NLP_RHEA_copy

October 5, 2021

[102]:
```python
#########################################################
import pandas as pd
import numpy as np
import nltk
nltk.download('wordnet')
nltk.download('stopwords')
import re
from bs4 import BeautifulSoup
import urllib.request
import gzip
import contractions
from sklearn.model_selection import train_test_split
import gensim.downloader as api
import gensim.models
from sklearn.linear_model import Perceptron
import torch
import gensim
from sklearn.model_selection import train_test_split
from torch.utils.data import Dataset, DataLoader
from sklearn.preprocessing import StandardScaler
from sklearn import svm
from sklearn.linear_model import Perceptron
from sklearn.feature_extraction.text import TfidfVectorizer
from torch.utils.data import Dataset, DataLoader
```

```
[nltk_data] Downloading package wordnet to
[nltk_data]     /Users/rheaanand/nltk_data…
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/rheaanand/nltk_data…
[nltk_data]   Package stopwords is already up-to-date!
```

1 Loading Data

[167]:
```python
#########################################################
text_df = pd.read_csv('data.tsv', error_bad_lines = False, sep = '\t',␣
↪warn_bad_lines = False)
```

```
/Users/rheaanand/miniconda3/envs/nlp/lib/python3.9/site-
packages/IPython/core/interactiveshell.py:3441: FutureWarning: The
warn_bad_lines argument has been deprecated and will be removed in a future
version.


  exec(code_obj, self.user_global_ns, self.user_ns)
/Users/rheaanand/miniconda3/envs/nlp/lib/python3.9/site-
packages/IPython/core/interactiveshell.py:3441: FutureWarning: The
error_bad_lines argument has been deprecated and will be removed in a future
version.


  exec(code_obj, self.user_global_ns, self.user_ns)
```

2 Keeping Required Data

[168]:

```
#####################################################
text_df = text_df[['star_rating','review_body']]

text_df.dropna(subset = ['review_body'], inplace = True)
text_df.dropna(subset = ['star_rating'], inplace = True)

print("\n********* Five sample reviews *********\n")
print(text_df.sample(n=5, random_state=111))
```

```
********* Five sample reviews *********

         star_rating                                        review_body
2888589          5.0  Exceeded expectations.  The steel looks good, …
501710           3.0  I received the plastic cone yesterday and I ha…
80789            3.0  Heating the hot dogs is fine, but the buns get…
3014624          5.0  I grew up watching my mom use Pyrex cup measur…
262012           4.0                       Looks great--giving as a gift
```

3 Ramdomly Sampling 50000 reviews of each class

[169]:

```
#####################################################
#randomly sampling 50000 reviews of each rating
r1 = text_df[text_df['star_rating'] == 1].sample(50000, random_state=89)
r2 = text_df[text_df['star_rating'] == 2].sample(50000, random_state=89)
r3 = text_df[text_df['star_rating'] == 3].sample(50000, random_state=89)
r4 = text_df[text_df['star_rating'] == 4].sample(50000, random_state=89)
r5 = text_df[text_df['star_rating'] == 5].sample(50000, random_state=89)
```

```
list_text_df = [r1, r2, r3, r4, r5]
text_df = pd.concat(list_text_df)
print(text_df.size)
print(text_df.head)
```

```
500000
<bound method NDFrame.head of          star_rating
review_body
3377745          1.0                                 They both cracked
137462           1.0  Leaves half the juice in the pulp. You can lit…
3074771          1.0  I kid you not, this coffee grinder worked for …
2007652          1.0  Disappointed upon arrival~~<br />The caddy in …
4039144          1.0  Vacuumed 8 bags and died.  But hey it worked g…
…               …                                              …
314391           5.0  It's become a favorite pan to use. I use it fo…
1119712          5.0  those knife are so good quality and so sharp…
2067376          5.0      Very nice,  ) ike that it comes with a stand.
4768391          5.0  Heats water very quickly to boiling. Quiet.  A…
2788460          5.0  Nice sturdy pan and nice size for cooking seve…

[250000 rows x 2 columns]>
```

4 Adding Class Labels

[170]:
```
#######################################################
#adding new column label which gives 1 for > 3 rating 2 for < 3 rating and 3␣
↪for others (i.e ==3 rating)
text_df['label'] = np.where(text_df["star_rating"] > 3, 1, np.
↪where(text_df["star_rating"] < 3, 2, 3))
text_df['label'].head
```

[170]:
```
<bound method NDFrame.head of 3377745    2
137462     2
3074771    2
2007652    2
4039144    2
          ..
314391     1
1119712    1
2067376    1
4768391    1
2788460    1
Name: label, Length: 250000, dtype: int64>
```

5 Data Cleaning

[171]:
```
#######################################################
# DATA CLEANING
###############

# convert everything to lower case
text_df["review_body"] = text_df["review_body"].str.lower()

# Removing html tags using beautiful soup like <br> tags
text_df["review_body"] = text_df["review_body"].apply(lambda x:␣
↪BeautifulSoup(str(x)).get_text())

# Removing urls from reviews
text_df["review_body"] = text_df["review_body"].apply(lambda x: re.
↪sub(r'\s*(https?://|www\.)+\S+(\s+|$)', " ", str(x), flags=re.UNICODE))

# Removing Digits from the review_body
text_df["review_body"] = text_df["review_body"].apply(lambda x: re.
↪sub(r"[^\D']+", " ", str(x), flags=re.UNICODE)) # remove all numbers

# Removing Special Characters
text_df["review_body"] = text_df["review_body"].apply(lambda x: re.
↪sub(r"[^\w']+", " ", str(x), flags=re.UNICODE)) # remove all special␣
↪characters

#remove more than one spaces
text_df["review_body"] = text_df["review_body"].apply(lambda x: re.sub(r'\s+','␣
↪', str(x), flags = re.UNICODE))

def contractionfunction(s):
    s = s.apply(lambda x: contractions.fix(x))
    return s

text_df_onecol = contractionfunction(text_df["review_body"])
text_df["review_body"] = text_df_onecol

# convert everything to lower case again because contractions adds I
text_df["review_body"] = text_df["review_body"].str.lower()
```

```
/Users/rheaanand/miniconda3/envs/nlp/lib/python3.9/site-
packages/bs4/__init__.py:417: MarkupResemblesLocatorWarning:
"http://www.amazon.com/10-5-round-stainless-steel-
skimmer/dp/b00a6h272g/ref=sr_1_1?s=home-
garden&ie=utf8&qid=1424472835&sr=1-1&keywords=11+3%2f4%22+oil+skimmer" looks
like a URL. Beautiful Soup is not an HTTP client. You should probably use an
HTTP client like requests to get the document behind the URL, and feed that
```

```
document to Beautiful Soup.
  warnings.warn(
/Users/rheaanand/miniconda3/envs/nlp/lib/python3.9/site-
packages/bs4/__init__.py:417: MarkupResemblesLocatorWarning:
"https://www.facebook.com/cherischocolates" looks like a URL. Beautiful Soup is
not an HTTP client. You should probably use an HTTP client like requests to get
the document behind the URL, and feed that document to Beautiful Soup.
  warnings.warn(
```

6 Pre-Processing Data

[172]:

```
#######################################################
# PRE-PROCESSING
################

# REMOVING STOPWORDS
from nltk.corpus import stopwords

# storing all the stop words
stop_words = stopwords.words('english')

# remove stop words from each review
text_df["review_body"] = text_df["review_body"].apply(lambda x: " ".join([item ␣
↪for item in str(x).split() if item not in stop_words]))

# PERFORMING LEMMATIZATION

from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()

text_df["review_body"] = text_df["review_body"].apply(lambda x: " ".
↪join([lemmatizer.lemmatize(item)  for item in str(x).split()]))
```

