [63]:
```
print('Each Model accuracy')
print("TF-IDF Perceptron:",perceptronCR['accuracy'])
print("TF-IDF SVM:",tfidfsvmacc['accuracy'])
print("Word2Vec Perceptron:",w2vpacc['accuracy'])
print("Word2Vec SVM:",w2vsvmacc['accuracy'])
print("Feedforwad neural network part(a):",fnnacc['accuracy'])
print("Feedforwad neural network part(b):",fnnconacc['accuracy'])
print("RNN:",rnnacc['accuracy'])
print("GRU:",gruacc['accuracy'])
print("LSTM:",lstmacc['accuracy'])
```

```
Each Model accuracy
TF-IDF Perceptron: 0.504
TF-IDF SVM: 0.6976666666666667
Word2Vec Perceptron: 0.504
Word2Vec SVM: 0.6746666666666666
Feedforwad neural network part(a): 0.6928333333333333
Feedforwad neural network part(b): 0.5068333333333334
RNN: 0.6274166666666666
GRU: 0.6363333333333333
LSTM: 0.63375
```

In [ ]: