In [3]:
```python
import pandas as pd
import numpy as np
import nltk
import warnings
warnings.filterwarnings("ignore")
nltk.download('wordnet')
nltk.download('stopwords')
nltk.download('punkt')
import re
from bs4 import BeautifulSoup
import contractions
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
import gensim.downloader as api
import gensim.models
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import Perceptron
from sklearn.svm import SVC
from sklearn.svm import LinearSVC
from sklearn.metrics import classification_report
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix

import gc
from sys import getsizeof
```

```
[nltk_data] Downloading package wordnet to
[nltk_data]     /home/ayanpatel_69/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     /home/ayanpatel_69/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to
[nltk_data]     /home/ayanpatel_69/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

Task 1: Dataset Generation

In [4]:
```python
df = pd.read_csv('./data.tsv', sep='\t', error_bad_lines=False, warn_bad_lines=False)
df = df[['star_rating', 'review_body']]

# Classwise dataset generation step
class_one = df[(df['star_rating']==1) | (df['star_rating']==2)]
class_two = df[df['star_rating']==3]
class_three = df[(df['star_rating']==4) | (df['star_rating']==5)]

class_one.loc[:, "label"] =1
class_two.loc[:, "label"] =2
class_three.loc[:, "label"] =3

class_one = class_one.sample(n=20000, random_state=100)
class_two = class_two.sample(n=20000, random_state=100)
class_three = class_three.sample(n=20000, random_state=100)
df = pd.concat([class_one, class_two, class_three])

df.reset_index(drop=True)

# Final Train Test 80/20 Split
train = df.sample(frac=0.8, random_state=100)
test = df.drop(train.index)

train = train.reset_index(drop = True)
test = test.reset_index(drop = True)

# clearing some memory
del globals()['class_one'], globals()['class_two'], globals()['class_three'], globals()['df']
df = dataset = [[99999, 99999]]
del df, dataset
gc.collect()
```

Out[4]:
```
0
```

In [5]:

```
# Covert all reviews to lower case
train['review_body'] = train['review_body'].str.lower()
test['review_body'] = test['review_body'].str.lower()

'''
URL Remover code
'''
train['review_body'] = train['review_body'].apply(lambda x: re.split('https:\/\/.*', str(x))[0])
test['review_body'] = test['review_body'].apply(lambda x: re.split('https:\/\/.*', str(x))[0])

def html_tag_remover(review):
    soup = BeautifulSoup(review, 'html.parser')
    review = soup.get_text()
    return review

train['review_body'] = train['review_body'].apply(lambda review: html_tag_remover(review))
test['review_body'] = test['review_body'].apply(lambda review: html_tag_remover(review))

'''
remove non-alphabetical characters
'''
train['review_body'] = train['review_body'].apply(lambda review: re.sub('[^a-zA-Z]+',' ', review))
test['review_body'] = test['review_body'].apply(lambda review: re.sub('[^a-zA-Z]+',' ', review))

'''
remove extra spaces
'''
train['review_body'] = train['review_body'].apply(lambda review: re.sub(' +', ' ', review))
test['review_body'] = test['review_body'].apply(lambda review: re.sub(' +', ' ', review))

'''
perform contractions on the reviews
'''
def expand_contractions(review):
    review = contractions.fix(review)
    return review

train['review_body'] = train['review_body'].apply(lambda review: expand_contractions(review))
test['review_body'] = test['review_body'].apply(lambda review: expand_contractions(review))
```

In [6]:

```
'''
remove the stop words AND perform lemmatization

'''
def remove_stopwords(review):
    stop_words_english = set(stopwords.words('english'))
    review_word_tokens = word_tokenize(review)
    filtered_review = [word for word in review_word_tokens if not word in stop_words_english]
    return filtered_review

train['review_body'] = train['review_body'].apply(lambda review: remove_stopwords(review))
test['review_body'] = test['review_body'].apply(lambda review: remove_stopwords(review))

def review_lemmatize(review):
    lemmatizer = WordNetLemmatizer()
    lemmatized_review = [lemmatizer.lemmatize(word) for word in review]
    return ' '.join(lemmatized_review)

train['review_body'] = train['review_body'].apply(lambda review: review_lemmatize(review))
test['review_body'] = test['review_body'].apply(lambda review: review_lemmatize(review))
```

Task 2: Word Embedding

Task 2(a) pretrained “word2vec-google-news-300” Word2Vec model.

In [7]:

```
# Loading Pretrained Word2Vec model:
pretrained_w2v = api.load('word2vec-google-news-300')
```

In [8]:

```
print('Check semantic similarities of the generated vectors:')
print(pretrained_w2v.most_similar(positive=['king', 'woman'], negative=['man'], topn = 1))
print('Excellent ~ Outstanding:', pretrained_w2v.similarity('excellent', 'outstanding'))
print('time ~ schedule:', pretrained_w2v.similarity('time', 'schedule'))
```

```
Check semantic similarities of the generated vectors:
[('queen', 0.7118193507194519)]
Excellent ~ Outstanding: 0.5567486
time ~ schedule: 0.26993576
```

Task 2(b) Word2Vec model using your own dataset

In [9]:
```
# Generating list of all the words corresponding to its sentence
all_Sentences = [sentence.split(' ') for sentence in train['review_body'].to_list()]
```

In [10]:
```
# Custom Word2Vec Setting the embedding size to be 300 and the window size to be 13.
custom_model = gensim.models.Word2Vec(all_Sentences, vector_size = 300, min_count=9, window=13)
```

In [11]:
```
print('Check semantic similarities of the generated vectors:')
print(custom_model.wv.most_similar(positive=['king', 'woman'], negative=['man'], topn = 1)[0])
print('Excellent ~ Outstanding:', custom_model.wv.similarity('excellent', 'outstanding'))
print('time ~ schedule:', custom_model.wv.similarity('time', 'schedule'))
```

