shweta

March 1, 2023

1 1. Dataset Generation

- In order to prepare the data before preprocessing first I read columns "review_body", and "star_rating" from the CSV file to store as my data frame(as per the HW guidelines).
- After this, I separated my data into three classes (1, 2, 3) using a dictionary and added randomly shuffled the data to get different values at each execution.
- Finally, I picked 20000 rows from each of the three classes to create 60000 data set for further analysis.

[ ]:
```
import pandas as pd
import numpy as np
import random
from tqdm import tqdm
import nltk
import warnings
warnings.filterwarnings("ignore")
import re
from bs4 import BeautifulSoup
```

[ ]:
```
dataframe = pd.read_csv("data.tsv",sep="\t",usecols =␣
↪["review_body","star_rating"],on_bad_lines='skip')
```

[ ]:
```
dataframe=dataframe.dropna()

ratingDict = {1:1, 2:1, 3:2, 4:3, 5:3}
filteredDataFrame = dataframe.replace({"star_rating": ratingDict})
filteredDataFrame = filteredDataFrame.sample(frac=1, random_state=14).
↪reset_index(drop=True)
```

[ ]:
```
rating1Data = filteredDataFrame[filteredDataFrame['star_rating']==1].head(20000)
rating2Data = filteredDataFrame[filteredDataFrame['star_rating']==2].head(20000)
rating3Data = filteredDataFrame[filteredDataFrame['star_rating']==3].head(20000)

finalRatingsData = rating1Data.append(rating2Data).append(rating3Data).
↪reset_index()
finalRatingsData['star_rating']=finalRatingsData['star_rating'].astype(int)
```

[ ]:
```
finalRatingsData['review_body'] = finalRatingsData["review_body"].apply(str)␣
↪#converts review_body column to string type
```

[ ]:
```
averageStringLenBeforeDataCleaning, stringLength = 0, 0
for ratings in finalRatingsData['review_body']:
  stringLength += len(ratings)

averageStringLenBeforeDataCleaning = stringLength /␣
↪len(finalRatingsData['review_body'])
print("Average reviews length before data cleaning : ",␣
↪averageStringLenBeforeDataCleaning)
```

```
Average reviews length before data cleaning :  292.6594333333333
```

1.0.1 Data Cleaning

For data cleaning, I have created a common function that will clean data for the following cases: - Convert the reviews data into lowercase characters - b. Remove numerical characters from the reviews data - c. Remove punctuation marks from the reviews data - d. Remove extra spaces from the reviews data - e. Remove URLs from the reviews data - f. Remove HTML tags from the reviews data

[ ]:
```
def cleanReviewData(column):
  column = column.lower() #converts string to lowercase
  column = re.sub(r'\d+','',column) #remove numerical characters from the string
  column = re.sub(r'[^\w\s]','', column) #removes punctuations from the string
  column = column.strip() #remove extra spaces from the string
  column = column.replace('\\S*\\.com\\b','') #remove .com ending URLs from the␣
↪string
  column = column.replace('http[s]?://(?:[a-zA-Z]|[0-9]|[$-_@.&+]|[!*\(\),]|(?:
↪%[0-9a-fA-F][0-9a-fA-F]))+', '') #remove http URLs from the string
  column = column.replace('https?://\S+|www\.\S+','') #remove www URLs from the␣
↪string
  column = column.replace('[\w\.-]+@[\w\.-]+\.\w', '') #remove email address␣
↪from the string
  column = column.replace('[^a-zA-Z ]', '') #remove non alphabetical characters␣
↪from the string
  column = column.replace('\s+', ' ')


  return column

def removehtml(text):
    return BeautifulSoup(text, 'html.parser').get_text()
```

```
finalRatingsData['review_body'] = finalRatingsData["review_body"].
 ↪apply(cleanReviewData)
finalRatingsData['review_body'] = finalRatingsData["review_body"].apply(lambda␣
 ↪x: removehtml(x))
finalRatingsData
```

```
[ ]:        index  star_rating                                        review_body
     0          6            1  this shampoo and conditioner left my hair feel…
     1          8            1                    made my scalp itch so i sent it back
     2         20            1  i was shocked im still looking for my return l…
     3         24            1  if this is real it is not as good as what i bo…
     4         26            1  not very polarized   tight fit gives me a head…
     …        …            …                                                …
     59995  31753            3  best leave in conditioner i have ever used  i …
     59996  31754            3  i was opening the borage capsules and now i ju…
     59997  31755            3  this is a wonderful fragrance  it is very allu…
     59998  31756            3  the most aromatic scent i have found thus far …
     59999  31758            3  razor works great i have had it for over a yea…

     [60000 rows x 3 columns]
```

```
[ ]: averageStringLengAfterDataCleaning, stringLength = 0, 0
     for ratings in finalRatingsData['review_body']:
       stringLength += len(ratings)

     averageStringLengAfterDataCleaning = stringLength /␣
      ↪len(finalRatingsData['review_body'])
     print("Average reviews length after data cleaning : ",␣
      ↪averageStringLengAfterDataCleaning)
```

```
Average reviews length after data cleaning :  281.15886666666665
```

