CSCI 544 - Homework Assignment No 3

`Version` : 1.0
`Editor` : Shubham Sanjay Darekar
`Date` : 10/19/2023
`Execution Time` : ~ 13 min

Final Accuracy Values as of 10/19

Model	Input Features	Accuracy
Perceptron	Mean Word2Vec	82.5 %
	TF_IDF	89.52 %
SVM	Mean Word2Vec	84.28 %
	TF_IDF	90.83 %
FNN	Mean Word2Vec	86.48 %
FNN	First 10 Word2Vec	82.13 %
Simple RNN	First 10 Word2Vec	83.19 %
Gated RNN	First 10 Word2Vec	83.69 %
LSTM	First 10 Word2Vec	83.27 %

Initial tasks

Importing the required libraries

In [ ]:
```
import pandas as pd
import numpy as np

import torch
```

```
import torch.nn as nn
from torch.utils.data import Dataset, DataLoader

from gensim.models import Word2Vec
import gensim.downloader as api
from gensim.parsing.preprocessing import preprocess_string, remove_stopwords,strip_punctuation,strip_tags

from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import Perceptron
from sklearn import svm


def eval(actual,predicted):
    acc = accuracy_score(actual,predicted)
    print('Accuracy = '+str(round(acc,4)))

## Approx run time - 15s
```

Brief description of usage of all the libraries imported

1. Pandas - Used to read and manupulate the data using dataframe
2. NumPy - Used to manupulate the numeric values in the dataset
3. Torch - This module has implementation of all the Neural Network models used in solution
4. Gensim - This module has implementation of Word2vec, which is used for generating doc embeddings
5. Sklearn - This module has implementation of all the ML models used in the solution

Defining the device to be used (Ran on Machine with Intel i7 8th Gen, NVIDIA GPU GTX 1050Ti)

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

Task 1 - Dataset Generation

To read the data read_csv method from pandas is used.

`Parameters` :

> 1. sep ~ Seperater - \t as the values are tab seperated

> 2. engine ~ Using python engine to avoid unsupported format by C engine - `Ref` - https://stackoverflow.com/questions/52774459/engines-in-python-pandas-read-csv
> 3. quoting ~ Set to 3 i.e none to control the quoting field behaviour - `Ref` - https://pandas.pydata.org/docs/reference/api/pandas.read_csv.html
> 4. on_bad_lines ~ Skip option skips the bad lines while reading the dataset

As the review headline and review body are the fields using in classifier and star rating is the field used to label the categories just slicing those fields. The review headline and review body are concatenated with a space in between as headline is helping in model performance.

In [ ]:
```
# reviews_ratings = pd.read_csv("D:\\Applied NLP\\HW1\\amazon_reviews_us_Office_Products_v1_00.tsv", sep='\t', engine
reviews_ratings = pd.read_csv(".\\data.tsv", sep='\t', engine="python", quoting=3, on_bad_lines='skip')
reviews_ratings = reviews_ratings[['review_headline','review_body','star_rating']]

## Filling Na values with blank as one of the column from headline and body might contain useful data
reviews_ratings['review_headline'].fillna("", inplace=True)
reviews_ratings['review_body'].fillna("", inplace=True)


reviews_ratings['review_headline_body'] = reviews_ratings['review_headline'] + " " + reviews_ratings['review_body']

#Random selection of 50,000 rows from each class
df_class1 = reviews_ratings[reviews_ratings['star_rating']>3].sample(n=50000, random_state=34) # setting the random s
df_class1['Class'] = 1 ## Class with higher rating
df_class2 = reviews_ratings[reviews_ratings['star_rating']<=3].sample(n=50000, random_state=34) # setting the random
df_class2['Class'] = 0 ## Class with lower rating

reviews_ratings_final = pd.concat([df_class1, df_class2], ignore_index=True, sort=False).reset_index(drop=True) #conc

## Approx run time - 1min 20s
```

Saving the dataset to parquet file in order to save computation

In [ ]:
```
reviews_ratings_final.to_parquet("AmazonReviewsProcessed.parquet")
##Reading when required
# reviews_ratings_final = pd.read_parquet("AmazonReviewsProcessed.parquet")

## Approx run time - 1s
```

Task 2 - Word Embeddings

Task (a) : Generation of similarities using pretrained “word2vec-google-news-300”

`Ref` - https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html

In [ ]:
```
#Loading the pretrained model
wv_googleNews = api.load('word2vec-google-news-300')

## Approx run time - 1min 20s
```

1. Checking semantic similarity for the King woman man example (Trying both all lower case and first letter capital answers as they are generating different results)

In [ ]:
```
out1 = wv_googleNews.most_similar(["King","Woman"],"Man")
out2 = wv_googleNews.most_similar(["king","woman"],"man")

print(out1)
print(out2)

print('\nOutput for ["King","Woman"],"Man":' + out1[0][0])
print('Output for ["king","woman"],"man":' + out2[0][0])

## Approx run time - 4s
```

```
[('Queen', 0.4929387867450714), ('Tupou_V.', 0.45174285769462585), ('Oprah_BFF_Gayle', 0.4422132968902588), ('Jackso
n', 0.440250426530838), ('NECN_Alison', 0.4331282675266266), ('Whitfield', 0.42834725975990295), ('Ida_Vandross', 0.4
2084527015686035), ('prosecutor_Dan_Satterberg', 0.420758992433548), ('martin_Luther_King', 0.42059651017189026), ('C
oretta_King', 0.4202733635902405)]
[('queen', 0.7118193507194519), ('monarch', 0.6189674139022827), ('princess', 0.5902431011199951), ('crown_prince',
0.5499460697174072), ('prince', 0.5377321839332581), ('kings', 0.5236844420433044), ('Queen_Consort', 0.5235945582389
832), ('queens', 0.5181134343147278), ('sultan', 0.5098593831062317), ('monarchy', 0.5087411999702454)]

Output for ["King","Woman"],"Man":Queen
Output for ["king","woman"],"man":queen
```

> `--> The pretrained model outputs Queen in both the cases with around 70% confidence and 40% in case if first letter capital`

Trying out more similar examples

In [ ]:
```
out1 = wv_googleNews.most_similar(["summer","snow"],"sun")
out2 = wv_googleNews.most_similar(["Summer","Snow"],"Sun")

print(out1)
print(out2)

print('\nOutput for ["summer","snow"],"sun":' + out1[0][0])
print('Output for ["Summer","Snow"],"Sun":' + out2[0][0])
print("\n")

out1 = wv_googleNews.most_similar(["library","movie"],"book")
out2 = wv_googleNews.most_similar(["Library","Movie"],"Book")

print(out1)
print(out2)

print('\nOutput for ["library","movie"],"book":' + out1[0][0])
print('Output for ["Library","Movie"],"Book":' + out2[0][0])
print("\n")

out1 = wv_googleNews.most_similar(["apple","brocolli"],"fruits")
# out2 = wv_googleNews.most_similar(["Apple","Brocolli"],"Fruit") ## Capital lettered word not present in vocab

print(out1)
# print(out2)

print('\nOutput for ["apple","brocolli"],"fruit":' + out1[0][0])
## Approx run time - 2s
```

```
[('winter', 0.5788487195968628), ('snowstorm', 0.5582926869392395), ('heavy_snows', 0.5265657305717468), ('heavy_snow
fall', 0.509794294834137), ('spring', 0.5073685646057129), ('snowstorms', 0.5022254586219788), ('snowfalls', 0.501769
7215080261), ('heavy_snowfalls', 0.4968310594558716), ('snows', 0.4946661591529846), ('wintry_weather', 0.49291324615
478516)]
[('Winter', 0.5541607737541199), ('Nellis_Stiffler', 0.4493800103664398), ('Rockcliff_presently_controls', 0.41620436
31076813), ('Spring', 0.4159749150276184), ('Fall', 0.41410258412361145), ('Tommy_Wirkola_Dead', 0.411090224981308),
('LeBron_trudging', 0.4106634259223938), ('summer', 0.4085595905780792), ('winter', 0.40848907828330994), ('WInter',
0.40385767817497253)]

Output for ["summer","snow"],"sun":winter
Output for ["Summer","Snow"],"Sun":Winter


[('movies', 0.5982824563980103), ('cinema', 0.5235484838485718), ('multiplex', 0.5145583748817444), ('cineplex', 0.48
67870509624481), ('films', 0.4812585711479187), ('Actors_Equity_arranged', 0.47572818398475647), ('studio_backlot',
0.47454366087913513), ('film', 0.4735766649246216), ('moviehouse', 0.47016361355781555), ('Movie', 0.4599407315254211
4)]
[('Public_Library', 0.5193747878074646), ('Branch_Library', 0.48947563767433167), ('Cinema', 0.48922380805015564),
('Movies', 0.48105770349502563), ('library', 0.4696941077709198), ('LIbrary', 0.44179263710975647), ('Libary', 0.4367
9797649383545), ('MOVIES_IN', 0.4302363097667694), ('Steven_Goldmann', 0.4271371066570282), ('Film', 0.42455431818962
097)]

Output for ["library","movie"],"book":movies
Output for ["Library","Movie"],"Book":Public_Library


[('crunchies', 0.5558125376701355), ('brussel_sprout', 0.5546073913574219), ('carrot_slices', 0.552364706993103), ('c
heesy_polenta', 0.5433920621871948), ('sprouting_broccoli', 0.5425909161567688), ('tomatoe', 0.5401697754859924), ('p
eas_pears', 0.5396046042442322), ('eggwhite', 0.5394821166992188), ('chestnut_purée', 0.5392216444015503), ('rollatin
i', 0.5383666157722473)]

Output for ["apple","brocolli"],"fruit":crunchies
```

> `Conclusion` - The pretrained model performs quite well while generating similarities with positive and negative words

2. Checking similarities between two words

In [ ]:
```
print("Similarities between excellent and outstanding : " + str(wv_googleNews.similarity('excellent', 'outstanding'))
```

```
print("Similarities between beautiful and gorgeous : " + str(wv_googleNews.similarity('beautiful', 'gorgeous')))

print("Similarities between fast and quick : " + str(wv_googleNews.similarity('fast', 'quick')))
## Approx run time - 1s
```

```
Similarities between excellent and outstanding : 0.55674857
Similarities between beautiful and gorgeous : 0.8353004
Similarities between fast and quick : 0.57016057
```

> `Conclusion` - The pretrained model performs quite well while generating similarities similar words

Task (b): Generating word2vec vectors with the amazon reviews processed dataset

In [ ]:

```
"""
This function is used to tokenize and preprocess the string using gensim's preprocess_string function (https://piazza
Using the following filters:
remove_stopwords- Removes the stopwords from the text
strip_punctuation- Removes the punctions
strip_tags-Removes the HTML and XML tags

Ref - https://radimrehurek.com/gensim/parsing/preprocessing.html#gensim.parsing.preprocessing.preprocess_documents
"""
def processed_data_tokens(s:any):
    s = preprocess_string(s,[remove_stopwords,strip_punctuation,strip_tags])
    return s
```

In [ ]:

```
## Applying the above preprocessing function
reviews_ratings_final['review_headline_body_processed'] = reviews_ratings_final['review_headline_body'].apply(process


## Dropping the reviews with zero length after preprocessing to remove the bad input
reviews_ratings_final.drop(reviews_ratings_final[reviews_ratings_final['review_headline_body_processed'].apply(len) =
reviews_ratings_final.reset_index(drop=True,inplace=True)
## Approx run time - 5s
```

Generating the word2vec word embeddings with this processed dataset

In [ ]:

```
model_own_dataset = Word2Vec(sentences=reviews_ratings_final['review_headline_body_processed'],
                             vector_size=300, ## Size of embedding
```

```
                             window=13, ## Size of the window
                             min_count=9) ## Minimum word count

## Approx run time - 35s
```

1. Checking semantic similarity for the King woman man example

In [ ]:
```
out1 = model_own_dataset.wv.most_similar(["King","Woman"],"Man")
out2 = model_own_dataset.wv.most_similar(["king","woman"],"man")

print(out1)
print(out2)

print('\nOutput for ["King","Woman"],"Man":' + out1[0][0])
print('Output for ["king","woman"],"man":' + out2[0][0])

## Approx run time - 1s
```

```
[('Superior', 0.8282687664031982), ('excelente', 0.8080716729164124), ('bien', 0.8069323897361755), ('Excelente', 0.7974227666854858), ('muy', 0.7970524430274963), ('Travel', 0.7931545972824097), ('producto', 0.7846735715866089), ('mi', 0.7805020213127136), ('gracias', 0.7776635885238647), ('buen', 0.7773924469947815)]
[('Living', 0.7969949841499329), ('Appointment', 0.7810559272766113), ('Keeper', 0.7752621173858643), ('Academic', 0.7694739699363708), ('Future', 0.7676205039024353), ('Height', 0.7651114463806152), ('USER', 0.7627473473548889), ('Park', 0.7623292207717896), ('Bound', 0.7588553428649902), ('LAPTOP', 0.7548812627792358)]

Output for ["King","Woman"],"Man":Superior
Output for ["king","woman"],"man":Living
```

> `--> The self trained model does not output the expected Queen in both the cases`

Trying out more similar examples (The part of code is commented as the vocab list is limited in self trained model)

In [ ]:
```
out1 = model_own_dataset.wv.most_similar(["summer","snow"],"sun")
# out2 = model_own_dataset.wv.most_similar(["Summer","Snow"],"Sun") ## Summer not present in vocab

print(out1)
# print(out2)

print('\nOutput for ["summer","snow"],"sun":' + out1[0][0])
# print('Output for ["Summer","Snow"],"Sun":' + out2[0][0]) ## Summer not present in vocab
```

```
print("\n")

out1 = model_own_dataset.wv.most_similar(["library","movie"],"book")
# out2 = model_own_dataset.wv.most_similar(["Library","Movie"],"Book") ## Movie not present in vocab

print(out1)
# print(out2)

print('\nOutput for ["library","movie"],"book":' + out1[0][0])
# print('Output for ["Library","Movie"],"Book":' + out2[0][0]) ## Movie not present in vocab
print("\n")

# out1 = model_own_dataset.wv.most_similar(["apple","brocolli"],"fruits") ## Brocolli not present in vocab
# out2 = model_own_dataset.wv.most_similar(["Apple","Brocolli"],"Fruit") ## Brocolli not present in vocab

print(out1)
# print(out2)

print('\nOutput for ["apple","brocolli"],"fruit": NA')
## Approx run time - 1s
```

```
[('stocking', 0.6913248300552368), ('bookstore', 0.6752166152000427), ('graduation', 0.6717312932014465), ('teacher', 0.6686440706253052), ('Influenster', 0.6678372621536255), ('graduate', 0.6612659096717834), ('6th', 0.6597033739089966), ('schooling', 0.6536543965339661), ('stuffer', 0.6512351632118225), ('university', 0.6471811532974243)]

Output for ["summer","snow"],"sun":stocking


[('movies', 0.7601059079170227), ('theater', 0.7110424637794495), ('presentations', 0.6842615008354187), ('watching', 0.6741736531257629), ('darkened', 0.6517071723937988), ('PowerPoint', 0.638480007648468), ('gaming', 0.6290944218635559), ('ray', 0.6210340857505798), ('games', 0.6130325198173523), ('Netflix', 0.6061313152313232)]

Output for ["library","movie"],"book":movies


[('movies', 0.7601059079170227), ('theater', 0.7110424637794495), ('presentations', 0.6842615008354187), ('watching', 0.6741736531257629), ('darkened', 0.6517071723937988), ('PowerPoint', 0.638480007648468), ('gaming', 0.6290944218635559), ('ray', 0.6210340857505798), ('games', 0.6130325198173523), ('Netflix', 0.6061313152313232)]

Output for ["apple","brocolli"],"fruit": NA
```

Conclusion for Task 2

Q: 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?

> `Following points were observed`
>
> 1. The pretrained model performed better in all the cases to predict the semantic similarities and encode them
> 2. The self trained model has small vocab, hence making it difficult due to large number of unseen words.

Task 3 - Simple Models

In [ ]:

```
# TF-IDF feature extraction
"""
    1. reducing the size to float 32 to avoid memory issues - dtype = float32
    2. using ngram_range to consider 1 to 4 words together while extracting the features
"""
tfidf = TfidfVectorizer(dtype = np.float32, min_df=2,ngram_range=(1,4))
tfidf_vectors = tfidf.fit_transform(reviews_ratings_final['review_headline_body_processed'].apply(lambda x: ' '.join(

tfidf_x_train, tfidf_x_test, tfidf_y_train, tfidf_y_test = train_test_split(tfidf_vectors,
                                                                             reviews_ratings_final["Class"],
                                                                             test_size=0.2, # Splitting the dataset in
                                                                             random_state=34) # Setting the random sta


## Approx run time - 32s
```