```
Check semantic similarities of the generated vectors:
('ray', 0.768917441368103)
Excellent ~ Outstanding: 0.751618
time ~ schedule: 0.16850077
```

In [12]:
```
# Clearing some memory
del all_Sentences, custom_model
gc.collect()
all_Sentences = [1]
custom_model = [1]
```

Question Task 2:

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?

Answer:

The pre-trained Word2vec model seems to encode semantic similarities better than my trained model. Pre-trained model encodes similarities better than my model because it got a lot of information from all the words it was trained on. Pretrained word2vec models that are trained on large, diverse corpus are generally known to perform well in capturing semantic similarities between words. I see that there are many more word keys generated by the pre-trained vort2vec model. This is because it has been trained on a very large dataset for a long time, making it a more accurate model for embedding words from many different words. Word2Vec is trained on the Google News dataset (about 100 billion words). It has several use cases like recommendation engines, knowledge discovery, and what we use it for, text classification.

Task 3: Simple Models

In [44]:
```
'''
Feature INPUT Vectors FOR BOTH TRAIN AND TEST DATA created for both Task 3 and Task 4 WHICH INCLUDES FEATURE VECTOR
OF BOTH AVERAGE WORD2VEC VECTORS AND CONCATENATED THE FIRST 10 Word2Vec VECTORS for each review
'''
# Calculates Average word2Vec vectors
def average_vectors(review, label):
    temp_review = review.split(' ')
    review_vector = np.array([pretrained_w2v[word] for word in temp_review if word in pretrained_w2v])
    if len(review_vector) >=1:
#         review_vector = []
#         for word in words:
#             review_vector.append(pretrained_w2v[word])
        return review_vector, label

# Calculates concatenated first 10 Word2Vec Vectors
def average_vectors_concat(review, label):
    temp_review = review.split(' ')
    words = np.array([word for word in temp_review[:10] if word in pretrained_w2v])
    review_vector = []
    for word in words:
        review_vector.append(pretrained_w2v[word])

    # can be the case where the words in the review are not found in the W2V vocabulary
    if len(review_vector)==0:
        review_vector = np.zeros((1, 300))
    review_vector = np.concatenate(review_vector, axis=0)

    # In the case where the total dim of the feature vector is <3000 add the padding with zeros
    if len(review_vector)<3000:
        review_vector = np.concatenate([review_vector, np.zeros(3000-len(review_vector))])
    return review_vector/10, label

def featurization(dataset, concat = False):
    features = []
    y_labels = []
    concat = concat

    for review, label in zip(dataset['review_body'], dataset['label']):
        try:
            if not concat:
                x, y = average_vectors(review, label)
                features.append(np.mean(x, axis=0))
            else:
                x, y = average_vectors_concat(review, label)
                features.append(x)

            y_labels.append(y)

        except:
            pass
    return features, y_labels

'''
Driver code for calculation of Word2Vec Vectors
'''
# Vectors without concatenation
# Average Word2Vec Vectors for train and test data
w2v_pretrain_train_x, w2v_pretrain_train_y = featurization(train)
w2v_pretrain_test_x, w2v_pretrain_test_y = featurization(test)

# Vectors with concatenation
# Concatenated first 10 Word2Vec Vectors for train and test data
w2v_pretrain_train_concat_x, w2v_pretrain_train_concat_y = featurization(train, True)
w2v_pretrain_test_concat_x, w2v_pretrain_test_concat_y = featurization(test, True)
```

In [45]:
```
'''
TF-IDF Feature Extraction for both train and test data
'''
tfidf_vectorizer = TfidfVectorizer(min_df = 0.001)

# Final TFIDF Features
tfidf_X_train = tfidf_vectorizer.fit_transform(list(train['review_body']))
tfidf_X_train = pd.DataFrame(tfidf_X_train.toarray())

tfidf_X_test = tfidf_vectorizer.transform(list(test['review_body']))
tfidf_X_test = pd.DataFrame(tfidf_X_test.toarray())

tfidf_Y_train = train['label']
tfidf_Y_test = test['label']

tfidf_Y_train = tfidf_Y_train.astype('int')
tfidf_Y_test = tfidf_Y_test.astype('int')
```

In [46]:
```
'''
Training Perceptron Model on Average Word2Vec Features
'''
perceptr_w2v = Perceptron(random_state = 100, eta0=0.1)
perceptr_w2v.fit(w2v_pretrain_train_x, w2v_pretrain_train_y)
Y_pred_w2v_test = perceptr_w2v.predict(w2v_pretrain_test_x)

'''
Training Perceptron Model on TF-IDF Features
'''
perceptr_tfidf = Perceptron(random_state = 100, eta0=0.1)
perceptr_tfidf.fit(tfidf_X_train, tfidf_Y_train)
Y_pred_tfidf_test = perceptr_tfidf.predict(tfidf_X_test)

# Accuracy Calculation
target_names = ['class 1', 'class 2', 'class 3']
report_w2v = classification_report(w2v_pretrain_test_y, Y_pred_w2v_test,
                                   target_names=target_names, output_dict=True)
report_tfidf = classification_report(tfidf_Y_test, Y_pred_tfidf_test,
                                     target_names=target_names, output_dict=True)
```

In [16]:
```
print('Accuracy values PERCEPTRON for w2v and tfidf features:')
print(report_w2v['accuracy'], report_tfidf['accuracy'])
```