1.0.2 Data Preprocessing

- For data preprocessing, I have used the NLTK package to first remove all the stop words from the dataset.
- After this, I performed lemmatization using WordNetLemmatizer from the NLTK package.
- And finally divided the dataset into training set (80%) and testing set (20%)

```
[ ]: nltk.download('stopwords')
     from nltk.corpus import stopwords
     stop_words = set(stopwords.words('english'))
     finalRatingsData['review_body'] = finalRatingsData['review_body'].apply(lambda␣
      ↪key: ' '.join([word for word in key.split() if word not in (stop_words)]))
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/shwetakumari/nltk_data…
```

```
[nltk_data]   Package stopwords is already up-to-date!
```

```
[ ]: from nltk.stem import WordNetLemmatizer
nltk.download('wordnet')
nltk.download('omw-1.4')
w_tokenizer = nltk.tokenize.WhitespaceTokenizer()
lemmatizer = nltk.stem.WordNetLemmatizer()
def lemmatize_text(text):
    return " ".join([lemmatizer.lemmatize(w) for w in w_tokenizer.
 ↪tokenize(text)])
finalRatingsData['review_body'] = finalRatingsData['review_body'].
 ↪apply(lemmatize_text)
```

```
[nltk_data] Downloading package wordnet to
[nltk_data]     /Users/shwetakumari/nltk_data…
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package omw-1.4 to
[nltk_data]     /Users/shwetakumari/nltk_data…
[nltk_data]   Package omw-1.4 is already up-to-date!
```

```
[ ]: averageStringLengAfterDataPreprocessing, stringLength = 0, 0
for ratings in finalRatingsData['review_body'].to_list():
  stringLength += len(ratings)

averageStringLengAfterDataPreprocessing = stringLength /␣
 ↪len(finalRatingsData['review_body'])
print("Average reviews length after data preprocessing : ",␣
 ↪averageStringLengAfterDataPreprocessing)
```

```
Average reviews length after data preprocessing :  175.12863333333334
```

```
[ ]: from sklearn.model_selection import train_test_split
X = finalRatingsData['review_body']
Y = finalRatingsData['star_rating']
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.2,␣
 ↪random_state=0)

X_train = X_train.reset_index(drop = True)
X_test = X_test.reset_index(drop = True)
Y_train = Y_train.reset_index(drop = True)
Y_test = Y_test.reset_index(drop = True)
```

2 2. Word Embeddings

- First I loaded pre-trained Google News dataset Word2vec model and tested it on two sample examples.

- Then I tested the model on three of my own examples to get the similarity score.
- After that I trained Word2vec model on reviews dataset to get similarith score on the given dataset.
- Results of both the cases are as follows:

2.0.1 a)

[ ]:
```
import gensim.models as genmodels
import gensim.downloader as api
googleW2V = api.load('word2vec-google-news-300')
```

[ ]:
```
# Sample Examples

print(googleW2V.most_similar(positive=['king','woman'], negative=['man'],␣
↪topn=1))
print(googleW2V.similarity('excellent', 'outstanding'))
```

```
[('queen', 0.7118192911148071)]
0.5567486
```

[ ]:
```
# Three different examples

print(googleW2V.most_similar(positive=['cat','kittens'], negative=['hamster'],␣
↪topn=1))
print(googleW2V.similarity('happy', 'pleased'))
print(googleW2V.similarity('coffee', 'brew'))
```

```
[('cats', 0.7502572536468506)]
0.6632171
0.5059545
```

2.0.2 b)

[ ]:
```
sent_corpus = X_train
for i in range(0, len(sent_corpus)):
  sent_corpus[i] = sent_corpus[i].split(" ")

sent_corpus_test = X_test
for i in range(0, len(sent_corpus_test)):
  sent_corpus_test[i] = sent_corpus_test[i].split(" ")
```

[ ]:
```
model = genmodels.Word2Vec(sentences=sent_corpus, vector_size=300, window=13,␣
↪min_count=9)
```

[ ]:
```python
print(model.wv.most_similar(positive=['king','woman'], negative=['man'],␣
 ↪topn=1))
print(model.wv.similarity('excellent', 'outstanding'))
```

```
[('skinceuticals', 0.7845896482467651)]
0.6814426
```

2.0.3 Conclusion

- Since we have used limited size of reviews dataset, the vocab size of Amazon's dataset will be much smaller than the pre-trained Google Word2vec model since it is trained on much larger dataset.
- In the above example we can see the result given for king, woman example is more acurate by pre-trained Google Word2vec model i.e 'Queen' than my model i.e 'skinceuticals' since words like king, queen are not related to amazon's review dataset.
- For the second example i.e 'execellet', 'outstanding' result given by my model is better than pre-trained model because possibility of repeating of these words in higher in amazon's dataset.
- Therefore we can conclude that pre-trained Word2Vec models seems to encode semantic similarities between words better.

3 3. Simple Models

- Using the Google Word2vec features, I am performing the vector averaging on my X_train and X_test data for each review.
- Firstly I am reading each review from the X_train and then checking each word of the review whether is it present in pre-trained model or not. If it is present then I'm appending it to a list. Once a review is parsed I'm taking mean of it if the length of vectors is greater than 0 else I'm appending zeroes to the list. Finally I'm appending the list to my result list.
- I have repeated the same for X_test dataset.
- Thereafter, I have trained and tested averaged word2vec features on Perceptron and SVM and reported the testing accuracy.

[ ]:
```python
X_simple_model = []
for sentence in sent_corpus:
    singleReview = []
    for word in sentence:
        if word in googleW2V:
            singleReview.append(googleW2V[word])
    if len(singleReview) > 0:
        sentence_avg = np.mean(singleReview, axis=0)
    else:
        sentence_avg = np.zeros((300,))
    X_simple_model.append(sentence_avg)
```

```
[ ]: X_simple_model_test = []
for sentence in sent_corpus_test:
    singleReview = []
    for word in sentence:
        if word in googleW2V:
            singleReview.append(googleW2V[word])
    if len(singleReview) > 0:
        sentence_avg = np.mean(singleReview, axis=0)
    else:
        sentence_avg = np.zeros((300,))
    X_simple_model_test.append(sentence_avg)
```

```
[ ]: from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score
svmModel = LinearSVC()
svmModel.fit(X_simple_model,Y_train)
svmModelPredictions=svmModel.predict(X_simple_model_test)
acc_svm=accuracy_score(Y_test,svmModelPredictions)
print("The accuracy obtained in SVM Model using Word Embeddings is:
 ↪",round(acc_svm*100, 4),'%')
```

```
The accuracy obtained in SVM Model using Word Embeddings is: 62.8333
```

```
[ ]: print("The accuracy obtained in SVM Model using TF-IDF is: 66.6872 %") # Values␣
 ↪taken from assignment 1
```

```
The accuracy obtained in SVM Model using TF-IDF is: 66.6872 %
```

```
[ ]: from sklearn.linear_model import Perceptron
from sklearn.metrics import accuracy_score
perceptronModel = Perceptron()
perceptronModel.fit(X_simple_model,Y_train)
perceptronPredictions=perceptronModel.predict(X_simple_model_test)
acc_perceptron=accuracy_score(Y_test,perceptronPredictions)
print("The accuracy obtained in Perceptron Model using Word Embeddings is:
 ↪",round(acc_perceptron*100, 4),'%')
```

```
The accuracy obtained in Perceptron Model using Word Embeddings is: 54.3333 %
```

```
[ ]: print("The accuracy obtained in Perceptron Model TF-IDF is: 59.7402 %")  #␣
 ↪Values taken from assignment 1
```

```
The accuracy obtained in Perceptron Model TF-IDF is: 59.7402 %
```

3.0.1 Conclusion

- As we can see the accuracy obtained by TF-IDF is better than Word Embeddings for both SVM and perceptron. The reason for this could be following:
- TF-IDF reflects the importance of a word in a document and takes into account both the frequency of a word in a document and the frequency of the word across all documents. Word embeddings, on the other hand, represent words as dense vectors in a high-dimensional space
- SVM and perceptron are linear models that rely on features that are linearly separable. TF-IDF scores are often more sparse and linearly separable than word embeddings, making them more suitable for linear models.
- Also, pre-trained word embeddings are typically learned on very large datasets and may not capture the nuances of the specific dataset and small dataset like amazon reviews dataset.

4 4. Feedforward Neural Networks

- I am using PyTorch for my implementation therefore initially I am importing all the required libraries and packages of PyTorch.
- I'm changing my classes index from 1 to 0 then as required by PyTorch.
- After that I have created a custom dataset which will be used by all of the models.
- Then I have set hyperparameters like learning rate, vector size and epochs to be used MLP model. Then I am using my custom dataset to convert numpy array into tensors.
- Finally using DataLoader functionality I am dividing my train and test split tensors into their batch sizes for training.

[ ]:
```
import torch
from torch.utils.data import DataLoader, Dataset, WeightedRandomSampler,␣
↪SubsetRandomSampler
import torch.nn as nn
import torch.nn.functional as F
import functools
import torch.optim as optim
from sklearn.metrics import accuracy_score
import torchvision
import torchvision.transforms as transforms
```

[ ]:
```
class_index ={}
for i in range(1,4):
    class_index[i]=i-1
Y_train.replace(class_index, inplace=True)
Y_test.replace(class_index, inplace=True)
```

[ ]:
```
class ClassifierDataset(Dataset):
    def __init__(self, X_data, Y_data):
        self.X_data = X_data
        self.Y_data = Y_data
```

```
    def __getitem__(self, index):
        return self.X_data[index], self.Y_data[index]

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

4.0.1 a)

- The input size for Feedforward Neural Network for Multilayer Perceptron is 300. The size for Hidden Layer 1 is 100, Hidden Layer 2 is 10 and for Output Layer is 3 as we have 3 classes. I have used relu as the activation function. The main advantage of using the ReLU function over other activation functions is that it does not activate all the neurons at the same time.
- Then in forward propogation, I applies relu function on input data.
- Then, after input layer the data is passed to Hidden Layer 1 and then Hidden Layer 2 and finally to the output layer. In each of these we apply relu.

[ ]:
```
hyperparameters_fnn = { 'lr': 0.0007, 'batch_size' : 10, 'epochs':13,␣
↪'vector_size' : 300,'num_classes' : 3 ,'hidden_size1' : 100,'hidden_size2' :␣
↪10}