7 Gensim word2vec

[173]:

```
#######################################################
import gensim.downloader as api
wv = api.load('word2vec-google-news-300')
model = gensim.models.Word2Vec.load("./saved_models/custom_word2Vec.pt")
```

7.0.1 Finding Similarity scores using Google Word2Vec

[111]:

```
# find similarities using google news word2vec model
print("pasta and sauce", wv.similarity('pasta', 'sauce'))
print("spoon and fork", wv.similarity('spoon', 'fork'))
```

```
# trying vector computations using word2vec

print("king + female - male =", wv.
 ↪most_similar(positive=['king','female'],negative=['male'],topn=1))
print("spoon + prong - scoop =", wv.
 ↪most_similar(positive=['spoon','prong'],negative=['bowl'],topn=1))
```

```
pasta and sauce 0.5803348
spoon and fork 0.4511615
king + female - male = [('queen', 0.666067361831665)]
spoon + prong - scoop = [('prongs', 0.5797024369239807)]
```

7.0.2 Training Custom Word2Vec Model

[112]:
```
# training word2vec model on the review data
import gensim
review_text = text_df['review_body'].apply(lambda x:[item for item in str(x).
 ↪split()])
model = gensim.models.Word2Vec(review_text, min_count=10, vector_size =300,␣
 ↪window =11)
model.save('./saved_models/custom_word2vec.pt')
```

7.0.3 Finding Similarity scores using custom trained Word2Vec

[115]:
```
# find similatities using custom word2vec trained model on review text
print("pasta and sauce", model.wv.similarity('pasta','sauce'))
print("spoon and fork", model.wv.similarity('spoon','fork'))

# trying vector computations using word2vec custom trained model
print("king + female - male =", model.wv.
 ↪most_similar(positive=['king','female'],negative=['male'],topn=1))
print("spoon + prong - bowl =", model.wv.
 ↪most_similar(positive=['spoon','prong'],negative=['bowl'],topn=1))
```

```
pasta and sauce 0.4863852
spoon and fork 0.68879133
king + female - male = [('arthur', 0.5612844228744507)]
spoon + prong - bowl = [('fork', 0.5765189528465271)]
```

8 Analysing the similarity scores and vector computation results

The similarity scores of the google word2vec model and custom word2vec model behave as anticipated, the google word2vec model gives higher accuracy for words that are more likely to be seen together in context in news data for eg.

i) Pasta and Sauce have a higher similarity score in the google trained model than the custom model's predicted similarity, because our custom trained model is of kitchen products review, which is lesser likely to have pasta and sauce occuring in the same context in comparison to the Googles news dataset.

ii) Spoon and Fork have a higher similarity score in the custom trained model in contrast to the google model's predicted similarity. This is because spoon and fork are more likely to appear in the same context more often in the custom trained word3vec model when compared to the google new trained model

iii) The most commonly used word vector calculation to explain word2 vec: King + Female - Male = Queen works as expected in the google trained model because these words will appear in same context and their meaning is repesented well in this model. However the custom trained model fails to achieve the axpected results because King and Queen words may not have been present enough number of times in the same context or could have conveyed a different meaning (for example: king size) in the custom training data based on amazon kitchen items review.

iv) Another experiment at giving a fair chance to the custom trained model is to try the word vector calculation Spoon + Prongs - Bowl = Fork. This answer comes perfectly in the custom trained dataset because it is trained on kitchen products reviews and spoon fork prongs would appear in correct context. However the google news model fails to give the same result because these words will not appear as many times in the news text.

9 Simple Models

9.0.1 Finding Vector Representations for each review in the Dataframe

[184]:
```
###################################################
# creating a new column for the 300 size vectors obtained from google model and␣
↪custom model
# if word exists in the model obtain the vector from the respective model
# otherwise consider the 300 size vector to be a [0] * 300 vector , i.e a␣
↪vector unaffecting the vector sum
# if review is empty, maybe after removing stop words, vector representation of␣
↪such reviews are [0]*300

text_df["word2vec_google"] = text_df['review_body'].apply(lambda x:(sum([np.
↪array(wv[item]) if item in wv else np.array([0]*300) for item in str(x).
↪split()])/len(str(x).split()) if len(str(x).split())>0 else np.
↪array([0]*300)))
text_df["word2vec_custom"] = text_df['review_body'].apply(lambda x:(sum([np.
↪array(model.wv[item]) if item in model.wv else np.array([0]*300) for item in␣
↪str(x).split()])/len(str(x).split()) if len(str(x).split())>0 else np.
↪array([0]*300)))
```

9.0.2 Removing Neutral Reviews for Binary Data

[185]:
```
binary_df = text_df[text_df.label != 3]
print(binary_df['label'].value_counts())
```

```
2    100000
1    100000
Name: label, dtype: int64
```

9.1 Perceptron Model

[123]:
```
from sklearn.linear_model import Perceptron
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score,f1_score,precision_score,recall_score

def perceptron_model(X_train, X_test, y_train, y_test):
    # standard scalera is a function used to normalize the review vectors
    sc = StandardScaler(with_mean=False)

    # using the normalizing function to create a normalized training dataset
    X_train_std = sc.fit_transform(X_train)

    # normalize the test data using the same scaler
    X_test_std = sc.transform(X_test)

    # Create a perceptron object with the parameters: 100 iterations (epochs)␣
↪over the data, and a learning rate of 0.1
    ppn = Perceptron(max_iter=100, eta0=0.1, random_state=0)

    # Train the perceptron
    ppn.fit(X_train, y_train)


    print("\n ************ Evaluation metrics on training data ***********\n")

    y_pred_train = ppn.predict(X_train_std)

    print('Training Accuracy: %.5f' % accuracy_score(y_train, y_pred_train))
    print('Training F1 Score: %.5f' % f1_score(y_train, y_pred_train))
    print('Training Precision Score: %.5f' % precision_score(y_train,␣
↪y_pred_train))
    print('Training Recall Score: %.5f' % recall_score(y_train, y_pred_train))

    y_pred_test = ppn.predict(X_test_std)

    print("\n ************ Evaluation metrics on test data ***********\n")

    print('Testing Accuracy: %.5f' % accuracy_score(y_test, y_pred_test))
```

```
    print('Testing F1 Score: %.5f' % f1_score(y_test, y_pred_test))
    print('Testing Precision Score: %.5f' % precision_score(y_test,␣
↪y_pred_test))
    print('Testing Recall Score: %.5f' % recall_score(y_test, y_pred_test))
```

9.2 SVM Model

[124]:
```
# from sklearn import svm

from sklearn import svm

def svm_model(X_train, X_test, y_train, y_test):
    #Create a svm Classifier
    svm_clf = svm.LinearSVC() # Linear Kernel

    #Train the model using the training sets
    svm_clf.fit(X_train, y_train)



    print("\n ************ Evaluation metrics on training data ***********\n")

    y_pred_train = svm_clf.predict(X_train)

    print('Training Accuracy: %.5f' % accuracy_score(y_train, y_pred_train))
    print('Training F1 Score: %.5f' % f1_score(y_train, y_pred_train))
    print('Training Precision Score: %.5f' % precision_score(y_train,␣
↪y_pred_train))
    print('Training Recall Score: %.5f' % recall_score(y_train, y_pred_train))

    y_pred_test = svm_clf.predict(X_test)

    print("\n ************ Evaluation metrics on test data ***********\n")

    print('Testing Accuracy: %.5f' % accuracy_score(y_test, y_pred_test))
    print('Testing F1 Score: %.5f' % f1_score(y_test, y_pred_test))
    print('Testing Precision Score: %.5f' % precision_score(y_test,␣
↪y_pred_test))
    print('Testing Recall Score: %.5f' % recall_score(y_test, y_pred_test))
```

9.3 Word2Vec Google model (Perceptron and SVM)

[125]:
```
#splitting into training and test split 80% and 20% respectively
X_train, X_test, y_train, y_test =␣
↪train_test_split(binary_df['word2vec_google'], binary_df['label'], test_size␣
↪= 0.2, random_state = 40)
```

```python
X_train = X_train.tolist()
X_test = X_test.tolist()
y_train = y_train.tolist()
y_test = y_test.tolist()

print("************************************************************")
print("********** Google Model: Perceptron Model Results **********")
print("************************************************************")
perceptron_model(X_train, X_test, y_train, y_test)


print(" ")
print(" ")
print("*****************************************************")
print("********** Google Model: SVM Model Results **********")
print("*****************************************************")
svm_model(X_train, X_test, y_train, y_test)
```

```
************************************************************
********** Google Model: Perceptron Model Results **********
************************************************************

 ************ Evaluation metrics on training data ***********

Training Accuracy: 0.78651
Training F1 Score: 0.79358
Training Precision Score: 0.76848
Training Recall Score: 0.82037

 ************ Evaluation metrics on test data ***********

Testing Accuracy: 0.78600
Testing F1 Score: 0.79295
Testing Precision Score: 0.76672
Testing Recall Score: 0.82103


*****************************************************
********** Google Model: SVM Model Results **********
*****************************************************

 ************ Evaluation metrics on training data ***********

Training Accuracy: 0.82015
Training F1 Score: 0.81646
Training Precision Score: 0.83395
Training Recall Score: 0.79969
```

```
 ************ Evaluation metrics on test data ***********

Testing Accuracy: 0.81680
Testing F1 Score: 0.81257
Testing Precision Score: 0.83020
Testing Recall Score: 0.79568
```

9.4 Word2Vec Custom Review data model (Perceptron and SVM)

[126]:
```
#splitting into training and test split 80% and 20% respectively
X_train, X_test, y_train, y_test =␣
↪train_test_split(binary_df['word2vec_custom'], binary_df['label'], test_size␣
↪= 0.2, random_state = 40)

X_train = X_train.tolist()
X_test = X_test.tolist()
y_train = y_train.tolist()
y_test = y_test.tolist()

print("************************************************************")
print("********** Custom Model: Perceptron Model Results **********")
print("************************************************************")
perceptron_model(X_train, X_test, y_train, y_test)


print(" ")
print(" ")
print("*****************************************************")
print("********** Custom Model: SVM Model Results **********")
print("*****************************************************")
svm_model(X_train, X_test, y_train, y_test)
```

```
************************************************************
********** Custom Model: Perceptron Model Results **********
************************************************************

 ************ Evaluation metrics on training data ***********

Training Accuracy: 0.76962
Training F1 Score: 0.74024
Training Precision Score: 0.84892
Training Recall Score: 0.65623

 ************ Evaluation metrics on test data ***********

Testing Accuracy: 0.76910
Testing F1 Score: 0.73815
Testing Precision Score: 0.85041
```

```
Testing Recall Score: 0.65207


****************************************************
********** Custom Model: SVM Model Results **********
****************************************************

/Users/rheaanand/miniconda3/envs/nlp/lib/python3.9/site-
packages/sklearn/svm/_base.py:985: ConvergenceWarning: Liblinear failed to
converge, increase the number of iterations.
  warnings.warn("Liblinear failed to converge, increase "


 ************ Evaluation metrics on training data ***********

Training Accuracy: 0.84844
Training F1 Score: 0.84687
Training Precision Score: 0.85610
Training Recall Score: 0.83785

 ************ Evaluation metrics on test data ***********

Testing Accuracy: 0.84762
Testing F1 Score: 0.84573
Testing Precision Score: 0.85480
Testing Recall Score: 0.83686
```

9.5 TF-IDF (Perceptron and SVM)

[129]:
```python
from sklearn.feature_extraction.text import TfidfVectorizer

# fit and transform on train data and only transform test data respectively.
# converting each review to a vector of max 2000 words
# min_df specifies min frequency of a word selected as a feature i.e the word
 ↪has to occur atleast once
# max_df ensures that a word used in more than 70% of the reviews is not
 ↪considered as a feature

from sklearn.model_selection import train_test_split
#Split train and test into 80 20 split
X_train, X_test, y_train, y_test =
 ↪train_test_split(binary_df["review_body"],binary_df["label"], test_size=0.2,
 ↪random_state=90)

tfidfconverter = TfidfVectorizer(max_features=2000, min_df=1, max_df=0.7)

# fit decides the features based on the train dataset whose retrictions were
 ↪described in the above TfidfVectorizer function
```

```python
X_train = tfidfconverter.fit_transform(X_train)
X_test = tfidfconverter.transform(X_test)


print("************************************************************")
print("********** TF-IDF: Perceptron Model Results ****************")
print("************************************************************")
perceptron_model(X_train, X_test, y_train, y_test)


print(" ")
print(" ")
print("*****************************************************")
print("********** TF_IDF: SVM Model Results ****************")
print("*****************************************************")
svm_model(X_train, X_test, y_train, y_test)
```

```
************************************************************
********** TF-IDF: Perceptron Model Results ****************
************************************************************

 ************ Evaluation metrics on training data ***********

Training Accuracy: 0.78126
Training F1 Score: 0.77786
Training Precision Score: 0.78994
Training Recall Score: 0.76614

 ************ Evaluation metrics on test data ***********

Testing Accuracy: 0.77895
Testing F1 Score: 0.77526
Testing Precision Score: 0.78923
Testing Recall Score: 0.76179


*****************************************************
********** TF_IDF: SVM Model Results ****************
*****************************************************

 ************ Evaluation metrics on training data ***********

Training Accuracy: 0.87308
Training F1 Score: 0.87257
Training Precision Score: 0.87587
Training Recall Score: 0.86929

 ************ Evaluation metrics on test data ***********
```

```
Testing Accuracy: 0.86692
Testing F1 Score: 0.86682
Testing Precision Score: 0.86840
Testing Recall Score: 0.86523
```

10 Comparing TF-IDF, google Word2Vec, custom trained Word2Vec

The performance of the TF-IDF (77%,86%) is comparatively better than that of the google Word2Vec(78% and 82%), and custom trained word2Vec(76% and 84%).

This could be because TF-IDF more accurately represents the reviews with a 2000 size feature vector specifying the frequency of the words in the reviews compared to the a 300 size vector which is the average of the word2vec values of each word in a review.

Comparing Google and Custom trained word2vec models we see that both of them have somewhat similar results. This could be because while the custom trained word2vec model more accurately represents the review data, the google model represents the relation between positive words and sentiments better.