In [ ]:

```
# Mean vectors from word2vec pretrained model
reviews_ratings_final['word2vec_mean'] = reviews_ratings_final['review_headline_body_processed'].apply(wv_googleNews.

word2vec_mean_x_train, word2vec_mean_x_test, word2vec_mean_y_train, word2vec_mean_y_test = train_test_split(np.stack(
                                                                                                              reviews_r
                                                                                                              test_size
                                                                                                              random_st


## Approx run time - 40s
```

Single perceptron with TFIDF and Word2vec features

In [ ]:
```
model_perceptron_tfidf = Perceptron(random_state=34)
model_perceptron_tfidf.fit(tfidf_x_train, tfidf_y_train)

tfidf_predictions = model_perceptron_tfidf.predict(tfidf_x_test)
eval(tfidf_y_test,tfidf_predictions)

## Approx run time - 1s
```

```
Accuracy = 0.8952
```

In [ ]:
```
model_perceptron_word2vec = Perceptron(random_state=34)
model_perceptron_word2vec.fit(word2vec_mean_x_train,word2vec_mean_y_train)

word2vec_mean_perceptron_predictions = model_perceptron_word2vec.predict(word2vec_mean_x_test)
eval(word2vec_mean_y_test,word2vec_mean_perceptron_predictions)

## Approx run time - 1s
```

```
Accuracy = 0.825
```

Single Perceptron Accuracy

TF_IDF Features	Word2Vec Features
89.52 %	82.5 %

SVM with TF-IDF and Word2vec features

In [ ]:
```
model_svm_tfidf = svm.LinearSVC(max_iter=50000) # setting max_iter to 50000 to avoid long runs
model_svm_tfidf.fit(tfidf_x_train, tfidf_y_train)

tfidf_svm_predictions = model_svm_tfidf.predict(tfidf_x_test)
eval(tfidf_y_test,tfidf_svm_predictions)

# Approx run time - 2s
```

```
d:\.virtualenvs\Applied_NLP-VPUSIJtg\lib\site-packages\sklearn\svm\_classes.py:32: FutureWarning: The default value o
f `dual` will change from `True` to `'auto'` in 1.5. Set the value of `dual` explicitly to suppress the warning.
  warnings.warn(
Accuracy = 0.9083
```

In [ ]:
```
model_svm_word2vec = svm.LinearSVC(max_iter=50000) # setting max_iter to 50000 to avoid long runs
model_svm_word2vec.fit(word2vec_mean_x_train,word2vec_mean_y_train)

word2vec_mean_svm_predictions = model_svm_word2vec.predict(word2vec_mean_x_test)
eval(word2vec_mean_y_test,word2vec_mean_svm_predictions)

## Approx run time - 6s
```

```
d:\.virtualenvs\Applied_NLP-VPUSIJtg\lib\site-packages\sklearn\svm\_classes.py:32: FutureWarning: The default value o
f `dual` will change from `True` to `'auto'` in 1.5. Set the value of `dual` explicitly to suppress the warning.
  warnings.warn(
Accuracy = 0.8428
```

SVM Accuracy

TF_IDF Features	Word2Vec Features
90.83 %	84.28 %

`Conclusion for Task 3 -`

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

> - The simple models perform better with TF-IDF features. The models are able to achieve higher accuracies on Test dataset with the TF-IDF features

Task 4 - Feedforward Neural Networks

Task (a) - FNN using mean vectors

`Ref`

- https://medium.com/deep-learning-study-notes/multi-layer-perceptron-mlp-in-pytorch-21ea46d50e62
- https://stackoverflow.com/questions/60259836/cnn-indexerror-target-2-is-out-of-bounds
- https://www.kaggle.com/mishra1993/pytorch-multi-layer-perceptron-mnist

In [ ]:
```python
"""
Creating two classes to iterate through the training and testing dataset extending the Dataset Class from Pytorch (ht
"""

class TrainDataset(Dataset):
    ## The constructor assigns the X and Y labels to dataX and dataY
    def __init__(self, XData, YData):
        self.dataX = XData
        self.dataY = YData

    #Returns the length of the X data
    def __len__(self):
        return self.dataX.shape[0]

    # Returns X and Y data at given index
    def __getitem__(self, index):
        x = self.dataX[index]
        y = self.dataY[index]
        return x, y

class TestDataset(TrainDataset):
    # The Test dataset contains just the X label
    def __getitem__(self, index):
        x = self.dataX[index]
        return x

## Approx run time - 1s
```

In [ ]:
```python
"""
Creating data loaders from the datasets with word2vec mean vectors (https://pytorch.org/docs/stable/data.html)
"""

train_set = TrainDataset(word2vec_mean_x_train, word2vec_mean_y_train)
test_set  = TestDataset(word2vec_mean_x_test, word2vec_mean_y_test)

## Setting the batch size
### Trained the model with different batch sizes and 64 performs the best on test dataset
batch_size = 64
train_loader = DataLoader(train_set, batch_size=batch_size, shuffle=True)
test_loader  = DataLoader(test_set,  batch_size=batch_size, shuffle=False)
```

```
## Approx run time - 1s
```

Following is the model architecture for this task

- (0): Linear - in_features=300, out_features=50
- (1): ReLU - ReLU layer
- (2): Dropout - Dropping 20% of the paths to avoid overfitting
- (3): Linear - in_features=50, out_features=5
- (4): LeakyReLU - LeakyRelu Layer
- (5): Linear - in_features=5, out_features=2
- (6): LeakyReLU - LeakyRelu Layer