mlp_train_torch = ClassifierDataset(torch.from_numpy(np.array(X_simple_model)).
↪float(), torch.from_numpy(Y_train.to_numpy()).long())
mlp_test_torch = ClassifierDataset(torch.from_numpy(np.
↪array(X_simple_model_test)).float(), torch.from_numpy(Y_test.to_numpy()).
↪long())

mlp_train_torch_loader = DataLoader(dataset=mlp_train_torch,␣
↪batch_size=hyperparameters_fnn['batch_size'])
mlp_test_torch_loader = DataLoader(dataset=mlp_test_torch, batch_size=1)
```

[ ]:
```
class MLP(nn.Module):
    def __init__(self, v_size, hidden_size1, hidden_size2, output_dim):
        super().__init__()
        self.relu = nn.ReLU()
        self.fc1 = nn.Linear(v_size, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.fcout = nn.Linear(hidden_size2, output_dim)

    def forward(self, x):
        x=self.fc1(x)
        x=self.relu(x)
        x=self.fc2(x)
        x=self.relu(x)
        preds = self.fcout(x)
        return preds
```

- The I have created MLP model as an object of the main model network class. I am using cross entropy loss and Adams optimizer here. Adam combines the best properties of the AdaGrad and RMSProp algorithms to provide an optimization algorithm that can handle sparse gradients on noisy problems.
- Then I am defining some functions to calculate training and testing accuracy.
- Then I trained my model for 13 epochs in batch size of 10 and learning rate 0.007. I do forward propogation, calculate the loss and accuracy and then perform backward propogation and optimization. Training accuracy after each epoch is calculated. zero_grad is used to reset all accumulated gradients after adam optimizer.
- After successful training of model, I am calculating testing split accuracy using accuracy function.

[ ]:
```
modelMLP =␣
 ↪MLP(hyperparameters_fnn['vector_size'],hyperparameters_fnn['hidden_size1'],hyperparameters_
criterion = nn.CrossEntropyLoss()
optimizer=torch.optim.Adam(modelMLP.parameters(),lr=hyperparameters_fnn['lr'])
```

[ ]:
```
def calculate_training_accuracy(y_pred, y_test):
    correct_pred = (y_pred== y_test)
    acc = correct_pred.sum() / len(correct_pred)
    acc = torch.round(acc * 100)
    return acc

def calculate_testing_accuracy(y_pred, y_test):
    acclist=[]
    for i, j in y_test:
        output = y_pred(i)
        _, predicted = torch.max(output.data, 1)
        correct_pred = (predicted== j)
        acc = correct_pred.sum() / len(correct_pred)
        acc = torch.round(acc * 100)
        acclist.append(acc.item())
    test_correct = sum(acclist)/len(y_test)
    return test_correct
```

[ ]:
```
epochs = hyperparameters_fnn['epochs']
for epoch in range(epochs):
    train_loss = 0.0
    acclist=[]
    modelMLP.train()
    for data, target in mlp_train_torch_loader:
        optimizer.zero_grad()
        output = modelMLP(data)
        loss = criterion(output, target)
        _,predicted = torch.max(output.data, 1)
        acc=calculate_training_accuracy(predicted,target)
        acclist.append(acc.item())
```

```
        loss.backward()
        optimizer.step()
        train_loss += loss.item()*data.size(0)
    train_loss = train_loss/len(mlp_train_torch_loader.dataset)
    train_correct = sum(acclist)/len(mlp_train_torch_loader)
    print('Epoch: {} \tTraining Loss: {:.4f} \tTraining Accuracy: {:.4f}'.
↪format(epoch+1, train_loss,train_correct))
```

```
Epoch: 1        Training Loss: 0.8087   Training Accuracy: 63.6167
Epoch: 2        Training Loss: 0.7921   Training Accuracy: 64.3958
Epoch: 3        Training Loss: 0.7793   Training Accuracy: 65.1229
Epoch: 4        Training Loss: 0.7677   Training Accuracy: 65.5938
Epoch: 5        Training Loss: 0.7568   Training Accuracy: 66.0938
Epoch: 6        Training Loss: 0.7470   Training Accuracy: 66.7500
Epoch: 7        Training Loss: 0.7374   Training Accuracy: 67.2458
Epoch: 8        Training Loss: 0.7281   Training Accuracy: 67.7208
Epoch: 9        Training Loss: 0.7189   Training Accuracy: 68.2125
Epoch: 10       Training Loss: 0.7101   Training Accuracy: 68.8833
Epoch: 11       Training Loss: 0.7011   Training Accuracy: 69.2396
Epoch: 12       Training Loss: 0.6926   Training Accuracy: 69.7042
Epoch: 13       Training Loss: 0.6841   Training Accuracy: 70.1750
```

[ ]:
```
avg_fnn_accuracy = calculate_testing_accuracy(modelMLP, mlp_test_torch_loader)
print("The testing accuracy obtained in FNN with average word2vec is:
↪",avg_fnn_accuracy)
```

