Data Preparation

Importing Important libraries

List of all the versions of the libraries used:

python 3.11.5

pandas 2.0.3

numpy 1.24.3

nltk 3.8.1

matplotlib 3.7.2

contractions 0.1.73

scikit-learn 1.2.2

tabulate 0.8.10

gensim 4.3.0

tensorflown 2.15.0

In [ ]:
```
!pip install contractions
```

```
Requirement already satisfied: contractions in /Users/namyashah/anaconda3/lib/python3.11/site-packages (
0.1.73)
Requirement already satisfied: textsearch>=0.0.21 in /Users/namyashah/anaconda3/lib/python3.11/site-packa
ges (from contractions) (0.0.24)
Requirement already satisfied: anyascii in /Users/namyashah/anaconda3/lib/python3.11/site-packages (from
textsearch>=0.0.21->contractions) (0.3.2)
Requirement already satisfied: pyahocorasick in /Users/namyashah/anaconda3/lib/python3.11/site-packages (
from textsearch>=0.0.21->contractions) (2.0.0)
```

In [ ]:
```
import pandas as pd
import numpy as np
import nltk
nltk.download('wordnet')
nltk.download('punkt')
import re
import matplotlib.pyplot as plt
import contractions
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
```

```
/var/folders/75/80lf4_793vn5xdm8_9jjr63h0000gn/T/ipykernel_7334/2486081064.py:1: DeprecationWarning:
Pyarrow will become a required dependency of pandas in the next major release of pandas (pandas 3.0),
(to allow more performant data types, such as the Arrow string type, and better interoperability with oth
er libraries)
but was not found to be installed on your system.
If this would cause problems for you,
please provide us feedback at https://github.com/pandas-dev/pandas/issues/54466

  import pandas as pd
[nltk_data] Downloading package wordnet to
[nltk_data]     /Users/namyashah/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package punkt to /Users/namyashah/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
```

In [ ]:
```
from sklearn.model_selection import train_test_split
```

In [ ]:
```
from nltk.corpus import stopwords
```

In [ ]:
```
nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/namyashah/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

Out[ ]: True

In [ ]:
```
from nltk.stem import WordNetLemmatizer
```

In [ ]:
```
!pip install tabulate
```

```
Requirement already satisfied: tabulate in /Users/namyashah/anaconda3/lib/python3.11/site-packages (0.8.1
0)
```

In [ ]:
```
from tabulate import tabulate
```

Reading Data

In [ ]:
```
path = '/Users/namyashah/Documents/USC Schooling/NLP/HW2/data/amazon_reviews_us_Office_Products_v1_00.tsv
```

In [ ]:
```
my_df = pd.read_csv(path,  sep='\t', header=0, on_bad_lines='skip')
my_df
```

```
/var/folders/75/80lf4_793vn5xdm8_9jjr63h0000gn/T/ipykernel_7334/1113296747.py:1: DtypeWarning: Columns (
7) have mixed types. Specify dtype option on import or set low_memory=False.
  my_df = pd.read_csv(path,  sep='\t', header=0, on_bad_lines='skip')
```

Out[ ]:

	marketplace	customer_id	review_id	product_id	product_parent	product_title	product_categ
0	US	43081963	R18RVCKGH1SSI9	B001BM2MAC	307809868	Scotch Cushion Wrap 7961, 12 Inches x 100 Feet	Office Produ
1	US	10951564	R3L4L6LW1PUOFY	B00DZYEXPQ	75004341	Dust-Off Compressed Gas Duster, Pack of 4	Office Produ
						Amram	

2	US	21143145	R2J8AWXWTDX2TF	B00RTMUHDW	529689027	Tagger Standard Tag Attaching Tagging Gu...	Office Produ
3	US	52782374	R1PR37BR7G3M6A	B00D7H8XB6	868449945	AmazonBasics 12-Sheet High-Security Micro-Cut ...	Office Produ
4	US	24045652	R3BDDDZMZBZDPU	B001XCWP34	33521401	Derwent Colored Pencils, Inktense Ink Pencils,...	Office Produ
...	...	...	...	...	...	...	
2640249	US	53005790	RLI7EI10S7SN0	B00000DM9M	223408988	PalmOne III Leather Belt Clip Case	Office Produ
2640250	US	52188548	R1F3SRK9MHE6A3	B00000DM9M	223408988	PalmOne III Leather Belt Clip Case	Office Produ
2640251	US	52090046	R23V0C4NRJL8EM	0807865001	307284585	Gods and Heroes of Ancient Greece	Office Produ
2640252	US	52503173	R13ZAE1ATEUC1T	1572313188	870359649	Microsoft EXCEL 97/ Visual Basic Step-by-Step ...	Office Produ
2640253	US	52585611	RE8J5O2GY04NN	1572313188	870359649	Microsoft EXCEL 97/ Visual Basic Step-by-Step ...	Office Produ

2640254 rows × 15 columns

Making Labels

In [ ]:
```
review=my_df['review_body'].tolist()
# print(review)
```

In [ ]:
```
rating=my_df['star_rating'].tolist()
#because these values have misread data of dates instead of ratings
rating[286835]=-1
rating[671556]=-1
rating[1523317]=-1
#print(rating)
```

In [ ]:
```
ndf = pd.DataFrame({'reviews': review})
ndf['ratings'] = rating
sample = ndf.iloc[2:5]
sample.head()
```

Out[ ]:

	reviews	ratings
2	Haven't used yet, but I am sure I will like it.	5
3	Although this was labeled as "new" the...	1
4	Gorgeous colors and easy to use	4

In [ ]:
```
#making sure all values of ratings are numeric values
ndf["ratings"]=pd.to_numeric(ndf["ratings"])
```

In [ ]:
```
#identifying that not all values of reviews are non string values
for a in ndf['reviews'].map(type):
  if a != str:
    print(a)
```

```
<class 'float'>
<class 'float'>
```

```
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
```

```
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
```

```
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
```

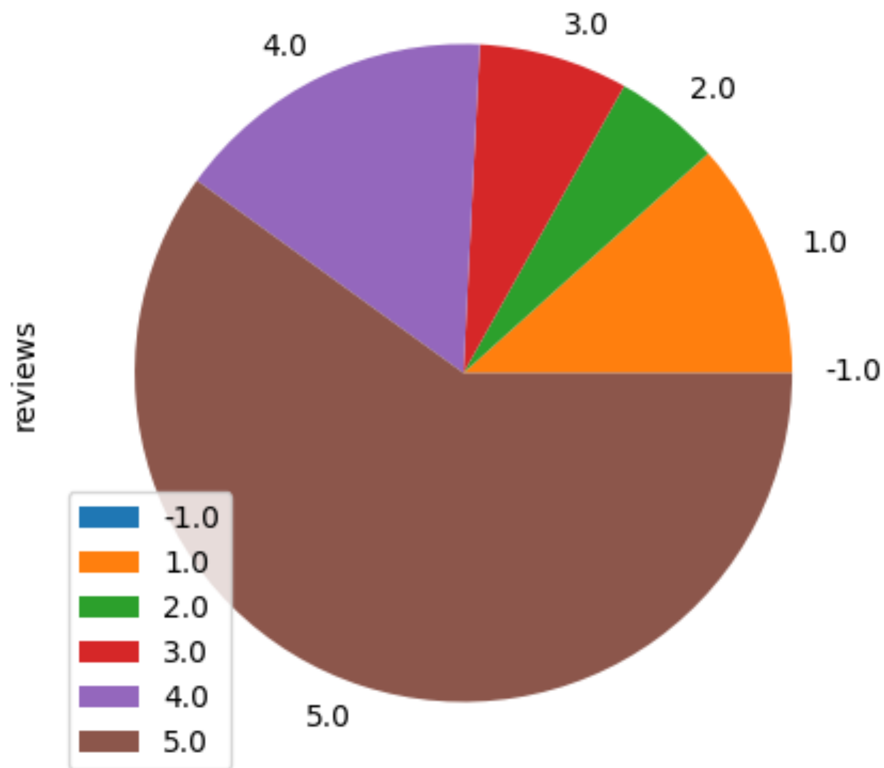
```
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
```

```
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
<class 'float'>
```

In [ ]:
```
#making sure all values of reviews are string values
ndf['reviews'] = ndf['reviews'].map(str)
```

In [ ]:
```
count = ndf.groupby(['ratings']).count()
print(count)
count.plot(kind='pie', subplots=True, figsize=(5,5))
plt.show()
```

```
         reviews
ratings
-1.0           3
 1.0      306979
 2.0      138384
 3.0      193691
 4.0      418371
 5.0     1582812
```



In [ ]:
```
def getTernaryLabel(ratings_Val):
  if ratings_Val > 3:
    return 0
  elif ratings_Val <= 2:
    return 1
  else:
    return 2

#adding new column for the binary labels
ndf['Labels'] = ndf['ratings'].apply(lambda x: getTernaryLabel(x))
ndf = ndf.drop('ratings', axis=1)
ndf
```

Out[ ]:

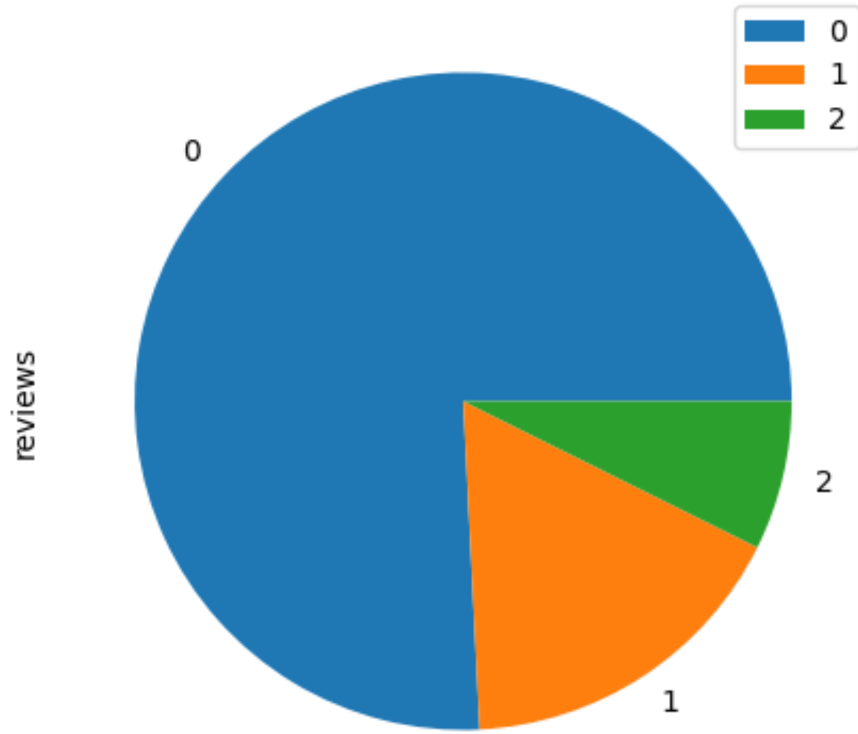
	reviews	Labels
0	Great product.	0
1	What's to say about this commodity item except...	0
2	Haven't used yet, but I am sure I will like it.	0
3	Although this was labeled as "new" the...	1
4	Gorgeous colors and easy to use	0
...	...	...
2640249	I can't live anymore whithout my Palm III. But...	0
2640250	Although the Palm Pilot is thin and compact it...	0
2640251	This book had a lot of great content without b...	0
2640252	I am teaching a course in Excel and am using t...	0
2640253	A very comprehensive layout of exactly how Vis...	0

2640254 rows × 2 columns

In [ ]:

```
count2 = ndf.groupby(['Labels']).count()
print(count2)
count2.plot(kind='pie', subplots=True, figsize=(5,5))
plt.show()
```

```
        reviews
Labels
0       2001183
1        445366
2        193705
```




```
In [ ]: ndf.shape
Out[ ]: (2640254, 2)
```

Making Samples of 50K and joining them to Binary and Ternary Dataset

```
In [ ]: #selecting all positive reviews
        pos_rec = ndf.loc[ndf['Labels'] == 0]
        print(pos_rec.shape)

        #selecting 100,000 of those positive reviews at random
        pos_rec = pos_rec.sample(n=50000)
        print(pos_rec.shape)
```

```
(2001183, 2)
(50000, 2)
```

In [ ]:
```
#selecting all negative reviews
neg_rec = ndf.loc[ndf['Labels'] == 1]
print(neg_rec.shape)

#selecting 100,000 of those negative reviews at random
neg_rec = neg_rec.sample(n=50000)
print(neg_rec.shape)
```

```
(445366, 2)
(50000, 2)
```

In [ ]:
```
#selecting all neutral reviews
neu_rec = ndf.loc[ndf['Labels'] == 2]
print(neu_rec.shape)

#selecting 100,000 of those neutral reviews at random
neu_rec = neu_rec.sample(n=50000)
print(neu_rec.shape)
```

```
(193705, 2)
(50000, 2)
```

Here I am combining the positive review data and negative review data into Binary Dataframe

In [ ]:
```
framesbin = [pos_rec, neg_rec]
binframes = pd.concat(framesbin)
binframes
```

Out[ ]:

	reviews	Labels
1512915	I like these Panasonic cordless phones. They h...	0
1851035	The three batteries arrived well packed and ea...	0
1420245	My kids use these all the time in their Art Cl...	0
1839567	I only replace/ install these as they run out ...	0
145903	Exactly what I needed for my music books.	0
...	...	...
2398822	I fear that I must agree fully with another re...	1
326453	They do not distribute the color evenly.Have t...	1
1795919	VERY POOR WIFI RECEPTER, CONTACTED CANON AND W...	1
1851482	I called Olympus at 800-622-6372 and was told ...	1
209273	This is the worst printer I have ever gotten. ...	1

100000 rows × 2 columns

Here I am combining the Positive, Negative and Neutral data into a Ternary Dataframe

In [ ]:

```
framester = [pos_rec, neg_rec, neu_rec]
terframes = pd.concat(framester)
terframes
```

Out[ ]:

	reviews	Labels
1512915	I like these Panasonic cordless phones. They h...	0
1851035	The three batteries arrived well packed and ea...	0
1420245	My kids use these all the time in their Art Cl...	0
1839567	I only replace/ install these as they run out ...	0
145903	Exactly what I needed for my music books.	0
...	...	...
2042827	My students were easily bored with the game. ...	2
1661195	I really wanted to like this, but the folder p...	2
71154	Bought it for a 50th anniversary slide show mo...	2
1199084	Product as described, but I am not convinced t...	2
1305877	Lasted 4 months and I rarely print things. Se...	2

150000 rows × 2 columns

Here we shuffle the dataframes

In [ ]:

```
binframes = binframes.sample(frac=1)
print(binframes.shape)
binframes
```

```
(100000, 2)
```

Out[ ]:

	reviews	Labels
1921057	This item came in a home made bubble wrap labe...	1
2032152	It works. I hope it continues working. Perha...	0
2590977	The software that comes with the printer does ...	1
1685277	It arrived on time and it came in a huge box. ...	0
29865	Works as well as Canon toner	0
...	...	...
1234437	The signage faded completely away after only a...	1
203972	Product was listed as usable on the MFC870DW. ...	1
2283458	I cleaned the heads 10 times and there are sti...	1
483295	The highlighters were not new or they were old...	1
556562	excellent, got one for my daughter. Perfect fo...	0

100000 rows × 2 columns

In [ ]:

```
terframes = terframes.sample(frac=1)
print(terframes.shape)
terframes
```

```
(150000, 2)
```

Out[ ]:

	reviews	Labels
1163606	Excellent pen! Actually this is the only pen ...	0
2069284	When you have a fresh set of batteries in this...	2
338411	These file folder labels are great. I use them...	0
37088	work's well ~ is Not the Heavy Duty All Steel...	2
752649	great quality, great value!	0
...	...	...
2482466	I purchased this pen specifically for outdoor ...	1
1504746	We always use Scotch Heavy Duty Tape for packi...	2
2599944	I'm on a quest to find a cordless phone with d...	2
1751783	Just received my new printer and love the size...	2
478502	Best phone bought to date	0

150000 rows × 2 columns

Data Cleaning and Pre-Processing

Data Cleaning

In [ ]:
```
avg_before_dc_bin = np.mean(binframes['reviews'].apply(lambda x: len(x.split())))
```

In [ ]:
```
avg_before_dc_ter = np.mean(terframes['reviews'].apply(lambda x: len(x.split())))
```

In [ ]:
```
#converting to lower case
binframes['reviews'] = binframes['reviews'].str.lower()
```

In [ ]:
```
#converting to lower case
```

```
terframes['reviews'] = terframes['reviews'].str.lower()
```

In [ ]:

```
#removing html

def clean_html(review_sent):
  review_sent = re.sub(r"<.*?>+","", review_sent)
  return " ".join(review_sent.split())
```

In [ ]:

```
#removing url

def clean_url(review_sent):
  review_sent = re.sub(r"http[^\s]+","", review_sent)
  return " ".join(review_sent.split())
```

In [ ]:

```
#fix contractions inspired from geeksforgeeks tutorial

def clean_contractions(review_sent):
  expanded_words = []
  for word in review_sent.split():
    expanded_words.append(contractions.fix(word))
  return " ".join(expanded_words)
```

In [ ]:

```
#fix mentions

def clean_mentions(tweet):
    tweet = re.sub(r"@[A-Za-z0-9_]+","", tweet)
    return " ".join(tweet.split())
```

In [ ]:

```
#removing non-alphabetic characters, numbers and extra spaces

binframes['reviews'] = binframes['reviews'].str.replace('\d+','')
terframes['reviews'] = terframes['reviews'].str.replace('\d+','')

non_alphabetic_chars = ['\\n','!','"','(',')','+',',','-','.','/',':',';','<','=','>','?','[','\\',']','

def preprocess_reviews(review_vals, non_alphabetic_chars):
    processed_review = review_vals
    processed_review = clean_html(processed_review)
    processed_review = clean_url(processed_review)
```

```
    processed_review = clean_contractions(processed_review)
    processed_review = clean_mentions(processed_review)
    for char_wd in non_alphabetic_chars:
      processed_review = processed_review.replace(char_wd,'')
    processed_review = processed_review + " "
    return(" ".join(processed_review.split()))
```

In [ ]:

```
binframes['reviews'] = binframes['reviews'].apply(lambda x: preprocess_reviews(x, non_alphabetic_chars))
binframes
```

Out[ ]:

	reviews	Labels
1921057	this item came in a home made bubble wrap labe...	1
2032152	it works i hope it continues working perhaps i...	0
2590977	the software that comes with the printer does ...	1
1685277	it arrived on time and it came in a huge box i...	0
29865	works as well as canon toner	0
...	...	...
1234437	the signage faded completely away after only a...	1
203972	product was listed as usable on the mfc870dw i...	1
2283458	i cleaned the heads 10 times and there are sti...	1
483295	the highlighters were not new or they were old...	1
556562	excellent got one for my daughter perfect for ...	0

100000 rows × 2 columns

In [ ]:

```
terframes['reviews'] = terframes['reviews'].apply(lambda x: preprocess_reviews(x, non_alphabetic_chars))
terframes
```

Out[ ]:

	reviews	Labels
1163606	excellent pen actually this is the only pen i ...	0
2069284	when you have a fresh set of batteries in this...	2
338411	these file folder labels are great i use them ...	0
37088	work's well is not the heavy duty all steel or...	2
752649	great quality great value	0
...	...	...
2482466	i purchased this pen specifically for outdoor ...	1
1504746	we always use scotch heavy duty tape for packi...	2
2599944	i am on a quest to find a cordless phone with ...	2
1751783	just received my new printer and love the size...	2
478502	best phone bought to date	0

150000 rows × 2 columns

In [ ]:
```
avg_after_dc_bin = np.mean(binframes['reviews'].apply(lambda x: len(x.split())))
```

In [ ]:
```
avg_after_dc_ter = np.mean(terframes['reviews'].apply(lambda x: len(x.split())))
```

In [ ]:
```
print('Average length of Reviews before and after Data Cleaning for Binary Data:',avg_before_dc_bin,',',a
print('Average length of Reviews before and after Data Cleaning for Ternary Data:',avg_before_dc_ter,',',
```

```
Average length of Reviews before and after Data Cleaning for Binary Data: 58.63282 , 58.67989
Average length of Reviews before and after Data Cleaning for Ternary Data: 61.25425333333333 , 61.32564
```

Data Preprocessing

Removing Stop Words

In [ ]:
```python
avg_before_pp_bin = np.mean(binframes['reviews'].apply(lambda x: len(x.split())))
```

In [ ]:
```python
avg_before_pp_ter = np.mean(terframes['reviews'].apply(lambda x: len(x.split())))
```

In [ ]:
```python
stops = set(stopwords.words('english'))
print(stops)
```

```
{'you', 'needn', "haven't", 'where', "wasn't", 'some', "don't", 'until', 'were', 'here', 'wouldn', 'm', '
below', 'aren', 'most', 'just', "you're", 'him', 'no', 'more', "doesn't", "mustn't", 'between', 's', 'her
s', 'wasn', "that'll", 'other', 'its', 'above', 'them', 'such', 'off', 'so', 'after', 'who', 'those', 'fr
om', 'being', 'her', 'as', 'it', 'for', 'any', 'over', 'that', 'did', 'your', 'by', 'further', 'too', 'h
e', 'have', 'against', 'or', 'through', 'i', 'ain', 'ours', 'my', 'how', 'am', 'of', 'an', 'mightn', 'und
er', 'll', "didn't", 'having', 'ourselves', 'can', 'than', 'y', 'doesn', 'into', 'when', 'hadn', "hadn'
t", 'been', 'don', "should've", 'with', 'then', 'same', 'are', 'couldn', 'they', "couldn't", 'should', 'm
ustn', 'doing', 'themselves', 'does', 'weren', 'do', 'own', 'but', 'all', 'ma', 'yours', 'haven', 'agai
n', "mightn't", 'up', 'on', 'once', 'was', 'in', "it's", "hasn't", 'while', "shouldn't", 'shan', 'if', 't
here', 'won', 'few', "weren't", "you'll", 'to', 'hasn', 'now', "needn't", 've', 'both', 'his', "won't", '
herself', "wouldn't", 'itself', 'is', 'not', 'because', 'at', 'shouldn', 'she', 'their', 'before', 'me',
'these', 't', 'didn', 'out', "shan't", "isn't", 'o', 'down', 'what', "she's", 're', 'why', "aren't", 'ver
y', 'each', 'only', 'myself', 'yourself', 'which', 'this', 'we', 'whom', "you'd", 'd', 'has', 'about', "y
ou've", 'himself', 'the', 'had', 'during', 'our', 'and', 'a', 'will', 'theirs', 'yourselves', 'be', 'is
n', 'nor'}
```

In [ ]:
```python
#remove stopwords

def clean_stopwords(review_sent):
  filtered_sentence = []
  for w in review_sent.split():
    if w not in stops:
        filtered_sentence.append(w)
  return " ".join(filtered_sentence)
```

In [ ]:
```python
binframes['reviews'] = binframes['reviews'].apply(lambda x: clean_stopwords(x))
binframes
```

Out[ ]:

	reviews	Labels
1921057	item came home made bubble wrap labeled new ou...	1
2032152	works hope continues working perhaps get disco...	0
2590977	software comes printer support newer versions ...	1
1685277	arrived time came huge box expecting big pictu...	0
29865	works well canon toner	0
...	...	...
1234437	signage faded completely away couple weeks	1
203972	product listed usable mfc870dw bought time bou...	1
2283458	cleaned heads 10 times still gaps printing goi...	1
483295	highlighters new old dried good buy	1
556562	excellent got one daughter perfect pill bottles	0

100000 rows × 2 columns

In [ ]:

```
terframes['reviews'] = terframes['reviews'].apply(lambda x: clean_stopwords(x))
terframes
```

Out[ ]:

	reviews	Labels
1163606	excellent pen actually pen like use write anyt...	0
2069284	fresh set batteries nice strong beam unfortuna...	2
338411	file folder labels great use folders	0
37088	work's well heavy duty steel original stapler ...	2
752649	great quality great value	0
...	...	...
2482466	purchased pen specifically outdoor use package...	1
1504746	always use scotch heavy duty tape packing supe...	2
2599944	quest find cordless phone decent features exce...	2
1751783	received new printer love size functions probl...	2
478502	best phone bought date	0

150000 rows × 2 columns

Perform Lemmatization

In [ ]:

```
def lets_lemmatize(review_sent):
  lemmatizer = WordNetLemmatizer()
  lemmatized_sentence = []
  for word in nltk.word_tokenize(review_sent):
    word =lemmatizer.lemmatize(word)
    lemmatized_sentence.append(word)
  return " ".join(lemmatized_sentence)
```

In [ ]:

```
binframes['reviews'] = binframes['reviews'].apply(lambda x: lets_lemmatize(x))
binframes
```

Out[ ]:

	reviews	Labels
1921057	item came home made bubble wrap labeled new ou...	1
2032152	work hope continues working perhaps get discou...	0
2590977	software come printer support newer version ma...	1
1685277	arrived time came huge box expecting big pictu...	0
29865	work well canon toner	0
...	...	...
1234437	signage faded completely away couple week	1
203972	product listed usable mfc870dw bought time bou...	1
2283458	cleaned head 10 time still gap printing going ...	1
483295	highlighter new old dried good buy	1
556562	excellent got one daughter perfect pill bottle	0

100000 rows × 2 columns

In [ ]:

```
terframes['reviews'] = terframes['reviews'].apply(lambda x: lets_lemmatize(x))
terframes
```

Out[ ]:

	reviews	Labels
1163606	excellent pen actually pen like use write anyt...	0
2069284	fresh set battery nice strong beam unfortunate...	2
338411	file folder label great use folder	0
37088	work 's well heavy duty steel original stapler...	2
752649	great quality great value	0
...	...	...
2482466	purchased pen specifically outdoor use package...	1
1504746	always use scotch heavy duty tape packing supe...	2
2599944	quest find cordless phone decent feature excel...	2
1751783	received new printer love size function proble...	2
478502	best phone bought date	0

150000 rows × 2 columns

In [ ]:

```
avg_after_pp_bin = np.mean(binframes['reviews'].apply(lambda x: len(x.split())))
avg_after_pp_ter = np.mean(terframes['reviews'].apply(lambda x: len(x.split())))
print('Average length of Reviews before and after Pre-Processing:',avg_before_pp_bin,',',avg_after_pp_bi
print('Average length of Reviews before and after Pre-Processing:',avg_before_pp_ter,',',avg_after_pp_te
```

```
Average length of Reviews before and after Pre-Processing: 58.67989 , 29.20605
Average length of Reviews before and after Pre-Processing: 61.32564 , 30.324166666666667
```

Word Embedding

Word2Vec using Pre-trained Model

In [ ]:

```
import tempfile
```

```
from gensim.models import KeyedVectors
```

```
/Users/namyashah/Library/Python/3.9/lib/python/site-packages/urllib3/__init__.py:35: NotOpenSSLWarning: u
rllib3 v2 only supports OpenSSL 1.1.1+, currently the 'ssl' module is compiled with 'LibreSSL 2.8.3'. Se
e: https://github.com/urllib3/urllib3/issues/3020
  warnings.warn(
```

In [ ]:
```
#importing gensim and pre-trained word2vec model
import gensim.downloader as api
wv = api.load('word2vec-google-news-300')
```

In [ ]:
```
import tempfile

with tempfile.NamedTemporaryFile(prefix='gensim-model-', delete=False) as tmp:
    temporary_filepath = tmp.name
    wv.save(temporary_filepath)
    #
    # The model is now safely stored in the filepath.
    # You can copy it to other machines, share it with others, etc.
    #
    # To load a saved model:
    #
    new_model = KeyedVectors.load(temporary_filepath)
```

In [ ]:
```
from nltk.tokenize import sent_tokenize, word_tokenize
```

In [ ]:
```
import time
```

In [ ]:
```
#tokenizing the reviews into words
binframes['MyReviews'] = [word_tokenize(t) for t in binframes['reviews']]
binframes.head(5)
```

Out[ ]:

	reviews	Labels	MyReviews
1921057	item came home made bubble wrap labeled new ou...	1	[item, came, home, made, bubble, wrap, labeled...
2032152	work hope continues working perhaps get discou...	0	[work, hope, continues, working, perhaps, get,...
2590977	software come printer support newer version ma...	1	[software, come, printer, support, newer, vers...
1685277	arrived time came huge box expecting big pictu...	0	[arrived, time, came, huge, box, expecting, bi...
29865	work well canon toner	0	[work, well, canon, toner]

In [ ]:

```
#Extracting Word Embeddings from the Pre-trained Model

def embeddingFun(sent):
    vectorsize = wv.vector_size
    PT_Embeddings = np.zeros(vectorsize)
    c=1

    for word in sent:
        if word in wv:
            c+=1
            PT_Embeddings+=wv[word]
    avg = PT_Embeddings/c
    return avg

binframes['gvectors']=binframes['MyReviews'].apply(embeddingFun)
```

In [ ]:

```
#Example 1 to check semantic similarities of the generated vectors
print(wv.most_similar(positive=['woman', 'king'], negative=['man'], topn=1))
```

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

In [ ]:

```
#Example 2 to check semantic similarities of the generated vectors
print('The similarity score between excellent and outstanding is:', wv.similarity('excellent','outstandi
```

```
The similarity score between excellent and outstanding is: 0.5567486
```

Custom Word2Vec Model

```
In [ ]: from gensim.test.utils import common_texts
        from gensim.models import Word2Vec
        from gensim.models import KeyedVectors
        import multiprocessing
```

```
In [ ]: cores = multiprocessing.cpu_count()
```

```
In [ ]: #making and saving our custom Word2Vec Model
        start = time.time()
        model = Word2Vec(sentences=binframes['MyReviews'], vector_size=300, window=11, min_count=10, workers=core
        end = round(time.time()-start,2)
        print("This process took",end,"seconds.")
```

```
This process took 85.48 seconds.
```

```
In [ ]: #corpus_iterable = MyReviews
```

```
In [ ]: #model.build_vocab(corpus_iterable=corpus_iterable, keep_raw_vocab=False)
```

```
In [ ]: model.save("word2vec.model")
```

```
In [ ]: model = Word2Vec.load("word2vec.model")
```

```
In [ ]: #model.train(corpus_iterable=corpus_iterable, total_examples=model.corpus_count, epochs=25)
```

```
In [ ]: #model.epochs
```

```
In [ ]: # Store just the words + their trained embeddings.
        word_vectors = model.wv
        word_vectors.save("word2vec.wordvectors")
```

```
In [ ]: # Load back with memory-mapping = read-only, shared across processes.
        wv2 = KeyedVectors.load("word2vec.wordvectors", mmap='r')
```

```
In [ ]: #Extracting Word Embeddings from the Custom trained Model

        def embeddingFun2(sent):
```

```
    vectorsize = wv2.vector_size
    CM_Embeddings = np.zeros(vectorsize)
    c=1

    for word in sent:
        if word in wv2:
            c+=1
            PT_Embeddings+=wv[word]
    avg = CM_Embeddings/c
    return avg

binframes['cvectors']=binframes['MyReviews'].apply(embeddingFun)
```

In [ ]:
```
#Example 1 to check semantic similarities of the generated vectors
print(model.wv.most_similar(positive=['woman', 'king'], negative=['man'], topn=5))
```

```
[('romney', 0.3914776146411896), ('men', 0.381064236164093), ('mitt', 0.3443523347377777), ('darling', 0.
3400265872478485), ('manly', 0.33894944190979004)]
```

In [ ]:
```
#Example 2 to check semantic similarities of the generated vectors
print('The similarity score between excellent and outstanding is:', model.wv.similarity('excellent','outs
```

```
The similarity score between excellent and outstanding is: 0.52947176
```

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

A. The vectors generated by the pre-trained model gives more accurate similar words and has a better similarity score compare to the custom model I created. This can mean that pre-trained model has a huge training set and hence it might have more context to say that King and Queen are semantically similar with the only difference of Man and Woman, while our pre-trained model is not as rich to understand that context. This also means that the word embeddings in the custom model is centric to the context of data we provied to the model.

Which of the Word2Vec models seems to encode semantic similarities between words better?

A. The Pre-trained Gensim Word2Vec Model seems to encode semantic similarities between words better than the Custom model I created.

Running Models for Pre-trained Word2Vec, Custom Word2Vec and TF-IDF Feature Extraction

In [ ]:
```
from sklearn.linear_model import Perceptron
from sklearn.svm import LinearSVC
from sklearn.model_selection import train_test_split
```

In [ ]:
```
Accuracy_Table = [['TFIDF','Percepton','Binary'],['TFIDF','SVM','Binary'],['Avg Pre-trained W2V','Percept
```

Running Models for TF-IDF Feature Extraxtion

In [ ]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
```

In [ ]:
```
tf_idf_feature_extracter = TfidfVectorizer()
Rev_tfidf = tf_idf_feature_extracter.fit_transform(binframes['reviews'])
#print(Rev_tfidf)
```

In [ ]:
```
# split the data into 80-20 train-test
x_train, x_test, y_train, y_test = train_test_split(Rev_tfidf, binframes['Labels'], test_size=0.2, randor
```

Percepton Model

In [ ]:
```
perceptron_model_tfidf = Perceptron()
perceptron_model_tfidf.fit(x_train, y_train)
```

Out[ ]:
```
▾ Perceptron

Perceptron()
```

In [ ]:
```
y_pred_perceptron_test = perceptron_model_tfidf.predict(x_test)
```

In [ ]:
```
accuracy_test_perceptron = accuracy_score(y_test, y_pred_perceptron_test)
print('Perceptron Test Metrics: Accuracy = ', accuracy_test_perceptron)
```

```
Perceptron Test Metrics: Accuracy =  0.84775
```

```
In [ ]: Accuracy_Table[0].append(accuracy_test_perceptron)
```

SVM Model

```
In [ ]: svm_model_tfidf = LinearSVC()
        svm_model_tfidf.fit(x_train, y_train)
```

```
/Users/namyashah/Library/Python/3.9/lib/python/site-packages/sklearn/svm/_classes.py:31: FutureWarning: T
he default value of `dual` will change from `True` to `'auto'` in 1.5. Set the value of `dual` explicitly
to suppress the warning.
  warnings.warn(
```

```
Out[ ]: ▾ LinearSVC
        LinearSVC()
```

```
In [ ]: y_pred_svm_test = svm_model_tfidf.predict(x_test)
```

```
In [ ]: accuracy_test_svm = accuracy_score(y_test, y_pred_svm_test)
        print('SVM Test Metrics: Accuracy = ', accuracy_test_svm)
```

```
SVM Test Metrics: Accuracy =  0.88845
```

```
In [ ]: Accuracy_Table[1].append(accuracy_test_svm)
```

Running Models for Pre-trained Word2Vec Feature Extraction

```
In [ ]: # split the data into 80-20 train-test
        x_train, x_test, y_train, y_test = train_test_split(binframes['gvectors'], binframes['Labels'], test_size
```

Percepton Model

```
In [ ]: perceptron_model = Perceptron()
```

```
perceptron_model.fit(x_train.to_list(), y_train.to_list())
```

Out[ ]:

```
▾ Perceptron
Perceptron()
```

In [ ]:

```
y_pred_perceptron_test = perceptron_model.predict(x_test.to_list())
```

In [ ]:

```
accuracy_test_perceptron = accuracy_score(y_test, y_pred_perceptron_test)
print('Perceptron Test Metrics: Accuracy = ', accuracy_test_perceptron)
```

```
Perceptron Test Metrics: Accuracy =  0.7971
```

In [ ]:

```
Accuracy_Table[2].append(accuracy_test_perceptron)
```

SVM Model

In [ ]:

```
svm_model_tfidf = LinearSVC()
svm_model_tfidf.fit(x_train.to_list(), y_train.to_list())
```

```
/Users/namyashah/Library/Python/3.9/lib/python/site-packages/sklearn/svm/_classes.py:31: FutureWarning: T
he default value of `dual` will change from `True` to `'auto'` in 1.5. Set the value of `dual` explicitly
to suppress the warning.
  warnings.warn(
```

Out[ ]:

```
▾ LinearSVC
LinearSVC()
```

In [ ]:

```
y_pred_svm_test = svm_model_tfidf.predict(x_test.to_list())
```

In [ ]:

```
accuracy_test_svm = accuracy_score(y_test, y_pred_svm_test)
print('SVM Test Metrics: Accuracy = ', accuracy_test_svm)
```

```
SVM Test Metrics: Accuracy =  0.8487
```

In [ ]:

```
Accuracy_Table[3].append(accuracy_test_svm)
```

Running Models for Custom Word2Vec Feature Extraction

In [ ]:
```
# split the data into 80-20 train-test
x_train, x_test, y_train, y_test = train_test_split(binframes['cvectors'], binframes['Labels'], test_size
```

Percepton Model

In [ ]:
```
perceptron_model = Perceptron()
perceptron_model.fit(x_train.to_list(), y_train.to_list())
```

Out[ ]:
```
Perceptron
Perceptron()
```

In [ ]:
```
y_pred_perceptron_test = perceptron_model.predict(x_test.to_list())
```

In [ ]:
```
accuracy_test_perceptron = accuracy_score(y_test, y_pred_perceptron_test)
print('Perceptron Train Metrics: Accuracy = ', accuracy_test_perceptron)
```

```
Perceptron Train Metrics: Accuracy =  0.7971
```

In [ ]:
```
Accuracy_Table[4].append(accuracy_test_perceptron)
```

SVM Model

In [ ]:
```
svm_model_tfidf = LinearSVC()
svm_model_tfidf.fit(x_train.to_list(), y_train.to_list())
```

```
/Users/namyashah/Library/Python/3.9/lib/python/site-packages/sklearn/svm/_classes.py:31: FutureWarning: T
he default value of `dual` will change from `True` to `'auto'` in 1.5. Set the value of `dual` explicitly
to suppress the warning.
  warnings.warn(
```

Out[ ]:
```
▾ LinearSVC
LinearSVC()
```

In [ ]:
```
y_pred_svm_test = svm_model_tfidf.predict(x_test.to_list())
```

In [ ]:
```
accuracy_test_svm = accuracy_score(y_test, y_pred_svm_test)
print('SVM Test Metrics: Accuracy = ', accuracy_test_svm)
```

```
SVM Test Metrics: Accuracy =  0.8487
```

In [ ]:
```
Accuracy_Table[5].append(accuracy_test_svm)
```

What do you conclude from comparing performances for the models trained using the three different feature types (TF-IDF, pretrained Word2Vec, your trained Word2Vec)?

A. From the results, it is apparent that the TF-IDF feature extraction gives better accuracy score compares to the Pre-trained Gensim Word2vec and my custom Word2Vec feature extracter. This could mean that TF-IDF features fit better with the SVM and Percepton model in comparison to the features from the other two models. This could also mean that TF-IDF is better at extracting features from this specific reviews dataset in comparison to the other two. It is also observed that SVM model gives a better accuracy score than the Percepton model, which can suggest that the SVM model does the job of classification for this specific reviews dataset better than the Percepton model. All this being said, one thing to be kept in mind is that these performances can have varied answers if we change parameters such as sample size of the data set, hyperparameter in the custom Word2Vec model or change the dataset in itself.

Feedforward Neural Networks

In [ ]:
```
#!pip install tensorflow
```

In [ ]:
```
import numpy as np
import tensorflow as tf
```

FNN for Binary Classification

Using Features extracted from Pre-trained Avg Word2Vec Model

Here the Feedforward Neural Network is used for the Binary Classification, where the feature extracter is the Pre-trained Gensim Word2Vec model and the vector for a review are the average of the vectors for each word in the review.

In [ ]:
```
# split the data into 80-20 train-test
x_train, x_test, y_train, y_test = train_test_split(binframes['gvectors'], binframes['Labels'], test_size
```

In [ ]:
```
# Converting NumPy arrays to TensorFlow tensors
X_train_tf = tf.convert_to_tensor(x_train.to_list(), dtype=tf.float32)
y_train_tf = tf.convert_to_tensor(y_train, dtype=tf.int32)
X_test_tf = tf.convert_to_tensor(x_test.to_list(), dtype=tf.float32)
y_test_tf = tf.convert_to_tensor(y_test, dtype=tf.int32)
```

In [ ]:
```
neural_network_1 = tf.keras.models.Sequential([
    tf.keras.layers.Dense(50, activation='relu', input_shape=(300,)),
    tf.keras.layers.Dense(10, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

In [ ]:
```
# Compiling the neural network
neural_network_1.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

In [ ]:
```
# Training the neural network
history1 = neural_network_1.fit(X_train_tf, y_train_tf, epochs=100, batch_size=32)
```

```
Epoch 1/100
2500/2500 [==============================] - 1s 432us/step - loss: 0.3755 - accuracy: 0.8339
Epoch 2/100
2500/2500 [==============================] - 1s 423us/step - loss: 0.3382 - accuracy: 0.8534
Epoch 3/100
2500/2500 [==============================] - 1s 457us/step - loss: 0.3253 - accuracy: 0.8598
Epoch 4/100
```

```
2500/2500 [==============================] - 1s 439us/step - loss: 0.3162 - accuracy: 0.8642
Epoch 5/100
2500/2500 [==============================] - 1s 452us/step - loss: 0.3078 - accuracy: 0.8687
Epoch 6/100
2500/2500 [==============================] - 1s 417us/step - loss: 0.3005 - accuracy: 0.8723
Epoch 7/100
2500/2500 [==============================] - 1s 411us/step - loss: 0.2947 - accuracy: 0.8751
Epoch 8/100
2500/2500 [==============================] - 1s 409us/step - loss: 0.2884 - accuracy: 0.8778
Epoch 9/100
2500/2500 [==============================] - 1s 409us/step - loss: 0.2835 - accuracy: 0.8800
Epoch 10/100
2500/2500 [==============================] - 1s 410us/step - loss: 0.2783 - accuracy: 0.8823
Epoch 11/100
2500/2500 [==============================] - 1s 410us/step - loss: 0.2739 - accuracy: 0.8848
Epoch 12/100
2500/2500 [==============================] - 1s 411us/step - loss: 0.2691 - accuracy: 0.8873
Epoch 13/100
2500/2500 [==============================] - 1s 416us/step - loss: 0.2656 - accuracy: 0.8884
Epoch 14/100
2500/2500 [==============================] - 1s 439us/step - loss: 0.2617 - accuracy: 0.8916
Epoch 15/100
2500/2500 [==============================] - 1s 423us/step - loss: 0.2584 - accuracy: 0.8926
Epoch 16/100
2500/2500 [==============================] - 1s 447us/step - loss: 0.2545 - accuracy: 0.8943
Epoch 17/100
2500/2500 [==============================] - 1s 447us/step - loss: 0.2509 - accuracy: 0.8964
Epoch 18/100
2500/2500 [==============================] - 1s 434us/step - loss: 0.2489 - accuracy: 0.8975
Epoch 19/100
2500/2500 [==============================] - 1s 426us/step - loss: 0.2453 - accuracy: 0.8987
Epoch 20/100
2500/2500 [==============================] - 1s 433us/step - loss: 0.2425 - accuracy: 0.9004
Epoch 21/100
2500/2500 [==============================] - 1s 415us/step - loss: 0.2402 - accuracy: 0.9005
Epoch 22/100
2500/2500 [==============================] - 1s 413us/step - loss: 0.2371 - accuracy: 0.9025
Epoch 23/100
2500/2500 [==============================] - 1s 411us/step - loss: 0.2336 - accuracy: 0.9046
Epoch 24/100
```

```
2500/2500 [==============================] - 1s 409us/step - loss: 0.2320 - accuracy: 0.9048
Epoch 25/100
2500/2500 [==============================] - 1s 414us/step - loss: 0.2294 - accuracy: 0.9066
Epoch 26/100
2500/2500 [==============================] - 1s 408us/step - loss: 0.2272 - accuracy: 0.9073
Epoch 27/100
2500/2500 [==============================] - 1s 407us/step - loss: 0.2244 - accuracy: 0.9076
Epoch 28/100
2500/2500 [==============================] - 1s 406us/step - loss: 0.2227 - accuracy: 0.9093
Epoch 29/100
2500/2500 [==============================] - 1s 436us/step - loss: 0.2203 - accuracy: 0.9105
Epoch 30/100
2500/2500 [==============================] - 1s 407us/step - loss: 0.2177 - accuracy: 0.9109
Epoch 31/100
2500/2500 [==============================] - 1s 407us/step - loss: 0.2162 - accuracy: 0.9127
Epoch 32/100
2500/2500 [==============================] - 1s 406us/step - loss: 0.2147 - accuracy: 0.9128
Epoch 33/100
2500/2500 [==============================] - 1s 407us/step - loss: 0.2126 - accuracy: 0.9134
Epoch 34/100
2500/2500 [==============================] - 1s 408us/step - loss: 0.2105 - accuracy: 0.9153
Epoch 35/100
2500/2500 [==============================] - 1s 407us/step - loss: 0.2088 - accuracy: 0.9144
Epoch 36/100
2500/2500 [==============================] - 1s 407us/step - loss: 0.2070 - accuracy: 0.9162
Epoch 37/100
2500/2500 [==============================] - 1s 433us/step - loss: 0.2051 - accuracy: 0.9166
Epoch 38/100
2500/2500 [==============================] - 1s 432us/step - loss: 0.2029 - accuracy: 0.9175
Epoch 39/100
2500/2500 [==============================] - 1s 407us/step - loss: 0.2022 - accuracy: 0.9176
Epoch 40/100
2500/2500 [==============================] - 1s 407us/step - loss: 0.1999 - accuracy: 0.9195
Epoch 41/100
2500/2500 [==============================] - 1s 408us/step - loss: 0.1984 - accuracy: 0.9195
Epoch 42/100
2500/2500 [==============================] - 1s 407us/step - loss: 0.1973 - accuracy: 0.9204
Epoch 43/100
2500/2500 [==============================] - 1s 409us/step - loss: 0.1947 - accuracy: 0.9208
Epoch 44/100
```

```
2500/2500 [==============================] - 1s 407us/step - loss: 0.1940 - accuracy: 0.9222
Epoch 45/100
2500/2500 [==============================] - 1s 432us/step - loss: 0.1928 - accuracy: 0.9216
Epoch 46/100
2500/2500 [==============================] - 1s 407us/step - loss: 0.1909 - accuracy: 0.9233
Epoch 47/100
2500/2500 [==============================] - 1s 407us/step - loss: 0.1904 - accuracy: 0.9224
Epoch 48/100
2500/2500 [==============================] - 1s 408us/step - loss: 0.1875 - accuracy: 0.9244
Epoch 49/100
2500/2500 [==============================] - 1s 408us/step - loss: 0.1862 - accuracy: 0.9256
Epoch 50/100
2500/2500 [==============================] - 1s 406us/step - loss: 0.1855 - accuracy: 0.9255
Epoch 51/100
2500/2500 [==============================] - 1s 405us/step - loss: 0.1832 - accuracy: 0.9261
Epoch 52/100
2500/2500 [==============================] - 1s 436us/step - loss: 0.1829 - accuracy: 0.9259
Epoch 53/100
2500/2500 [==============================] - 1s 411us/step - loss: 0.1812 - accuracy: 0.9270
Epoch 54/100
2500/2500 [==============================] - 1s 407us/step - loss: 0.1803 - accuracy: 0.9274
Epoch 55/100
2500/2500 [==============================] - 1s 413us/step - loss: 0.1785 - accuracy: 0.9280
Epoch 56/100
2500/2500 [==============================] - 1s 429us/step - loss: 0.1782 - accuracy: 0.9285
Epoch 57/100
2500/2500 [==============================] - 1s 450us/step - loss: 0.1761 - accuracy: 0.9295
Epoch 58/100
2500/2500 [==============================] - 1s 419us/step - loss: 0.1755 - accuracy: 0.9291
Epoch 59/100
2500/2500 [==============================] - 1s 426us/step - loss: 0.1742 - accuracy: 0.9303
Epoch 60/100
2500/2500 [==============================] - 1s 455us/step - loss: 0.1730 - accuracy: 0.9311
Epoch 61/100
2500/2500 [==============================] - 1s 456us/step - loss: 0.1728 - accuracy: 0.9307
Epoch 62/100
2500/2500 [==============================] - 1s 412us/step - loss: 0.1701 - accuracy: 0.9316
Epoch 63/100
2500/2500 [==============================] - 1s 409us/step - loss: 0.1704 - accuracy: 0.9316
Epoch 64/100
```

```
2500/2500 [==============================] - 1s 463us/step - loss: 0.1686 - accuracy: 0.9331
Epoch 65/100
2500/2500 [==============================] - 1s 480us/step - loss: 0.1674 - accuracy: 0.9330
Epoch 66/100
2500/2500 [==============================] - 1s 464us/step - loss: 0.1668 - accuracy: 0.9333
Epoch 67/100
2500/2500 [==============================] - 1s 470us/step - loss: 0.1657 - accuracy: 0.9337
Epoch 68/100
2500/2500 [==============================] - 1s 476us/step - loss: 0.1659 - accuracy: 0.9342
Epoch 69/100
2500/2500 [==============================] - 1s 486us/step - loss: 0.1641 - accuracy: 0.9345
Epoch 70/100
2500/2500 [==============================] - 1s 470us/step - loss: 0.1626 - accuracy: 0.9361
Epoch 71/100
2500/2500 [==============================] - 1s 483us/step - loss: 0.1627 - accuracy: 0.9350
Epoch 72/100
2500/2500 [==============================] - 1s 470us/step - loss: 0.1607 - accuracy: 0.9359
Epoch 73/100
2500/2500 [==============================] - 1s 413us/step - loss: 0.1603 - accuracy: 0.9363
Epoch 74/100
2500/2500 [==============================] - 1s 407us/step - loss: 0.1604 - accuracy: 0.9360
Epoch 75/100
2500/2500 [==============================] - 1s 406us/step - loss: 0.1584 - accuracy: 0.9368
Epoch 76/100
2500/2500 [==============================] - 1s 404us/step - loss: 0.1577 - accuracy: 0.9377
Epoch 77/100
2500/2500 [==============================] - 1s 407us/step - loss: 0.1567 - accuracy: 0.9383
Epoch 78/100
2500/2500 [==============================] - 1s 408us/step - loss: 0.1555 - accuracy: 0.9381
Epoch 79/100
2500/2500 [==============================] - 1s 432us/step - loss: 0.1556 - accuracy: 0.9379
Epoch 80/100
2500/2500 [==============================] - 1s 408us/step - loss: 0.1535 - accuracy: 0.9389
Epoch 81/100
2500/2500 [==============================] - 1s 405us/step - loss: 0.1540 - accuracy: 0.9384
Epoch 82/100
2500/2500 [==============================] - 1s 409us/step - loss: 0.1522 - accuracy: 0.9399
Epoch 83/100
2500/2500 [==============================] - 1s 407us/step - loss: 0.1517 - accuracy: 0.9406
Epoch 84/100
```

```
2500/2500 [==============================] - 1s 407us/step - loss: 0.1512 - accuracy: 0.9404
Epoch 85/100
2500/2500 [==============================] - 1s 406us/step - loss: 0.1494 - accuracy: 0.9415
Epoch 86/100
2500/2500 [==============================] - 1s 443us/step - loss: 0.1486 - accuracy: 0.9415
Epoch 87/100
2500/2500 [==============================] - 1s 413us/step - loss: 0.1481 - accuracy: 0.9412
Epoch 88/100
2500/2500 [==============================] - 1s 406us/step - loss: 0.1479 - accuracy: 0.9418
Epoch 89/100
2500/2500 [==============================] - 1s 408us/step - loss: 0.1460 - accuracy: 0.9419
Epoch 90/100
2500/2500 [==============================] - 1s 406us/step - loss: 0.1453 - accuracy: 0.9428
Epoch 91/100
2500/2500 [==============================] - 1s 408us/step - loss: 0.1449 - accuracy: 0.9420
Epoch 92/100
2500/2500 [==============================] - 1s 406us/step - loss: 0.1452 - accuracy: 0.9430
Epoch 93/100
2500/2500 [==============================] - 1s 433us/step - loss: 0.1435 - accuracy: 0.9426
Epoch 94/100
2500/2500 [==============================] - 1s 405us/step - loss: 0.1421 - accuracy: 0.9437
Epoch 95/100
2500/2500 [==============================] - 1s 406us/step - loss: 0.1425 - accuracy: 0.9439
Epoch 96/100
2500/2500 [==============================] - 1s 406us/step - loss: 0.1412 - accuracy: 0.9443
Epoch 97/100
2500/2500 [==============================] - 1s 408us/step - loss: 0.1402 - accuracy: 0.9449
Epoch 98/100
2500/2500 [==============================] - 1s 407us/step - loss: 0.1410 - accuracy: 0.9445
Epoch 99/100
2500/2500 [==============================] - 1s 431us/step - loss: 0.1393 - accuracy: 0.9456
Epoch 100/100
2500/2500 [==============================] - 1s 407us/step - loss: 0.1397 - accuracy: 0.9451
```

In [ ]:

```
# Evaluating the neural network on the test set
test_loss, test_acc = neural_network_1.evaluate(X_test_tf, y_test_tf)
print('Test accuracy:', test_acc)
```

```
625/625 [==============================] - 0s 297us/step - loss: 0.5328 - accuracy: 0.8498
Test accuracy: 0.8498499989509583
```

In [ ]:
```
Accuracy_Table[6].append(test_acc)
```

Using Features extracted from Custom Avg Word2Vec Model

Here the Feedforward Neural Network is used for the Binary Classification, where the feature extracter is the Custom Word2Vec model and the vector for a review are the average of the vectors for each word in the review.

In [ ]:
```
# split the data into 80-20 train-test
x_train, x_test, y_train, y_test = train_test_split(binframes['cvectors'], binframes['Labels'], test_size
```

In [ ]:
```
# Converting NumPy arrays to TensorFlow tensors
X_train_tf = tf.convert_to_tensor(x_train.to_list(), dtype=tf.float32)
y_train_tf = tf.convert_to_tensor(y_train, dtype=tf.int32)
X_test_tf = tf.convert_to_tensor(x_test.to_list(), dtype=tf.float32)
y_test_tf = tf.convert_to_tensor(y_test, dtype=tf.int32)
```

In [ ]:
```
neural_network_2 = tf.keras.models.Sequential([
    tf.keras.layers.Dense(50, activation='relu', input_shape=(300,)),
    tf.keras.layers.Dense(10, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

In [ ]:
```
# Compiling the neural network
neural_network_2.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

In [ ]:
```
# Training the neural network
history2 = neural_network_2.fit(X_train_tf, y_train_tf, epochs=100, batch_size=32)
```

```
Epoch 1/100
  87/2500 [>.............................] - ETA: 1s - loss: 0.6104 - accuracy: 0.7432
2500/2500 [==============================] - 1s 403us/step - loss: 0.3771 - accuracy: 0.8369
Epoch 2/100
2500/2500 [==============================] - 1s 393us/step - loss: 0.3375 - accuracy: 0.8539
Epoch 3/100
2500/2500 [==============================] - 1s 400us/step - loss: 0.3233 - accuracy: 0.8623
```

```
Epoch 4/100
2500/2500 [==============================] - 1s 425us/step - loss: 0.3127 - accuracy: 0.8657
Epoch 5/100
2500/2500 [==============================] - 1s 415us/step - loss: 0.3048 - accuracy: 0.8703
Epoch 6/100
2500/2500 [==============================] - 1s 424us/step - loss: 0.2984 - accuracy: 0.8735
Epoch 7/100
2500/2500 [==============================] - 1s 405us/step - loss: 0.2929 - accuracy: 0.8761
Epoch 8/100
2500/2500 [==============================] - 1s 466us/step - loss: 0.2866 - accuracy: 0.8794
Epoch 9/100
2500/2500 [==============================] - 1s 435us/step - loss: 0.2818 - accuracy: 0.8805
Epoch 10/100
2500/2500 [==============================] - 1s 406us/step - loss: 0.2774 - accuracy: 0.8834
Epoch 11/100
2500/2500 [==============================] - 1s 411us/step - loss: 0.2720 - accuracy: 0.8852
Epoch 12/100
2500/2500 [==============================] - 1s 404us/step - loss: 0.2680 - accuracy: 0.8876
Epoch 13/100
2500/2500 [==============================] - 1s 431us/step - loss: 0.2647 - accuracy: 0.8889
Epoch 14/100
2500/2500 [==============================] - 1s 404us/step - loss: 0.2606 - accuracy: 0.8910
Epoch 15/100
2500/2500 [==============================] - 1s 406us/step - loss: 0.2570 - accuracy: 0.8933
Epoch 16/100
2500/2500 [==============================] - 1s 407us/step - loss: 0.2535 - accuracy: 0.8947
Epoch 17/100
2500/2500 [==============================] - 1s 402us/step - loss: 0.2504 - accuracy: 0.8965
Epoch 18/100
2500/2500 [==============================] - 1s 403us/step - loss: 0.2464 - accuracy: 0.8989
Epoch 19/100
2500/2500 [==============================] - 1s 405us/step - loss: 0.2447 - accuracy: 0.8990
Epoch 20/100
2500/2500 [==============================] - 1s 402us/step - loss: 0.2418 - accuracy: 0.9002
Epoch 21/100
2500/2500 [==============================] - 1s 402us/step - loss: 0.2387 - accuracy: 0.9017
Epoch 22/100
2500/2500 [==============================] - 1s 429us/step - loss: 0.2355 - accuracy: 0.9037
Epoch 23/100
2500/2500 [==============================] - 1s 405us/step - loss: 0.2339 - accuracy: 0.9045
```

```
Epoch 24/100
2500/2500 [==============================] - 1s 403us/step - loss: 0.2309 - accuracy: 0.9060
Epoch 25/100
2500/2500 [==============================] - 1s 403us/step - loss: 0.2288 - accuracy: 0.9072
Epoch 26/100
2500/2500 [==============================] - 1s 400us/step - loss: 0.2254 - accuracy: 0.9086
Epoch 27/100
2500/2500 [==============================] - 1s 405us/step - loss: 0.2238 - accuracy: 0.9086
Epoch 28/100
2500/2500 [==============================] - 1s 408us/step - loss: 0.2210 - accuracy: 0.9109
Epoch 29/100
2500/2500 [==============================] - 1s 403us/step - loss: 0.2197 - accuracy: 0.9103
Epoch 30/100
2500/2500 [==============================] - 1s 401us/step - loss: 0.2175 - accuracy: 0.9122
Epoch 31/100
2500/2500 [==============================] - 1s 427us/step - loss: 0.2155 - accuracy: 0.9127
Epoch 32/100
2500/2500 [==============================] - 1s 402us/step - loss: 0.2134 - accuracy: 0.9136
Epoch 33/100
2500/2500 [==============================] - 1s 403us/step - loss: 0.2115 - accuracy: 0.9148
Epoch 34/100
2500/2500 [==============================] - 1s 402us/step - loss: 0.2096 - accuracy: 0.9151
Epoch 35/100
2500/2500 [==============================] - 1s 404us/step - loss: 0.2091 - accuracy: 0.9151
Epoch 36/100
2500/2500 [==============================] - 1s 401us/step - loss: 0.2052 - accuracy: 0.9183
Epoch 37/100
2500/2500 [==============================] - 1s 404us/step - loss: 0.2050 - accuracy: 0.9173
Epoch 38/100
2500/2500 [==============================] - 1s 401us/step - loss: 0.2027 - accuracy: 0.9194
Epoch 39/100
2500/2500 [==============================] - 1s 402us/step - loss: 0.2008 - accuracy: 0.9194
Epoch 40/100
2500/2500 [==============================] - 1s 421us/step - loss: 0.1993 - accuracy: 0.9196
Epoch 41/100
2500/2500 [==============================] - 1s 404us/step - loss: 0.1984 - accuracy: 0.9207
Epoch 42/100
2500/2500 [==============================] - 1s 402us/step - loss: 0.1969 - accuracy: 0.9208
Epoch 43/100
2500/2500 [==============================] - 1s 402us/step - loss: 0.1959 - accuracy: 0.9205
```

```
Epoch 44/100
2500/2500 [==============================] - 1s 402us/step - loss: 0.1945 - accuracy: 0.9222
Epoch 45/100
2500/2500 [==============================] - 1s 401us/step - loss: 0.1930 - accuracy: 0.9230
Epoch 46/100
2500/2500 [==============================] - 1s 403us/step - loss: 0.1908 - accuracy: 0.9236
Epoch 47/100
2500/2500 [==============================] - 1s 403us/step - loss: 0.1890 - accuracy: 0.9246
Epoch 48/100
2500/2500 [==============================] - 1s 427us/step - loss: 0.1881 - accuracy: 0.9249
Epoch 49/100
2500/2500 [==============================] - 1s 407us/step - loss: 0.1863 - accuracy: 0.9259
Epoch 50/100
2500/2500 [==============================] - 1s 433us/step - loss: 0.1858 - accuracy: 0.9255
Epoch 51/100
2500/2500 [==============================] - 1s 465us/step - loss: 0.1842 - accuracy: 0.9264
Epoch 52/100
2500/2500 [==============================] - 1s 479us/step - loss: 0.1832 - accuracy: 0.9271
Epoch 53/100
2500/2500 [==============================] - 1s 507us/step - loss: 0.1817 - accuracy: 0.9280
Epoch 54/100
2500/2500 [==============================] - 1s 454us/step - loss: 0.1816 - accuracy: 0.9277
Epoch 55/100
2500/2500 [==============================] - 1s 412us/step - loss: 0.1794 - accuracy: 0.9288
Epoch 56/100
2500/2500 [==============================] - 1s 428us/step - loss: 0.1776 - accuracy: 0.9291
Epoch 57/100
2500/2500 [==============================] - 1s 460us/step - loss: 0.1782 - accuracy: 0.9293
Epoch 58/100
2500/2500 [==============================] - 1s 461us/step - loss: 0.1760 - accuracy: 0.9308
Epoch 59/100
2500/2500 [==============================] - 1s 419us/step - loss: 0.1749 - accuracy: 0.9313
Epoch 60/100
2500/2500 [==============================] - 1s 412us/step - loss: 0.1726 - accuracy: 0.9317
Epoch 61/100
2500/2500 [==============================] - 1s 436us/step - loss: 0.1727 - accuracy: 0.9319
Epoch 62/100
2500/2500 [==============================] - 1s 419us/step - loss: 0.1722 - accuracy: 0.9317
Epoch 63/100
2500/2500 [==============================] - 1s 406us/step - loss: 0.1706 - accuracy: 0.9322
```

```
Epoch 64/100
2500/2500 [==============================] - 1s 400us/step - loss: 0.1684 - accuracy: 0.9331
Epoch 65/100
2500/2500 [==============================] - 1s 396us/step - loss: 0.1689 - accuracy: 0.9335
Epoch 66/100
2500/2500 [==============================] - 1s 420us/step - loss: 0.1667 - accuracy: 0.9342
Epoch 67/100
2500/2500 [==============================] - 1s 396us/step - loss: 0.1660 - accuracy: 0.9343
Epoch 68/100
2500/2500 [==============================] - 1s 394us/step - loss: 0.1639 - accuracy: 0.9355
Epoch 69/100
2500/2500 [==============================] - 1s 401us/step - loss: 0.1644 - accuracy: 0.9349
Epoch 70/100
2500/2500 [==============================] - 1s 397us/step - loss: 0.1629 - accuracy: 0.9357
Epoch 71/100
2500/2500 [==============================] - 1s 401us/step - loss: 0.1625 - accuracy: 0.9359
Epoch 72/100
2500/2500 [==============================] - 1s 404us/step - loss: 0.1621 - accuracy: 0.9361
Epoch 73/100
2500/2500 [==============================] - 1s 423us/step - loss: 0.1605 - accuracy: 0.9371
Epoch 74/100
2500/2500 [==============================] - 1s 398us/step - loss: 0.1591 - accuracy: 0.9369
Epoch 75/100
2500/2500 [==============================] - 1s 400us/step - loss: 0.1589 - accuracy: 0.9373
Epoch 76/100
2500/2500 [==============================] - 1s 394us/step - loss: 0.1581 - accuracy: 0.9376
Epoch 77/100
2500/2500 [==============================] - 1s 398us/step - loss: 0.1577 - accuracy: 0.9380
Epoch 78/100
2500/2500 [==============================] - 1s 419us/step - loss: 0.1557 - accuracy: 0.9393
Epoch 79/100
2500/2500 [==============================] - 1s 398us/step - loss: 0.1548 - accuracy: 0.9382
Epoch 80/100
2500/2500 [==============================] - 1s 396us/step - loss: 0.1548 - accuracy: 0.9390
Epoch 81/100
2500/2500 [==============================] - 1s 403us/step - loss: 0.1530 - accuracy: 0.9397
Epoch 82/100
2500/2500 [==============================] - 1s 430us/step - loss: 0.1537 - accuracy: 0.9397
Epoch 83/100
2500/2500 [==============================] - 1s 403us/step - loss: 0.1520 - accuracy: 0.9398
```

```
Epoch 84/100
2500/2500 [==============================] - 1s 398us/step - loss: 0.1509 - accuracy: 0.9405
Epoch 85/100
2500/2500 [==============================] - 1s 400us/step - loss: 0.1509 - accuracy: 0.9412
Epoch 86/100
2500/2500 [==============================] - 1s 436us/step - loss: 0.1497 - accuracy: 0.9413
Epoch 87/100
2500/2500 [==============================] - 1s 399us/step - loss: 0.1485 - accuracy: 0.9423
Epoch 88/100
2500/2500 [==============================] - 1s 432us/step - loss: 0.1488 - accuracy: 0.9422
Epoch 89/100
2500/2500 [==============================] - 1s 412us/step - loss: 0.1476 - accuracy: 0.9418
Epoch 90/100
2500/2500 [==============================] - 1s 395us/step - loss: 0.1474 - accuracy: 0.9424
Epoch 91/100
2500/2500 [==============================] - 1s 417us/step - loss: 0.1452 - accuracy: 0.9439
Epoch 92/100
2500/2500 [==============================] - 1s 449us/step - loss: 0.1460 - accuracy: 0.9428
Epoch 93/100
2500/2500 [==============================] - 1s 406us/step - loss: 0.1447 - accuracy: 0.9433
Epoch 94/100
2500/2500 [==============================] - 1s 405us/step - loss: 0.1438 - accuracy: 0.9441
Epoch 95/100
2500/2500 [==============================] - 1s 419us/step - loss: 0.1436 - accuracy: 0.9438
Epoch 96/100
2500/2500 [==============================] - 1s 413us/step - loss: 0.1422 - accuracy: 0.9445
Epoch 97/100
2500/2500 [==============================] - 1s 410us/step - loss: 0.1418 - accuracy: 0.9445
Epoch 98/100
2500/2500 [==============================] - 1s 453us/step - loss: 0.1406 - accuracy: 0.9451
Epoch 99/100
2500/2500 [==============================] - 1s 413us/step - loss: 0.1404 - accuracy: 0.9451
Epoch 100/100
2500/2500 [==============================] - 1s 408us/step - loss: 0.1401 - accuracy: 0.9456
```

In [ ]:

```python
# Evaluating the neural network on the test set
test_loss, test_acc = neural_network_2.evaluate(X_test_tf, y_test_tf)
print('Test accuracy:', test_acc)
```

```
625/625 [==============================] - 0s 300us/step - loss: 0.5405 - accuracy: 0.8510
Test accuracy: 0.8510000109672546
```

In [ ]:
```
Accuracy_Table[7].append(test_acc)
```

Using Pre-trained Word2Vec for Concatenanting Extracted Features in Binary Data Set

Here we are preparing the concatenated vectors

In [ ]:
```
binframes.head(5)
```

Out[ ]:

	reviews	Labels	MyReviews	gvectors	cvectors
1921057	item came home made bubble wrap labeled new ou...	1	[item, came, home, made, bubble, wrap, labeled...	[0.027547836303710938, 0.09846019744873047, 0....	[0.027547836303710938, 0.09846019744873047, 0....
2032152	work hope continues working perhaps get discou...	0	[work, hope, continues, working, perhaps, get,...	[0.05845424107142857, 0.016178676060267856, -0...	[0.05845424107142857, 0.016178676060267856, -0...
2590977	software come printer support newer version ma...	1	[software, come, printer, support, newer, vers...	[0.06760212912488339, -0.03568415855293843, -0...	[0.06760212912488339, -0.03568415855293843, -0...
1685277	arrived time came huge box expecting big pictu...	0	[arrived, time, came, huge, box, expecting, bi...	[0.03836669921875, 0.05717875162760417, -0.008...	[0.03836669921875, 0.05717875162760417, -0.008...
29865	work well canon toner	0	[work, well, canon, toner]	[0.0806640625, 0.07841796875, -0.0003662109375...	[0.0806640625, 0.07841796875, -0.0003662109375...

In [ ]:
```
new_model = KeyedVectors.load(temporary_filepath)
```

The padding function does the job of padding the concatenated vectors in a way that each vector has a fixed length of 10 vectors, each containing 300 values.

In [ ]:
```
def padding(list_of_lists, pad_value, desired_length, max_num_lists=10):
    padded_list = []
```

```python
    # Iterate over the original list, keeping the original content unchanged
    for sublist in list_of_lists[:max_num_lists]:
        padded_sublist = sublist[:]  # Create a copy of the sublist

        # Pad or truncate the copied sublist
        if len(sublist) < desired_length:
            padded_sublist.extend([pad_value] * (desired_length - len(sublist)))
        else:
            padded_sublist = padded_sublist[:desired_length]  # truncate if longer

        padded_list.append(padded_sublist)

    # Add empty lists if there are fewer than max_num_lists lists
    while len(padded_list) < max_num_lists:
        padded_list.append([pad_value] * desired_length)

    return padded_list
```

In [ ]:
```python
#Extracting Word Embeddings from the Pre-trained Model

def embeddingFun(sent):
    vectorsize = new_model.vector_size
    PT_Embeddings_temp = np.zeros(vectorsize)
    PT_Embeddings = []
    c=0
    pad_value=0
    desired_length=300

    for word in sent:
        if word in new_model:
            PT_Embeddings_temp=wv[word]
        if c in range(0,10):
            PT_Embeddings.append(PT_Embeddings_temp)
            c+=1

    PT_Embeddings = padding(PT_Embeddings, pad_value, desired_length)
    return np.array(PT_Embeddings)

binframes['congvectors']=binframes['MyReviews'].apply(embeddingFun)
binframes.head(5)
```

```
#temp =pd.DataFrame()
#temp['congvectors'] = binframes['MyReviews'].apply(embeddingFun)
#temp.head(10)
```

Out[ ]:

	reviews	Labels	MyReviews	gvectors	cvectors	congvectors
1921057	item came home made bubble wrap labeled new ou...	1	[item, came, home, made, bubble, wrap, labeled...	[0.027547836303710938, 0.09846019744873047, 0....	[0.027547836303710938, 0.09846019744873047, 0....	[[0.024291992, 0.010803223, -0.107421875, 0.30...
2032152	work hope continues working perhaps get discou...	0	[work, hope, continues, working, perhaps, get,...	[0.05845424107142857, 0.016178676060267856, -0...	[0.05845424107142857, 0.016178676060267856, -0...	[[-0.075683594, 0.033691406, -0.064941406, 0.1...
2590977	software come printer support newer version ma...	1	[software, come, printer, support, newer, vers...	[0.06760212912488339, -0.03568415855293843, -0...	[0.06760212912488339, -0.03568415855293843, -0...	[[0.20410156, -0.30078125, -0.013916016, 0.119...
1685277	arrived time came huge box expecting big pictu...	0	[arrived, time, came, huge, box, expecting, bi...	[0.03836669921875, 0.05717875162760417, -0.008...	[0.03836669921875, 0.05717875162760417, -0.008...	[[0.15429688, 0.26757812, 0.09326172, -0.15234...
29865	work well canon toner	0	[work, well, canon, toner]	[0.0806640625, 0.07841796875, -0.0003662109375...	[0.0806640625, 0.07841796875, -0.0003662109375...	[[-0.07568359375, 0.03369140625, -0.0649414062...

In [ ]:

```
'''
#just checking vectorizing was done correctly
count0 =0
count1 =0
count2 =0
count3 =0
for i in range(100000):
    if len(temp['congvectors'].iloc[i]) >10:
```

```
        count0+=1
    elif len(temp['congvectors'].iloc[i]) == 10:
        count1+=1
    elif len(temp['congvectors'].iloc[i]) in range(5,10):
        count2+=1
    else:
        count3+=1
print("Vectors of size greater than 10: ",count0)
print("Vectors of size 10: ",count1)
print("Vectors of size between 5 and 10: ",count2)
print("Vectors of size between 0 and 5: ",count3)
print("Sentences vectorized",(count0+count1+count2+count3))
'''
```

Out[ ]:
```
'\n#just checking vectorizing was done correctly\ncount0 =0\ncount1 =0\ncount2 =0\ncount3 =0\nfor i in range(100000):\n    if len(temp[\'congvectors\'].iloc[i]) >10:\n        count0+=1\n    elif len(temp[\'congvectors\'].iloc[i]) == 10:\n        count1+=1\n    elif len(temp[\'congvectors\'].iloc[i]) in range(5,10):\n        count2+=1\n    else:\n        count3+=1\nprint("Vectors of size greater than 10: ",count0)\nprint("Vectors of size 10: ",count1)\nprint("Vectors of size between 5 and 10: ",count2)\nprint("Vectors of size between 0 and 5: ",count3)\nprint("Sentences vectorized",(count0+count1+count2+count3))\n'
```

In [ ]:
```
#print(temp['congvectors'].iloc[2000].shape)
```

Using Custom Word2Vec for Concatenanting Extracted Features in Binary Data Set

Here we are preparing the concatenated vectors

In [ ]:
```
#loading the word vectors from the previously custom trained model
wvcon = KeyedVectors.load("word2vec.wordvectors", mmap='r')
```

In [ ]:
```
#Extracting Word Embeddings from the Pre-trained Model

def embeddingFun(sent):
    vectorsize = new_model.vector_size
    CM_Embeddings_temp = np.zeros(vectorsize)
    CM_Embeddings = []
    c=0
    pad_value=0
```

```
    desired_length=300

    for word in sent:
        if word in wvcon:
            CM_Embeddings_temp=wvcon[word]
        if c in range(0,10):
            CM_Embeddings.append(CM_Embeddings_temp)
            c+=1

    CM_Embeddings = padding(CM_Embeddings, pad_value, desired_length)
    return np.array(CM_Embeddings)

binframes['concvectors']=binframes['MyReviews'].apply(embeddingFun)
binframes.head(5)
```

Out[ ]:

	reviews	Labels	MyReviews	gvectors	cvectors	congvectors	
1921057	item came home made bubble wrap labeled new ou...	1	[item, came, home, made, bubble, wrap, labeled...	[0.027547836303710938, 0.09846019744873047, 0....	[0.027547836303710938, 0.09846019744873047, 0....	[[0.024291992, 0.010803223, -0.107421875, 0.30...	-0.0182
2032152	work hope continues working perhaps get discou...	0	[work, hope, continues, working, perhaps, get,...	[0.05845424107142857, 0.016178676060267856, -0...	[0.05845424107142857, 0.016178676060267856, -0...	[[-0.075683594, 0.033691406, -0.064941406, 0.1...	0.06
2590977	software come printer support newer version ma...	1	[software, come, printer, support, newer, vers...	[0.06760212912488339, -0.03568415855293843, -0...	[0.06760212912488339, -0.03568415855293843, -0...	[[0.20410156, -0.30078125, -0.013916016, 0.119...	-0
1685277	arrived time came huge box expecting big pictu...	0	[arrived, time, came, huge, box, expecting, bi...	[0.03836669921875, 0.05717875162760417, -0.008...	[0.03836669921875, 0.05717875162760417, -0.008...	[[0.15429688, 0.26757812, 0.09326172, -0.15234...	-0.154
29865	work well canon toner	0	[work, well, canon, toner]	[0.0806640625, 0.07841796875, -0.0003662109375...	[0.0806640625, 0.07841796875, -0.0003662109375...	[[-0.07568359375, 0.03369140625, -0.0649414062...	[[0.0448 -0.032

FNN Using Features extracted from Pre-trained Concatenated Word2Vec Model for Binary Data

Here the Feedforward Neural Network is used for the Binary Classification, where the feature extracter is the Pre-trained Gensim Word2Vec model and the vector for a review are the concatenation of the vectors for each word in the review.

```
In [ ]: # split the data into 80-20 train-test
        x_train, x_test, y_train, y_test = train_test_split(binframes['congvectors'], binframes['Labels'], test_s
```

```
In [ ]: # Converting NumPy arrays to TensorFlow tensors
        X_train_tf = tf.convert_to_tensor(x_train.to_list(), dtype=tf.float32)
        y_train_tf = tf.convert_to_tensor(y_train, dtype=tf.int32)
        X_test_tf = tf.convert_to_tensor(x_test.to_list(), dtype=tf.float32)
        y_test_tf = tf.convert_to_tensor(y_test, dtype=tf.int32)
```

```
In [ ]: # Flattening the tensor into a single feature vector
        x_train_tf_flat = tf.reshape(X_train_tf, (X_train_tf.shape[0],-1))
        X_test_tf_flat = tf.reshape(X_test_tf, (X_test_tf.shape[0],-1))
```

```
In [ ]: neural_network_3 = tf.keras.models.Sequential([
            tf.keras.layers.Dense(50, activation='relu', input_shape=((3000),)),
            tf.keras.layers.Dense(10, activation='relu'),
            tf.keras.layers.Dense(1, activation='sigmoid')
        ])
```

```
In [ ]: # Compiling the neural network
        neural_network_3.compile(optimizer='adam',
                      loss='binary_crossentropy',
                      metrics=['accuracy'])
```

```
In [ ]: # Training the neural network
        history3 = neural_network_3.fit(x_train_tf_flat, y_train_tf, epochs=100, batch_size=32)
```

```
Epoch 1/100
2500/2500 [==============================] - 2s 653us/step - loss: 0.4609 - accuracy: 0.7789
Epoch 2/100
2500/2500 [==============================] - 2s 622us/step - loss: 0.3993 - accuracy: 0.8150
Epoch 3/100
2500/2500 [==============================] - 1s 597us/step - loss: 0.3422 - accuracy: 0.8475
Epoch 4/100
2500/2500 [==============================] - 1s 590us/step - loss: 0.2763 - accuracy: 0.8829
```

```
Epoch 5/100
2500/2500 [==============================] - 2s 625us/step - loss: 0.2119 - accuracy: 0.9147
Epoch 6/100
2500/2500 [==============================] - 2s 619us/step - loss: 0.1612 - accuracy: 0.9380
Epoch 7/100
2500/2500 [==============================] - 2s 819us/step - loss: 0.1235 - accuracy: 0.9537
Epoch 8/100
2500/2500 [==============================] - 2s 929us/step - loss: 0.0999 - accuracy: 0.9626
Epoch 9/100
2500/2500 [==============================] - 2s 720us/step - loss: 0.0855 - accuracy: 0.9686
Epoch 10/100
2500/2500 [==============================] - 2s 794us/step - loss: 0.0718 - accuracy: 0.9742
Epoch 11/100
2500/2500 [==============================] - 2s 684us/step - loss: 0.0669 - accuracy: 0.9761
Epoch 12/100
2500/2500 [==============================] - 2s 756us/step - loss: 0.0600 - accuracy: 0.9786
Epoch 13/100
2500/2500 [==============================] - 2s 726us/step - loss: 0.0559 - accuracy: 0.9798
Epoch 14/100
2500/2500 [==============================] - 2s 793us/step - loss: 0.0518 - accuracy: 0.9816
Epoch 15/100
2500/2500 [==============================] - 2s 701us/step - loss: 0.0499 - accuracy: 0.9827
Epoch 16/100
2500/2500 [==============================] - 2s 659us/step - loss: 0.0459 - accuracy: 0.9843
Epoch 17/100
2500/2500 [==============================] - 2s 703us/step - loss: 0.0437 - accuracy: 0.9847
Epoch 18/100
2500/2500 [==============================] - 2s 675us/step - loss: 0.0418 - accuracy: 0.9853
Epoch 19/100
2500/2500 [==============================] - 2s 642us/step - loss: 0.0402 - accuracy: 0.9861
Epoch 20/100
2500/2500 [==============================] - 2s 713us/step - loss: 0.0401 - accuracy: 0.9858
Epoch 21/100
2500/2500 [==============================] - 2s 659us/step - loss: 0.0357 - accuracy: 0.9876
Epoch 22/100
2500/2500 [==============================] - 2s 639us/step - loss: 0.0357 - accuracy: 0.9874
Epoch 23/100
2500/2500 [==============================] - 2s 646us/step - loss: 0.0341 - accuracy: 0.9880
Epoch 24/100
2500/2500 [==============================] - 2s 661us/step - loss: 0.0332 - accuracy: 0.9888
```

```
Epoch 25/100
2500/2500 [==============================] - 2s 623us/step - loss: 0.0321 - accuracy: 0.9889
Epoch 26/100
2500/2500 [==============================] - 2s 622us/step - loss: 0.0322 - accuracy: 0.9888
Epoch 27/100
2500/2500 [==============================] - 2s 623us/step - loss: 0.0312 - accuracy: 0.9892
Epoch 28/100
2500/2500 [==============================] - 2s 600us/step - loss: 0.0325 - accuracy: 0.9888
Epoch 29/100
2500/2500 [==============================] - 2s 604us/step - loss: 0.0292 - accuracy: 0.9898
Epoch 30/100
2500/2500 [==============================] - 2s 692us/step - loss: 0.0290 - accuracy: 0.9898
Epoch 31/100
2500/2500 [==============================] - 2s 656us/step - loss: 0.0302 - accuracy: 0.9894
Epoch 32/100
2500/2500 [==============================] - 2s 629us/step - loss: 0.0260 - accuracy: 0.9909
Epoch 33/100
2500/2500 [==============================] - 2s 643us/step - loss: 0.0269 - accuracy: 0.9905
Epoch 34/100
2500/2500 [==============================] - 2s 638us/step - loss: 0.0260 - accuracy: 0.9911
Epoch 35/100
2500/2500 [==============================] - 2s 632us/step - loss: 0.0279 - accuracy: 0.9906
Epoch 36/100
2500/2500 [==============================] - 2s 663us/step - loss: 0.0245 - accuracy: 0.9919
Epoch 37/100
2500/2500 [==============================] - 2s 619us/step - loss: 0.0261 - accuracy: 0.9908
Epoch 38/100
2500/2500 [==============================] - 2s 621us/step - loss: 0.0241 - accuracy: 0.9920
Epoch 39/100
2500/2500 [==============================] - 2s 624us/step - loss: 0.0222 - accuracy: 0.9922
Epoch 40/100
2500/2500 [==============================] - 2s 628us/step - loss: 0.0241 - accuracy: 0.9915
Epoch 41/100
2500/2500 [==============================] - 2s 657us/step - loss: 0.0233 - accuracy: 0.9922
Epoch 42/100
2500/2500 [==============================] - 2s 626us/step - loss: 0.0236 - accuracy: 0.9918
Epoch 43/100
2500/2500 [==============================] - 2s 632us/step - loss: 0.0225 - accuracy: 0.9923
Epoch 44/100
2500/2500 [==============================] - 2s 628us/step - loss: 0.0223 - accuracy: 0.9922
```

```
Epoch 45/100
2500/2500 [==============================] - 2s 628us/step - loss: 0.0216 - accuracy: 0.9924
Epoch 46/100
2500/2500 [==============================] - 2s 655us/step - loss: 0.0242 - accuracy: 0.9916
Epoch 47/100
2500/2500 [==============================] - 2s 742us/step - loss: 0.0204 - accuracy: 0.9927
Epoch 48/100
2500/2500 [==============================] - 2s 836us/step - loss: 0.0211 - accuracy: 0.9928
Epoch 49/100
2500/2500 [==============================] - 2s 682us/step - loss: 0.0226 - accuracy: 0.9922
Epoch 50/100
2500/2500 [==============================] - 2s 730us/step - loss: 0.0223 - accuracy: 0.9922
Epoch 51/100
2500/2500 [==============================] - 2s 666us/step - loss: 0.0203 - accuracy: 0.9927
Epoch 52/100
2500/2500 [==============================] - 2s 681us/step - loss: 0.0207 - accuracy: 0.9927
Epoch 53/100
2500/2500 [==============================] - 2s 648us/step - loss: 0.0188 - accuracy: 0.9933
Epoch 54/100
2500/2500 [==============================] - 2s 637us/step - loss: 0.0206 - accuracy: 0.9927
Epoch 55/100
2500/2500 [==============================] - 2s 658us/step - loss: 0.0187 - accuracy: 0.9936
Epoch 56/100
2500/2500 [==============================] - 2s 649us/step - loss: 0.0205 - accuracy: 0.9930
Epoch 57/100
2500/2500 [==============================] - 2s 700us/step - loss: 0.0193 - accuracy: 0.9937
Epoch 58/100
2500/2500 [==============================] - 2s 653us/step - loss: 0.0194 - accuracy: 0.9934
Epoch 59/100
2500/2500 [==============================] - 2s 610us/step - loss: 0.0200 - accuracy: 0.9931
Epoch 60/100
2500/2500 [==============================] - 2s 693us/step - loss: 0.0194 - accuracy: 0.9932
Epoch 61/100
2500/2500 [==============================] - 2s 650us/step - loss: 0.0188 - accuracy: 0.9933
Epoch 62/100
2500/2500 [==============================] - 1s 594us/step - loss: 0.0184 - accuracy: 0.9934
Epoch 63/100
2500/2500 [==============================] - 2s 602us/step - loss: 0.0201 - accuracy: 0.9931
Epoch 64/100
2500/2500 [==============================] - 2s 681us/step - loss: 0.0182 - accuracy: 0.9936
```

```
Epoch 65/100
2500/2500 [==============================] - 2s 650us/step - loss: 0.0183 - accuracy: 0.9934
Epoch 66/100
2500/2500 [==============================] - 2s 733us/step - loss: 0.0187 - accuracy: 0.9934
Epoch 67/100
2500/2500 [==============================] - 2s 739us/step - loss: 0.0182 - accuracy: 0.9937
Epoch 68/100
2500/2500 [==============================] - 2s 686us/step - loss: 0.0187 - accuracy: 0.9933
Epoch 69/100
2500/2500 [==============================] - 2s 695us/step - loss: 0.0171 - accuracy: 0.9941
Epoch 70/100
2500/2500 [==============================] - 2s 657us/step - loss: 0.0176 - accuracy: 0.9939
Epoch 71/100
2500/2500 [==============================] - 2s 650us/step - loss: 0.0175 - accuracy: 0.9937
Epoch 72/100
2500/2500 [==============================] - 2s 641us/step - loss: 0.0175 - accuracy: 0.9940
Epoch 73/100
2500/2500 [==============================] - 2s 682us/step - loss: 0.0166 - accuracy: 0.9941
Epoch 74/100
2500/2500 [==============================] - 2s 639us/step - loss: 0.0171 - accuracy: 0.9940
Epoch 75/100
2500/2500 [==============================] - 2s 613us/step - loss: 0.0165 - accuracy: 0.9939
Epoch 76/100
2500/2500 [==============================] - 2s 614us/step - loss: 0.0176 - accuracy: 0.9937
Epoch 77/100
2500/2500 [==============================] - 1s 583us/step - loss: 0.0166 - accuracy: 0.9941
Epoch 78/100
2500/2500 [==============================] - 2s 623us/step - loss: 0.0172 - accuracy: 0.9939
Epoch 79/100
2500/2500 [==============================] - 2s 640us/step - loss: 0.0165 - accuracy: 0.9942
Epoch 80/100
2500/2500 [==============================] - 2s 676us/step - loss: 0.0166 - accuracy: 0.9943
Epoch 81/100
2500/2500 [==============================] - 2s 630us/step - loss: 0.0153 - accuracy: 0.9944
Epoch 82/100
2500/2500 [==============================] - 2s 635us/step - loss: 0.0159 - accuracy: 0.9944
Epoch 83/100
2500/2500 [==============================] - 2s 691us/step - loss: 0.0170 - accuracy: 0.9940
Epoch 84/100
2500/2500 [==============================] - 1s 577us/step - loss: 0.0164 - accuracy: 0.9944
```

```
Epoch 85/100
2500/2500 [==============================] - 1s 588us/step - loss: 0.0176 - accuracy: 0.9940
Epoch 86/100
2500/2500 [==============================] - 2s 610us/step - loss: 0.0158 - accuracy: 0.9942
Epoch 87/100
2500/2500 [==============================] - 2s 603us/step - loss: 0.0155 - accuracy: 0.9944
Epoch 88/100
2500/2500 [==============================] - 2s 618us/step - loss: 0.0153 - accuracy: 0.9945
Epoch 89/100
2500/2500 [==============================] - 2s 600us/step - loss: 0.0172 - accuracy: 0.9939
Epoch 90/100
2500/2500 [==============================] - 1s 591us/step - loss: 0.0157 - accuracy: 0.9944
Epoch 91/100
2500/2500 [==============================] - 1s 598us/step - loss: 0.0155 - accuracy: 0.9942
Epoch 92/100
2500/2500 [==============================] - 2s 605us/step - loss: 0.0162 - accuracy: 0.9942
Epoch 93/100
2500/2500 [==============================] - 2s 627us/step - loss: 0.0157 - accuracy: 0.9944
Epoch 94/100
2500/2500 [==============================] - 2s 621us/step - loss: 0.0153 - accuracy: 0.9947
Epoch 95/100
2500/2500 [==============================] - 1s 599us/step - loss: 0.0145 - accuracy: 0.9948
Epoch 96/100
2500/2500 [==============================] - 1s 595us/step - loss: 0.0158 - accuracy: 0.9945
Epoch 97/100
2500/2500 [==============================] - 2s 705us/step - loss: 0.0147 - accuracy: 0.9945
Epoch 98/100
2500/2500 [==============================] - 2s 605us/step - loss: 0.0149 - accuracy: 0.9948
Epoch 99/100
2500/2500 [==============================] - 2s 675us/step - loss: 0.0157 - accuracy: 0.9943
Epoch 100/100
2500/2500 [==============================] - 2s 688us/step - loss: 0.0144 - accuracy: 0.9948
```

In [ ]:

```python
# Evaluating the neural network on the test set
test_loss, test_acc = neural_network_3.evaluate(X_test_tf_flat, y_test_tf)
print('Test accuracy:', test_acc)
```

```
625/625 [==============================] - 0s 517us/step - loss: 3.0111 - accuracy: 0.7652
Test accuracy: 0.7652000188827515
```

In [ ]:

```python
Accuracy_Table[8].append(test_acc)
```

FNN Using Features extracted from Custom Concatenated Word2Vec Model For Binary Data

Here the Feedforward Neural Network is used for the Binary Classification, where the feature extracter is the Custom Word2Vec model and the vector for a review are the concatenation of the vectors for each word in the review.