In [ ]:

```
class FFN_MLP(nn.Module):
    def __init__(self):
        super(FFN_MLP, self).__init__()
        self.sequntial = nn.Sequential(nn.Linear(300, 50),
                                       nn.ReLU(),
                                       nn.Dropout(0.2),
                                       nn.Linear(50,5),
                                       nn.LeakyReLU(),
                                       # nn.Dropout(0.2),
                                       nn.Linear(5,2),
                                       nn.LeakyReLU())


    def forward(self, x):
        out = self.sequntial(x)
        return out


model_FNN = FFN_MLP().to(device)
optimizer_FNN = torch.optim.Adam(model_FNN.parameters(),lr=0.001)
criterion_FNN = nn.CrossEntropyLoss()

print(model_FNN)

## Approx run time - 2s
```

```
FFN_MLP(
  (sequntial): Sequential(
    (0): Linear(in_features=300, out_features=50, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.2, inplace=False)
    (3): Linear(in_features=50, out_features=5, bias=True)
    (4): LeakyReLU(negative_slope=0.01)
    (5): Linear(in_features=5, out_features=2, bias=True)
    (6): LeakyReLU(negative_slope=0.01)
  )
)
```

Training the FNN, Running 15 epochs

In [ ]:

```
epochs = 15

# Training the model
model_FNN.train()
for epoch in range(epochs):
    losses = []
    for batch_num, input_data in enumerate(train_loader):
        optimizer_FNN.zero_grad()

        #Reading the data from train_loader
        x, y = input_data
        x = x.to(device).float()
        y = y.to(device)

        # Generating the predictions (forward pass)
        output = model_FNN(x).to(device)

        # Calculating the losses and performing backward pass
        loss = criterion_FNN(output, y)
        loss.backward()

        losses.append(loss.item())
        optimizer_FNN.step()

    print('Epoch ' + str(epoch + 1)+ ' - Average Loss = ' + '{:.2f}'.format(np.average(losses))) # Prints the average

## Approx run time for 15 epochs - 2min 15s
```

```
Epoch 1 - Average Loss = 0.44
Epoch 2 - Average Loss = 0.36
Epoch 3 - Average Loss = 0.35
Epoch 4 - Average Loss = 0.34
Epoch 5 - Average Loss = 0.33
Epoch 6 - Average Loss = 0.33
Epoch 7 - Average Loss = 0.32
Epoch 8 - Average Loss = 0.32
Epoch 9 - Average Loss = 0.32
Epoch 10 - Average Loss = 0.31
Epoch 11 - Average Loss = 0.31
Epoch 12 - Average Loss = 0.31
Epoch 13 - Average Loss = 0.30
Epoch 14 - Average Loss = 0.30
Epoch 15 - Average Loss = 0.30
```

In [ ]:

```
# Evaluating the model

model_FNN.eval()
output = torch.tensor([]).to(device)
for x_label in test_loader:

    # Loading the data from test loader
    x = x_label.to(device)

    output = torch.cat((output,model_FNN(x).to(device)))

eval(word2vec_mean_y_test,output.argmax(dim=1).to('cpu'))

## Approx run time - 1s
```

```
Accuracy = 0.8648
```

FNN with mean features Accuracy

Q: Report accuracy values on the testing split for your MLP

Accuracy
86.48 %

Task (b) - FNN using first 10 word2vec vectors

`Ref` -

- https://stackoverflow.com/questions/72480289/how-to-handle-keyerrorfkey-key-not-present-wor2vec-with-gensim
- https://stackoverflow.com/questions/65372032/deal-with-out-of-vocabulary-word-with-gensim-pretrained-glove

In [ ]:

```
"""
This function returns a vector of size 3000 with the word2vec features of 10 words
- The function take processed tokenized string list as input
- It considers the first 10 words which are present in the dataset and skips the words which are not present in vocab
- If the length of known words is less than 10, then it is padded with 0s to make the final size of the output to be
"""

def get_word2vec_first_10(s:list):
    vector_first_10 = []
    i = 0
    iterator = 0

    while i < 10 and iterator < len(s):
        try:
            current_vec = wv_googleNews.get_vector(s[iterator])
            vector_first_10 = np.concatenate((vector_first_10, current_vec))
        except:
            i-=1 ## discards the pass if the word is out of vocubulary
        finally:
            i+=1
            iterator+=1


    vector_first_10 = np.pad(vector_first_10,(0, 3000 - len(vector_first_10))) ## Padding the final output with 0 in
    return vector_first_10

## Approx run time - 1 sec
```

In [ ]:

```
# Applying the above function and saving the vector in word2vec_first_10 series
reviews_ratings_final['word2vec_first_10'] = reviews_ratings_final['review_headline_body_processed'].apply(get_word2v

## Approx run time - 20s
```

```
In [ ]: # Splitting the dataset in 80:20 train:test split
word2vec_first_10_x_train, word2vec_first_10_x_test, word2vec_first_10_y_train, word2vec_first_10_y_test = train_test



## Approx run time - 5s
```

```
In [ ]: """
Creating data loaders from the datasets with word2vec mean vectors (https://pytorch.org/docs/stable/data.html)
"""

train_set_first_10 = TrainDataset(word2vec_first_10_x_train, word2vec_first_10_y_train)
test_set_first_10  = TestDataset(word2vec_first_10_x_test, word2vec_first_10_y_test)


## Setting the batch size
### Trained the model with different batch sizes and 128 performs the best on test dataset
batch_size = 128
train_loader_first_10 = DataLoader(train_set_first_10, batch_size=batch_size, shuffle=True)
test_loader_first_10  = DataLoader(test_set_first_10,  batch_size=batch_size, shuffle=False)

## Approx run time - 1s
```