```
The testing accuracy obtained in FNN with average word2vec is: 63.075
```

4.0.2 b)

- In part b of FNN to generate input features I have just considered first 10 vectors and concatenated them.
- In this case input size would be 3000 (300 * 10 words) and overall shape of word embeddings for train split is (48000,3000) and for test split is (12000,3000).
- The rest of the code remains same for training the model and calculating accuracy

[ ]:
```
X_fnn_concat_train = []
for sentence in sent_corpus:
    singleReview = []
    for word in sentence:
        if word in googleW2V:
            singleReview.append(googleW2V[word])
        if len(singleReview) == 10:
            break
    if len(singleReview) < 10:
        for i in range(len(singleReview),10):
            singleReview.append(np.zeros((300,)))
```

```
    singleReview = np.concatenate(singleReview, axis=0)
    X_fnn_concat_train.append(singleReview)
X_fnn_concat_train = np.array(X_fnn_concat_train)
```

[ ]:
```
X_fnn_concat_test = []
for sentence in sent_corpus_test:
    singleReview = []
    for word in sentence:
        if word in googleW2V:
            singleReview.append(googleW2V[word])
        if len(singleReview) == 10:
            break
    if len(singleReview) < 10:
        for i in range(len(singleReview),10):
            singleReview.append(np.zeros((300,)))
    singleReview = np.concatenate(singleReview, axis=0)
    X_fnn_concat_test.append(singleReview)
X_fnn_concat_test = np.array(X_fnn_concat_test)
```

[ ]:
```
hyperparameters_fnn.update([('vector_size',3000)])

mlp_train_torch_concat = ClassifierDataset(torch.from_numpy(np.
 ↪array(X_fnn_concat_train)).float(), torch.from_numpy(Y_train.to_numpy()).
 ↪long())
mlp_test_torch_concat = ClassifierDataset(torch.from_numpy(np.
 ↪array(X_fnn_concat_test)).float(), torch.from_numpy(Y_test.to_numpy()).
 ↪long())

mlp_train_torch_loader_concat = DataLoader(dataset=mlp_train_torch_concat,␣
 ↪batch_size=hyperparameters_fnn['batch_size'])
mlp_test_torch_loader_concat = DataLoader(dataset=mlp_test_torch_concat,␣
 ↪batch_size=1)

modelMLP2 =␣
 ↪MLP(hyperparameters_fnn['vector_size'],hyperparameters_fnn['hidden_size1'],hyperparameters_
criterion = nn.CrossEntropyLoss()
optimizer=torch.optim.Adam(modelMLP2.parameters(),lr=hyperparameters_fnn['lr'])
```

[ ]:
```
epochs = hyperparameters_fnn['epochs']

for epoch in range(epochs):
    train_loss_concat = 0.0
    accuracylist_concat=[]
    modelMLP2.train()
    for data, target in mlp_train_torch_loader_concat:
        optimizer.zero_grad()
```

```
        output = modelMLP2(data)
        loss = criterion(output, target)
        _,predicted = torch.max(output.data, 1)
        acc=calculate_training_accuracy(predicted,target)
        accuracylist_concat.append(acc.item())
        loss.backward()
        optimizer.step()
        train_loss_concat += loss.item()*data.size(0)
    train_loss_concat = train_loss_concat/len(mlp_train_torch_loader_concat.
 ↪dataset)
    train_correct_concat = sum(accuracylist_concat)/
 ↪len(mlp_train_torch_loader_concat)
    print('Epoch: {} \tTraining Loss: {:.4f} \tTraining Accuracy: {:.4f}'.
 ↪format(epoch+1, train_loss_concat, train_correct_concat))
```

```
Epoch: 1        Training Loss: 0.8329   Training Accuracy: 61.7021
Epoch: 2        Training Loss: 0.7229   Training Accuracy: 68.3875
Epoch: 3        Training Loss: 0.5738   Training Accuracy: 76.2562
Epoch: 4        Training Loss: 0.4223   Training Accuracy: 83.4500
Epoch: 5        Training Loss: 0.3119   Training Accuracy: 88.1250
Epoch: 6        Training Loss: 0.2502   Training Accuracy: 90.4292
Epoch: 7        Training Loss: 0.2129   Training Accuracy: 91.7938
Epoch: 8        Training Loss: 0.1852   Training Accuracy: 92.9104
Epoch: 9        Training Loss: 0.1640   Training Accuracy: 93.7667
Epoch: 10       Training Loss: 0.1536   Training Accuracy: 94.1833
Epoch: 11       Training Loss: 0.1381   Training Accuracy: 94.8396
Epoch: 12       Training Loss: 0.1314   Training Accuracy: 95.1896
Epoch: 13       Training Loss: 0.1193   Training Accuracy: 95.5563
```

[ ]:
```
fnn_accuracy_concat = calculate_testing_accuracy(modelMLP2,␣
 ↪mlp_test_torch_loader_concat)
print("The testing accuracy obtained in FNN with concatenated word2vec of␣
 ↪length 10 is:",fnn_accuracy_concat)
```