```
Accuracy values PERCEPTRON for w2v and tfidf features:
0.5805374728759807 0.6170833333333333
```

In [17]:
```
'''
Training SVM Model on Average Word2Vec Features
'''
svm_w2v = LinearSVC(random_state=100, max_iter=1000)
svm_w2v.fit(w2v_pretrain_train_x, w2v_pretrain_train_y)
Y_pred_w2v_svm_test = svm_w2v.predict(w2v_pretrain_test_x)

'''
Training SVM Model on TFIDF Features
'''
svm_tfidf = LinearSVC(random_state=100, max_iter=1000)
svm_tfidf.fit(tfidf_X_train, tfidf_Y_train)
Y_pred_tfidf_svm_test = svm_tfidf.predict(tfidf_X_test)

# Accuracy Calculation
report_svm_w2v = classification_report(w2v_pretrain_test_y, Y_pred_w2v_svm_test,
                                       target_names=target_names, output_dict=True)
report_svm_tfidf = classification_report(tfidf_Y_test, Y_pred_tfidf_svm_test,
                                         target_names=target_names, output_dict=True)
```

In [18]:
```
print('Accuracy values SVM for w2v and tfidf features:')
print(report_svm_w2v['accuracy'], report_svm_tfidf['accuracy'])
```

```
Accuracy values SVM for w2v and tfidf features:
0.627691537305959 0.6685
```

Question Task 3:

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

Answer:

In my experiments, I found that TF-IDF features outperformed Word2vec features in both the Perceptron and SVM models. Although the SVM model with Word2vec took a long time to train, possibly due to the time required to determine the margin, overall the SVM model performed better than the Perceptron model. The TF-IDF feature set contained 48,000 features per review, while the Word2vec feature set contained only 300 features obtained by averaging all the words in a review. The reason for the poorer performance of Word2vec may be that averaging the word vector values results in the loss of information connecting the feature to the label, and this data is not suitable for simple models like the Perceptron. Furthermore, TF-IDF is a statistical measure that is specific to the dataset, whereas Word2vec embeddings are based on a pretrained vector that may not be specific to this dataset and contain a large amount of unrelated information.

Task 4: Feedforward Neural Networks

In [19]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader,TensorDataset
```

In [20]:
```python
device = torch.device('cpu')
```

In [21]:
```python
'''
FNN: network with two hidden layers, each with 100 and 10 nodes
'''
# For 4(a)
class MLP(nn.Module):
    def __init__(self, classification = "binary", vocab_size = 300):
        super(MLP, self).__init__()
        hidden_1 = 100
        hidden_2 = 10
        self.fc1 = nn.Linear(vocab_size, hidden_1)
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        self.fc3 = nn.Linear(hidden_2, 3)

    def forward(self, x):
        x = x.view(-1, x.shape[1])
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

# For 4(b)
class MLP_concat(nn.Module):
    def __init__(self, classification = "binary", vocab_size = 3000):
        super(MLP_concat, self).__init__()
        hidden_1 = 100
        hidden_2 = 10
        self.fc1 = nn.Linear(vocab_size, hidden_1)
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        self.fc3 = nn.Linear(hidden_2, 3)

    def forward(self, x):
        x = x.view(-1, x.shape[1])
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

model = MLP()
model_concat = MLP_concat()
model = model
model_concat = model_concat
print(model)
print(model_concat)
```

```
MLP(
  (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)
)
MLP_concat(
  (fc1): Linear(in_features=3000, 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)
)
```