In [ ]:
```
# split the data into 80-20 train-test
x_train, x_test, y_train, y_test = train_test_split(binframes['concvectors'], binframes['Labels'], test_s
```

In [ ]:
```
# Converting NumPy arrays to TensorFlow tensors
X_train_tf = tf.convert_to_tensor(x_train.to_list(), dtype=tf.float32)
y_train_tf = tf.convert_to_tensor(y_train, dtype=tf.int32)
X_test_tf = tf.convert_to_tensor(x_test.to_list(), dtype=tf.float32)
y_test_tf = tf.convert_to_tensor(y_test, dtype=tf.int32)
```

In [ ]:
```
# Flattening the tensor into a single feature vector
x_train_tf_flat = tf.reshape(X_train_tf, (X_train_tf.shape[0],-1))
X_test_tf_flat = tf.reshape(X_test_tf, (X_test_tf.shape[0],-1))
```

In [ ]:
```
neural_network_4 = tf.keras.models.Sequential([
    tf.keras.layers.Dense(50, activation='relu', input_shape=((3000),)),
    tf.keras.layers.Dense(10, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
```

In [ ]:
```
# Compiling the neural network
neural_network_4.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

In [ ]:
```
# Training the neural network
history4 = neural_network_4.fit(x_train_tf_flat, y_train_tf, epochs=100, batch_size=32)
```

```
Epoch 1/100
  50/2500 [..............................] - ETA: 2s - loss: 0.5508 - accuracy: 0.7300
2500/2500 [==============================] - 2s 775us/step - loss: 0.4087 - accuracy: 0.8116
```

```
Epoch 2/100
2500/2500 [==============================] - 2s 617us/step - loss: 0.3618 - accuracy: 0.8364
Epoch 3/100
2500/2500 [==============================] - 2s 610us/step - loss: 0.3208 - accuracy: 0.8581
Epoch 4/100
2500/2500 [==============================] - 1s 596us/step - loss: 0.2655 - accuracy: 0.8888
Epoch 5/100
2500/2500 [==============================] - 2s 664us/step - loss: 0.2055 - accuracy: 0.9185
Epoch 6/100
2500/2500 [==============================] - 2s 683us/step - loss: 0.1514 - accuracy: 0.9430
Epoch 7/100
2500/2500 [==============================] - 2s 612us/step - loss: 0.1124 - accuracy: 0.9587
Epoch 8/100
2500/2500 [==============================] - 1s 595us/step - loss: 0.0861 - accuracy: 0.9692
Epoch 9/100
2500/2500 [==============================] - 2s 603us/step - loss: 0.0688 - accuracy: 0.9756
Epoch 10/100
2500/2500 [==============================] - 2s 602us/step - loss: 0.0617 - accuracy: 0.9790
Epoch 11/100
2500/2500 [==============================] - 2s 609us/step - loss: 0.0569 - accuracy: 0.9804
Epoch 12/100
2500/2500 [==============================] - 2s 661us/step - loss: 0.0495 - accuracy: 0.9831
Epoch 13/100
2500/2500 [==============================] - 2s 660us/step - loss: 0.0456 - accuracy: 0.9845
Epoch 14/100
2500/2500 [==============================] - 2s 653us/step - loss: 0.0424 - accuracy: 0.9850
Epoch 15/100
2500/2500 [==============================] - 2s 695us/step - loss: 0.0422 - accuracy: 0.9855
Epoch 16/100
2500/2500 [==============================] - 1s 597us/step - loss: 0.0373 - accuracy: 0.9871
Epoch 17/100
2500/2500 [==============================] - 2s 609us/step - loss: 0.0393 - accuracy: 0.9862
Epoch 18/100
2500/2500 [==============================] - 2s 606us/step - loss: 0.0349 - accuracy: 0.9878
Epoch 19/100
2500/2500 [==============================] - 2s 600us/step - loss: 0.0336 - accuracy: 0.9882
Epoch 20/100
2500/2500 [==============================] - 2s 636us/step - loss: 0.0324 - accuracy: 0.9890
Epoch 21/100
2500/2500 [==============================] - 2s 719us/step - loss: 0.0331 - accuracy: 0.9888
```

```
Epoch 22/100
2500/2500 [==============================] - 2s 711us/step - loss: 0.0311 - accuracy: 0.9893
Epoch 23/100
2500/2500 [==============================] - 2s 660us/step - loss: 0.0332 - accuracy: 0.9887
Epoch 24/100
2500/2500 [==============================] - 2s 656us/step - loss: 0.0278 - accuracy: 0.9902
Epoch 25/100
2500/2500 [==============================] - 2s 622us/step - loss: 0.0290 - accuracy: 0.9899
Epoch 26/100
2500/2500 [==============================] - 1s 593us/step - loss: 0.0290 - accuracy: 0.9900
Epoch 27/100
2500/2500 [==============================] - 2s 609us/step - loss: 0.0264 - accuracy: 0.9911
Epoch 28/100
2500/2500 [==============================] - 1s 597us/step - loss: 0.0249 - accuracy: 0.9914
Epoch 29/100
2500/2500 [==============================] - 1s 594us/step - loss: 0.0283 - accuracy: 0.9901
Epoch 30/100
2500/2500 [==============================] - 2s 646us/step - loss: 0.0257 - accuracy: 0.9913
Epoch 31/100
2500/2500 [==============================] - 2s 606us/step - loss: 0.0261 - accuracy: 0.9916
Epoch 32/100
2500/2500 [==============================] - 2s 674us/step - loss: 0.0255 - accuracy: 0.9912
Epoch 33/100
2500/2500 [==============================] - 2s 659us/step - loss: 0.0246 - accuracy: 0.9913
Epoch 34/100
2500/2500 [==============================] - 2s 600us/step - loss: 0.0233 - accuracy: 0.9921
Epoch 35/100
2500/2500 [==============================] - 2s 639us/step - loss: 0.0227 - accuracy: 0.9920
Epoch 36/100
2500/2500 [==============================] - 1s 589us/step - loss: 0.0241 - accuracy: 0.9917
Epoch 37/100
2500/2500 [==============================] - 1s 586us/step - loss: 0.0220 - accuracy: 0.9924
Epoch 38/100
2500/2500 [==============================] - 1s 599us/step - loss: 0.0217 - accuracy: 0.9925
Epoch 39/100
2500/2500 [==============================] - 1s 591us/step - loss: 0.0224 - accuracy: 0.9923
Epoch 40/100
2500/2500 [==============================] - 2s 651us/step - loss: 0.0218 - accuracy: 0.9926
Epoch 41/100
2500/2500 [==============================] - 2s 707us/step - loss: 0.0214 - accuracy: 0.9925
```

```
Epoch 42/100
2500/2500 [==============================] - 2s 601us/step - loss: 0.0222 - accuracy: 0.9924
Epoch 43/100
2500/2500 [==============================] - 1s 596us/step - loss: 0.0216 - accuracy: 0.9927
Epoch 44/100
2500/2500 [==============================] - 1s 593us/step - loss: 0.0212 - accuracy: 0.9925
Epoch 45/100
2500/2500 [==============================] - 1s 592us/step - loss: 0.0204 - accuracy: 0.9929
Epoch 46/100
2500/2500 [==============================] - 2s 604us/step - loss: 0.0199 - accuracy: 0.9929
Epoch 47/100
2500/2500 [==============================] - 2s 640us/step - loss: 0.0205 - accuracy: 0.9929
Epoch 48/100
2500/2500 [==============================] - 2s 602us/step - loss: 0.0203 - accuracy: 0.9930
Epoch 49/100
2500/2500 [==============================] - 1s 599us/step - loss: 0.0187 - accuracy: 0.9935
Epoch 50/100
2500/2500 [==============================] - 2s 613us/step - loss: 0.0208 - accuracy: 0.9931
Epoch 51/100
2500/2500 [==============================] - 2s 613us/step - loss: 0.0174 - accuracy: 0.9937
Epoch 52/100
2500/2500 [==============================] - 2s 605us/step - loss: 0.0190 - accuracy: 0.9936
Epoch 53/100
2500/2500 [==============================] - 2s 634us/step - loss: 0.0180 - accuracy: 0.9936
Epoch 54/100
2500/2500 [==============================] - 1s 598us/step - loss: 0.0178 - accuracy: 0.9936
Epoch 55/100
2500/2500 [==============================] - 1s 593us/step - loss: 0.0188 - accuracy: 0.9936
Epoch 56/100
2500/2500 [==============================] - 1s 595us/step - loss: 0.0180 - accuracy: 0.9938
Epoch 57/100
2500/2500 [==============================] - 2s 603us/step - loss: 0.0178 - accuracy: 0.9937
Epoch 58/100
2500/2500 [==============================] - 2s 610us/step - loss: 0.0165 - accuracy: 0.9940
Epoch 59/100
2500/2500 [==============================] - 2s 634us/step - loss: 0.0194 - accuracy: 0.9933
Epoch 60/100
2500/2500 [==============================] - 1s 586us/step - loss: 0.0187 - accuracy: 0.9936
Epoch 61/100
2500/2500 [==============================] - 1s 586us/step - loss: 0.0183 - accuracy: 0.9934
```

```
Epoch 62/100
2500/2500 [==============================] - 1s 589us/step - loss: 0.0175 - accuracy: 0.9939
Epoch 63/100
2500/2500 [==============================] - 1s 594us/step - loss: 0.0160 - accuracy: 0.9944
Epoch 64/100
2500/2500 [==============================] - 2s 708us/step - loss: 0.0171 - accuracy: 0.9940
Epoch 65/100
2500/2500 [==============================] - 2s 630us/step - loss: 0.0184 - accuracy: 0.9936
Epoch 66/100
2500/2500 [==============================] - 1s 594us/step - loss: 0.0182 - accuracy: 0.9937
Epoch 67/100
2500/2500 [==============================] - 1s 596us/step - loss: 0.0165 - accuracy: 0.9943
Epoch 68/100
2500/2500 [==============================] - 1s 581us/step - loss: 0.0163 - accuracy: 0.9945
Epoch 69/100
2500/2500 [==============================] - 1s 592us/step - loss: 0.0159 - accuracy: 0.9943
Epoch 70/100
2500/2500 [==============================] - 2s 676us/step - loss: 0.0163 - accuracy: 0.9943
Epoch 71/100
2500/2500 [==============================] - 1s 571us/step - loss: 0.0155 - accuracy: 0.9944
Epoch 72/100
2500/2500 [==============================] - 1s 564us/step - loss: 0.0177 - accuracy: 0.9939
Epoch 73/100
2500/2500 [==============================] - 1s 560us/step - loss: 0.0147 - accuracy: 0.9946
Epoch 74/100
2500/2500 [==============================] - 1s 592us/step - loss: 0.0163 - accuracy: 0.9944
Epoch 75/100
2500/2500 [==============================] - 2s 652us/step - loss: 0.0160 - accuracy: 0.9943
Epoch 76/100
2500/2500 [==============================] - 1s 568us/step - loss: 0.0153 - accuracy: 0.9947
Epoch 77/100
2500/2500 [==============================] - 1s 566us/step - loss: 0.0155 - accuracy: 0.9945
Epoch 78/100
2500/2500 [==============================] - 1s 570us/step - loss: 0.0152 - accuracy: 0.9946
Epoch 79/100
2500/2500 [==============================] - 2s 646us/step - loss: 0.0155 - accuracy: 0.9944
Epoch 80/100
2500/2500 [==============================] - 2s 911us/step - loss: 0.0164 - accuracy: 0.9944
Epoch 81/100
2500/2500 [==============================] - 2s 682us/step - loss: 0.0171 - accuracy: 0.9938
```

```
Epoch 82/100
2500/2500 [==============================] - 2s 668us/step - loss: 0.0153 - accuracy: 0.9945
Epoch 83/100
2500/2500 [==============================] - 2s 614us/step - loss: 0.0148 - accuracy: 0.9948
Epoch 84/100
2500/2500 [==============================] - 2s 670us/step - loss: 0.0145 - accuracy: 0.9945
Epoch 85/100
2500/2500 [==============================] - 2s 657us/step - loss: 0.0153 - accuracy: 0.9946
Epoch 86/100
2500/2500 [==============================] - 2s 642us/step - loss: 0.0142 - accuracy: 0.9947
Epoch 87/100
2500/2500 [==============================] - 2s 675us/step - loss: 0.0147 - accuracy: 0.9945
Epoch 88/100
2500/2500 [==============================] - 2s 640us/step - loss: 0.0149 - accuracy: 0.9947
Epoch 89/100
2500/2500 [==============================] - 2s 650us/step - loss: 0.0149 - accuracy: 0.9946
Epoch 90/100
2500/2500 [==============================] - 2s 721us/step - loss: 0.0136 - accuracy: 0.9951
Epoch 91/100
2500/2500 [==============================] - 2s 636us/step - loss: 0.0162 - accuracy: 0.9943
Epoch 92/100
2500/2500 [==============================] - 2s 650us/step - loss: 0.0153 - accuracy: 0.9945
Epoch 93/100
2500/2500 [==============================] - 1s 576us/step - loss: 0.0152 - accuracy: 0.9945
Epoch 94/100
2500/2500 [==============================] - 1s 597us/step - loss: 0.0166 - accuracy: 0.9944
Epoch 95/100
2500/2500 [==============================] - 2s 641us/step - loss: 0.0133 - accuracy: 0.9952
Epoch 96/100
2500/2500 [==============================] - 2s 632us/step - loss: 0.0143 - accuracy: 0.9949
Epoch 97/100
2500/2500 [==============================] - 1s 580us/step - loss: 0.0143 - accuracy: 0.9949
Epoch 98/100
2500/2500 [==============================] - 2s 618us/step - loss: 0.0142 - accuracy: 0.9950
Epoch 99/100
2500/2500 [==============================] - 2s 635us/step - loss: 0.0151 - accuracy: 0.9948
Epoch 100/100
2500/2500 [==============================] - 2s 712us/step - loss: 0.0130 - accuracy: 0.9953
```

In [ ]:

```
# Evaluating the neural network on the test set
```

```
test_loss, test_acc = neural_network_4.evaluate(X_test_tf_flat, y_test_tf)
print('Test accuracy:', test_acc)
```

```
625/625 [==============================] - 0s 398us/step - loss: 3.0079 - accuracy: 0.7911
Test accuracy: 0.7910500168800354
```

In [ ]:
```
Accuracy_Table[9].append(test_acc)
```

FNN for Ternary Classification

Using Pre-trained Avg Word2Vec for Feature Extraction in Ternary DataSet

Here we are performing the Pre-trained Gensim Word2Vec feature extraction for Ternary Data

In [ ]:
```
#tokenizing the reviews into words
terframes['MyReviews'] = [word_tokenize(t) for t in terframes['reviews']]
terframes.head(5)
```

Out[ ]:

	reviews	Labels	MyReviews
1163606	excellent pen actually pen like use write anyt...	0	[excellent, pen, actually, pen, like, use, wri...
2069284	fresh set battery nice strong beam unfortunate...	2	[fresh, set, battery, nice, strong, beam, unfo...
338411	file folder label great use folder	0	[file, folder, label, great, use, folder]
37088	work 's well heavy duty steel original stapler...	2	[work, 's, well, heavy, duty, steel, original,...
752649	great quality great value	0	[great, quality, great, value]

In [ ]:
```
new_model = KeyedVectors.load(temporary_filepath)
```

In [ ]:
```
#Extracting Word Embeddings from the Pre-trained Model

def embeddingFun(sent):
    vectorsize = new_model.vector_size
    PT_Embeddings = np.zeros(vectorsize)
    c=1
```

```python
    for word in sent:
        if word in new_model:
            c+=1
            PT_Embeddings+=wv[word]
    avg = PT_Embeddings/c
    return avg

terframes['gvectors']=terframes['MyReviews'].apply(embeddingFun)
```

Using Custom Avg Word2Vec for Feature Extraction in Ternary DataSet

Here we are performing the Custom Word2Vec feature extraction for Ternary Data

In [ ]:
```python
#making and saving our custom Word2Vec Model
start = time.time()
model = Word2Vec(sentences=terframes['MyReviews'], vector_size=300, window=11, min_count=10, workers=core
end = round(time.time()-start,2)
print("This process took",end,"seconds.")
```

```
This process took 158.33 seconds.
```

In [ ]:
```python
model.save("word2vecTer.model")
```

In [ ]:
```python
model = Word2Vec.load("word2vecTer.model")
```

In [ ]:
```python
word_vectors = model.wv
word_vectors.save("word2vecTer.wordvectors")
```

In [ ]:
```python
# Load back with memory-mapping = read-only, shared across processes.
wv2 = KeyedVectors.load("word2vecTer.wordvectors", mmap='r')
```

In [ ]:
```python
#Extracting Word Embeddings from the Pre-trained Model

def embeddingFun2(sent):
    vectorsize = wv2.vector_size
    CM_Embeddings = np.zeros(vectorsize)
    c=1
```

```
    for word in sent:
        if word in wv2:
            c+=1
            PT_Embeddings+=wv[word]
    avg = CM_Embeddings/c
    return avg

terframes['cvectors']=terframes['MyReviews'].apply(embeddingFun)
```

In [ ]:
```
terframes.head(5)
```

Out[ ]:

	reviews	Labels	MyReviews	gvectors	cvectors
1163606	excellent pen actually pen like use write anyt...	0	[excellent, pen, actually, pen, like, use, wri...	[0.013108995225694444, -0.015750461154513888, ...	[0.013108995225694444, -0.015750461154513888, ...
2069284	fresh set battery nice strong beam unfortunate...	2	[fresh, set, battery, nice, strong, beam, unfo...	[0.035248976487379804, 0.09482046274038461, 0....	[0.035248976487379804, 0.09482046274038461, 0....
338411	file folder label great use folder	0	[file, folder, label, great, use, folder]	[0.11321149553571429, 0.029767717633928572, -0...	[0.11321149553571429, 0.029767717633928572, -0...
37088	work 's well heavy duty steel original stapler...	2	[work, 's, well, heavy, duty, steel, original,...	[0.07191051136363637, 0.07883522727272728, -0....	[0.07191051136363637, 0.07883522727272728, -0....
752649	great quality great value	0	[great, quality, great, value]	[-0.00732421875, 0.141259765625, 0.03442382812...	[-0.00732421875, 0.141259765625, 0.03442382812...

Using Pre-trained Word2Vec for Concatenanting Extracted Features in Ternary Data Set

Here we are preparing the concatenated vectors

In [ ]:
```
terframes.head(5)
```

Out[ ]:

	reviews	Labels	MyReviews	gvectors	cvectors
1163606	excellent pen actually pen like use write anyt...	0	[excellent, pen, actually, pen, like, use, wri...	[0.013108995225694444, -0.015750461154513888, ...	[0.013108995225694444, -0.015750461154513888, ...
2069284	fresh set battery nice strong beam unfortunate...	2	[fresh, set, battery, nice, strong, beam, unfo...	[0.035248976487379804, 0.09482046274038461, 0....	[0.035248976487379804, 0.09482046274038461, 0....
338411	file folder label great use folder	0	[file, folder, label, great, use, folder]	[0.11321149553571429, 0.029767717633928572, -0...	[0.11321149553571429, 0.029767717633928572, -0...
37088	work 's well heavy duty steel original stapler...	2	[work, 's, well, heavy, duty, steel, original,...	[0.07191051136363637, 0.07883522727272728, -0....	[0.07191051136363637, 0.07883522727272728, -0....
752649	great quality great value	0	[great, quality, great, value]	[-0.00732421875, 0.141259765625, 0.03442382812...	[-0.00732421875, 0.141259765625, 0.03442382812...

In [ ]:

```
new_model = KeyedVectors.load(temporary_filepath)
```

In [ ]:

```
#Extracting Word Embeddings from the Pre-trained Model

def embeddingFun(sent):
    vectorsize = new_model.vector_size
    PT_Embeddings_temp = np.zeros(vectorsize)
    PT_Embeddings = []
    c=0
    pad_value=0
    desired_length=300

    for word in sent:
        if word in new_model:
            PT_Embeddings_temp=wv[word]
        if c in range(0,10):
            PT_Embeddings.append(PT_Embeddings_temp)
            c+=1
```

```python
    PT_Embeddings = padding(PT_Embeddings, pad_value, desired_length)
    return np.array(PT_Embeddings)

terframes['congvectors']=terframes['MyReviews'].apply(embeddingFun)
terframes.head(5)
```

Out[ ]:

	reviews	Labels	MyReviews	gvectors	cvectors	congvectors
1163606	excellent pen actually pen like use write anyt...	0	[excellent, pen, actually, pen, like, use, wri...	[0.013108995225694444, -0.015750461154513888, ...	[0.013108995225694444, -0.015750461154513888, ...	[[-0.212890625, -0.004302978515625, -0.1806640...
2069284	fresh set battery nice strong beam unfortunate...	2	[fresh, set, battery, nice, strong, beam, unfo...	[0.035248976487379804, 0.09482046274038461, 0....	[0.035248976487379804, 0.09482046274038461, 0....	[[-0.042236328, 0.018066406, 0.22070312, -0.01...
338411	file folder label great use folder	0	[file, folder, label, great, use, folder]	[0.11321149553571429, 0.029767717633928572, -0...	[0.11321149553571429, 0.029767717633928572, -0...	[[0.28515625, 0.023193359375, -0.03173828125, ...
37088	work 's well heavy duty steel original stapler...	2	[work, 's, well, heavy, duty, steel, original,...	[0.07191051136363637, 0.07883522727272728, -0....	[0.07191051136363637, 0.07883522727272728, -0....	[[-0.075683594, 0.033691406, -0.064941406, 0.1...
752649	great quality great value	0	[great, quality, great, value]	[-0.00732421875, 0.141259765625, 0.03442382812...	[-0.00732421875, 0.141259765625, 0.03442382812...	[[0.07177734375, 0.2080078125, -0.028442382812...

Using Custom Word2Vec for Concatenanting Extracted Features in Ternary Data Set

Here we are preparing the concatenated vectors

In [ ]:

```python
#loading the word vectors from the previously custom trained model
```

```
wvcon = KeyedVectors.load("word2vec.wordvectors", mmap='r')
```

In [ ]:

```
#Extracting Word Embeddings from the Pre-trained Model

def embeddingFun(sent):
    vectorsize = new_model.vector_size
    CM_Embeddings_temp = np.zeros(vectorsize)
    CM_Embeddings = []
    c=0
    pad_value=0
    desired_length=300

    for word in sent:
        if word in wvcon:
            CM_Embeddings_temp=wvcon[word]
        if c in range(0,10):
            CM_Embeddings.append(CM_Embeddings_temp)
            c+=1

    CM_Embeddings = padding(CM_Embeddings, pad_value, desired_length)
    return np.array(CM_Embeddings)

terframes['concvectors']=terframes['MyReviews'].apply(embeddingFun)
terframes.head(5)
```

Out[ ]:

	reviews	Labels	MyReviews	gvectors	cvectors	congvectors
1163606	excellent pen actually pen like use write anyt...	0	[excellent, pen, actually, pen, like, use, wri...	[0.013108995225694444, -0.015750461154513888, ...	[0.013108995225694444, -0.015750461154513888, ...	[[-0.212890625, -0.004302978515625, -0.1806640...
2069284	fresh set battery nice strong beam unfortunate...	2	[fresh, set, battery, nice, strong, beam, unfo...	[0.035248976487379804, 0.09482046274038461, 0....	[0.035248976487379804, 0.09482046274038461, 0....	[[-0.042236328, 0.018066406, 0.22070312, -0.01...
338411	file folder label great use folder	0	[file, folder, label, great, use, folder]	[0.11321149553571429, 0.029767717633928572, -0...	[0.11321149553571429, 0.029767717633928572, -0...	[[0.28515625, 0.023193359375, -0.03173828125, ...
37088	work 's well heavy duty steel original stapler...	2	[work, 's, well, heavy, duty, steel, original,...	[0.07191051136363637, 0.07883522727272728, -0....	[0.07191051136363637, 0.07883522727272728, -0....	[[-0.075683594, 0.033691406, -0.064941406, 0.1...
752649	great quality great value	0	[great, quality, great, value]	[-0.00732421875, 0.141259765625, 0.03442382812...	[-0.00732421875, 0.141259765625, 0.03442382812...	[[0.07177734375, 0.2080078125, -0.028442382812...

FNN Using Features extracted from Pre-trained Avg Word2Vec Model

Here the Feedforward Neural Network is used for the Ternary Classification, where the feature extracter is the Pre-trained Gensim Word2Vec model and the vector for a review is the Average of the vectors for each word in the review.

In [ ]:

```
# split the data into 80-20 train-test
x_train, x_test, y_train, y_test = train_test_split(terframes['gvectors'], terframes['Labels'], test_size
```

In [ ]:

```
# Converting NumPy arrays to TensorFlow tensors
X_train_tf = tf.convert_to_tensor(x_train.to_list(), dtype=tf.float32)
```

```
y_train_tf = tf.convert_to_tensor(y_train, dtype=tf.int32)
X_test_tf = tf.convert_to_tensor(x_test.to_list(), dtype=tf.float32)
y_test_tf = tf.convert_to_tensor(y_test, dtype=tf.int32)
```

In [ ]:
```
neural_network_5 = tf.keras.models.Sequential([
    tf.keras.layers.Dense(50, activation='relu', input_shape=(300,)),
    tf.keras.layers.Dense(10, activation='relu'),
    tf.keras.layers.Dense(3, activation='softmax')
])
```

In [ ]:
```
# Compiling the neural network
neural_network_5.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