```
The testing accuracy obtained in FNN with concatenated word2vec of length 10 is:
51.65833333333333
```

4.0.3 Conclusion

- The accuracies obtained in both the cases 4(a) using weighted average and 4(b) using first 10 words only is better than Single Layer Perceptron that we are using in Simple Models. The reason being that in FNN Multilayer Perceptron we have 2 hidden layers which helps in better optimization of model through forward and backward propogation.
- After comparing the accuracies of 4(a) and 4(b) with those Single Layer Perceptron used in Simple Models, we can see that it is better for SVM and similar for Perceptron because in

FNN Multilayer Perceptron we have 2 hidden layers which helps in better optimization of model through forward and backward propogation.
- Additionally, comparing 4(a) and 4(b) testing accuracy in part 4(a) is better than part (b). This is because we are using average of all the words in the review which captures the essence of all the important words in that review whereas in part (b) there is a possibility of not considering important words in the review as they may not be the first 10 words which results in loss of words.

5 5. Recurrent Neural Networks

- For RNN to I have created the input features by appending of first 20 words of each review. If the length of review is shorter than 20 I am padding it zero values.
- The overal size of train split using this word embedding is (48000,20,300) and test split is (12000,20,300).
- Also, I have used the same similar hyperparameter as that of FNN for these models to get an accurate comparison between the two.
- The rest of the code is similar to FNN except the model name.

[ ]:
```
X_rnn_concat_train = []
for sentence in sent_corpus:
    singleReview = []
    for i in range(len(sentence)):
        if sentence[i] in googleW2V:
            singleReview.append(googleW2V[sentence[i]])
        if len(singleReview) == 20:
            break
    if len(singleReview) < 20:
        for i in range(len(singleReview),20):
            singleReview.append(np.zeros((300,)))
    singleReview = np.array(singleReview)
    X_rnn_concat_train.append(singleReview)
X_rnn_concat_train = np.array(X_rnn_concat_train)
```

[ ]:
```
X_rnn_concat_test = []
for sentence in sent_corpus_test:
    singleReview = []
    for i in range(len(sentence)):
        if sentence[i] in googleW2V:
            singleReview.append(googleW2V[sentence[i]])
        if len(singleReview) == 20:
            break
    if len(singleReview) < 20:
        for i in range(len(singleReview),20):
            singleReview.append(np.zeros((300,)))
    singleReview = np.array(singleReview)
    X_rnn_concat_test.append(singleReview)
```

```
X_rnn_concat_test = np.array(X_rnn_concat_test)
```

[ ]:
```
hyperparameters_rnn={ 'lr': 0.0007, 'batch_size' : 10, 'epochs':9,␣
 ↪'vector_size' : 300, 'dropout': 0.2,'num_classes' : 3 ,'hidden_size' : 20}

rnn_train_torch = ClassifierDataset(torch.from_numpy(np.
 ↪array(X_rnn_concat_train)).float(), torch.from_numpy(Y_train.to_numpy()).
 ↪long())
rnn_test_torch = ClassifierDataset(torch.from_numpy(np.
 ↪array(X_rnn_concat_test)).float(), torch.from_numpy(Y_test.to_numpy()).
 ↪long())

rnn_train_torch_loader= DataLoader(dataset=rnn_train_torch,␣
 ↪batch_size=hyperparameters_rnn['batch_size'])
rnn_test_torch_loader= DataLoader(dataset=rnn_test_torch, batch_size=1)
```

5.0.1 a) RNN

- In this RNN model we have an RNN cell with the hidden state size of 20. The input size is 300 and the output size is 3.
- I haved used nn.RNN model available to us through PyTorch. Then I create the object of our RNN model and use the same cross entropy loss and optimizer.
- Then I trained my model for 9 epochs in batch size of 10 and learning rate 0.007. I have used similar training method as in FNN.
- After successful training of model, I am calculating testing split accuracy using accuracy function. Finally the accuracy for testing split is also calculated.

[ ]:
```
class ModelRNN(nn.Module):
    def __init__(self, v_size=300, hidden_size=20, num_classes=3):
        super(ModelRNN, self).__init__()
        self.relu = nn.ReLU()
        self.input_dim= v_size
        self.output_dim = num_classes
        self.hidden_size = hidden_size
        self.fc1 = nn.RNN(v_size, hidden_size, batch_first=True)
        self.fc2 = nn.Linear(hidden_size ,num_classes)

    def forward(self, x):
        batch_size = x.shape[0]
        hidden = torch.zeros(1,batch_size, self.hidden_size)
        op,h1=self.fc1(x,hidden)
        output = self.fc2(op[:,-1])
        return output
```

[ ]:
```
model =␣
 ↪ModelRNN(hyperparameters_rnn['vector_size'],hyperparameters_rnn['hidden_size'],hyperparamet
```