-- Task 4(a) using the average Word2Vec vectors

In [22]:
```
train_data=TensorDataset(torch.FloatTensor(w2v_pretrain_train_x), torch.LongTensor(w2v_pretrain_train_y))
test_data=TensorDataset(torch.FloatTensor(w2v_pretrain_test_x), torch.LongTensor(w2v_pretrain_test_y))

# Data Loader
train_batch_size=16
train_loader=DataLoader(train_data, batch_size=train_batch_size, shuffle=True)

test_batch_size=16
test_loader=DataLoader(test_data, batch_size=test_batch_size, shuffle=True)

# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()
criterion = criterion
# specify optimizer (stochastic gradient descent) and learning rate = 0.01
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)

# number of epochs to train the model
n_epochs = 20

# initialize tracker for minimum validation loss
valid_loss_min = np.Inf # set initial "min" to infinity
best_acc = 0

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

    # train the model #
    model.train() # prep model for training
    for data, target in train_loader: # iterates upto number of batch size
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, (target-1))
        # backward pass: compute gradient of the loss with respect to model parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update running training loss
        train_loss += loss.item()*data.size(0)

    # validate the model #
    model.eval() # prep model for evaluation
    correct = 0
    for data, target in test_loader:
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, (target-1))
        # update running validation loss
        valid_loss += loss.item()*data.size(0)
        ypred = output.argmax(dim = 1)
        correct += (ypred == (target-1)).float().sum()

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


    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f} \tEpoch Accuracy: {:.6f}'.format(
        epoch+1,
        train_loss,
        valid_loss,
        correct/len(test_loader.dataset)
        ))
```

```
Epoch: 1	Training Loss: 1.095245	Validation Loss: 1.085394	Epoch Accuracy: 0.438241
Epoch: 2	Training Loss: 1.039120	Validation Loss: 0.984465	Epoch Accuracy: 0.529377
Epoch: 3	Training Loss: 0.932123	Validation Loss: 0.912589	Epoch Accuracy: 0.561008
Epoch: 4	Training Loss: 0.889675	Validation Loss: 0.886198	Epoch Accuracy: 0.579369
Epoch: 5	Training Loss: 0.866262	Validation Loss: 0.868845	Epoch Accuracy: 0.598565
Epoch: 6	Training Loss: 0.846607	Validation Loss: 0.858551	Epoch Accuracy: 0.603155
Epoch: 7	Training Loss: 0.833461	Validation Loss: 0.838942	Epoch Accuracy: 0.622601
Epoch: 8	Training Loss: 0.824110	Validation Loss: 0.836921	Epoch Accuracy: 0.620347
Epoch: 9	Training Loss: 0.818204	Validation Loss: 0.832157	Epoch Accuracy: 0.624019
Epoch: 10	Training Loss: 0.812506	Validation Loss: 0.829871	Epoch Accuracy: 0.625438
Epoch: 11	Training Loss: 0.808316	Validation Loss: 0.823904	Epoch Accuracy: 0.629778
Epoch: 12	Training Loss: 0.804297	Validation Loss: 0.857766	Epoch Accuracy: 0.608746
Epoch: 13	Training Loss: 0.801441	Validation Loss: 0.817335	Epoch Accuracy: 0.633283
Epoch: 14	Training Loss: 0.798735	Validation Loss: 0.823436	Epoch Accuracy: 0.629194
Epoch: 15	Training Loss: 0.794912	Validation Loss: 0.814164	Epoch Accuracy: 0.632699
Epoch: 16	Training Loss: 0.792207	Validation Loss: 0.813080	Epoch Accuracy: 0.635453
Epoch: 17	Training Loss: 0.789171	Validation Loss: 0.815732	Epoch Accuracy: 0.633951
Epoch: 18	Training Loss: 0.785690	Validation Loss: 0.808095	Epoch Accuracy: 0.636872
Epoch: 19	Training Loss: 0.783736	Validation Loss: 0.809173	Epoch Accuracy: 0.640210
Epoch: 20	Training Loss: 0.781308	Validation Loss: 0.846638	Epoch Accuracy: 0.616007
```

-- Test Dataset Accuracy

In [23]:

```
model.eval() # prep model for evaluation
main_tar = []
predss = []
with torch.no_grad():
    for data, target in test_loader:
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, (target-1))
        ypred = output.argmax(dim = 1)
        for i in np.array(target-1):
            main_tar.append(i)
        for j in np.array(ypred):
            predss.append(j)
```

In [24]:

```
print("Accuracy Value")
report = classification_report(main_tar, predss, digits=6, output_dict=True)
print(report['accuracy'])
```

```
Accuracy Value
0.6160073443498582
```

Task 4(b) 10 word vectors concatenated

In [25]:

```
train_data=TensorDataset(torch.FloatTensor(w2v_pretrain_train_concat_x),
                         torch.LongTensor(w2v_pretrain_train_concat_y))

test_data=TensorDataset(torch.FloatTensor(w2v_pretrain_test_concat_x),
                        torch.LongTensor(w2v_pretrain_test_concat_y))

# Data Loader
train_batch_size=16
train_loader=DataLoader(train_data, batch_size=train_batch_size, shuffle=True)

test_batch_size=16
test_loader=DataLoader(test_data, batch_size=test_batch_size, shuffle=True)

# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()
criterion = criterion
optimizer = torch.optim.Adam(model_concat.parameters(), lr=0.002)

# number of epochs to train the model
n_epochs = 20

# initialize tracker for minimum validation loss
valid_loss_min = np.Inf # set initial "min" to infinity
best_acc = 0

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

    # train the model #
    model_concat.train() # prep model for training
    for data, target in train_loader: # iterates upto number of batch size
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model_concat(data)
        # calculate the loss
        loss = criterion(output, target-1)
        # backward pass: compute gradient of the loss with respect to model parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update running training loss
        train_loss += loss.item()*data.size(0)

    # validate the model #
    model_concat.eval() # prep model for evaluation
    correct = 0
    for data, target in test_loader:
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model_concat(data)
        # calculate the loss
        loss = criterion(output, target-1)
        # update running validation loss
        valid_loss += loss.item()*data.size(0)
        ypred = output.argmax(dim = 1)
        correct += (ypred == target-1).float().sum()

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


    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f} \tEpoch Accuracy: {:.6f}'.format(
        epoch+1,
        train_loss,
        valid_loss,
        correct/len(test_loader.dataset)
        ))
```

```
Epoch: 1        Training Loss: 0.941390         Validation Loss: 0.911243       Epoch Accuracy: 0.561833
Epoch: 2        Training Loss: 0.857803         Validation Loss: 0.889943       Epoch Accuracy: 0.579833
Epoch: 3        Training Loss: 0.801910         Validation Loss: 0.908848       Epoch Accuracy: 0.575667
Epoch: 4        Training Loss: 0.720994         Validation Loss: 0.957854       Epoch Accuracy: 0.569333
Epoch: 5        Training Loss: 0.607623         Validation Loss: 1.062071       Epoch Accuracy: 0.560583
Epoch: 6        Training Loss: 0.479041         Validation Loss: 1.256614       Epoch Accuracy: 0.548333
Epoch: 7        Training Loss: 0.363243         Validation Loss: 1.564067       Epoch Accuracy: 0.535583
Epoch: 8        Training Loss: 0.275674         Validation Loss: 1.829120       Epoch Accuracy: 0.534000
Epoch: 9        Training Loss: 0.215885         Validation Loss: 2.109466       Epoch Accuracy: 0.533000
Epoch: 10       Training Loss: 0.182354         Validation Loss: 2.507781       Epoch Accuracy: 0.529833
Epoch: 11       Training Loss: 0.159975         Validation Loss: 2.592043       Epoch Accuracy: 0.527500
Epoch: 12       Training Loss: 0.141952         Validation Loss: 2.872877       Epoch Accuracy: 0.524500
Epoch: 13       Training Loss: 0.132474         Validation Loss: 3.010039       Epoch Accuracy: 0.524583
Epoch: 14       Training Loss: 0.118733         Validation Loss: 3.029299       Epoch Accuracy: 0.525500
Epoch: 15       Training Loss: 0.107735         Validation Loss: 3.221778       Epoch Accuracy: 0.523667
Epoch: 16       Training Loss: 0.103279         Validation Loss: 3.461644       Epoch Accuracy: 0.525583
Epoch: 17       Training Loss: 0.098106         Validation Loss: 3.558471       Epoch Accuracy: 0.524333
Epoch: 18       Training Loss: 0.095432         Validation Loss: 3.643724       Epoch Accuracy: 0.524167
Epoch: 19       Training Loss: 0.097249         Validation Loss: 3.791314       Epoch Accuracy: 0.528083
Epoch: 20       Training Loss: 0.088351         Validation Loss: 3.740162       Epoch Accuracy: 0.516167
```