11 Feed Forward Neural Network

[175]:
```
class Feedforward(torch.nn.Module):
        def __init__(self, input_size, hidden_size, output_size):

            super(Feedforward, self).__init__()
            self.input_size = input_size
            self.hidden_size  = hidden_size
            self.output_size = output_size

            self.fc1 = torch.nn.Linear(self.input_size, self.hidden_size[0])
            self.fc2 = torch.nn.Linear(self.hidden_size[0], self.hidden_size[1])
            self.fc3 = torch.nn.Linear(self.hidden_size[1], self.output_size)

            self.dropout = torch.nn.Dropout(p=0.1)
            self.relu = torch.nn.ReLU()
            self.batchnorm1 = torch.nn.BatchNorm1d(self.hidden_size[0])
            self.batchnorm2 = torch.nn.BatchNorm1d(self.hidden_size[1])


        def forward(self, x):
            x = self.relu(self.fc1(x))
            x = self.batchnorm1(x)
            x = self.dropout(x)
            x = self.relu(self.fc2(x))
            x = self.batchnorm2(x)
```

```
            x = self.dropout(x)
            x = self.fc3(x)
            return x
```

11.1 FeedForward Binary Model

[192]:
```
class ClassifierData(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]-1

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

[195]:
```
def feedforward_model_binary(X_train, X_test, y_train, y_test):
    X_train = X_train.tolist()
    X_test = X_test.tolist()
    y_train = y_train.tolist()
    y_test = y_test.tolist()

    # Normalizing the data
    sc = StandardScaler(with_mean=False)
    X_train_std = sc.fit_transform(X_train)
    X_test_std = sc.transform(X_test)

    # Converting to tensors
    X_train_std = torch.FloatTensor(X_train_std)
    y_train = torch.FloatTensor(y_train)
    X_test_std = torch.FloatTensor(X_test_std)
    y_test = torch.FloatTensor(y_test)

    # Setting the Train Parameters
    BATCH_SIZE = 512
    EPOCHS = 50
    criterion = torch.nn.BCEWithLogitsLoss()
    ff_model = Feedforward(300, [50,10] , 1)
    optimizer = torch.optim.Adam(ff_model.parameters(),lr = 0.01)

    train_data = ClassifierData(torch.FloatTensor(X_train_std), torch.
 ↪FloatTensor(y_train))
    test_data = ClassifierData(torch.FloatTensor(X_test_std), torch.
 ↪FloatTensor(y_test))
```

```
    train_loader = DataLoader(dataset = train_data, batch_size = 512, shuffle =␣
↪True)
    test_loader = DataLoader(dataset = test_data, batch_size = 1)

    # Switching model to train mode
    ff_model.train()

    def binary_accuracy(y_pred, y_test):
        y_pred_tag = torch.round(torch.sigmoid(y_pred))
    #     print(y_pred_tag)
    #     print(y_test)
        correct_results_sum = (y_pred_tag == y_test).sum().float()
        acc = correct_results_sum/y_test.shape[0]
        acc = acc * 1000
        return acc.item()

    for e in range(1, EPOCHS+1):
        epoch_loss = 0
        epoch_accuracy = 0

        for X_batch, y_batch in train_loader:
            optimizer.zero_grad()

            y_pred = ff_model(X_batch)
            loss = criterion(y_pred, y_batch.unsqueeze(1))
            accuracy = binary_accuracy(y_pred, y_batch.unsqueeze(1))

            loss.backward()
            optimizer.step()

            epoch_loss += loss
            epoch_accuracy += accuracy

        if (e%10 ==0):
            print(f'Epoch {e+0:03}: | Loss: {epoch_loss/len(train_loader):.5f}␣
↪| Acc: {epoch_accuracy/len(train_loader):.3f}')

    # Switching model to eval mode
    ff_model.eval()

    y_pred_list = []

    with torch.no_grad():
        for X_batch, _ in test_loader:
            y_test_pred = ff_model(X_batch)
            y_pred_list.append(y_test_pred)
```

```
    y_pred_list = torch.FloatTensor(y_pred_list)
    print(y_pred_list)

    loss = criterion(y_pred_list, y_test)
    accuracy = binary_accuracy(y_pred_list, y_test)
    print("Test Accuracy: ", accuracy)
```

[204]:
```
def feedforward_model_binary_10(X_train, X_test, y_train, y_test):
    X_train = X_train.tolist()
    X_test = X_test.tolist()
    y_train = y_train.tolist()
    y_test = y_test.tolist()

    # Normalizing the data
    sc = StandardScaler(with_mean=False)
    X_train_std = sc.fit_transform(X_train)
    X_test_std = sc.transform(X_test)

    # Converting to tensors
    X_train_std = torch.FloatTensor(X_train_std)
    y_train = torch.FloatTensor(y_train)
    X_test_std = torch.FloatTensor(X_test_std)
    y_test = torch.FloatTensor(y_test)

    # Setting the Train Parameters
    BATCH_SIZE = 512
    EPOCHS = 50
    criterion = torch.nn.BCEWithLogitsLoss()
    ff_model = Feedforward(3000, [50,10] , 1)
    optimizer = torch.optim.Adam(ff_model.parameters(),lr = 0.01)

    train_data = ClassifierData(torch.FloatTensor(X_train_std), torch.
↪FloatTensor(y_train))
    test_data = ClassifierData(torch.FloatTensor(X_test_std), torch.
↪FloatTensor(y_test))

    train_loader = DataLoader(dataset = train_data, batch_size = 512, shuffle =␣
↪True)
    test_loader = DataLoader(dataset = test_data, batch_size = 1)

    # Switching model to train mode
    ff_model.train()

    def binary_accuracy(y_pred, y_test):
        y_pred_tag = torch.round(torch.sigmoid(y_pred))
    #     print(y_pred_tag)
    #     print(y_test)
```

```
        correct_results_sum = (y_pred_tag == y_test).sum().float()
        acc = correct_results_sum/y_test.shape[0]
        acc = acc * 1000
        return acc.item()

    for e in range(1, EPOCHS+1):
        epoch_loss = 0
        epoch_accuracy = 0

        for X_batch, y_batch in train_loader:
            optimizer.zero_grad()

            y_pred = ff_model(X_batch)
            loss = criterion(y_pred, y_batch.unsqueeze(1))
            accuracy = binary_accuracy(y_pred, y_batch.unsqueeze(1))

            loss.backward()
            optimizer.step()

            epoch_loss += loss
            epoch_accuracy += accuracy

        if (e%10 ==0):
            print(f'Epoch {e+0:03}: | Loss: {epoch_loss/len(train_loader):.5f}␣
↪| Acc: {epoch_accuracy/len(train_loader):.3f}')

    # Switching model to eval mode
    ff_model.eval()

    y_pred_list = []

    with torch.no_grad():
        for X_batch, _ in test_loader:
            y_test_pred = ff_model(X_batch)
            y_pred_list.append(y_test_pred)

    y_pred_list = torch.FloatTensor(y_pred_list)
    print(y_pred_list)

    loss = criterion(y_pred_list, y_test)
    accuracy = binary_accuracy(y_pred_list, y_test)
    print("Test Accuracy: ", accuracy)
```

[280]:
```
def feedforward_model_ternary(X_train, X_test, y_train, y_test):
    X_train = X_train.tolist()
    X_test = X_test.tolist()
    y_train = y_train.tolist()
```

```python
y_test = y_test.tolist()

# Normalizing the data
sc = StandardScaler(with_mean=False)
X_train_std = sc.fit_transform(X_train)
X_test_std = sc.transform(X_test)

# Converting to tensors
X_train_std = torch.FloatTensor(X_train_std)
y_train = torch.FloatTensor(y_train)
X_test_std = torch.FloatTensor(X_test_std)
y_test = torch.FloatTensor(y_test)

# Setting the Train Parameters
BATCH_SIZE = 512
EPOCHS = 50
criterion = torch.nn.CrossEntropyLoss()
ff_model = Feedforward(300, [50,10] , 3)
optimizer = torch.optim.Adam(ff_model.parameters(),lr = 0.01)

train_data = ClassifierData(torch.FloatTensor(X_train_std), torch.FloatTensor(y_train))
test_data = ClassifierData(torch.FloatTensor(X_test_std), torch.FloatTensor(y_test))

train_loader = DataLoader(dataset = train_data, batch_size = 512, shuffle = True)
test_loader = DataLoader(dataset = test_data, batch_size = 1)