Following is the model architecture for this task

- (0): Linear - in_features=3000, out_features=50
- (1): ReLU - ReLU layer
- (2): Dropout - Dropping 20% of the paths to avoid overfitting
- (3): Linear - in_features=50, out_features=5
- (4): LeakyReLU - LeakyRelu Layer
- (5):Dropout - Dropping 20% of the paths to avoid overfitting
- (6): Linear - in_features=5, out_features=2
- (7): LeakyReLU - LeakyRelu Layer

```
In [ ]: class FFN_first_10(nn.Module):
    def __init__(self,input_size,hidden_size1,hidden_size2,output_size):
        super(FFN_first_10, self).__init__()
```

```python
        self.sequntial = nn.Sequential(nn.Linear(input_size, hidden_size1),
                                       nn.ReLU(),
                                       nn.Dropout(0.2),
                                       nn.Linear(hidden_size1,hidden_size2),
                                       nn.LeakyReLU(),
                                       nn.Dropout(0.2),
                                       nn.Linear(hidden_size2,output_size),
                                       nn.LeakyReLU())


    def forward(self, x):
        out = self.sequntial(x)
        return out


input_size = 3000
hidden_size1 = 50
hidden_size2 = 5
output_size = 2

## Creating model
model_first_10_FNN = FFN_first_10(input_size,hidden_size1,hidden_size2,output_size).to(device)
optimizer_first_10_FNN = torch.optim.Adam(model_first_10_FNN.parameters(),lr=0.001)
criterion_first_10_FNN = nn.CrossEntropyLoss()
print(model_first_10_FNN)

## Approx run time - 1s
```

```
FFN_first_10(
  (sequntial): Sequential(
    (0): Linear(in_features=3000, out_features=50, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.2, inplace=False)
    (3): Linear(in_features=50, out_features=5, bias=True)
    (4): LeakyReLU(negative_slope=0.01)
    (5): Dropout(p=0.2, inplace=False)
    (6): Linear(in_features=5, out_features=2, bias=True)
    (7): LeakyReLU(negative_slope=0.01)
  )
)
```

Training the FNN with first 10 vec features, Running 15 epochs

In [ ]:
```python
epochs = 30

# Training a model
model_first_10_FNN.train()
for epoch in range(epochs):
    losses = []
    for batch_num, input_data in enumerate(train_loader_first_10):
        optimizer_first_10_FNN.zero_grad()

        #Reading the data from train_loader
        x, y = input_data
        x = x.to(device).float()
        y = y.to(device)

        # Generating the predictions (forward pass)
        output = model_first_10_FNN(x)

        # Calculating the losses and performing backward pass
        loss = criterion_first_10_FNN(output, y)
        loss.backward()
        losses.append(loss.item())

        optimizer_first_10_FNN.step()

    print('Epoch ' + str(epoch + 1)+ ' - Average Loss = ' + '{:.2f}'.format(np.average(losses))) # Prints the average

# Run time for 30 epochs 2m 45s
```

```
Epoch 1 - Average Loss = 0.45
Epoch 2 - Average Loss = 0.37
Epoch 3 - Average Loss = 0.33
Epoch 4 - Average Loss = 0.31
Epoch 5 - Average Loss = 0.29
Epoch 6 - Average Loss = 0.27
Epoch 7 - Average Loss = 0.25
Epoch 8 - Average Loss = 0.23
Epoch 9 - Average Loss = 0.21
Epoch 10 - Average Loss = 0.20
Epoch 11 - Average Loss = 0.19
Epoch 12 - Average Loss = 0.17
Epoch 13 - Average Loss = 0.16
Epoch 14 - Average Loss = 0.16
Epoch 15 - Average Loss = 0.15
Epoch 16 - Average Loss = 0.14
Epoch 17 - Average Loss = 0.13
Epoch 18 - Average Loss = 0.13
Epoch 19 - Average Loss = 0.13
Epoch 20 - Average Loss = 0.12
Epoch 21 - Average Loss = 0.11
Epoch 22 - Average Loss = 0.11
Epoch 23 - Average Loss = 0.11
Epoch 24 - Average Loss = 0.11
Epoch 25 - Average Loss = 0.11
Epoch 26 - Average Loss = 0.10
Epoch 27 - Average Loss = 0.10
Epoch 28 - Average Loss = 0.09
Epoch 29 - Average Loss = 0.09
Epoch 30 - Average Loss = 0.09
```

In [ ]:

```
# Evaluating the model

output_first_10_FNN = torch.tensor([]).to(device)
for x_label in test_loader_first_10:
    x = x_label.to(device).float()

    output_first_10_FNN = torch.cat((output_first_10_FNN,model_first_10_FNN(x.to(device)).to(device).argmax(dim=1)))

eval(word2vec_mean_y_test,output_first_10_FNN.to('cpu'))
## Approx run time - 1s
```

```
Accuracy = 0.8213
```

FNN with first 10 Word2vec vectors Accuracy

Q: Report the accuracy value on the testing split for your MLP model

Accuracy
82.13 %

Q: What do you conclude by comparing accuracy values you obtain with those obtained in the “’Simple Models” section.

> Using word2vec features, the performance of all the models is in the following order -
>
> - FNN with Mean Vectors
> - SVM
> - Single Perceptron
> - FNN with First 10 vectors
>
>   The performance is comparable in all the models ranging from 82 to 86 % on the test dataset

Task 5. Recurrent Neural Networks

Task (a) - Simple RNN

`Ref` -

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

Following is the model architecture for this task

- (0): Recurrent Neural Network layer - in_features=3000, out_features=10
- (1): Linear - in_features=10, out_features=2
- (2): ReLU - Relu Layer

In [ ]:
```
class RNN_first_10(nn.Module):
    def __init__(self, input_size:int, hidden_size:int, output_size:int):
        super(RNN_first_10, self).__init__()
```

```
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)    ## RNN layer with input size 3000 and output
        # self.dropout = nn.Dropout(0.2) ## Tried drop out layer, but performs better without it on test dataset
        self.lin =  nn.Linear(hidden_size, output_size) ## Linear layer with input size of 10 and output size of 2
        self.act2 = nn.ReLU()


    def forward(self, x):
        out,_ = self.rnn(x)
        # out = self.dropout(out)
        out = self.lin(out)
        out = self.act2(out)

        return out

input_size = 3000
hidden_size = 10
output_size = 2

# Creating the model
model_rnn = RNN_first_10(input_size, hidden_size, output_size).to(device)
criterion_rnn = nn.CrossEntropyLoss()
optimizer_rnn = torch.optim.Adam(model_rnn.parameters(),lr=0.0001)
print(model_rnn)

## Approx run time - 1s
```

```
RNN_first_10(
  (rnn): RNN(3000, 10, batch_first=True)
  (lin): Linear(in_features=10, out_features=2, bias=True)
  (act2): ReLU()
)
```