In [26]:

```
model_concat.eval() # prep model for evaluation
main_tar = []
predss = []
with torch.no_grad():
    for data, target in test_loader:
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model_concat(data)
        # calculate the loss
        loss = criterion(output, target-1)
        # update running validation loss
        ypred = output.argmax(dim = 1)
        for i in np.array(target-1):
            main_tar.append(i)
        for j in np.array(ypred):
            predss.append(j)
```

In [27]:

```
print("Accuracy Value: After contenating first 10 review vectors")
concate_report = classification_report(main_tar, predss, digits=6, output_dict=True)
print(concate_report['accuracy'])
```

```
Accuracy Value: After contenating first 10 review vectors
0.5161666666666667
```

Question Task 4:

What do you conclude by comparing accuracy values you obtain with those obtained in the “Simple Models” section?

Answer:

Based on the outcomes of the basic models, the accuracy values obtained were as follows:

- For W2V: 0.5805 (Perceptron) and 0.6276 (SVM)
- For TF-IDF: 0.6170 (Perceptron) and 0.6685 (SVM)
- For FNN model a): 0.6160
- For FNN model b): 0.5161
- In my opinion, the utilization of W2V features showed improvement in FNN model a), whereas in FNN model b), the concatenated W2V features of the first 10 words did not seem to have a strong connection with the labels. This is because not all reviews express their sentiment in the first 10 words, and some reviews have less than 10 words, which were concatenated with zero value vectors. Consequently, the performance of model b) was not as good as that of a). Furthermore, Neural Network models are less sensitive to hyperparameters, and the preparation of training data is more straightforward and systematic. With the advancement of computing power, FNN models have become far more effective than traditional SVM and Perceptron models.

In [28]:
```
# clearing some memory
del globals()['tfidf_X_train'], globals()['tfidf_X_test'],
globals()['tfidf_Y_train'], globals()['tfidf_Y_test']

del globals()['w2v_pretrain_train_x'], globals()['w2v_pretrain_train_y']
del globals()['w2v_pretrain_test_x'], globals()['w2v_pretrain_test_y']

del globals()['w2v_pretrain_train_concat_x'], globals()['w2v_pretrain_train_concat_y']
del globals()['w2v_pretrain_test_concat_x'], globals()['w2v_pretrain_test_concat_y']

del globals()['model'], globals()['model_concat'], globals()['train_data'], globals()['test_data']
del globals()['Y_pred_w2v_test'], globals()['Y_pred_tfidf_test'],
globals()['Y_pred_w2v_svm_test'], globals()['Y_pred_tfidf_svm_test']

del globals()['train_loader'], globals()['test_loader']
del globals()['main_tar'], globals()['predss']

gc.collect()
```

Out[28]: 63

Task 5 Recurrent Neural Networks

In [29]:
```
'''
limiting the maximum review length to 20 by truncating longer reviews and padding
shorter reviews with a null value (0)
'''
# Average word2Vec vectors
def average_vectors_rnn(review):
    temp_review = review.split(' ')

    words = np.array([word for word in temp_review[:20] if word in pretrained_w2v])

    review_vector = []
    for word in words:
        review_vector.append(pretrained_w2v[word])
    review_vector = np.array(review_vector)

    # can be the case where the words in the review are not found in the W2V vocabulary
    if len(review_vector)==0:
        review_vector = np.zeros((20, 300))

    # In the case where the total dim of the feature vector is <20 add the padding with zeros
    elif len(review_vector)<20:
        review_vector = np.concatenate([review_vector, np.zeros((20-len(review_vector), 300))])

    return review_vector

def featurization_rnn(dataset):
    features = []

    for review in dataset['review_body']:
        x = average_vectors_rnn(review)
        features.append(x)

    return features

'''
Review Vectors for  first 20 words each
'''
w2v_pretrain_train_x = featurization_rnn(train)
w2v_pretrain_train_y = train['label']
w2v_pretrain_test_x = featurization_rnn(test)
w2v_pretrain_test_y = test['label']
```

- Task 5(a): Train a simple RNN for sentiment analysis.