In [ ]:
```
# Training the neural network
history5 = neural_network_5.fit(X_train_tf, y_train_tf, epochs=100, batch_size=32)
```

```
Epoch 1/100
3750/3750 [==============================] - 2s 401us/step - loss: 0.8060 - accuracy: 0.6350
Epoch 2/100
3750/3750 [==============================] - 2s 402us/step - loss: 0.7695 - accuracy: 0.6550
Epoch 3/100
3750/3750 [==============================] - 2s 427us/step - loss: 0.7558 - accuracy: 0.6606
Epoch 4/100
3750/3750 [==============================] - 2s 402us/step - loss: 0.7461 - accuracy: 0.6652
Epoch 5/100
3750/3750 [==============================] - 2s 404us/step - loss: 0.7376 - accuracy: 0.6694
Epoch 6/100
3750/3750 [==============================] - 2s 404us/step - loss: 0.7301 - accuracy: 0.6719
Epoch 7/100
3750/3750 [==============================] - 2s 400us/step - loss: 0.7233 - accuracy: 0.6760
Epoch 8/100
3750/3750 [==============================] - 2s 414us/step - loss: 0.7175 - accuracy: 0.6799
Epoch 9/100
3750/3750 [==============================] - 2s 425us/step - loss: 0.7130 - accuracy: 0.6811
Epoch 10/100
3750/3750 [==============================] - 2s 404us/step - loss: 0.7080 - accuracy: 0.6842
Epoch 11/100
3750/3750 [==============================] - 1s 398us/step - loss: 0.7047 - accuracy: 0.6856
```

```
Epoch 12/100
3750/3750 [==============================] - 1s 399us/step - loss: 0.7010 - accuracy: 0.6877
Epoch 13/100
3750/3750 [==============================] - 2s 403us/step - loss: 0.6979 - accuracy: 0.6897
Epoch 14/100
3750/3750 [==============================] - 2s 426us/step - loss: 0.6953 - accuracy: 0.6910
Epoch 15/100
3750/3750 [==============================] - 2s 408us/step - loss: 0.6912 - accuracy: 0.6938
Epoch 16/100
3750/3750 [==============================] - 2s 405us/step - loss: 0.6888 - accuracy: 0.6939
Epoch 17/100
3750/3750 [==============================] - 2s 427us/step - loss: 0.6859 - accuracy: 0.6961
Epoch 18/100
3750/3750 [==============================] - 2s 410us/step - loss: 0.6837 - accuracy: 0.6962
Epoch 19/100
3750/3750 [==============================] - 2s 428us/step - loss: 0.6805 - accuracy: 0.6982
Epoch 20/100
3750/3750 [==============================] - 2s 408us/step - loss: 0.6784 - accuracy: 0.7006
Epoch 21/100
3750/3750 [==============================] - 2s 407us/step - loss: 0.6763 - accuracy: 0.7011
Epoch 22/100
3750/3750 [==============================] - 2s 441us/step - loss: 0.6745 - accuracy: 0.7022
Epoch 23/100
3750/3750 [==============================] - 1s 395us/step - loss: 0.6723 - accuracy: 0.7033
Epoch 24/100
3750/3750 [==============================] - 1s 391us/step - loss: 0.6696 - accuracy: 0.7039
Epoch 25/100
3750/3750 [==============================] - 1s 392us/step - loss: 0.6684 - accuracy: 0.7050
Epoch 26/100
3750/3750 [==============================] - 2s 431us/step - loss: 0.6670 - accuracy: 0.7062
Epoch 27/100
3750/3750 [==============================] - 1s 396us/step - loss: 0.6648 - accuracy: 0.7076
Epoch 28/100
3750/3750 [==============================] - 1s 396us/step - loss: 0.6635 - accuracy: 0.7071
Epoch 29/100
3750/3750 [==============================] - 2s 423us/step - loss: 0.6617 - accuracy: 0.7089
Epoch 30/100
3750/3750 [==============================] - 2s 407us/step - loss: 0.6594 - accuracy: 0.7094
Epoch 31/100
3750/3750 [==============================] - 2s 426us/step - loss: 0.6587 - accuracy: 0.7103
```

```
Epoch 32/100
3750/3750 [==============================] - 2s 405us/step - loss: 0.6568 - accuracy: 0.7118
Epoch 33/100
3750/3750 [==============================] - 2s 433us/step - loss: 0.6557 - accuracy: 0.7125
Epoch 34/100
3750/3750 [==============================] - 2s 425us/step - loss: 0.6549 - accuracy: 0.7127
Epoch 35/100
3750/3750 [==============================] - 2s 449us/step - loss: 0.6534 - accuracy: 0.7131
Epoch 36/100
3750/3750 [==============================] - 2s 406us/step - loss: 0.6519 - accuracy: 0.7135
Epoch 37/100
3750/3750 [==============================] - 2s 408us/step - loss: 0.6504 - accuracy: 0.7151
Epoch 38/100
3750/3750 [==============================] - 2s 427us/step - loss: 0.6495 - accuracy: 0.7154
Epoch 39/100
3750/3750 [==============================] - 1s 400us/step - loss: 0.6481 - accuracy: 0.7159
Epoch 40/100
3750/3750 [==============================] - 2s 402us/step - loss: 0.6470 - accuracy: 0.7162
Epoch 41/100
3750/3750 [==============================] - 2s 429us/step - loss: 0.6456 - accuracy: 0.7164
Epoch 42/100
3750/3750 [==============================] - 2s 406us/step - loss: 0.6448 - accuracy: 0.7182
Epoch 43/100
3750/3750 [==============================] - 2s 412us/step - loss: 0.6434 - accuracy: 0.7182
Epoch 44/100
3750/3750 [==============================] - 2s 420us/step - loss: 0.6430 - accuracy: 0.7180
Epoch 45/100
3750/3750 [==============================] - 2s 413us/step - loss: 0.6415 - accuracy: 0.7188
Epoch 46/100
3750/3750 [==============================] - 2s 402us/step - loss: 0.6405 - accuracy: 0.7199
Epoch 47/100
3750/3750 [==============================] - 2s 401us/step - loss: 0.6397 - accuracy: 0.7205
Epoch 48/100
3750/3750 [==============================] - 2s 406us/step - loss: 0.6389 - accuracy: 0.7205
Epoch 49/100
3750/3750 [==============================] - 2s 406us/step - loss: 0.6377 - accuracy: 0.7209
Epoch 50/100
3750/3750 [==============================] - 2s 423us/step - loss: 0.6372 - accuracy: 0.7211
Epoch 51/100
3750/3750 [==============================] - 1s 400us/step - loss: 0.6360 - accuracy: 0.7214
```

```
Epoch 52/100
3750/3750 [==============================] - 2s 406us/step - loss: 0.6351 - accuracy: 0.7228
Epoch 53/100
3750/3750 [==============================] - 2s 402us/step - loss: 0.6339 - accuracy: 0.7229
Epoch 54/100
3750/3750 [==============================] - 2s 423us/step - loss: 0.6337 - accuracy: 0.7230
Epoch 55/100
3750/3750 [==============================] - 2s 401us/step - loss: 0.6326 - accuracy: 0.7227
Epoch 56/100
3750/3750 [==============================] - 2s 401us/step - loss: 0.6321 - accuracy: 0.7243
Epoch 57/100
3750/3750 [==============================] - 2s 403us/step - loss: 0.6313 - accuracy: 0.7232
Epoch 58/100
3750/3750 [==============================] - 2s 400us/step - loss: 0.6308 - accuracy: 0.7251
Epoch 59/100
3750/3750 [==============================] - 2s 400us/step - loss: 0.6297 - accuracy: 0.7255
Epoch 60/100
3750/3750 [==============================] - 2s 434us/step - loss: 0.6294 - accuracy: 0.7249
Epoch 61/100
3750/3750 [==============================] - 2s 406us/step - loss: 0.6285 - accuracy: 0.7257
Epoch 62/100
3750/3750 [==============================] - 2s 408us/step - loss: 0.6279 - accuracy: 0.7248
Epoch 63/100
3750/3750 [==============================] - 2s 408us/step - loss: 0.6275 - accuracy: 0.7266
Epoch 64/100
3750/3750 [==============================] - 2s 427us/step - loss: 0.6262 - accuracy: 0.7268
Epoch 65/100
3750/3750 [==============================] - 2s 421us/step - loss: 0.6261 - accuracy: 0.7266
Epoch 66/100
3750/3750 [==============================] - 2s 404us/step - loss: 0.6248 - accuracy: 0.7263
Epoch 67/100
3750/3750 [==============================] - 2s 407us/step - loss: 0.6243 - accuracy: 0.7283
Epoch 68/100
3750/3750 [==============================] - 2s 429us/step - loss: 0.6240 - accuracy: 0.7277
Epoch 69/100
3750/3750 [==============================] - 2s 417us/step - loss: 0.6236 - accuracy: 0.7278
Epoch 70/100
3750/3750 [==============================] - 2s 418us/step - loss: 0.6226 - accuracy: 0.7296
Epoch 71/100
3750/3750 [==============================] - 2s 442us/step - loss: 0.6222 - accuracy: 0.7284
```

```
Epoch 72/100
3750/3750 [==============================] - 2s 411us/step - loss: 0.6218 - accuracy: 0.7289
Epoch 73/100
3750/3750 [==============================] - 2s 434us/step - loss: 0.6210 - accuracy: 0.7287
Epoch 74/100
3750/3750 [==============================] - 2s 410us/step - loss: 0.6208 - accuracy: 0.7300
Epoch 75/100
3750/3750 [==============================] - 2s 406us/step - loss: 0.6202 - accuracy: 0.7297
Epoch 76/100
3750/3750 [==============================] - 2s 406us/step - loss: 0.6196 - accuracy: 0.7291
Epoch 77/100
3750/3750 [==============================] - 2s 430us/step - loss: 0.6193 - accuracy: 0.7303
Epoch 78/100
3750/3750 [==============================] - 2s 413us/step - loss: 0.6185 - accuracy: 0.7316
Epoch 79/100
3750/3750 [==============================] - 2s 431us/step - loss: 0.6179 - accuracy: 0.7309
Epoch 80/100
3750/3750 [==============================] - 2s 405us/step - loss: 0.6175 - accuracy: 0.7303
Epoch 81/100
3750/3750 [==============================] - 2s 402us/step - loss: 0.6167 - accuracy: 0.7310
Epoch 82/100
3750/3750 [==============================] - 2s 429us/step - loss: 0.6171 - accuracy: 0.7313
Epoch 83/100
3750/3750 [==============================] - 2s 440us/step - loss: 0.6162 - accuracy: 0.7312
Epoch 84/100
3750/3750 [==============================] - 2s 413us/step - loss: 0.6164 - accuracy: 0.7306
Epoch 85/100
3750/3750 [==============================] - 2s 432us/step - loss: 0.6150 - accuracy: 0.7314
Epoch 86/100
3750/3750 [==============================] - 2s 406us/step - loss: 0.6146 - accuracy: 0.7307
Epoch 87/100
3750/3750 [==============================] - 2s 438us/step - loss: 0.6139 - accuracy: 0.7325
Epoch 88/100
3750/3750 [==============================] - 2s 430us/step - loss: 0.6139 - accuracy: 0.7324
Epoch 89/100
3750/3750 [==============================] - 2s 435us/step - loss: 0.6131 - accuracy: 0.7323
Epoch 90/100
3750/3750 [==============================] - 2s 409us/step - loss: 0.6128 - accuracy: 0.7329
Epoch 91/100
3750/3750 [==============================] - 2s 408us/step - loss: 0.6121 - accuracy: 0.7329
```

```
Epoch 92/100
3750/3750 [==============================] - 2s 431us/step - loss: 0.6116 - accuracy: 0.7331
Epoch 93/100
3750/3750 [==============================] - 2s 405us/step - loss: 0.6122 - accuracy: 0.7327
Epoch 94/100
3750/3750 [==============================] - 2s 411us/step - loss: 0.6116 - accuracy: 0.7341
Epoch 95/100
3750/3750 [==============================] - 2s 429us/step - loss: 0.6118 - accuracy: 0.7335
Epoch 96/100
3750/3750 [==============================] - 2s 405us/step - loss: 0.6106 - accuracy: 0.7339
Epoch 97/100
3750/3750 [==============================] - 2s 403us/step - loss: 0.6099 - accuracy: 0.7358
Epoch 98/100
3750/3750 [==============================] - 2s 434us/step - loss: 0.6104 - accuracy: 0.7343
Epoch 99/100
3750/3750 [==============================] - 2s 410us/step - loss: 0.6098 - accuracy: 0.7343
Epoch 100/100
3750/3750 [==============================] - 2s 428us/step - loss: 0.6092 - accuracy: 0.7341
```

In [ ]:
```
# Evaluating the neural network on the test set
test_loss, test_acc = neural_network_5.evaluate(X_test_tf, y_test_tf)
print('Test accuracy:', test_acc)
```

```
938/938 [==============================] - 0s 304us/step - loss: 0.8378 - accuracy: 0.6428
Test accuracy: 0.642799973487854
```

In [ ]:
```
Accuracy_Table[10].append(test_acc)
```

FNN Using Features extracted from Custom Avg Word2Vec Model

Here the Feedforward Neural Network is used for the Ternary Classification, where the feature extracter is the Custom Word2Vec model and the vector for a review is the Average of the vectors for each word in the review.

In [ ]:
```
# split the data into 80-20 train-test
x_train, x_test, y_train, y_test = train_test_split(terframes['cvectors'], terframes['Labels'], test_size
```

In [ ]:
```
# Converting NumPy arrays to TensorFlow tensors
X_train_tf = tf.convert_to_tensor(x_train.to_list(), dtype=tf.float32)
y_train_tf = tf.convert_to_tensor(y_train, dtype=tf.int32)
```

```python
X_test_tf = tf.convert_to_tensor(x_test.to_list(), dtype=tf.float32)
y_test_tf = tf.convert_to_tensor(y_test, dtype=tf.int32)
```

In [ ]:
```python
neural_network_6 = tf.keras.models.Sequential([
    tf.keras.layers.Dense(50, activation='relu', input_shape=(300,)),
    tf.keras.layers.Dense(10, activation='relu'),
    tf.keras.layers.Dense(3, activation='softmax')
])
```

In [ ]:
```python
# Compiling the neural network
neural_network_6.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

In [ ]:
```python
# Training the neural network
history6 = neural_network_6.fit(X_train_tf, y_train_tf, epochs=100, batch_size=32)
```

```
Epoch 1/100
3750/3750 [==============================] - 2s 410us/step - loss: 0.8057 - accuracy: 0.6346
Epoch 2/100
3750/3750 [==============================] - 2s 404us/step - loss: 0.7651 - accuracy: 0.6552
Epoch 3/100
3750/3750 [==============================] - 2s 424us/step - loss: 0.7518 - accuracy: 0.6618
Epoch 4/100
3750/3750 [==============================] - 2s 403us/step - loss: 0.7417 - accuracy: 0.6668
Epoch 5/100
3750/3750 [==============================] - 2s 414us/step - loss: 0.7334 - accuracy: 0.6703
Epoch 6/100
3750/3750 [==============================] - 2s 407us/step - loss: 0.7272 - accuracy: 0.6741
Epoch 7/100
3750/3750 [==============================] - 2s 403us/step - loss: 0.7215 - accuracy: 0.6763
Epoch 8/100
3750/3750 [==============================] - 2s 404us/step - loss: 0.7166 - accuracy: 0.6806
Epoch 9/100
3750/3750 [==============================] - 2s 439us/step - loss: 0.7120 - accuracy: 0.6804
Epoch 10/100
3750/3750 [==============================] - 2s 409us/step - loss: 0.7074 - accuracy: 0.6844
Epoch 11/100
3750/3750 [==============================] - 2s 406us/step - loss: 0.7032 - accuracy: 0.6871
Epoch 12/100
```

```
3750/3750 [==============================] - 2s 433us/step - loss: 0.6996 - accuracy: 0.6881
Epoch 13/100
3750/3750 [==============================] - 2s 412us/step - loss: 0.6962 - accuracy: 0.6896
Epoch 14/100
3750/3750 [==============================] - 2s 410us/step - loss: 0.6935 - accuracy: 0.6918
Epoch 15/100
3750/3750 [==============================] - 2s 429us/step - loss: 0.6898 - accuracy: 0.6932
Epoch 16/100
3750/3750 [==============================] - 2s 412us/step - loss: 0.6868 - accuracy: 0.6950
Epoch 17/100
3750/3750 [==============================] - 2s 410us/step - loss: 0.6838 - accuracy: 0.6959
Epoch 18/100
3750/3750 [==============================] - 2s 411us/step - loss: 0.6817 - accuracy: 0.6984
Epoch 19/100
3750/3750 [==============================] - 2s 410us/step - loss: 0.6790 - accuracy: 0.6987
Epoch 20/100
3750/3750 [==============================] - 2s 425us/step - loss: 0.6769 - accuracy: 0.7005
Epoch 21/100
3750/3750 [==============================] - 2s 412us/step - loss: 0.6746 - accuracy: 0.7005
Epoch 22/100
3750/3750 [==============================] - 2s 412us/step - loss: 0.6722 - accuracy: 0.7025
Epoch 23/100
3750/3750 [==============================] - 2s 410us/step - loss: 0.6708 - accuracy: 0.7028
Epoch 24/100
3750/3750 [==============================] - 2s 424us/step - loss: 0.6687 - accuracy: 0.7043
Epoch 25/100
3750/3750 [==============================] - 2s 403us/step - loss: 0.6667 - accuracy: 0.7050
Epoch 26/100
3750/3750 [==============================] - 2s 405us/step - loss: 0.6650 - accuracy: 0.7067
Epoch 27/100
3750/3750 [==============================] - 2s 405us/step - loss: 0.6630 - accuracy: 0.7082
Epoch 28/100
3750/3750 [==============================] - 2s 405us/step - loss: 0.6621 - accuracy: 0.7076
Epoch 29/100
3750/3750 [==============================] - 2s 404us/step - loss: 0.6603 - accuracy: 0.7084
Epoch 30/100
3750/3750 [==============================] - 2s 404us/step - loss: 0.6586 - accuracy: 0.7096
Epoch 31/100
3750/3750 [==============================] - 2s 423us/step - loss: 0.6570 - accuracy: 0.7106
Epoch 32/100
```

```
3750/3750 [==============================] - 2s 406us/step - loss: 0.6555 - accuracy: 0.7111
Epoch 33/100
3750/3750 [==============================] - 2s 404us/step - loss: 0.6550 - accuracy: 0.7109
Epoch 34/100
3750/3750 [==============================] - 2s 404us/step - loss: 0.6526 - accuracy: 0.7120
Epoch 35/100
3750/3750 [==============================] - 2s 403us/step - loss: 0.6521 - accuracy: 0.7128
Epoch 36/100
3750/3750 [==============================] - 2s 406us/step - loss: 0.6504 - accuracy: 0.7126
Epoch 37/100
3750/3750 [==============================] - 2s 407us/step - loss: 0.6493 - accuracy: 0.7136
Epoch 38/100
3750/3750 [==============================] - 2s 424us/step - loss: 0.6479 - accuracy: 0.7144
Epoch 39/100
3750/3750 [==============================] - 2s 404us/step - loss: 0.6469 - accuracy: 0.7152
Epoch 40/100
3750/3750 [==============================] - 2s 421us/step - loss: 0.6457 - accuracy: 0.7154
Epoch 41/100
3750/3750 [==============================] - 2s 404us/step - loss: 0.6450 - accuracy: 0.7159
Epoch 42/100
3750/3750 [==============================] - 2s 410us/step - loss: 0.6446 - accuracy: 0.7158
Epoch 43/100
3750/3750 [==============================] - 2s 434us/step - loss: 0.6432 - accuracy: 0.7167
Epoch 44/100
3750/3750 [==============================] - 2s 409us/step - loss: 0.6415 - accuracy: 0.7184
Epoch 45/100
3750/3750 [==============================] - 2s 408us/step - loss: 0.6410 - accuracy: 0.7179
Epoch 46/100
3750/3750 [==============================] - 2s 405us/step - loss: 0.6402 - accuracy: 0.7187
Epoch 47/100
3750/3750 [==============================] - 2s 426us/step - loss: 0.6394 - accuracy: 0.7191
Epoch 48/100
3750/3750 [==============================] - 2s 408us/step - loss: 0.6380 - accuracy: 0.7191
Epoch 49/100
3750/3750 [==============================] - 2s 410us/step - loss: 0.6379 - accuracy: 0.7191
Epoch 50/100
3750/3750 [==============================] - 2s 406us/step - loss: 0.6366 - accuracy: 0.7196
Epoch 51/100
3750/3750 [==============================] - 2s 430us/step - loss: 0.6363 - accuracy: 0.7204
Epoch 52/100
```

```
3750/3750 [==============================] - 2s 413us/step - loss: 0.6348 - accuracy: 0.7211
Epoch 53/100
3750/3750 [==============================] - 2s 409us/step - loss: 0.6343 - accuracy: 0.7208
Epoch 54/100
3750/3750 [==============================] - 2s 405us/step - loss: 0.6335 - accuracy: 0.7221
Epoch 55/100
3750/3750 [==============================] - 2s 426us/step - loss: 0.6329 - accuracy: 0.7228
Epoch 56/100
3750/3750 [==============================] - 2s 407us/step - loss: 0.6321 - accuracy: 0.7224
Epoch 57/100
3750/3750 [==============================] - 2s 425us/step - loss: 0.6317 - accuracy: 0.7226
Epoch 58/100
3750/3750 [==============================] - 2s 443us/step - loss: 0.6306 - accuracy: 0.7237
Epoch 59/100
3750/3750 [==============================] - 2s 408us/step - loss: 0.6308 - accuracy: 0.7229
Epoch 60/100
3750/3750 [==============================] - 2s 401us/step - loss: 0.6299 - accuracy: 0.7240
Epoch 61/100
3750/3750 [==============================] - 2s 408us/step - loss: 0.6293 - accuracy: 0.7245
Epoch 62/100
3750/3750 [==============================] - 2s 407us/step - loss: 0.6282 - accuracy: 0.7243
Epoch 63/100
3750/3750 [==============================] - 2s 429us/step - loss: 0.6276 - accuracy: 0.7252
Epoch 64/100
3750/3750 [==============================] - 2s 406us/step - loss: 0.6270 - accuracy: 0.7245
Epoch 65/100
3750/3750 [==============================] - 2s 412us/step - loss: 0.6262 - accuracy: 0.7259
Epoch 66/100
3750/3750 [==============================] - 2s 431us/step - loss: 0.6260 - accuracy: 0.7254
Epoch 67/100
3750/3750 [==============================] - 2s 425us/step - loss: 0.6251 - accuracy: 0.7258
Epoch 68/100
3750/3750 [==============================] - 2s 462us/step - loss: 0.6242 - accuracy: 0.7263
Epoch 69/100
3750/3750 [==============================] - 2s 424us/step - loss: 0.6245 - accuracy: 0.7257
Epoch 70/100
3750/3750 [==============================] - 2s 413us/step - loss: 0.6240 - accuracy: 0.7270
Epoch 71/100
3750/3750 [==============================] - 2s 416us/step - loss: 0.6232 - accuracy: 0.7264
Epoch 72/100
```

```
3750/3750 [==============================] - 2s 435us/step - loss: 0.6231 - accuracy: 0.7271
Epoch 73/100
3750/3750 [==============================] - 2s 408us/step - loss: 0.6222 - accuracy: 0.7268
Epoch 74/100
3750/3750 [==============================] - 2s 409us/step - loss: 0.6216 - accuracy: 0.7275
Epoch 75/100
3750/3750 [==============================] - 2s 409us/step - loss: 0.6210 - accuracy: 0.7272
Epoch 76/100
3750/3750 [==============================] - 2s 409us/step - loss: 0.6205 - accuracy: 0.7287
Epoch 77/100
3750/3750 [==============================] - 2s 432us/step - loss: 0.6205 - accuracy: 0.7287
Epoch 78/100
3750/3750 [==============================] - 2s 414us/step - loss: 0.6196 - accuracy: 0.7293
Epoch 79/100
3750/3750 [==============================] - 2s 408us/step - loss: 0.6190 - accuracy: 0.7292
Epoch 80/100
3750/3750 [==============================] - 2s 408us/step - loss: 0.6192 - accuracy: 0.7294
Epoch 81/100
3750/3750 [==============================] - 2s 427us/step - loss: 0.6182 - accuracy: 0.7290
Epoch 82/100
3750/3750 [==============================] - 2s 408us/step - loss: 0.6178 - accuracy: 0.7297
Epoch 83/100
3750/3750 [==============================] - 2s 433us/step - loss: 0.6167 - accuracy: 0.7299
Epoch 84/100
3750/3750 [==============================] - 2s 412us/step - loss: 0.6168 - accuracy: 0.7307
Epoch 85/100
3750/3750 [==============================] - 2s 409us/step - loss: 0.6165 - accuracy: 0.7303
Epoch 86/100
3750/3750 [==============================] - 2s 409us/step - loss: 0.6159 - accuracy: 0.7301
Epoch 87/100
3750/3750 [==============================] - 2s 425us/step - loss: 0.6158 - accuracy: 0.7307
Epoch 88/100
3750/3750 [==============================] - 2s 410us/step - loss: 0.6156 - accuracy: 0.7309
Epoch 89/100
3750/3750 [==============================] - 2s 404us/step - loss: 0.6149 - accuracy: 0.7302
Epoch 90/100
3750/3750 [==============================] - 2s 415us/step - loss: 0.6143 - accuracy: 0.7305
Epoch 91/100
3750/3750 [==============================] - 2s 463us/step - loss: 0.6135 - accuracy: 0.7318
Epoch 92/100
```