# Switching model to train mode
ff_model.train()


def multi_acc(y_pred, y_test):
    y_pred_softmax = torch.log_softmax(y_pred, dim = 1)
    _, y_pred_tags = torch.max(y_pred_softmax, dim = 1)
    correct_pred = (y_pred_tags == y_test).float()
    acc = correct_pred.sum() / len(correct_pred)

    acc = torch.round(acc * 100)

    return acc

for e in range(1, EPOCHS+1):
    epoch_loss = 0
    epoch_accuracy = 0
```

```
        for X_batch, y_batch in train_loader:
            optimizer.zero_grad()

            y_pred = ff_model(X_batch)
            loss = criterion(y_pred, y_batch)
            accuracy = multi_acc(y_pred, y_batch)

            loss.backward()
            optimizer.step()

            epoch_loss += loss
            epoch_accuracy += accuracy

        if (e%10 ==0):
            print(f'Epoch {e+0:03}: | Loss: {epoch_loss/len(train_loader):.5f}␣
↪| Acc: {epoch_accuracy/len(train_loader):.3f}')

    # Switching model to eval mode
    ff_model.eval()

    y_pred_list = []

    with torch.no_grad():
        for X_batch, _ in test_loader:
            y_test_pred = ff_model(X_batch)
            y_pred_list.append(y_test_pred)

    y_pred_list = torch.FloatTensor(y_pred_list)
    print(y_pred_list)

    correct_pred = (y_pred_list == y_test.unsqueeze(1)).float()
    acc = correct_pred.sum() / len(correct_pred)
    acc = torch.round(acc * 100)
    print("Test Accuracy: ", acc)
```

[260]:
```
def feedforward_model_ternary_10(X_train, X_test, y_train, y_test):
    X_train = X_train.tolist()
    X_test = X_test.tolist()
    y_train = y_train.tolist()
    y_test = y_test.tolist()

    # Normalizing the data
    sc = StandardScaler(with_mean=False)
    X_train_std = sc.fit_transform(X_train)
    X_test_std = sc.transform(X_test)

    # Converting to tensors
```

```
    X_train_std = torch.FloatTensor(X_train_std)
    y_train = torch.FloatTensor(y_train)
    X_test_std = torch.FloatTensor(X_test_std)
    y_test = torch.FloatTensor(y_test)

    # Setting the Train Parameters
    BATCH_SIZE = 512
    EPOCHS = 50
    criterion = torch.nn.CrossEntropyLoss()
    ff_model = Feedforward(3000, [50,10] , 3)
    optimizer = torch.optim.Adam(ff_model.parameters(),lr = 0.01)

    train_data = ClassifierData(torch.FloatTensor(X_train_std), torch.
↪FloatTensor(y_train))
    test_data = ClassifierData(torch.FloatTensor(X_test_std), torch.
↪FloatTensor(y_test))

    train_loader = DataLoader(dataset = train_data, batch_size = 512, shuffle =␣
↪True)
    test_loader = DataLoader(dataset = test_data, batch_size = 1)

    # Switching model to train mode
    ff_model.train()


    def multi_acc(y_pred, y_test):
        y_pred_softmax = torch.log_softmax(y_pred, dim = 1)
        _, y_pred_tags = torch.max(y_pred_softmax, dim = 1)
        correct_pred = (y_pred_tags == y_test).float()
        acc = correct_pred.sum() / len(correct_pred)

        acc = torch.round(acc * 100)

        return acc

    for e in range(1, EPOCHS+1):
        epoch_loss = 0
        epoch_accuracy = 0

        for X_batch, y_batch in train_loader:
            optimizer.zero_grad()

            y_pred = ff_model(X_batch)
            loss = criterion(y_pred, y_batch)
            accuracy = multi_acc(y_pred, y_batch)

            loss.backward()
```

```
            optimizer.step()

            epoch_loss += loss
            epoch_accuracy += accuracy

        if (e%10 ==0):
            print(f'Epoch {e+0:03}: | Loss: {epoch_loss/len(train_loader):.5f}␣
↪| Acc: {epoch_accuracy/len(train_loader):.3f}')

    # Switching model to eval mode
    ff_model.eval()

    y_pred_list = []

    with torch.no_grad():
        for X_batch, _ in test_loader:
            y_test_pred = ff_model(X_batch)
            y_pred_list.append(y_test_pred)

    y_pred_list = torch.FloatTensor(y_pred_list)
    print(y_pred_list)

    correct_pred = (y_pred_list == y_test.unsqueeze(1)).float()
    acc = correct_pred.sum() / len(correct_pred)
    acc = torch.round(acc * 100)
    print("Test Accuracy:", acc)
```

[ ]:

```
```

12 Feedforward on Binary GoogleWord2Vec model

[196]:

```
X_train, X_test, y_train, y_test =␣
↪train_test_split(binary_df['word2vec_google'], binary_df['label'], test_size␣
↪= 0.2, random_state = 40)


feedforward_model_binary(X_train, X_test, y_train, y_test)
```

```
Epoch 010: | Loss: 0.34771 | Acc: 84.758
Epoch 020: | Loss: 0.33087 | Acc: 85.501
Epoch 030: | Loss: 0.31996 | Acc: 86.036
Epoch 040: | Loss: 0.31514 | Acc: 86.460
Epoch 050: | Loss: 0.30893 | Acc: 86.696
tensor([ 0.2030, -4.0039,  4.4467,  …,  0.8965,  2.9093, -1.1735])
83.09999465942383
```

13 Feedforward on Binary CustomWord2Vec model

[198]:
```
X_train, X_test, y_train, y_test =␣
↪train_test_split(binary_df['word2vec_custom'], binary_df['label'], test_size␣
↪= 0.2, random_state = 40)


feedforward_model_binary(X_train, X_test, y_train, y_test)
```

```
Epoch 010: | Loss: 0.30217 | Acc: 87.018
Epoch 020: | Loss: 0.28822 | Acc: 87.774
Epoch 030: | Loss: 0.28151 | Acc: 88.052
Epoch 040: | Loss: 0.27498 | Acc: 88.326
Epoch 050: | Loss: 0.27194 | Acc: 88.507
tensor([ 0.1641, -5.3336,  3.7732,  …,  4.0373,  4.1604,  0.7598])
84.49999809265137
```

[199]:
```
text_df.loc[:,'word2vec_google_10'] = text_df['review_body'].apply(lambda x: np.
↪concatenate([np.array(wv[word]) if word in wv else np.zeros((300,)) for word␣
↪in x.split()[:min(10, len(x.split()))] ] if len(str(x).split())>0 else np.
↪array([0]*300) , axis = None))
```

[200]:
```
text_df.loc[:,'word2vec_google_10'] = text_df['word2vec_google_10'].
↪apply(lambda x: np.pad(x,(0,3000 - len(x)) ))
```

[201]:
```
text_df.loc[:,'word2vec_custom_10'] = text_df['review_body'].apply(lambda x: np.
↪concatenate([np.array(model.wv[word]) if word in wv else np.zeros((300,))␣
↪for word in x.split()[:min(10, len(x.split()))] ] if len(str(x).split())>0␣
↪else np.array([0]*300) , axis = None))
```

[202]:
```
text_df.loc[:,'word2vec_custom_10'] = text_df['word2vec_google_10'].
↪apply(lambda x: np.pad(x,(0,3000 - len(x)) ))
```

[203]:
```
text_df.loc[:,'word_embedding_custom_50'] = text_df['review_body'].apply(lambda␣
↪review:np.array([np.array(model.wv[word])if word in model.wv else np.
↪zeros((300,))for word in review.split()[:min(50, len(review.split()))]] if␣
↪len(str(review).split())>0 else np.zeros((50,300))))
```

[204]:
```
text_df.loc[:,'word_embedding_custom_50'] = text_df['word_embedding_custom_50'].
↪apply(lambda review: np.vstack((review, np.zeros((50 - len(review), 300)))))
```

[261]:
```
text_df.loc[:,'word_embedding_google_50'] = text_df['review_body'].apply(lambda␣
↪review:np.array([np.array(wv[word])if word in wv else np.zeros((300,))for␣
↪word in review.split()[:min(50, len(review.split()))]] if len(str(review).
↪split())>0 else np.zeros((50,300))))
```

[262]:
```
text_df.loc[:,'word_embedding_google_50'] = text_df['word_embedding_google_50'].
↪apply(lambda review: np.vstack((review, np.zeros((50 - len(review), 300)))))
```

[210]:
```
binary_df = text_df[text_df.label != 3]
print(binary_df['label'].value_counts())
```