In [30]:
```
class rnn_model(nn.Module):
    def __init__(self):
        super(rnn_model, self).__init__()

        self.rnn_layer = nn.RNN(300, 20, batch_first = True)
        self.fc = nn.Linear(20,3)

    def forward(self, input):
        output = input.view(-1,20,300)
        output, hidden = self.rnn_layer(output)
        output=self.fc(output[:,-1,:])
        return output

model = rnn_model()
print(model)
```

```
rnn_model(
  (rnn_layer): RNN(300, 20, batch_first=True)
  (fc): Linear(in_features=20, out_features=3, bias=True)
)
```

In [31]:
```
train_data=TensorDataset(torch.FloatTensor(w2v_pretrain_train_x), torch.LongTensor(w2v_pretrain_train_y))
test_data=TensorDataset(torch.FloatTensor(w2v_pretrain_test_x), torch.LongTensor(w2v_pretrain_test_y))

# Data Loader
train_batch_size=200
train_loader=DataLoader(train_data, batch_size=train_batch_size, shuffle=True)

test_batch_size=200
test_loader=DataLoader(test_data, batch_size=test_batch_size, shuffle=True)
```

In [32]:
```python
# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()
criterion = criterion
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# number of epochs to train the model
n_epochs = 20

# initialize tracker for minimum validation loss
valid_loss_min = np.Inf # set initial "min" to infinity
best_acc = 0

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

    # train the model #
    model.train() # prep model for training
    for data, target in train_loader: # iterates upto number of batch size
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target-1)
        # backward pass: compute gradient of the loss with respect to model parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update running training loss
        train_loss += loss.item()*data.size(0)

    # validate the model #
    model.eval() # prep model for evaluation
    correct = 0
    for data, target in test_loader:
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target-1)
        # update running validation loss
        valid_loss += loss.item()*data.size(0)
        ypred = output.argmax(dim = 1)
        correct += (ypred == target-1).float().sum()

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


    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f} \tEpoch Accuracy: {:.6f}'.format(
        epoch+1,
        train_loss,
        valid_loss,
        correct/len(test_loader.dataset)
        ))

model.eval() # prep model for evaluation
main_tar = []
predss = []
with torch.no_grad():
    for data, target in test_loader:
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model(data)
        # calculate the loss
        loss = criterion(output, target-1)
        ypred = output.argmax(dim = 1)
        for i in np.array(target-1):
            main_tar.append(i)
        for j in np.array(ypred):
            predss.append(j)
```

```
Epoch: 1        Training Loss: 1.068041        Validation Loss: 0.972443        Epoch Accuracy: 0.515667
Epoch: 2        Training Loss: 0.946077        Validation Loss: 0.931448        Epoch Accuracy: 0.541500
Epoch: 3        Training Loss: 0.907505        Validation Loss: 0.898142        Epoch Accuracy: 0.570083
Epoch: 4        Training Loss: 0.886334        Validation Loss: 0.894273        Epoch Accuracy: 0.573083
Epoch: 5        Training Loss: 0.873569        Validation Loss: 0.891884        Epoch Accuracy: 0.570833
Epoch: 6        Training Loss: 0.868208        Validation Loss: 0.889946        Epoch Accuracy: 0.574583
Epoch: 7        Training Loss: 0.864372        Validation Loss: 0.876857        Epoch Accuracy: 0.588250
Epoch: 8        Training Loss: 0.857649        Validation Loss: 0.876751        Epoch Accuracy: 0.583750
Epoch: 9        Training Loss: 0.851378        Validation Loss: 0.871992        Epoch Accuracy: 0.591083
Epoch: 10       Training Loss: 0.849071        Validation Loss: 0.881663        Epoch Accuracy: 0.582083
Epoch: 11       Training Loss: 0.844632        Validation Loss: 0.865923        Epoch Accuracy: 0.598500
Epoch: 12       Training Loss: 0.841565        Validation Loss: 0.861431        Epoch Accuracy: 0.599083
Epoch: 13       Training Loss: 0.837749        Validation Loss: 0.866031        Epoch Accuracy: 0.591917
Epoch: 14       Training Loss: 0.836802        Validation Loss: 0.857653        Epoch Accuracy: 0.607250
Epoch: 15       Training Loss: 0.831219        Validation Loss: 0.857527        Epoch Accuracy: 0.606167
Epoch: 16       Training Loss: 0.828503        Validation Loss: 0.852041        Epoch Accuracy: 0.608917
Epoch: 17       Training Loss: 0.823943        Validation Loss: 0.854108        Epoch Accuracy: 0.617333
Epoch: 18       Training Loss: 0.827398        Validation Loss: 0.853338        Epoch Accuracy: 0.607333
Epoch: 19       Training Loss: 0.817418        Validation Loss: 0.847530        Epoch Accuracy: 0.614417
Epoch: 20       Training Loss: 0.814677        Validation Loss: 0.859046        Epoch Accuracy: 0.603333
```

In [33]:
```
print("Accuracy Value: RNN")
rnn_report = classification_report(main_tar, predss, digits=6, output_dict=True)
print(rnn_report['accuracy'])
```

```
Accuracy Value: RNN
0.6033333333333334
```

Task 5(b): Considering a gated recurrent unit cell.