```
3750/3750 [==============================] - 2s 419us/step - loss: 0.6140 - accuracy: 0.7311
Epoch 93/100
3750/3750 [==============================] - 2s 415us/step - loss: 0.6134 - accuracy: 0.7321
Epoch 94/100
3750/3750 [==============================] - 2s 412us/step - loss: 0.6124 - accuracy: 0.7321
Epoch 95/100
3750/3750 [==============================] - 2s 425us/step - loss: 0.6127 - accuracy: 0.7324
Epoch 96/100
3750/3750 [==============================] - 2s 405us/step - loss: 0.6120 - accuracy: 0.7314
Epoch 97/100
3750/3750 [==============================] - 2s 405us/step - loss: 0.6124 - accuracy: 0.7327
Epoch 98/100
3750/3750 [==============================] - 2s 403us/step - loss: 0.6114 - accuracy: 0.7331
Epoch 99/100
3750/3750 [==============================] - 2s 413us/step - loss: 0.6111 - accuracy: 0.7329
Epoch 100/100
3750/3750 [==============================] - 2s 424us/step - loss: 0.6102 - accuracy: 0.7332
```

In [ ]:
```
# Evaluating the neural network on the test set
test_loss, test_acc = neural_network_6.evaluate(X_test_tf, y_test_tf)
print('Test accuracy:', test_acc)
```

```
938/938 [==============================] - 0s 309us/step - loss: 0.8307 - accuracy: 0.6476
Test accuracy: 0.6476333141326904
```

In [ ]:
```
Accuracy_Table[11].append(test_acc)
```

FNN Using Features extracted from Pre-trained Concatenated Word2Vec Model For Ternary Data

Here the Feedforward Neural Network is used for the Ternary Classification, where the feature extracter is the Pre-trained Gensim Word2Vec model and the vector for a review is the concatenation of the vectors for each word in the review.

In [ ]:
```
# split the data into 80-20 train-test
x_train, x_test, y_train, y_test = train_test_split(terframes['congvectors'], terframes['Labels'], test_s
```

In [ ]:
```
# Converting NumPy arrays to TensorFlow tensors
X_train_tf = tf.convert_to_tensor(x_train.to_list(), dtype=tf.float32)
```

```
y_train_tf = tf.convert_to_tensor(y_train, dtype=tf.int32)
X_test_tf = tf.convert_to_tensor(x_test.to_list(), dtype=tf.float32)
y_test_tf = tf.convert_to_tensor(y_test, dtype=tf.int32)
```

In [ ]:
```
# Flattening the tensor into a single feature vector
x_train_tf_flat = tf.reshape(X_train_tf, (X_train_tf.shape[0],-1))
X_test_tf_flat = tf.reshape(X_test_tf, (X_test_tf.shape[0],-1))
```

In [ ]:
```
neural_network_7 = tf.keras.models.Sequential([
    tf.keras.layers.Dense(50, activation='relu', input_shape=((3000),)),
    tf.keras.layers.Dense(10, activation='relu'),
    tf.keras.layers.Dense(3, activation='softmax')
])
```

In [ ]:
```
# Compiling the neural network
neural_network_7.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

In [ ]:
```
# Training the neural network
history7 = neural_network_7.fit(x_train_tf_flat, y_train_tf, epochs=100, batch_size=32)
```

```
Epoch 1/100
3750/3750 [==============================] - 3s 697us/step - loss: 0.8927 - accuracy: 0.5740
Epoch 2/100
3750/3750 [==============================] - 2s 637us/step - loss: 0.8281 - accuracy: 0.6147
Epoch 3/100
3750/3750 [==============================] - 2s 603us/step - loss: 0.7793 - accuracy: 0.6441
Epoch 4/100
3750/3750 [==============================] - 2s 627us/step - loss: 0.7219 - accuracy: 0.6770
Epoch 5/100
3750/3750 [==============================] - 2s 656us/step - loss: 0.6634 - accuracy: 0.7098
Epoch 6/100
3750/3750 [==============================] - 2s 645us/step - loss: 0.6048 - accuracy: 0.7417
Epoch 7/100
3750/3750 [==============================] - 3s 667us/step - loss: 0.5534 - accuracy: 0.7673
Epoch 8/100
3750/3750 [==============================] - 2s 634us/step - loss: 0.5072 - accuracy: 0.7896
Epoch 9/100
```

```
3750/3750 [==============================] - 3s 680us/step - loss: 0.4661 - accuracy: 0.8097
Epoch 10/100
3750/3750 [==============================] - 2s 641us/step - loss: 0.4313 - accuracy: 0.8257
Epoch 11/100
3750/3750 [==============================] - 3s 707us/step - loss: 0.3970 - accuracy: 0.8407
Epoch 12/100
3750/3750 [==============================] - 2s 660us/step - loss: 0.3695 - accuracy: 0.8530
Epoch 13/100
3750/3750 [==============================] - 2s 637us/step - loss: 0.3433 - accuracy: 0.8654
Epoch 14/100
3750/3750 [==============================] - 3s 690us/step - loss: 0.3238 - accuracy: 0.8724
Epoch 15/100
3750/3750 [==============================] - 2s 639us/step - loss: 0.3042 - accuracy: 0.8810
Epoch 16/100
3750/3750 [==============================] - 3s 686us/step - loss: 0.2883 - accuracy: 0.8866
Epoch 17/100
3750/3750 [==============================] - 2s 649us/step - loss: 0.2723 - accuracy: 0.8944
Epoch 18/100
3750/3750 [==============================] - 3s 694us/step - loss: 0.2598 - accuracy: 0.8994
Epoch 19/100
3750/3750 [==============================] - 2s 611us/step - loss: 0.2505 - accuracy: 0.9032
Epoch 20/100
3750/3750 [==============================] - 2s 645us/step - loss: 0.2396 - accuracy: 0.9071
Epoch 21/100
3750/3750 [==============================] - 2s 650us/step - loss: 0.2280 - accuracy: 0.9118
Epoch 22/100
3750/3750 [==============================] - 3s 676us/step - loss: 0.2230 - accuracy: 0.9138
Epoch 23/100
3750/3750 [==============================] - 2s 645us/step - loss: 0.2124 - accuracy: 0.9181
Epoch 24/100
3750/3750 [==============================] - 2s 665us/step - loss: 0.2059 - accuracy: 0.9205
Epoch 25/100
3750/3750 [==============================] - 2s 636us/step - loss: 0.1982 - accuracy: 0.9244
Epoch 26/100
3750/3750 [==============================] - 3s 668us/step - loss: 0.1934 - accuracy: 0.9260
Epoch 27/100
3750/3750 [==============================] - 2s 661us/step - loss: 0.1885 - accuracy: 0.9280
Epoch 28/100
3750/3750 [==============================] - 3s 814us/step - loss: 0.1840 - accuracy: 0.9296
Epoch 29/100
```

```
3750/3750 [==============================] - 2s 666us/step - loss: 0.1762 - accuracy: 0.9326
Epoch 30/100
3750/3750 [==============================] - 3s 753us/step - loss: 0.1730 - accuracy: 0.9342
Epoch 31/100
3750/3750 [==============================] - 2s 643us/step - loss: 0.1698 - accuracy: 0.9356
Epoch 32/100
3750/3750 [==============================] - 2s 605us/step - loss: 0.1637 - accuracy: 0.9378
Epoch 33/100
3750/3750 [==============================] - 2s 612us/step - loss: 0.1632 - accuracy: 0.9380
Epoch 34/100
3750/3750 [==============================] - 3s 684us/step - loss: 0.1594 - accuracy: 0.9397
Epoch 35/100
3750/3750 [==============================] - 3s 671us/step - loss: 0.1553 - accuracy: 0.9415
Epoch 36/100
3750/3750 [==============================] - 2s 641us/step - loss: 0.1507 - accuracy: 0.9430
Epoch 37/100
3750/3750 [==============================] - 2s 618us/step - loss: 0.1497 - accuracy: 0.9442
Epoch 38/100
3750/3750 [==============================] - 3s 691us/step - loss: 0.1448 - accuracy: 0.9451
Epoch 39/100
3750/3750 [==============================] - 2s 613us/step - loss: 0.1436 - accuracy: 0.9459
Epoch 40/100
3750/3750 [==============================] - 2s 615us/step - loss: 0.1402 - accuracy: 0.9467
Epoch 41/100
3750/3750 [==============================] - 2s 621us/step - loss: 0.1373 - accuracy: 0.9482
Epoch 42/100
3750/3750 [==============================] - 2s 657us/step - loss: 0.1363 - accuracy: 0.9486
Epoch 43/100
3750/3750 [==============================] - 2s 622us/step - loss: 0.1334 - accuracy: 0.9496
Epoch 44/100
3750/3750 [==============================] - 2s 639us/step - loss: 0.1319 - accuracy: 0.9504
Epoch 45/100
3750/3750 [==============================] - 3s 708us/step - loss: 0.1289 - accuracy: 0.9515
Epoch 46/100
3750/3750 [==============================] - 2s 656us/step - loss: 0.1273 - accuracy: 0.9517
Epoch 47/100
3750/3750 [==============================] - 3s 744us/step - loss: 0.1240 - accuracy: 0.9538
Epoch 48/100
3750/3750 [==============================] - 3s 700us/step - loss: 0.1252 - accuracy: 0.9530
Epoch 49/100
```

```
3750/3750 [==============================] - 2s 661us/step - loss: 0.1220 - accuracy: 0.9546
Epoch 50/100
3750/3750 [==============================] - 2s 663us/step - loss: 0.1197 - accuracy: 0.9554
Epoch 51/100
3750/3750 [==============================] - 2s 665us/step - loss: 0.1177 - accuracy: 0.9554
Epoch 52/100
3750/3750 [==============================] - 3s 689us/step - loss: 0.1148 - accuracy: 0.9567
Epoch 53/100
3750/3750 [==============================] - 3s 667us/step - loss: 0.1160 - accuracy: 0.9561
Epoch 54/100
3750/3750 [==============================] - 2s 650us/step - loss: 0.1145 - accuracy: 0.9569
Epoch 55/100
3750/3750 [==============================] - 2s 631us/step - loss: 0.1132 - accuracy: 0.9575
Epoch 56/100
3750/3750 [==============================] - 2s 652us/step - loss: 0.1108 - accuracy: 0.9583
Epoch 57/100
3750/3750 [==============================] - 2s 605us/step - loss: 0.1105 - accuracy: 0.9588
Epoch 58/100
3750/3750 [==============================] - 2s 620us/step - loss: 0.1097 - accuracy: 0.9590
Epoch 59/100
3750/3750 [==============================] - 2s 624us/step - loss: 0.1068 - accuracy: 0.9603
Epoch 60/100
3750/3750 [==============================] - 2s 639us/step - loss: 0.1079 - accuracy: 0.9601
Epoch 61/100
3750/3750 [==============================] - 2s 612us/step - loss: 0.1025 - accuracy: 0.9609
Epoch 62/100
3750/3750 [==============================] - 2s 626us/step - loss: 0.1035 - accuracy: 0.9610
Epoch 63/100
3750/3750 [==============================] - 2s 613us/step - loss: 0.1034 - accuracy: 0.9616
Epoch 64/100
3750/3750 [==============================] - 2s 638us/step - loss: 0.1028 - accuracy: 0.9608
Epoch 65/100
3750/3750 [==============================] - 2s 650us/step - loss: 0.1017 - accuracy: 0.9620
Epoch 66/100
3750/3750 [==============================] - 3s 727us/step - loss: 0.0980 - accuracy: 0.9636
Epoch 67/100
3750/3750 [==============================] - 3s 788us/step - loss: 0.1006 - accuracy: 0.9624
Epoch 68/100
3750/3750 [==============================] - 3s 740us/step - loss: 0.0963 - accuracy: 0.9639
Epoch 69/100
```

```
3750/3750 [==============================] - 2s 663us/step - loss: 0.0968 - accuracy: 0.9638
Epoch 70/100
3750/3750 [==============================] - 2s 630us/step - loss: 0.0958 - accuracy: 0.9645
Epoch 71/100
3750/3750 [==============================] - 2s 636us/step - loss: 0.0947 - accuracy: 0.9639
Epoch 72/100
3750/3750 [==============================] - 2s 644us/step - loss: 0.0932 - accuracy: 0.9656
Epoch 73/100
3750/3750 [==============================] - 2s 642us/step - loss: 0.0944 - accuracy: 0.9647
Epoch 74/100
3750/3750 [==============================] - 2s 613us/step - loss: 0.0919 - accuracy: 0.9657
Epoch 75/100
3750/3750 [==============================] - 2s 649us/step - loss: 0.0908 - accuracy: 0.9660
Epoch 76/100
3750/3750 [==============================] - 2s 613us/step - loss: 0.0917 - accuracy: 0.9659
Epoch 77/100
3750/3750 [==============================] - 2s 621us/step - loss: 0.0896 - accuracy: 0.9670
Epoch 78/100
3750/3750 [==============================] - 2s 654us/step - loss: 0.0911 - accuracy: 0.9661
Epoch 79/100
3750/3750 [==============================] - 2s 637us/step - loss: 0.0881 - accuracy: 0.9672
Epoch 80/100
3750/3750 [==============================] - 2s 605us/step - loss: 0.0899 - accuracy: 0.9662
Epoch 81/100
3750/3750 [==============================] - 2s 643us/step - loss: 0.0887 - accuracy: 0.9674
Epoch 82/100
3750/3750 [==============================] - 2s 642us/step - loss: 0.0874 - accuracy: 0.9678
Epoch 83/100
3750/3750 [==============================] - 2s 615us/step - loss: 0.0855 - accuracy: 0.9679
Epoch 84/100
3750/3750 [==============================] - 2s 599us/step - loss: 0.0857 - accuracy: 0.9684
Epoch 85/100
3750/3750 [==============================] - 2s 638us/step - loss: 0.0857 - accuracy: 0.9680
Epoch 86/100
3750/3750 [==============================] - 2s 611us/step - loss: 0.0835 - accuracy: 0.9690
Epoch 87/100
3750/3750 [==============================] - 3s 711us/step - loss: 0.0851 - accuracy: 0.9689
Epoch 88/100
3750/3750 [==============================] - 3s 779us/step - loss: 0.0826 - accuracy: 0.9693
Epoch 89/100
```

```
3750/3750 [==============================] - 2s 661us/step - loss: 0.0838 - accuracy: 0.9696
Epoch 90/100
3750/3750 [==============================] - 2s 666us/step - loss: 0.0812 - accuracy: 0.9692
Epoch 91/100
3750/3750 [==============================] - 2s 666us/step - loss: 0.0815 - accuracy: 0.9695
Epoch 92/100
3750/3750 [==============================] - 2s 662us/step - loss: 0.0801 - accuracy: 0.9701
Epoch 93/100
3750/3750 [==============================] - 3s 695us/step - loss: 0.0802 - accuracy: 0.9704
Epoch 94/100
3750/3750 [==============================] - 3s 678us/step - loss: 0.0809 - accuracy: 0.9697
Epoch 95/100
3750/3750 [==============================] - 3s 821us/step - loss: 0.0778 - accuracy: 0.9710
Epoch 96/100
3750/3750 [==============================] - 3s 694us/step - loss: 0.0797 - accuracy: 0.9708
Epoch 97/100
3750/3750 [==============================] - 3s 696us/step - loss: 0.0779 - accuracy: 0.9705
Epoch 98/100
3750/3750 [==============================] - 2s 648us/step - loss: 0.0767 - accuracy: 0.9710
Epoch 99/100
3750/3750 [==============================] - 3s 739us/step - loss: 0.0751 - accuracy: 0.9717
Epoch 100/100
3750/3750 [==============================] - 3s 823us/step - loss: 0.0789 - accuracy: 0.9712
```

In [ ]:
```
# Evaluating the neural network on the test set
test_loss, test_acc = neural_network_7.evaluate(X_test_tf_flat, y_test_tf)
print('Test accuracy:', test_acc)
```

```
938/938 [==============================] - 1s 530us/step - loss: 6.6856 - accuracy: 0.5260
Test accuracy: 0.526033341884613
```

In [ ]:
```
Accuracy_Table[12].append(test_acc)
```

FNN Using Features extracted from Custom Concatenated Word2Vec Model For Ternary Data

Here the Feedforward Neural Network is used for the Ternary Classification, where the feature extracter is the Custom Word2Vec model and the vector for a review is the concatenation of the vectors for each word in the review.

In [ ]:
```
# split the data into 80-20 train-test
x_train, x_test, y_train, y_test = train_test_split(terframes['concvectors'], terframes['Labels'], test_s
```

In [ ]:
```
# Converting NumPy arrays to TensorFlow tensors
X_train_tf = tf.convert_to_tensor(x_train.to_list(), dtype=tf.float32)
y_train_tf = tf.convert_to_tensor(y_train, dtype=tf.int32)
X_test_tf = tf.convert_to_tensor(x_test.to_list(), dtype=tf.float32)
y_test_tf = tf.convert_to_tensor(y_test, dtype=tf.int32)
```

In [ ]:
```
# Flattening the tensor into a single feature vector
x_train_tf_flat = tf.reshape(X_train_tf, (X_train_tf.shape[0],-1))
X_test_tf_flat = tf.reshape(X_test_tf, (X_test_tf.shape[0],-1))
```

In [ ]:
```
neural_network_8 = tf.keras.models.Sequential([
    tf.keras.layers.Dense(50, activation='relu', input_shape=((3000),)),
    tf.keras.layers.Dense(10, activation='relu'),
    tf.keras.layers.Dense(3, activation='softmax')
])
```

In [ ]:
```
# Compiling the neural network
neural_network_8.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

In [ ]:
```
# Training the neural network
history8 = neural_network_8.fit(x_train_tf_flat, y_train_tf, epochs=100, batch_size=32)
```

```
Epoch 1/100
3750/3750 [==============================] - 3s 718us/step - loss: 0.8425 - accuracy: 0.6041
Epoch 2/100
3750/3750 [==============================] - 2s 651us/step - loss: 0.7878 - accuracy: 0.6377
Epoch 3/100
3750/3750 [==============================] - 3s 694us/step - loss: 0.7412 - accuracy: 0.6631
Epoch 4/100
3750/3750 [==============================] - 3s 713us/step - loss: 0.6828 - accuracy: 0.6966
Epoch 5/100
3750/3750 [==============================] - 3s 705us/step - loss: 0.6206 - accuracy: 0.7305
Epoch 6/100
3750/3750 [==============================] - 3s 737us/step - loss: 0.5612 - accuracy: 0.7601
```

```
Epoch 7/100
3750/3750 [==============================] - 3s 713us/step - loss: 0.5063 - accuracy: 0.7871
Epoch 8/100
3750/3750 [==============================] - 3s 670us/step - loss: 0.4588 - accuracy: 0.8099
Epoch 9/100
3750/3750 [==============================] - 3s 770us/step - loss: 0.4176 - accuracy: 0.8284
Epoch 10/100
3750/3750 [==============================] - 3s 783us/step - loss: 0.3816 - accuracy: 0.8443
Epoch 11/100
3750/3750 [==============================] - 3s 714us/step - loss: 0.3512 - accuracy: 0.8584
Epoch 12/100
3750/3750 [==============================] - 3s 717us/step - loss: 0.3242 - accuracy: 0.8685
Epoch 13/100
3750/3750 [==============================] - 3s 708us/step - loss: 0.3019 - accuracy: 0.8801
Epoch 14/100
3750/3750 [==============================] - 3s 673us/step - loss: 0.2794 - accuracy: 0.8887
Epoch 15/100
3750/3750 [==============================] - 2s 656us/step - loss: 0.2646 - accuracy: 0.8957
Epoch 16/100
3750/3750 [==============================] - 3s 835us/step - loss: 0.2478 - accuracy: 0.9032
Epoch 17/100
3750/3750 [==============================] - 3s 682us/step - loss: 0.2356 - accuracy: 0.9087
Epoch 18/100
3750/3750 [==============================] - 3s 692us/step - loss: 0.2226 - accuracy: 0.9132
Epoch 19/100
3750/3750 [==============================] - 2s 662us/step - loss: 0.2146 - accuracy: 0.9165
Epoch 20/100
3750/3750 [==============================] - 3s 690us/step - loss: 0.2018 - accuracy: 0.9228
Epoch 21/100
3750/3750 [==============================] - 3s 806us/step - loss: 0.1951 - accuracy: 0.9250
Epoch 22/100
3750/3750 [==============================] - 3s 709us/step - loss: 0.1862 - accuracy: 0.9281
Epoch 23/100
3750/3750 [==============================] - 3s 733us/step - loss: 0.1785 - accuracy: 0.9322
Epoch 24/100
3750/3750 [==============================] - 3s 690us/step - loss: 0.1755 - accuracy: 0.9327
Epoch 25/100
3750/3750 [==============================] - 3s 691us/step - loss: 0.1675 - accuracy: 0.9359
Epoch 26/100
3750/3750 [==============================] - 3s 676us/step - loss: 0.1618 - accuracy: 0.9382
```

```
Epoch 27/100
3750/3750 [==============================] - 2s 655us/step - loss: 0.1585 - accuracy: 0.9401
Epoch 28/100
3750/3750 [==============================] - 3s 677us/step - loss: 0.1517 - accuracy: 0.9428
Epoch 29/100
3750/3750 [==============================] - 3s 722us/step - loss: 0.1498 - accuracy: 0.9432
Epoch 30/100
3750/3750 [==============================] - 3s 750us/step - loss: 0.1439 - accuracy: 0.9459
Epoch 31/100
3750/3750 [==============================] - 3s 675us/step - loss: 0.1413 - accuracy: 0.9470
Epoch 32/100
3750/3750 [==============================] - 3s 702us/step - loss: 0.1400 - accuracy: 0.9478
Epoch 33/100
3750/3750 [==============================] - 3s 719us/step - loss: 0.1365 - accuracy: 0.9490
Epoch 34/100
3750/3750 [==============================] - 3s 706us/step - loss: 0.1312 - accuracy: 0.9518
Epoch 35/100
3750/3750 [==============================] - 3s 672us/step - loss: 0.1290 - accuracy: 0.9517
Epoch 36/100
3750/3750 [==============================] - 3s 815us/step - loss: 0.1267 - accuracy: 0.9532
Epoch 37/100
3750/3750 [==============================] - 3s 683us/step - loss: 0.1241 - accuracy: 0.9541
Epoch 38/100
3750/3750 [==============================] - 2s 658us/step - loss: 0.1243 - accuracy: 0.9542
Epoch 39/100
3750/3750 [==============================] - 3s 698us/step - loss: 0.1198 - accuracy: 0.9561
Epoch 40/100
3750/3750 [==============================] - 2s 653us/step - loss: 0.1178 - accuracy: 0.9569
Epoch 41/100
3750/3750 [==============================] - 2s 644us/step - loss: 0.1167 - accuracy: 0.9575
Epoch 42/100
3750/3750 [==============================] - 3s 676us/step - loss: 0.1145 - accuracy: 0.9580
Epoch 43/100
3750/3750 [==============================] - 3s 676us/step - loss: 0.1105 - accuracy: 0.9594
Epoch 44/100
3750/3750 [==============================] - 3s 691us/step - loss: 0.1102 - accuracy: 0.9596
Epoch 45/100
3750/3750 [==============================] - 3s 791us/step - loss: 0.1086 - accuracy: 0.9606
Epoch 46/100
3750/3750 [==============================] - 3s 684us/step - loss: 0.1057 - accuracy: 0.9617
```