```
criterion = nn.CrossEntropyLoss()
optimizer=torch.optim.Adam(model.parameters(),lr=hyperparameters_rnn['lr'])
```

[ ]:
```
epochs = hyperparameters_rnn['epochs']

for epoch in range(epochs):
    train_loss_rnn = 0.0
    accuracylist_rnn=[]
    model.train()
    for data, target in rnn_train_torch_loader:
        optimizer.zero_grad()
        data = torch.tensor(np.array(data),dtype=torch.float)
        output = model(data)
        loss = criterion(output, target)
        _,predicted = torch.max(output.data, 1)
        acc=calculate_training_accuracy(predicted,target)
        accuracylist_rnn.append(acc.item())
        loss.backward()
        optimizer.step()
        train_loss_rnn += loss.item()*data.size(0)
    train_loss_rnn = train_loss_rnn/len(rnn_train_torch_loader.dataset)
    train_correct_rnn = sum(accuracylist_rnn)/len(rnn_train_torch_loader)
    print('Epoch: {} \tTraining Loss: {:.4f} \tTraining Accuracy: {:.4f}'.
 ↪format(epoch+1, train_loss_concat,train_correct_rnn))
```

```
Epoch: 1        Training Loss: 0.1193   Training Accuracy: 49.3417
Epoch: 2        Training Loss: 0.1193   Training Accuracy: 57.1146
Epoch: 3        Training Loss: 0.1193   Training Accuracy: 58.2396
Epoch: 4        Training Loss: 0.1193   Training Accuracy: 59.0271
Epoch: 5        Training Loss: 0.1193   Training Accuracy: 59.6042
Epoch: 6        Training Loss: 0.1193   Training Accuracy: 60.6271
Epoch: 7        Training Loss: 0.1193   Training Accuracy: 61.2917
Epoch: 8        Training Loss: 0.1193   Training Accuracy: 61.3188
Epoch: 9        Training Loss: 0.1193   Training Accuracy: 61.6854
```

[ ]:
```
rnn_accuracy = calculate_testing_accuracy(model, rnn_test_torch_loader)
print("The testing accuracy obtained in RNN with word2vec of length 20 is: ",␣
 ↪rnn_accuracy)
```

```
The testing accuracy obtained in RNN with word2vec of length 20 is:  57.275
```

RNN Conclusion

- The accuracy obtained using FNN with weighted average is better than RNN with using first 20 words since in FNN we are considering weighted average of all the words in the review than first 20 words of the review. Also, in RNN the previous word can affect the value of next word as the output of first word goes to next word.

- The accuracy obtained using RNN using first 20 words is better than FNN with first 10 words since for longer reviews i.e 20 instead of 10, it may capture the word embeddings better.

5.0.2 b) GRU

- The code for GRU is similar to RNN with the only differenc being in using nn.GRU instead of nn.RNN.
- Thus, I have defined the model in GRU with same forward propogation, hyperparamteres and performing training as well with the same method. At the end, I am calculating the testing accuracy for testing split.

[ ]:
```
class ModelGRU(nn.Module):
    def __init__(self, v_size=300, hidden_size=20, num_classes=3):
        super(ModelGRU, self).__init__()
        self.relu = nn.ReLU()
        self.input_dim= v_size
        self.output_dim = num_classes
        self.hidden_size = hidden_size
        self.fc1 = nn.GRU(v_size, hidden_size, batch_first=True)
        self.fc2 = nn.Linear(hidden_size ,num_classes)
    def forward(self, x):
        batch_size = x.shape[0]
        hidden = torch.zeros(1,batch_size, self.hidden_size)
        op,h1=self.fc1(x,hidden)
        output = self.fc2(op[:,-1])
        return output
```

[ ]:
```
model = 
 ↪ModelGRU(hyperparameters_rnn['vector_size'],hyperparameters_rnn['hidden_size'],hyperparamet
criterion = nn.CrossEntropyLoss()
optimizer=torch.optim.Adam(model.parameters(),lr=hyperparameters_rnn['lr'])
```

[ ]:
```
epochs = hyperparameters_rnn['epochs']

for epoch in range(epochs):
    train_loss_gru = 0.0
    accuracylist_gru=[]
    model.train()
    for data, target in rnn_train_torch_loader:
        optimizer.zero_grad()
        data = torch.tensor(np.array(data),dtype=torch.float)
        output = model(data)
        loss = criterion(output, target)
        _,predicted = torch.max(output.data, 1)
        acc=calculate_training_accuracy(predicted,target)
        accuracylist_gru.append(acc.item())
        loss.backward()
```

```
        optimizer.step()
        train_loss_gru += loss.item()*data.size(0)
    train_loss_gru = train_loss_gru/len(rnn_train_torch_loader.dataset)
    train_correct_gru = sum(accuracylist_gru)/len(rnn_train_torch_loader)
    print('Epoch: {} \tTraining Loss: {:.4f} \tTraining Accuracy: {:.4f}'.
 ↪format(epoch+1, train_loss_gru,train_correct_gru ))
```

```
Epoch: 1        Training Loss: 0.8965   Training Accuracy: 55.6708
Epoch: 2        Training Loss: 0.7966   Training Accuracy: 63.9417
Epoch: 3        Training Loss: 0.7676   Training Accuracy: 65.6229
Epoch: 4        Training Loss: 0.7490   Training Accuracy: 66.5187
Epoch: 5        Training Loss: 0.7348   Training Accuracy: 67.2333
Epoch: 6        Training Loss: 0.7228   Training Accuracy: 67.9604
Epoch: 7        Training Loss: 0.7121   Training Accuracy: 68.4562
Epoch: 8        Training Loss: 0.7020   Training Accuracy: 68.9667
Epoch: 9        Training Loss: 0.6925   Training Accuracy: 69.5104
```

[ ]:
```
gru_accuracy = calculate_testing_accuracy(model, rnn_test_torch_loader)
print("The testing accuracy obtained in GRU with word2vec of length 20 is: ",␣
 ↪gru_accuracy)
```