In [34]:
```
class gru_model(nn.Module):
    def __init__(self):
        super(gru_model, self).__init__()
        self.gru_layer = nn.GRU(300, 20, batch_first = True)
        self.fc = nn.Linear(20,3)

    def forward(self, input):
        output = input.view(-1,20,300)
        output, hidden = self.gru_layer(output)
        output=self.fc(output[:,-1,:])
        return output

model_gru = gru_model()
print(model_gru)
```

```
gru_model(
  (gru_layer): GRU(300, 20, batch_first=True)
  (fc): Linear(in_features=20, out_features=3, bias=True)
)
```

In [35]:
```python
# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()
criterion = criterion
optimizer = torch.optim.Adam(model_gru.parameters(), lr=0.001)

# number of epochs to train the model
n_epochs = 20

# initialize tracker for minimum validation loss
valid_loss_min = np.Inf # set initial "min" to infinity
best_acc = 0

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

    # train the model #
    model_gru.train() # prep model for training
    for data, target in train_loader: # iterates upto number of batch size
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model_gru(data)
        # calculate the loss
        loss = criterion(output, target-1)
        # backward pass: compute gradient of the loss with respect to model parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update running training loss
        train_loss += loss.item()*data.size(0)

    # validate the model #
    model_gru.eval() # prep model for evaluation
    correct = 0
    for data, target in test_loader:
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model_gru(data)
        # calculate the loss
        loss = criterion(output, target-1)
        # update running validation loss
        valid_loss += loss.item()*data.size(0)
        ypred = output.argmax(dim = 1)
        correct += (ypred == target-1).float().sum()

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


    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f} \tEpoch Accuracy: {:.6f}'.format(
        epoch+1,
        train_loss,
        valid_loss,
        correct/len(test_loader.dataset)
        ))

model_gru.eval() # prep model for evaluation
main_tar = []
predss = []
with torch.no_grad():
    for data, target in test_loader:
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model_gru(data)
        # calculate the loss
        loss = criterion(output, target-1)
        ypred = output.argmax(dim = 1)
        for i in np.array(target-1):
            main_tar.append(i)
        for j in np.array(ypred):
            predss.append(j)
```

```
Epoch: 1        Training Loss: 1.008304         Validation Loss: 0.906597         Epoch Accuracy: 0.556667
Epoch: 2        Training Loss: 0.871335         Validation Loss: 0.866509         Epoch Accuracy: 0.584833
Epoch: 3        Training Loss: 0.833902         Validation Loss: 0.827483         Epoch Accuracy: 0.618750
Epoch: 4        Training Loss: 0.806703         Validation Loss: 0.802922         Epoch Accuracy: 0.629500
Epoch: 5        Training Loss: 0.785005         Validation Loss: 0.804980         Epoch Accuracy: 0.636667
Epoch: 6        Training Loss: 0.769362         Validation Loss: 0.777338         Epoch Accuracy: 0.649000
Epoch: 7        Training Loss: 0.758515         Validation Loss: 0.770910         Epoch Accuracy: 0.656667
Epoch: 8        Training Loss: 0.749954         Validation Loss: 0.770134         Epoch Accuracy: 0.654833
Epoch: 9        Training Loss: 0.741368         Validation Loss: 0.766501         Epoch Accuracy: 0.656750
Epoch: 10       Training Loss: 0.734971         Validation Loss: 0.764203         Epoch Accuracy: 0.658583
Epoch: 11       Training Loss: 0.727820         Validation Loss: 0.765951         Epoch Accuracy: 0.652750
Epoch: 12       Training Loss: 0.722957         Validation Loss: 0.768525         Epoch Accuracy: 0.652750
Epoch: 13       Training Loss: 0.718677         Validation Loss: 0.761530         Epoch Accuracy: 0.656333
Epoch: 14       Training Loss: 0.713302         Validation Loss: 0.769965         Epoch Accuracy: 0.652000
Epoch: 15       Training Loss: 0.708194         Validation Loss: 0.764820         Epoch Accuracy: 0.657500
Epoch: 16       Training Loss: 0.706288         Validation Loss: 0.765403         Epoch Accuracy: 0.660750
Epoch: 17       Training Loss: 0.700849         Validation Loss: 0.765382         Epoch Accuracy: 0.655500
Epoch: 18       Training Loss: 0.695064         Validation Loss: 0.766828         Epoch Accuracy: 0.659417
Epoch: 19       Training Loss: 0.691395         Validation Loss: 0.765902         Epoch Accuracy: 0.658417
Epoch: 20       Training Loss: 0.688022         Validation Loss: 0.769943         Epoch Accuracy: 0.657083
```

In [36]:

```python
print("Accuracy Value: GRU")
gru_report = classification_report(main_tar, predss, digits=6, output_dict=True)
print(gru_report['accuracy'])
```

```
Accuracy Value: GRU
0.6570833333333334
```

- Task 5(c): Considering a LSTM unit cell.