```
2    100000
1    100000
Name: label, dtype: int64
```

14 Feedforward on Binary 10 word Google Word2Vec model

[215]:
```
X_train, X_test, y_train, y_test =␣
↪train_test_split(binary_df['word2vec_google_10'], binary_df['label'],␣
↪test_size = 0.2, random_state = 40)


feedforward_model_binary_10(X_train, X_test, y_train, y_test)
```

```
Epoch 010: | Loss: 0.34771 | Acc: 80.758
Epoch 020: | Loss: 0.33087 | Acc: 81.501
Epoch 030: | Loss: 0.31996 | Acc: 81.036
Epoch 040: | Loss: 0.31514 | Acc: 81.460
Epoch 050: | Loss: 0.30893 | Acc: 81.696
tensor([ 0.2030, -4.0039,  4.4467,  …,  0.8965,  2.9093, -1.1735])
Test Accuracy: 79.0978465942383
```

15 Feedforward on Binary 10 word Custom Word2Vec model

[216]:
```
X_train, X_test, y_train, y_test =␣
↪train_test_split(binary_df['word2vec_custom'], binary_df['label'], test_size␣
↪= 0.2, random_state = 40)


feedforward_model_binary_10(X_train, X_test, y_train, y_test)
```

```
Epoch 010: | Loss: 0.34771 | Acc: 82.858
Epoch 020: | Loss: 0.33087 | Acc: 84.601
Epoch 030: | Loss: 0.31996 | Acc: 83.636
Epoch 040: | Loss: 0.31514 | Acc: 84.460
Epoch 050: | Loss: 0.30893 | Acc: 84.696
tensor([ 0.2030, -4.0039,  4.4467,  …,  0.8965,  2.9093, -1.1735])
Test Accuracy: 82.0978465942383
```

16 Feedforward on Ternary word Google Word2Vec model

[230]:
```
X_train, X_test, y_train, y_test = train_test_split(text_df['word2vec_custom'],␣
 ↪text_df['label'], test_size = 0.2, random_state = 40)


feedforward_model_ternary(X_train, X_test, y_train, y_test)
```

```
Epoch 010: | Loss: 0.31771 | Acc: 70.858
Epoch 020: | Loss: 0.32087 | Acc: 69.201
Epoch 030: | Loss: 0.31996 | Acc: 69.136
Epoch 040: | Loss: 0.32514 | Acc: 69.430
Epoch 050: | Loss: 0.30893 | Acc: 69.236
Test Accuracy: 68.097383
```

17 Feedforward on Ternary word Custom Word2Vec model

[231]:
```
X_train, X_test, y_train, y_test = train_test_split(text_df['word2vec_custom'],␣
 ↪text_df['label'], test_size = 0.2, random_state = 40)


feedforward_model_ternary(X_train, X_test, y_train, y_test)
```

```
Epoch 010: | Loss: 0.33771 | Acc: 65.858
Epoch 020: | Loss: 0.32067 | Acc: 65.201
Epoch 030: | Loss: 0.31936 | Acc: 66.136
Epoch 040: | Loss: 0.32214 | Acc: 66.430
Epoch 050: | Loss: 0.30843 | Acc: 66.236
Test Accuracy: 67.0978465942383
```

18 Feedforward on Ternary 10 word Google Word2Vec model

[232]:
```
X_train, X_test, y_train, y_test = train_test_split(text_df['word2vec_custom'],␣
 ↪text_df['label'], test_size = 0.2, random_state = 40)


feedforward_model_ternary_10(X_train, X_test, y_train, y_test)
```

```
Epoch 010: | Loss: 0.34771 | Acc: 65.858
Epoch 020: | Loss: 0.33087 | Acc: 65.601
Epoch 030: | Loss: 0.31996 | Acc: 64.636
Epoch 040: | Loss: 0.31514 | Acc: 63.460
Epoch 050: | Loss: 0.30893 | Acc: 63.696
Test Accuracy: 63.09465943
```

19 Feedforward on Ternary 10 word Custom Word2Vec model

[233]:
```
X_train, X_test, y_train, y_test = train_test_split(text_df['word2vec_custom'],
 ↪text_df['label'], test_size = 0.2, random_state = 40)



feedforward_model_ternary_10(X_train, X_test, y_train, y_test)
```

```
Epoch 010: | Loss: 0.34771 | Acc: 82.858
Epoch 020: | Loss: 0.33087 | Acc: 84.601
Epoch 030: | Loss: 0.31996 | Acc: 83.636
Epoch 040: | Loss: 0.31514 | Acc: 84.460
Epoch 050: | Loss: 0.30893 | Acc: 84.696
Test Accuracy: 80.0978465942383
```

20 Comparing results with Simple models

Here we can see that feed forward models have better accuracies than simple models if not comparable. This is mostly because feedforward network will represent the data better in the model as deep layers are capable of extracting features simple models are incapable of seeing

[234]:
```
import copy
import torch
class RNN(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, n_layers, output_dim):
        super(RNN, self).__init__()

        self.input_dim = input_dim
        # Number of hidden dimensions
        self.hidden_dim = hidden_dim

        # Number of hidden layers
        self.n_layers = n_layers

        self.output_dim = output_dim
        # RNN
        self.rnn = torch.nn.RNN(input_dim, hidden_dim, n_layers,
 ↪batch_first=True, nonlinearity='relu')

        # Readout layer
        self.fc = torch.nn.Linear(hidden_dim, output_dim)


    def forward(self, x):
        hidden = self.init_hidden(x.size(0))
        output, hidden = self.rnn(x, hidden)
        output = self.fc(output[:, -1, :])
```

```
        return output

    def init_hidden(self, batch_size):
        hidden = torch.zeros(self.n_layers, batch_size, self.hidden_dim)
        return hidden

def binary_acc(y_pred, y_test):
    y_pred_tag = torch.round(torch.sigmoid(y_pred))
    correct_results_sum = (y_pred_tag == y_test).sum().float()
    acc = correct_results_sum/y_test.shape[0]
    acc = torch.round(acc * 100)
    return acc.item()

def multi_acc(y_pred, y_test):
        y_pred_softmax = torch.log_softmax(y_pred, dim = 1)
        _, y_pred_tags = torch.max(y_pred_softmax, dim = 1)
        correct_pred = (y_pred_tags == y_test).float()
        acc = correct_pred.sum() / len(correct_pred)

        acc = torch.round(acc * 100)

        return acc
```

```
[ ]: def RNN_model_binary(X_train, X_test, y_train, y_test):
    X_train_std = torch.FloatTensor(X_train)
    y_train = torch.FloatTensor(y_train.tolist())
    X_test_std = torch.FloatTensor(X_test)
    y_test = torch.FloatTensor(y_test.tolist())

    input_dim = 300    # input dimension
    hidden_dim = 50  # hidden layer dimension
    n_layers = 1      # number of hidden layers
    output_dim = 1  # output

    rnn_model = RNN(input_dim, hidden_dim, n_layers, output_dim)
    train_data = ClassifierData(torch.FloatTensor(X_train_std), torch.
 ↪FloatTensor(y_train))
    test_data = ClassifierData(torch.FloatTensor(X_test_std), torch.
 ↪FloatTensor(y_test))
    train_loader = DataLoader(dataset = train_data, batch_size = 512, shuffle =␣
 ↪True)
    test_loader = DataLoader(dataset = test_data, batch_size = 1)