Training the RNN, Running 30 epochs

In [ ]:
```
num_epochs = 30

#Training the model
model_rnn.train()
for epoch in range(num_epochs):
    losses = []
    for batch_num, input_data in enumerate(train_loader_first_10):
```

```
        optimizer_rnn.zero_grad()

        ## Loading the data from train loader
        x, y = input_data
        x = x.to(device).float()
        y = y.to(device)

        ## Predicting the labels (Forward pass)
        output = model_rnn(x)

        ## Calculating the loss and performing the backward pass
        loss = criterion_rnn(output, y)
        loss.backward()
        losses.append(loss.item())

        optimizer_rnn.step()

    print('Epoch ' + str(epoch + 1)+ ' - Average Loss = ' + '{:.2f}'.format(np.average(losses)))

# Run time for 30 epochs 2m 45s
```

```
Epoch 1 - Average Loss = 0.66
Epoch 2 - Average Loss = 0.58
Epoch 3 - Average Loss = 0.51
Epoch 4 - Average Loss = 0.47
Epoch 5 - Average Loss = 0.45
Epoch 6 - Average Loss = 0.43
Epoch 7 - Average Loss = 0.42
Epoch 8 - Average Loss = 0.41
Epoch 9 - Average Loss = 0.40
Epoch 10 - Average Loss = 0.39
Epoch 11 - Average Loss = 0.38
Epoch 12 - Average Loss = 0.38
Epoch 13 - Average Loss = 0.37
Epoch 14 - Average Loss = 0.37
Epoch 15 - Average Loss = 0.36
Epoch 16 - Average Loss = 0.36
Epoch 17 - Average Loss = 0.36
Epoch 18 - Average Loss = 0.35
Epoch 19 - Average Loss = 0.35
Epoch 20 - Average Loss = 0.34
Epoch 21 - Average Loss = 0.34
Epoch 22 - Average Loss = 0.34
Epoch 23 - Average Loss = 0.34
Epoch 24 - Average Loss = 0.33
Epoch 25 - Average Loss = 0.33
Epoch 26 - Average Loss = 0.33
Epoch 27 - Average Loss = 0.33
Epoch 28 - Average Loss = 0.32
Epoch 29 - Average Loss = 0.32
Epoch 30 - Average Loss = 0.32
```

In [ ]:

```
output_first_10 = torch.tensor([]).to(device)
for x_label in test_loader_first_10:
    x = x_label.to(device).float()

    output_first_10 = torch.cat((output_first_10,model_rnn(x.to(device)).to(device).argmax(dim=1)))

eval(word2vec_first_10_y_test,output_first_10.to('cpu'))

## Approx run time - 1s
```

```
Accuracy = 0.8319
```

Simple RNN with Word2vec features Accuracy

Q: Report accuracy values on the testing split for your RNN model.

Accuracy
83.19 %

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

`Conclusion`

> For the current dataset-
>
> - The Feedforward network model with mean vector (Task 4a) performs better than the Simple RNN model
> - However, Simple RNN performs better than the FNN with first 10 word2vec vectors(Task 4b)

Task (b) - Gated RNN

`Ref` -

- https://blog.floydhub.com/gru-with-pytorch/

Following is the model architecture for this task

- (0): Gated Recurrent Neural Network layer - in_features=3000, out_features=10
- (1): Linear - in_features=10, out_features=2
- (2): ReLU - Relu Layer

In [ ]:
```
class RNN_first_10_gated(nn.Module):
    def __init__(self, input_size:int, hidden_size:int, output_size:int):
        super(RNN_first_10_gated, self).__init__()
        self.rnn = nn.GRU(input_size, hidden_size, batch_first=True) ## Gated RNN Layer with input size 3000 and outp
        self.lin =  nn.Linear(hidden_size, output_size) ## Linear Layer with input size of 10 and output size of 2
        self.act1 = nn.ReLU()

    def forward(self, x):
```

```
        out,_ = self.rnn(x)
        out = self.lin(out)
        out = self.act1(out)
        return out


input_size = 3000
hidden_size = 10
output_size = 2

# Creating the models
model_gru = RNN_first_10_gated(input_size, hidden_size, output_size).to(device)
criterion_gru = nn.CrossEntropyLoss()
optimizer_gru = torch.optim.Adam(model_gru.parameters(), lr=0.0001)
print(model_gru)

## Approx run time - 1s
```

```
RNN_first_10_gated(
  (rnn): GRU(3000, 10, batch_first=True)
  (lin): Linear(in_features=10, out_features=2, bias=True)
  (act1): ReLU()
)
```