In [37]:

```python
class lstm_model(nn.Module):
    def __init__(self):
        super(lstm_model, self).__init__()
        self.lstm_layer = nn.LSTM(300, 20, batch_first = True)
        self.fc = nn.Linear(20,3)

    def forward(self, input):
        output = input.view(-1,20,300)
        output, hidden = self.lstm_layer(output)
        output=self.fc(output[:,-1,:])
        return output

model_lstm = lstm_model()
print(model_lstm)
```

```
lstm_model(
  (lstm_layer): LSTM(300, 20, batch_first=True)
  (fc): Linear(in_features=20, out_features=3, bias=True)
)
```

In [38]:
```
# specify loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()
criterion = criterion
# specify optimizer (stochastic gradient descent) and learning rate = 0.01
optimizer = torch.optim.Adam(model_lstm.parameters(), lr=0.001)

# number of epochs to train the model
n_epochs = 20

# initialize tracker for minimum validation loss
valid_loss_min = np.Inf # set initial "min" to infinity
best_acc = 0

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

    # train the model #
    model_lstm.train() # prep model for training
    for data, target in train_loader: # iterates upto number of batch size
        # clear the gradients of all optimized variables
        optimizer.zero_grad()
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model_lstm(data)
        # calculate the loss
        loss = criterion(output, target-1)
        # backward pass: compute gradient of the loss with respect to model parameters
        loss.backward()
        # perform a single optimization step (parameter update)
        optimizer.step()
        # update running training loss
        train_loss += loss.item()*data.size(0)

    # validate the model #
    model_lstm.eval() # prep model for evaluation
    correct = 0
    for data, target in test_loader:
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model_lstm(data)
        # calculate the loss
        loss = criterion(output, target-1)
        # update running validation loss
        valid_loss += loss.item()*data.size(0)
        ypred = output.argmax(dim = 1)
        correct += (ypred == target-1).float().sum()

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


    print('Epoch: {} \tTraining Loss: {:.6f} \tValidation Loss: {:.6f} \tEpoch Accuracy: {:.6f}'.format(
        epoch+1,
        train_loss,
        valid_loss,
        correct/len(test_loader.dataset)
        ))

model_lstm.eval() # prep model for evaluation
main_tar = []
predss = []
with torch.no_grad():
    for data, target in test_loader:
        # forward pass: compute predicted outputs by passing inputs to the model
        output = model_lstm(data)
        # calculate the loss
        loss = criterion(output, target-1)
        ypred = output.argmax(dim = 1)
        for i in np.array(target-1):
            main_tar.append(i)
        for j in np.array(ypred):
            predss.append(j)
```

```
Epoch: 1        Training Loss: 1.018164         Validation Loss: 0.918034         Epoch Accuracy: 0.549583
Epoch: 2        Training Loss: 0.875050         Validation Loss: 0.864892         Epoch Accuracy: 0.594083
Epoch: 3        Training Loss: 0.839194         Validation Loss: 0.835017         Epoch Accuracy: 0.619000
Epoch: 4        Training Loss: 0.814308         Validation Loss: 0.813177         Epoch Accuracy: 0.631583
Epoch: 5        Training Loss: 0.793888         Validation Loss: 0.799908         Epoch Accuracy: 0.642583
Epoch: 6        Training Loss: 0.779068         Validation Loss: 0.807967         Epoch Accuracy: 0.637917
Epoch: 7        Training Loss: 0.767645         Validation Loss: 0.794317         Epoch Accuracy: 0.641583
Epoch: 8        Training Loss: 0.754578         Validation Loss: 0.789441         Epoch Accuracy: 0.649917
Epoch: 9        Training Loss: 0.746755         Validation Loss: 0.777910         Epoch Accuracy: 0.657167
Epoch: 10       Training Loss: 0.737703         Validation Loss: 0.784912         Epoch Accuracy: 0.654667
Epoch: 11       Training Loss: 0.729838         Validation Loss: 0.772910         Epoch Accuracy: 0.655083
Epoch: 12       Training Loss: 0.723032         Validation Loss: 0.772639         Epoch Accuracy: 0.653833
Epoch: 13       Training Loss: 0.715835         Validation Loss: 0.771529         Epoch Accuracy: 0.654500
Epoch: 14       Training Loss: 0.709885         Validation Loss: 0.772301         Epoch Accuracy: 0.655917
Epoch: 15       Training Loss: 0.702764         Validation Loss: 0.770811         Epoch Accuracy: 0.658583
Epoch: 16       Training Loss: 0.697297         Validation Loss: 0.773639         Epoch Accuracy: 0.658083
Epoch: 17       Training Loss: 0.695691         Validation Loss: 0.778288         Epoch Accuracy: 0.658667
Epoch: 18       Training Loss: 0.686277         Validation Loss: 0.767325         Epoch Accuracy: 0.656583
Epoch: 19       Training Loss: 0.683824         Validation Loss: 0.770309         Epoch Accuracy: 0.657667
Epoch: 20       Training Loss: 0.678004         Validation Loss: 0.769233         Epoch Accuracy: 0.657833
```

In [41]:

```
print("Accuracy Value: LSTM")
lstm_report = classification_report(main_tar, predss, digits=6, output_dict=True)
print(lstm_report['accuracy'])
```

```
Accuracy Value: LSTM
0.6578333333333334
```

Question Task 5:

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

Answer:

For Simple RNN, GRU and LSTM we got the respective accuracies of 0.6033, 0.6570 and 0.6578. To summarize, the GRU has shown a slight improvement in accuracy compared to traditional RNNs and LSTM also performs slightly better than GRU. While RNNs may encounter issues with vanishing or exploding gradients, leading to decreased accuracy, Gated RNNs, such as the GRU, have mechanisms that allow them to learn long-term dependencies and regulate the amount of information they pass on. The use of the tanh function in GRUs further helps to address the problem of vanishing and exploding gradients. One reason for why LSTM outperforms GRU is that LSTMs have more gating mechanisms than GRUs. Specifically, LSTMs have three gating mechanisms: input, forget, and output gates. This extra gate, the forget gate, allows the LSTM to selectively forget information from previous time steps, which can be useful for tasks where some of the historical information is no longer relevant.