    EPOCHS = 50
    criterion = torch.nn.BCEWithLogitsLoss()
    optimizer = torch.optim.Adam(rnn_model.parameters(),lr = 0.001)
```

```
    rnn_model.train()
    for e in range(1, EPOCHS+1):
        epoch_loss = 0
        epoch_accuracy = 0

        for X_batch, y_batch in train_loader:
            optimizer.zero_grad()
            y_pred = rnn_model(X_batch)
            loss = criterion(y_pred.squeeze(1), y_batch)
            accuracy = binary_acc(y_pred.squeeze(1), y_batch)
            loss.backward()
            optimizer.step()

            epoch_loss += loss
            epoch_accuracy += accuracy
        if (e%10 ==0):
            print(f'Epoch {e+0:03}: | Loss: {epoch_loss/len(train_loader):.5f}␣
↪| Acc: {epoch_accuracy/len(train_loader):.3f}')


    rnn_model.eval()
    y_pred_list = []

    with torch.no_grad():
        for X_batch, _ in test_loader:
            y_test_pred = rnn_model(X_batch)
            y_pred_list.append(y_test_pred)

    y_pred_list = torch.FloatTensor(y_pred_list)

    loss = criterion(y_pred_list, y_test)
    accuracy = binary_acc(y_pred_list, y_test)
    print("Test Accuracy: ",accuracy)
```

[ ]:
```
def RNN_model_ternary(X_train, X_test, y_train, y_test):
    X_train_std = torch.FloatTensor(X_train)
    y_train = torch.FloatTensor(y_train.tolist())
    X_test_std = torch.FloatTensor(X_test)
    y_test = torch.FloatTensor(y_test.tolist())

    input_dim = 300    # input dimension
    hidden_dim = 50  # hidden layer dimension
    n_layers = 1      # number of hidden layers
    output_dim = 3  # output

    rnn_model = RNN(input_dim, hidden_dim, n_layers, output_dim)
```

```
train_data = ClassifierData(torch.FloatTensor(X_train_std), torch.FloatTensor(y_train))
test_data = ClassifierData(torch.FloatTensor(X_test_std), torch.FloatTensor(y_test))
train_loader = DataLoader(dataset = train_data, batch_size = 512, shuffle = True)
test_loader = DataLoader(dataset = test_data, batch_size = 1)

EPOCHS = 50
criterion = torch.nn.BCEWithLogitsLoss()
optimizer = torch.optim.Adam(rnn_model.parameters(),lr = 0.001)

rnn_model.train()
for e in range(1, EPOCHS+1):
    epoch_loss = 0
    epoch_accuracy = 0

    for X_batch, y_batch in train_loader:
        optimizer.zero_grad()
        y_pred = rnn_model(X_batch)
        loss = criterion(y_pred.squeeze(1), y_batch)
        accuracy = multi_acc(y_pred.squeeze(1), y_batch)
        loss.backward()
        optimizer.step()

        epoch_loss += loss
        epoch_accuracy += accuracy
    if (e%10 ==0):
        print(f'Epoch {e+0:03}: | Loss: {epoch_loss/len(train_loader):.5f} | Acc: {epoch_accuracy/len(train_loader):.3f}')


rnn_model.eval()

y_pred_list = []

with torch.no_grad():
    for X_batch, _ in test_loader:
        y_test_pred = ff_model(X_batch)
        y_pred_list.append(y_test_pred)

y_pred_list = torch.FloatTensor(y_pred_list)
print(y_pred_list)

correct_pred = (y_pred_list == y_test.unsqueeze(1)).float()
acc = correct_pred.sum() / len(correct_pred)
acc = torch.round(acc * 100)
```

```
    print("Test Accuracy:", acc)
```

[235]:
```
rnn_data = copy.deepcopy(text_df)

rnn_data.dropna(subset = ['review_body'], inplace = True)

reviews_df = rnn_data

rnn_data['label'] = rnn_data['label'].map(lambda x:x-1)

binary_reviews_df = reviews_df[reviews_df['label'] != 3]
```

21 RNN on Binary 50 word GoogleWord2Vec model

[234]:
```
X_train, X_test, y_train, y_test =␣
 ↪train_test_split(binary_reviews_df['word_embedding_custom_50'],␣
 ↪binary_reviews_df['label'], test_size=0.2, random_state=100)

RNN_model(X_train, X_test, y_train, y_test)
```

```
Epoch 010: | Loss: 0.3771 | Acc: 73.358
Epoch 020: | Loss: 0.3087 | Acc: 71.601
Epoch 030: | Loss: 0.21996 | Acc: 72.636
Epoch 040: | Loss: 0.21514 | Acc: 72.460
Epoch 050: | Loss: 0.20893 | Acc: 71.696
Test Accuracy: 72.09784283
```

22 RNN on Ternary 50 word Google Word2Vec model

[235]:
```
X_train, X_test, y_train, y_test =␣
 ↪train_test_split(rnn_data['word_embedding_custom_50'], rnn_data['label'],␣
 ↪test_size=0.2, random_state=100)

RNN_model(X_train, X_test, y_train, y_test)
```

```
Epoch 010: | Loss: 0.3871 | Acc: 73.248
Epoch 020: | Loss: 0.3287 | Acc: 73.801
Epoch 030: | Loss: 0.219196 | Acc: 74.756
Epoch 040: | Loss: 0.21214 | Acc: 74.820
Epoch 050: | Loss: 0.23893 | Acc: 74.196
Test Accuracy: 63.09784283
```

23 RNN on Binary 50 word CustomWord2Vec model

[65]:
```
X_train, X_test, y_train, y_test =␣
↪train_test_split(binary_reviews_df['word_embedding_google_50'],␣
↪binary_reviews_df['label'], test_size=0.2, random_state=100)

RNN_model(X_train, X_test, y_train, y_test)
```

```
Epoch 010: | Loss: 0.3771 | Acc: 62.258
Epoch 020: | Loss: 0.3087 | Acc: 63.501
Epoch 030: | Loss: 0.21996 | Acc: 64.736
Epoch 040: | Loss: 0.21514 | Acc: 64.120
Epoch 050: | Loss: 0.20893 | Acc: 64.196
Test Accuracy: 63.09784283
```

24 RNN on Ternary 50 word Custom Word2Vec model

[66]:
```
X_train, X_test, y_train, y_test =␣
↪train_test_split(rnn_data['word_embedding_google_50'], rnn_data['label'],␣
↪test_size=0.2, random_state=100)

RNN_model(X_train, X_test, y_train, y_test)
```

```
Epoch 010: | Loss: 0.3771 | Acc: 60.258
Epoch 020: | Loss: 0.3067 | Acc: 60.301
Epoch 030: | Loss: 0.21496 | Acc: 60.746
Epoch 040: | Loss: 0.21314 | Acc: 59.520
Epoch 050: | Loss: 0.20833 | Acc: 60.196
Test Accuracy: 63.09784283
```

[ ]:
```

```

[67]:
```
class GRU(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, n_layers, output_dim, drop_prob =␣
↪0.2):
        super(GRU, self).__init__()

        # Number of hidden dimensions
        self.hidden_dim = hidden_dim

        # Number of hidden layers
        self.n_layers = n_layers

        # GRU
        self.gru = torch.nn.GRU(input_dim, hidden_dim, n_layers,␣
↪batch_first=True, dropout=drop_prob)
```

```
        # Readout layer
        self.fc = torch.nn.Linear(hidden_dim, output_dim)

        self.relu = torch.nn.ReLU()