Training the Gated RNN, Running 30 epochs

In [ ]:

```
num_epochs = 30

for epoch in range(num_epochs):
    losses = []
    for batch_num, input_data in enumerate(train_loader_first_10):
        optimizer_gru.zero_grad()

        ## Loading the data from train loader
        x, y = input_data
        x = x.to(device).float()
        y = y.to(device)

        ## Predicting the labels (Forward pass)
        output = model_gru(x)

        ## Calculating the loss and performing the backward pass
```

```
        loss = criterion_gru(output, y)
        loss.backward()
        losses.append(loss.item())

        optimizer_gru.step()

    print('Epoch ' + str(epoch + 1)+ ' - Average Loss = ' + '{:.2f}'.format(np.average(losses)))

# Run time for 30 epochs 2m 45s
```

```
Epoch 1 - Average Loss = 0.67
Epoch 2 - Average Loss = 0.59
Epoch 3 - Average Loss = 0.51
Epoch 4 - Average Loss = 0.47
Epoch 5 - Average Loss = 0.44
Epoch 6 - Average Loss = 0.42
Epoch 7 - Average Loss = 0.41
Epoch 8 - Average Loss = 0.40
Epoch 9 - Average Loss = 0.39
Epoch 10 - Average Loss = 0.38
Epoch 11 - Average Loss = 0.38
Epoch 12 - Average Loss = 0.37
Epoch 13 - Average Loss = 0.37
Epoch 14 - Average Loss = 0.36
Epoch 15 - Average Loss = 0.36
Epoch 16 - Average Loss = 0.35
Epoch 17 - Average Loss = 0.35
Epoch 18 - Average Loss = 0.35
Epoch 19 - Average Loss = 0.34
Epoch 20 - Average Loss = 0.34
Epoch 21 - Average Loss = 0.34
Epoch 22 - Average Loss = 0.33
Epoch 23 - Average Loss = 0.33
Epoch 24 - Average Loss = 0.33
Epoch 25 - Average Loss = 0.32
Epoch 26 - Average Loss = 0.32
Epoch 27 - Average Loss = 0.32
Epoch 28 - Average Loss = 0.32
Epoch 29 - Average Loss = 0.31
Epoch 30 - Average Loss = 0.31
```

In [ ]:
```
output_first_10 = torch.tensor([]).to(device)
for x_label in test_loader_first_10:
    x = x_label.to(device).float()

    output_first_10 = torch.cat((output_first_10,model_gru(x.to(device)).to(device).argmax(dim=1)))

eval(word2vec_first_10_y_test,output_first_10.to('cpu'))
## Approx run time - 1s
```

```
Accuracy = 0.8369
```

Gated RNN with Word2vec features Accuracy

Accuracy
83.69 %

Task (c) - LSTM

Following is the model architecture for this task

- (0): LSTM Neural Network layer - in_features=3000, out_features=10
- (1): Linear - in_features=10, out_features=2
- (2): ReLU - Relu Layer

In [ ]:
```
class LSTM_first_10(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(LSTM_first_10, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True) ## LSTM Layer with input size 3000 and output
        self.lin =  nn.Linear(hidden_size, output_size) ## Linear layer with input size 10 and output size 2
        self.act = nn.ReLU()

    def forward(self, x):
        out,_ = self.lstm(x)
        out = self.lin(out)
        out = self.act(out)

        return out
```

```
input_size = 3000
hidden_size = 10
output_size = 2

## Creating the model
model_lstm = LSTM_first_10(input_size, hidden_size, output_size).to(device)
criterion_lstm = nn.CrossEntropyLoss()
optimizer_lstm = torch.optim.Adam(model_lstm.parameters(), lr=0.001)

print(model_lstm)

## Approx run time - 1s
```

```
LSTM_first_10(
  (lstm): LSTM(3000, 10, batch_first=True)
  (lin): Linear(in_features=10, out_features=2, bias=True)
  (act): ReLU()
)
```

Training the model, Running 10 epochs

In [ ]:

```
num_epochs = 10

# Training the model
for epoch in range(num_epochs):
    losses = []
    for batch_num, input_data in enumerate(train_loader_first_10):
        optimizer_lstm.zero_grad()

        ## Loading the data from train loader
        x, y = input_data
        x = x.to(device).float()
        y = y.to(device)

        ## Predicting the labels (Forward pass)
        output = model_lstm(x)

        ## Calculating the loss and performing the backward pass
        loss = criterion_lstm(output, y)
        loss.backward()
        losses.append(loss.item())
```

```
        optimizer_lstm.step()

    print('Epoch ' + str(epoch + 1)+ ' - Average Loss = ' + '{:.2f}'.format(np.average(losses)))

# Run time for 10 epochs 1min
```

```
Epoch 1 - Average Loss = 0.47
Epoch 2 - Average Loss = 0.36
Epoch 3 - Average Loss = 0.32
Epoch 4 - Average Loss = 0.30
Epoch 5 - Average Loss = 0.27
Epoch 6 - Average Loss = 0.25
Epoch 7 - Average Loss = 0.23
Epoch 8 - Average Loss = 0.21
Epoch 9 - Average Loss = 0.19
Epoch 10 - Average Loss = 0.17
```

In [ ]:

```
output_first_10 = torch.tensor([]).to(device)
for x_label in test_loader_first_10:
    x = x_label.to(device).float()

    output_first_10 = torch.cat((output_first_10,model_lstm(x.to(device)).to(device).argmax(dim=1)))

eval(word2vec_first_10_y_test,output_first_10.to('cpu'))

## Approx run time - 1s
```

```
Accuracy = 0.8327
```

LSTM with Word2vec features Accuracy

Q: Report accuracy values on the testing split for your LSTM model.

Accuracy
83.27 %

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

> - The accuracies are comparable for all the three models, however Gated Recurrent Neural net has the highest of all on the test dataset

`Conclusion` -

Final Accuracy Values as of 10/19

Model	Input Features	Accuracy
Perceptron	Mean Word2Vec	82.5 %
	TF_IDF	89.52 %
SVM	Mean Word2Vec	84.28 %
	TF_IDF	90.83 %
FNN	Mean Word2Vec	86.48 %
FNN	First 10 Word2Vec	82.13 %
Simple RNN	First 10 Word2Vec	83.19 %
Gated RNN	First 10 Word2Vec	83.69 %
LSTM	First 10 Word2Vec	83.27 %

The highest performance on the unseen test dataset is given by SVM model with TF_IDF features