```
The testing accuracy obtained in GRU with word2vec of length 20 is:  64.15
```

5.0.3 c) LSTM

- The code for LSTM is similar to RNN and GRU with the only difference being in using nn.GRU instead of nn.RNN and I have used same hyperparameters for all three models..
- At the end, I am calculating the testing accuracy for testing split.

[ ]:
```
class ModelLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(ModelLSTM, self).__init__()
        self.hidden_size = hidden_size
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        batch_size = x.size(0)
        hidden_state = torch.zeros(1, batch_size, self.hidden_size).to(x.device)
        cell_state = torch.zeros(1, batch_size, self.hidden_size).to(x.device)

        lstm_output, (hidden_state, cell_state) = self.lstm(x, (hidden_state,␣
 ↪cell_state))
        output = self.fc(lstm_output[:, -1, :])

        return output
```

[ ]:
```
model =␣
↪ModelLSTM(hyperparameters_rnn['vector_size'],hyperparameters_rnn['hidden_size'],hyperparame
criterion = nn.CrossEntropyLoss()
optimizer=torch.optim.Adam(model.parameters(),lr=hyperparameters_rnn['lr'])
```

[ ]:
```
epochs = hyperparameters_rnn['epochs']

for epoch in range(epochs):
    train_loss_lstm = 0.0
    accuracylist_lstm=[]
    model.train()
    for data, target in rnn_train_torch_loader:
        optimizer.zero_grad()
        data = torch.tensor(np.array(data),dtype=torch.float)
        output = model(data)
        loss = criterion(output, target)
        _,predicted = torch.max(output.data, 1)
        acc=calculate_training_accuracy(predicted,target)
        accuracylist_lstm.append(acc.item())
        loss.backward()
        optimizer.step()
        train_loss_lstm += loss.item()*data.size(0)
    train_loss_lstm = train_loss_lstm/len(rnn_train_torch_loader.dataset)
    train_correct_lstm = sum(accuracylist_lstm)/len(rnn_train_torch_loader)
    print('Epoch: {} \tTraining Loss: {:.4f} \tTraining Accuracy: {:.4f}'.
↪format(epoch+1, train_loss_lstm,train_correct_lstm ))
```

```
Epoch: 1        Training Loss: 0.9114   Training Accuracy: 54.5708
Epoch: 2        Training Loss: 0.8240   Training Accuracy: 62.1667
Epoch: 3        Training Loss: 0.7856   Training Accuracy: 64.4688
Epoch: 4        Training Loss: 0.7611   Training Accuracy: 65.8083
Epoch: 5        Training Loss: 0.7425   Training Accuracy: 66.8104
Epoch: 6        Training Loss: 0.7266   Training Accuracy: 67.8000
Epoch: 7        Training Loss: 0.7122   Training Accuracy: 68.6833
Epoch: 8        Training Loss: 0.6989   Training Accuracy: 69.5417
Epoch: 9        Training Loss: 0.6862   Training Accuracy: 70.1896
```

[ ]:
```
lstm_accuracy = calculate_testing_accuracy(model, rnn_test_torch_loader)
print("The testing accuracy obtained in LSTM with word2vec of length 20 is: ",␣
↪lstm_accuracy)
```

```
The testing accuracy obtained in LSTM with word2vec of length 20 is:
63.916666666666664
```

5.0.4 Conclusion

- From the above values we can see that GRU performs better than RNN and LSTM and the testing accuracy obtained in GRU is more than RNN as well as LSTM because of the following reasons:
- GRU overcomes the problem of gradient vanishing problem whereas RNNs and LSTMs are prone to the problem of vanishing and exploding gradients.
- LSTMs can be prone to overfitting because they have a large number of parameters that can be tuned. GRUs, on the other hand, have fewer parameters and are therefore less likely to overfit.

6 Final Values Of All Models

Simple Models

- Accuracy for SVM: 62.8333 %
- Accuracy for single layer Perceptron: 54.3333 %

Feedforward Neural Networks

- Accuracy for MLP with average word embeddings: 63.075 %
- Accuracy for MLP with concatenated word embeddings: 51.6583 %

Recurrent Neural Networks

- Accuracy for RNN: 57.275 %
- Accuracy for GRU: 64.15 %
- Accuracy for LSTM: 63.91666 %

6.0.1 References

- https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html
- https://www.kaggle.com/code/mishra1993/pytorch-multi-layer-perceptron-mnist/notebook
- https://dipikabaad.medium.com/finding-the-hidden-sentiments-using-rnns-in-pytorch-f1e1e9638e9c
- https://towardsdatascience.com/pytorch-tabular-multiclass-classification-9f8211a123ab
- https://subscription.packtpub.com/book/data/9781789614381/6/ch06lvl1sec28/training-rnns-for-sentiment-analysis