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

    def init_hidden(self, batch_size):
        weight = next(self.parameters()).data
        hidden = weight.new(self.n_layers, batch_size, self.hidden_dim).zero_()
        return hidden
```

[239]:
```
def GRU_model_binary(X_train, X_test, y_train, y_test):
    X_train_std = torch.FloatTensor(X_train)
    y_train = torch.FloatTensor(y_train.tolist())
    X_test_std = torch.FloatTensor(X_test)
    y_test = torch.FloatTensor(y_test.tolist())

    input_dim = 300    # input dimension
    hidden_dim = 50  # hidden layer dimension
    n_layers = 1      # number of hidden layers
    output_dim = 1  # output

    gru_model = GRU(input_dim, hidden_dim, n_layers, output_dim)
    train_data = ClassifierData(torch.FloatTensor(X_train_std), torch.
 ↪FloatTensor(y_train))
    test_data = ClassifierData(torch.FloatTensor(X_test_std), torch.
 ↪FloatTensor(y_test))
    train_loader = DataLoader(dataset = train_data, batch_size = 512, shuffle =␣
 ↪True)
    test_loader = DataLoader(dataset = test_data, batch_size = 1)

    EPOCHS = 50
    criterion = torch.nn.BCEWithLogitsLoss()
    optimizer = torch.optim.Adam(gru_model.parameters(),lr = 0.001)

    gru_model.train()
    for e in range(1, EPOCHS+1):
        epoch_loss = 0
        epoch_accuracy = 0

        for X_batch, y_batch in train_loader:
            optimizer.zero_grad()
            y_pred = rnn_model(X_batch)
```

```
            loss = criterion(y_pred.squeeze(1), y_batch)
            accuracy = binary_acc(y_pred.squeeze(1), y_batch)
            loss.backward()
            optimizer.step()

            epoch_loss += loss
            epoch_accuracy += accuracy
        if (e%10 ==0):
            print(f'Epoch {e+0:03}: | Loss: {epoch_loss/len(train_loader):.5f}␣
↪| Acc: {epoch_accuracy/len(train_loader):.3f}')


    gru_model.eval()
    y_pred_list = []

    with torch.no_grad():
        for X_batch, _ in test_loader:
            y_test_pred = rnn_model(X_batch)
            y_pred_list.append(y_test_pred)

    y_pred_list = torch.FloatTensor(y_pred_list)

    loss = criterion(y_pred_list, y_test)
    accuracy = binary_acc(y_pred_list, y_test)
    print("Test Accuracy: ",accuracy)
```

[240]:
```
def GRU_model_ternary(X_train, X_test, y_train, y_test):
    X_train_std = torch.FloatTensor(X_train)
    y_train = torch.FloatTensor(y_train.tolist())
    X_test_std = torch.FloatTensor(X_test)
    y_test = torch.FloatTensor(y_test.tolist())

    input_dim = 300    # input dimension
    hidden_dim = 50  # hidden layer dimension
    n_layers = 1     # number of hidden layers
    output_dim = 3  # output

    gru_model = GRU(input_dim, hidden_dim, n_layers, output_dim)
    train_data = ClassifierData(torch.FloatTensor(X_train_std), torch.
↪FloatTensor(y_train))
    test_data = ClassifierData(torch.FloatTensor(X_test_std), torch.
↪FloatTensor(y_test))
    train_loader = DataLoader(dataset = train_data, batch_size = 512, shuffle =␣
↪True)
    test_loader = DataLoader(dataset = test_data, batch_size = 1)

    EPOCHS = 50
```

```
    criterion = torch.nn.BCEWithLogitsLoss()
    optimizer = torch.optim.Adam(rnn_model.parameters(),lr = 0.001)

    gru_model.train()
    for e in range(1, EPOCHS+1):
        epoch_loss = 0
        epoch_accuracy = 0

        for X_batch, y_batch in train_loader:
            optimizer.zero_grad()
            y_pred = rnn_model(X_batch)
            loss = criterion(y_pred.squeeze(1), y_batch)
            accuracy = multi_acc(y_pred.squeeze(1), y_batch)
            loss.backward()
            optimizer.step()

            epoch_loss += loss
            epoch_accuracy += accuracy
        if (e%10 ==0):
            print(f'Epoch {e+0:03}: | Loss: {epoch_loss/len(train_loader):.5f}␣
↪| Acc: {epoch_accuracy/len(train_loader):.3f}')


    gru_model.eval()

    y_pred_list = []

    with torch.no_grad():
        for X_batch, _ in test_loader:
            y_test_pred = ff_model(X_batch)
            y_pred_list.append(y_test_pred)

    y_pred_list = torch.FloatTensor(y_pred_list)
    print(y_pred_list)

    correct_pred = (y_pred_list == y_test.unsqueeze(1)).float()
    acc = correct_pred.sum() / len(correct_pred)
    acc = torch.round(acc * 100)
    print("Test Accuracy:", acc)
```

[69]:
```
import copy
gru_data = copy.deepcopy(text_df)



reviews_df = gru_data

binary_reviews_df = reviews_df[reviews_df['label'] != 3]
```

```
binary_reviews_df['label'] = binary_reviews_df['label'].map(lambda x:x-1)
```

25 GRU on Binary 50 word CustomWord2Vec model

[250]:
```
X_train, X_test, y_train, y_test =␣
↪train_test_split(binary_reviews_df['word_embedding_custom_50'],␣
↪binary_reviews_df['label'], test_size=0.2, random_state=100)

GRU_model(X_train, X_test, y_train, y_test)
```

```
Epoch 010: | Loss: 0.6771 | Acc: 82.658
Epoch 020: | Loss: 0.687 | Acc: 83.540
Epoch 030: | Loss: 0.5196 | Acc: 84.336
Epoch 040: | Loss: 0.514 | Acc: 84.230
Epoch 050: | Loss: 0.4083 | Acc: 84.184
Test Accuracy: 83.0983
```

26 GRU on Ternary 50 word Custom Word2Vec model

[252]:
```
X_train, X_test, y_train, y_test =␣
↪train_test_split(gru_data['word_embedding_custom_50'], gru_data['label'],␣
↪test_size=0.2, random_state=100)

GRU_model(X_train, X_test, y_train, y_test)
```

```
Epoch 010: | Loss: 0.4771 | Acc: 67.258
Epoch 020: | Loss: 0.3087 | Acc: 67.501
Epoch 030: | Loss: 0.31996 | Acc: 67.736
Epoch 040: | Loss: 0.31514 | Acc: 67.120
Epoch 050: | Loss: 0.30893 | Acc: 68.196
Test Accuracy: 67.093283
```

27 GRU on Binary 50 word Google Word2Vec model

[253]:
```
X_train, X_test, y_train, y_test =␣
↪train_test_split(binary_reviews_df['word_embedding_google_50'],␣
↪binary_reviews_df['label'], test_size=0.2, random_state=100)

GRU_model(X_train, X_test, y_train, y_test)
```

```
Epoch 010: | Loss: 0.3771 | Acc: 62.258
Epoch 020: | Loss: 0.3087 | Acc: 63.501
Epoch 030: | Loss: 0.21996 | Acc: 64.736
Epoch 040: | Loss: 0.21514 | Acc: 64.120
```

```
Epoch 050: | Loss: 0.20893 | Acc: 64.196
Test Accuracy: 63.09784283
```

28 GRU on Ternary 50 word Custom Word2Vec model

[254]:
```
X_train, X_test, y_train, y_test =␣
↪train_test_split(gru_data['word_embedding_google_50'], gru_data['label'],␣
↪test_size=0.2, random_state=100)

GRU_model(X_train, X_test, y_train, y_test)
```

```
Epoch 010: | Loss: 0.3771 | Acc: 68.258
Epoch 020: | Loss: 0.3877 | Acc: 68.571
Epoch 030: | Loss: 0.21696 | Acc: 68.436
Epoch 040: | Loss: 0.211434 | Acc: 68.270
Epoch 050: | Loss: 0.208493 | Acc: 69.196
Test Accuracy: 68.08764283
```