```
Epoch 47/100
3750/3750 [==============================] - 3s 813us/step - loss: 0.1056 - accuracy: 0.9612
Epoch 48/100
3750/3750 [==============================] - 3s 756us/step - loss: 0.1033 - accuracy: 0.9618
Epoch 49/100
3750/3750 [==============================] - 3s 679us/step - loss: 0.1023 - accuracy: 0.9621
Epoch 50/100
3750/3750 [==============================] - 3s 670us/step - loss: 0.0991 - accuracy: 0.9636
Epoch 51/100
3750/3750 [==============================] - 3s 683us/step - loss: 0.1017 - accuracy: 0.9634
Epoch 52/100
3750/3750 [==============================] - 3s 683us/step - loss: 0.0979 - accuracy: 0.9646
Epoch 53/100
3750/3750 [==============================] - 3s 688us/step - loss: 0.0976 - accuracy: 0.9649
Epoch 54/100
3750/3750 [==============================] - 3s 736us/step - loss: 0.0973 - accuracy: 0.9642
Epoch 55/100
3750/3750 [==============================] - 3s 736us/step - loss: 0.0941 - accuracy: 0.9656
Epoch 56/100
3750/3750 [==============================] - 3s 669us/step - loss: 0.0944 - accuracy: 0.9661
Epoch 57/100
3750/3750 [==============================] - 3s 838us/step - loss: 0.0919 - accuracy: 0.9668
Epoch 58/100
3750/3750 [==============================] - 3s 696us/step - loss: 0.0919 - accuracy: 0.9667
Epoch 59/100
3750/3750 [==============================] - 3s 704us/step - loss: 0.0910 - accuracy: 0.9674
Epoch 60/100
3750/3750 [==============================] - 2s 638us/step - loss: 0.0886 - accuracy: 0.9683
Epoch 61/100
3750/3750 [==============================] - 2s 632us/step - loss: 0.0897 - accuracy: 0.9673
Epoch 62/100
3750/3750 [==============================] - 2s 635us/step - loss: 0.0877 - accuracy: 0.9684
Epoch 63/100
3750/3750 [==============================] - 3s 671us/step - loss: 0.0853 - accuracy: 0.9692
Epoch 64/100
3750/3750 [==============================] - 4s 1ms/step - loss: 0.0855 - accuracy: 0.9691
Epoch 65/100
3750/3750 [==============================] - 4s 969us/step - loss: 0.0856 - accuracy: 0.9690
Epoch 66/100
3750/3750 [==============================] - 3s 866us/step - loss: 0.0845 - accuracy: 0.9692
```

```
Epoch 67/100
3750/3750 [==============================] - 3s 869us/step - loss: 0.0825 - accuracy: 0.9704
Epoch 68/100
3750/3750 [==============================] - 3s 707us/step - loss: 0.0827 - accuracy: 0.9699
Epoch 69/100
3750/3750 [==============================] - 3s 751us/step - loss: 0.0820 - accuracy: 0.9707
Epoch 70/100
3750/3750 [==============================] - 3s 694us/step - loss: 0.0826 - accuracy: 0.9701
Epoch 71/100
3750/3750 [==============================] - 3s 706us/step - loss: 0.0800 - accuracy: 0.9710
Epoch 72/100
3750/3750 [==============================] - 4s 1ms/step - loss: 0.0824 - accuracy: 0.9703
Epoch 73/100
3750/3750 [==============================] - 3s 688us/step - loss: 0.0791 - accuracy: 0.9713
Epoch 74/100
3750/3750 [==============================] - 3s 846us/step - loss: 0.0788 - accuracy: 0.9718
Epoch 75/100
3750/3750 [==============================] - 3s 715us/step - loss: 0.0780 - accuracy: 0.9719
Epoch 76/100
3750/3750 [==============================] - 3s 746us/step - loss: 0.0775 - accuracy: 0.9714
Epoch 77/100
3750/3750 [==============================] - 3s 716us/step - loss: 0.0764 - accuracy: 0.9721
Epoch 78/100
3750/3750 [==============================] - 3s 719us/step - loss: 0.0779 - accuracy: 0.9717
Epoch 79/100
3750/3750 [==============================] - 3s 823us/step - loss: 0.0751 - accuracy: 0.9726
Epoch 80/100
3750/3750 [==============================] - 3s 736us/step - loss: 0.0758 - accuracy: 0.9729
Epoch 81/100
3750/3750 [==============================] - 3s 778us/step - loss: 0.0737 - accuracy: 0.9736
Epoch 82/100
3750/3750 [==============================] - 3s 741us/step - loss: 0.0728 - accuracy: 0.9736
Epoch 83/100
3750/3750 [==============================] - 3s 749us/step - loss: 0.0749 - accuracy: 0.9730
Epoch 84/100
3750/3750 [==============================] - 3s 785us/step - loss: 0.0705 - accuracy: 0.9745
Epoch 85/100
3750/3750 [==============================] - 3s 734us/step - loss: 0.0741 - accuracy: 0.9732
Epoch 86/100
3750/3750 [==============================] - 3s 680us/step - loss: 0.0725 - accuracy: 0.9738
```

```
Epoch 87/100
3750/3750 [==============================] - 3s 718us/step - loss: 0.0724 - accuracy: 0.9737
Epoch 88/100
3750/3750 [==============================] - 3s 676us/step - loss: 0.0722 - accuracy: 0.9739
Epoch 89/100
3750/3750 [==============================] - 3s 692us/step - loss: 0.0730 - accuracy: 0.9741
Epoch 90/100
3750/3750 [==============================] - 3s 697us/step - loss: 0.0734 - accuracy: 0.9738
Epoch 91/100
3750/3750 [==============================] - 3s 681us/step - loss: 0.0693 - accuracy: 0.9746
Epoch 92/100
3750/3750 [==============================] - 3s 706us/step - loss: 0.0721 - accuracy: 0.9748
Epoch 93/100
3750/3750 [==============================] - 3s 708us/step - loss: 0.0670 - accuracy: 0.9756
Epoch 94/100
3750/3750 [==============================] - 2s 657us/step - loss: 0.0681 - accuracy: 0.9757
Epoch 95/100
3750/3750 [==============================] - 3s 707us/step - loss: 0.0686 - accuracy: 0.9750
Epoch 96/100
3750/3750 [==============================] - 3s 683us/step - loss: 0.0682 - accuracy: 0.9753
Epoch 97/100
3750/3750 [==============================] - 3s 704us/step - loss: 0.0678 - accuracy: 0.9755
Epoch 98/100
3750/3750 [==============================] - 3s 693us/step - loss: 0.0680 - accuracy: 0.9754
Epoch 99/100
3750/3750 [==============================] - 2s 662us/step - loss: 0.0669 - accuracy: 0.9755
Epoch 100/100
3750/3750 [==============================] - 3s 687us/step - loss: 0.0657 - accuracy: 0.9758
```

In [ ]:

```
# Evaluating the neural network on the test set
test_loss, test_acc = neural_network_8.evaluate(X_test_tf_flat, y_test_tf)
print('Test accuracy:', test_acc)
```

```
938/938 [==============================] - 0s 424us/step - loss: 7.5096 - accuracy: 0.5455
Test accuracy: 0.5455333590507507
```

In [ ]:

```
Accuracy_Table[13].append(test_acc)
```

What do you conclude by comparing accuracy values you obtain with those obtained in the “'Simple Models” section (note you can compare the accuracy values for binary classification).

A. From the results, it can be observed that the accuracy scores for the feed forward neural network is better than that of the svm or the percepton model. This can mean that the feed forward neural network gives a better result due to the backpropagation and helps the machine to learn the classification beter than the previously used 'simple models'. However, these performances can have varied answers if we change parameters such as sample size of the data set, hyperparameters of the FNN, hyperparameters in the custom Word2Vec model or change the dataset in itself.

Convolutional Neural Networks

CNN for Binary Classification

In [ ]:
```
binframes.head(5)
```

Out[ ]:

	reviews	Labels	MyReviews	gvectors	cvectors	congvectors	
1921057	item came home made bubble wrap labeled new ou...	1	[item, came, home, made, bubble, wrap, labeled...	[0.027547836303710938, 0.09846019744873047, 0....	[0.027547836303710938, 0.09846019744873047, 0....	[[0.024291992, 0.010803223, -0.107421875, 0.30...	-0.0182
2032152	work hope continues working perhaps get discou...	0	[work, hope, continues, working, perhaps, get,...	[0.05845424107142857, 0.016178676060267856, -0...	[0.05845424107142857, 0.016178676060267856, -0...	[[-0.075683594, 0.033691406, -0.064941406, 0.1...	0.06
2590977	software come printer support newer version ma...	1	[software, come, printer, support, newer, vers...	[0.06760212912488339, -0.03568415855293843, -0...	[0.06760212912488339, -0.03568415855293843, -0...	[[0.20410156, -0.30078125, -0.013916016, 0.119...	-
1685277	arrived time came huge box expecting big pictu...	0	[arrived, time, came, huge, box, expecting, bi...	[0.03836669921875, 0.05717875162760417, -0.008...	[0.03836669921875, 0.05717875162760417, -0.008...	[[0.15429688, 0.26757812, 0.09326172, -0.15234...	-0.154
29865	work well canon toner	0	[work, well, canon, toner]	[0.0806640625, 0.07841796875, -0.0003662109375...	[0.0806640625, 0.07841796875, -0.0003662109375...	[[-0.07568359375, 0.03369140625, -0.0649414062...	[[0.0448 -0.032

CNN Using Features extracted from Pre-trained Word2Vec Model For Binary Data

Here the Convolutional Neural Network is used for the Binary Classification, where the feature extracter is the Pre-trained Gensim Word2Vec model and the vector for a review is the average of the vectors for each word in the review.

In [ ]:
```
# split the data into 80-20 train-test
x_train, x_test, y_train, y_test = train_test_split(binframes['gvectors'], binframes['Labels'], test_size
```

In [ ]:
```
# Converting NumPy arrays to TensorFlow tensors
X_train_tf = tf.convert_to_tensor(x_train.to_list(), dtype=tf.float32)
y_train_tf = tf.convert_to_tensor(y_train, dtype=tf.int32)
X_test_tf = tf.convert_to_tensor(x_test.to_list(), dtype=tf.float32)
y_test_tf = tf.convert_to_tensor(y_test, dtype=tf.int32)
```

In [ ]:
```
# Define cnn architecture
cnn_network = tf.keras.models.Sequential()
cnn_network.add(tf.keras.layers.Conv1D(50, 3, activation='relu', input_shape=(300,1) ))
cnn_network.add(tf.keras.layers.MaxPooling1D(3))
cnn_network.add(tf.keras.layers.Conv1D(10, 3, activation='relu'))
cnn_network.add(tf.keras.layers.MaxPooling1D(2))
cnn_network.add(tf.keras.layers.Flatten())
cnn_network.add(tf.keras.layers.Dense(1, activation='sigmoid'))
```

In [ ]:
```
# Compile the cnn
cnn_network.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

In [ ]:
```
# Train the cnn
cnn_network.fit(X_train_tf, y_train_tf, epochs=10, batch_size=64, verbose=1)
```

```
Epoch 1/10
1250/1250 [==============================] - 6s 4ms/step - loss: 0.4780 - accuracy: 0.7767
Epoch 2/10
1250/1250 [==============================] - 6s 5ms/step - loss: 0.4101 - accuracy: 0.8163
Epoch 3/10
1250/1250 [==============================] - 6s 4ms/step - loss: 0.3947 - accuracy: 0.8256
Epoch 4/10
1250/1250 [==============================] - 6s 5ms/step - loss: 0.3851 - accuracy: 0.8303
Epoch 5/10
1250/1250 [==============================] - 7s 5ms/step - loss: 0.3783 - accuracy: 0.8344
Epoch 6/10
1250/1250 [==============================] - 6s 5ms/step - loss: 0.3744 - accuracy: 0.8350
Epoch 7/10
1250/1250 [==============================] - 6s 5ms/step - loss: 0.3704 - accuracy: 0.8382
Epoch 8/10
1250/1250 [==============================] - 6s 5ms/step - loss: 0.3673 - accuracy: 0.8399
Epoch 9/10
1250/1250 [==============================] - 6s 5ms/step - loss: 0.3650 - accuracy: 0.8416
Epoch 10/10
1250/1250 [==============================] - 6s 5ms/step - loss: 0.3632 - accuracy: 0.8418
```

Out[ ]: <keras.src.callbacks.History at 0x13807a400>

In [ ]:
```
# Evaluate the model on the test set
loss_ternary_gavg_cnn, ternary_gavg_cnn = cnn_network.evaluate(X_test_tf, y_test_tf, verbose=0)
print('Test Accuracy: ', ternary_gavg_cnn)
```

```
Test Accuracy:  0.8415499925613403
```

In [ ]:
```
Accuracy_Table[14].append(ternary_gavg_cnn)
```

CNN Using Features extracted from Custom Word2Vec Model For Binary Data

Here the Convolutional Neural Network is used for the Binary Classification, where the feature extracter is the Custom Word2Vec model and the vector for a review is the average of the vectors for each word in the review.

In [ ]:
```
# split the data into 80-20 train-test
x_train, x_test, y_train, y_test = train_test_split(binframes['cvectors'], binframes['Labels'], test_size
```

In [ ]:
```python
# Converting NumPy arrays to TensorFlow tensors
X_train_tf = tf.convert_to_tensor(x_train.to_list(), dtype=tf.float32)
y_train_tf = tf.convert_to_tensor(y_train, dtype=tf.int32)
X_test_tf = tf.convert_to_tensor(x_test.to_list(), dtype=tf.float32)
y_test_tf = tf.convert_to_tensor(y_test, dtype=tf.int32)
```

In [ ]:
```python
# Define cnn architecture
cnn_network = tf.keras.models.Sequential()
cnn_network.add(tf.keras.layers.Conv1D(50, 3, activation='relu', input_shape=(300,1) ))
cnn_network.add(tf.keras.layers.MaxPooling1D(3))
cnn_network.add(tf.keras.layers.Conv1D(10, 3, activation='relu'))
cnn_network.add(tf.keras.layers.MaxPooling1D(2))
cnn_network.add(tf.keras.layers.Flatten())
cnn_network.add(tf.keras.layers.Dense(1, activation='sigmoid'))
```

In [ ]:
```python
# Compile the cnn
cnn_network.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
```

In [ ]:
```python
# Train the cnn
cnn_network.fit(X_train_tf, y_train_tf, epochs=10, batch_size=64, verbose=1)
```

```
Epoch 1/10
1250/1250 [==============================] - 6s 5ms/step - loss: 0.4825 - accuracy: 0.7719
Epoch 2/10
1250/1250 [==============================] - 6s 4ms/step - loss: 0.4188 - accuracy: 0.8098
Epoch 3/10
1250/1250 [==============================] - 6s 4ms/step - loss: 0.4032 - accuracy: 0.8211
Epoch 4/10
1250/1250 [==============================] - 6s 5ms/step - loss: 0.3930 - accuracy: 0.8257
Epoch 5/10
1250/1250 [==============================] - 7s 5ms/step - loss: 0.3855 - accuracy: 0.8303
Epoch 6/10
1250/1250 [==============================] - 6s 5ms/step - loss: 0.3803 - accuracy: 0.8328
Epoch 7/10
1250/1250 [==============================] - 6s 5ms/step - loss: 0.3757 - accuracy: 0.8348
Epoch 8/10
1250/1250 [==============================] - 6s 5ms/step - loss: 0.3716 - accuracy: 0.8364
Epoch 9/10
1250/1250 [==============================] - 6s 5ms/step - loss: 0.3684 - accuracy: 0.8393
Epoch 10/10
1250/1250 [==============================] - 6s 5ms/step - loss: 0.3665 - accuracy: 0.8396
```

Out[ ]: <keras.src.callbacks.History at 0x1399c56d0>

In [ ]:

```python
# Evaluate the model on the test set
loss_ternary_cavg_cnn, ternary_cavg_cnn = cnn_network.evaluate(X_test_tf, y_test_tf, verbose=0)
print('Test Accuracy: ', ternary_cavg_cnn)
```

```
Test Accuracy:  0.8417999744415283
```

In [ ]:

```python
Accuracy_Table[15].append(ternary_cavg_cnn)
```

CNN for Ternary Classification

In [ ]:

```python
terframes.head(5)
```

Out[ ]:

	reviews	Labels	MyReviews	gvectors	cvectors	congvectors	
1163606	excellent pen actually pen like use write anyt...	0	[excellent, pen, actually, pen, like, use, wri...	[0.013108995225694444, -0.015750461154513888, ...	[0.013108995225694444, -0.015750461154513888, ...	[[-0.212890625, -0.004302978515625, -0.1806640...	
2069284	fresh set battery nice strong beam unfortunate...	2	[fresh, set, battery, nice, strong, beam, unfo...	[0.035248976487379804, 0.09482046274038461, 0....	[0.035248976487379804, 0.09482046274038461, 0....	[[-0.042236328, 0.018066406, 0.22070312, -0.01...	
338411	file folder label great use folder	0	[file, folder, label, great, use, folder]	[0.11321149553571429, 0.029767717633928572, -0...	[0.11321149553571429, 0.029767717633928572, -0...	[[0.28515625, 0.023193359375, -0.03173828125, ...	
37088	work 's well heavy duty steel original stapler...	2	[work, 's, well, heavy, duty, steel, original,...	[0.07191051136363637, 0.07883522727272728, -0....	[0.07191051136363637, 0.07883522727272728, -0....	[[-0.075683594, 0.033691406, -0.064941406, 0.1...	-
752649	great quality great value	0	[great, quality, great, value]	[-0.00732421875, 0.141259765625, 0.03442382812...	[-0.00732421875, 0.141259765625, 0.03442382812...	[[0.07177734375, 0.2080078125, -0.028442382812...	[

CNN Using Features extracted from Pre-trained Word2Vec Model For Ternary Data

Here the Convolutional Neural Network is used for the Ternary Classification, where the feature extracter is the Pre-trained Gensim Word2Vec model and the vector for a review is the average of the vectors for each word in the review.

In [ ]:

```
# split the data into 80-20 train-test
x_train, x_test, y_train, y_test = train_test_split(terframes['gvectors'], terframes['Labels'], test_size
```

In [ ]:

```
# Converting NumPy arrays to TensorFlow tensors
X_train_tf = tf.convert_to_tensor(x_train.to_list(), dtype=tf.float32)
```

```python
y_train_tf = tf.convert_to_tensor(y_train, dtype=tf.int32)
X_test_tf = tf.convert_to_tensor(x_test.to_list(), dtype=tf.float32)
y_test_tf = tf.convert_to_tensor(y_test, dtype=tf.int32)
```

In [ ]:
```python
# Define cnn architecture
cnn_network = tf.keras.models.Sequential()
cnn_network.add(tf.keras.layers.Conv1D(50, 3, activation='relu', input_shape=(300,1) ))
cnn_network.add(tf.keras.layers.MaxPooling1D(3))
cnn_network.add(tf.keras.layers.Conv1D(10, 3, activation='relu'))
cnn_network.add(tf.keras.layers.MaxPooling1D(2))
cnn_network.add(tf.keras.layers.Flatten())
cnn_network.add(tf.keras.layers.Dense(3, activation='softmax'))
```

In [ ]:
```python
# Compile the cnn
cnn_network.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

In [ ]:
```python
# Train the cnn
cnn_network.fit(X_train_tf, y_train_tf, epochs=10, batch_size=64, verbose=1)
```

```
Epoch 1/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.8888 - accuracy: 0.5833
Epoch 2/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.8362 - accuracy: 0.6166
Epoch 3/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.8200 - accuracy: 0.6262
Epoch 4/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.8122 - accuracy: 0.6299
Epoch 5/10
1875/1875 [==============================] - 8s 5ms/step - loss: 0.8066 - accuracy: 0.6341
Epoch 6/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.8034 - accuracy: 0.6355
Epoch 7/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.8007 - accuracy: 0.6363
Epoch 8/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.7982 - accuracy: 0.6374
Epoch 9/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.7963 - accuracy: 0.6392
Epoch 10/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.7947 - accuracy: 0.6387
```

Out[ ]: <keras.src.callbacks.History at 0x138245e50>

In [ ]:
```
# Evaluate the model on the test set
loss_ternary_gavg_cnn, ternary_gavg_cnn = cnn_network.evaluate(X_test_tf, y_test_tf, verbose=0)
print('Test Accuracy: ', ternary_gavg_cnn)
```

```
Test Accuracy:  0.635200023651123
```

In [ ]:
```
Accuracy_Table[16].append(ternary_gavg_cnn)
```

CNN Using Features extracted from Custom Word2Vec Model For Ternary Data

Here the Convolutional Neural Network is used for the Ternary Classification, where the feature extracter is the Custom Word2Vec model and the vector for a review is the average of the vectors for each word in the review.

In [ ]:
```
# split the data into 80-20 train-test
x_train, x_test, y_train, y_test = train_test_split(terframes['gvectors'], terframes['Labels'], test_size
```

In [ ]:
```
# Converting NumPy arrays to TensorFlow tensors
X_train_tf = tf.convert_to_tensor(x_train.to_list(), dtype=tf.float32)
y_train_tf = tf.convert_to_tensor(y_train, dtype=tf.int32)
X_test_tf = tf.convert_to_tensor(x_test.to_list(), dtype=tf.float32)
y_test_tf = tf.convert_to_tensor(y_test, dtype=tf.int32)
```

In [ ]:
```
# Define cnn architecture
cnn_network = tf.keras.models.Sequential()
cnn_network.add(tf.keras.layers.Conv1D(50, 3, activation='relu', input_shape=(300,1) ))
cnn_network.add(tf.keras.layers.MaxPooling1D(3))
cnn_network.add(tf.keras.layers.Conv1D(10, 3, activation='relu'))
cnn_network.add(tf.keras.layers.MaxPooling1D(2))
cnn_network.add(tf.keras.layers.Flatten())
cnn_network.add(tf.keras.layers.Dense(3, activation='softmax'))
```

In [ ]:
```
# Compile the cnn
cnn_network.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

In [ ]:
```
# Train the cnn
cnn_network.fit(X_train_tf, y_train_tf, epochs=10, batch_size=64, verbose=1)
```

```
Epoch 1/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.8961 - accuracy: 0.5797
Epoch 2/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.8441 - accuracy: 0.6119
Epoch 3/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.8313 - accuracy: 0.6197
Epoch 4/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.8227 - accuracy: 0.6244
Epoch 5/10
1875/1875 [==============================] - 10s 5ms/step - loss: 0.8177 - accuracy: 0.6270
Epoch 6/10
1875/1875 [==============================] - 8s 4ms/step - loss: 0.8144 - accuracy: 0.6272
Epoch 7/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.8110 - accuracy: 0.6297
Epoch 8/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.8082 - accuracy: 0.6317
Epoch 9/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.8065 - accuracy: 0.6320
Epoch 10/10
1875/1875 [==============================] - 9s 5ms/step - loss: 0.8050 - accuracy: 0.6341
```

Out[ ]:

```
<keras.src.callbacks.History at 0x138376850>
```

In [ ]:

```
# Evaluate the model on the test set
loss_ternary_cavg_cnn, ternary_cavg_cnn = cnn_network.evaluate(X_test_tf, y_test_tf, verbose=0)
print('Test Accuracy: ', ternary_cavg_cnn)
```

```
Test Accuracy:  0.6302666664123535
```

In [ ]:

```
Accuracy_Table[17].append(ternary_cavg_cnn)
```

Reporting Accuracy Values

In [ ]:

```
print(tabulate(Accuracy_Table, headers=["Feature Extracter Type", "Neural Network Type", "Classification
```

Feature Extracter Type	Neural Network Type	Classification Type	Accuracy Value
TFIDF	Percepton	Binary	0.84775
TFIDF	SVM	Binary	0.88845
Avg Pre-trained W2V	Percepton	Binary	0.7971
Avg Pre-trained W2V	SVM	Binary	0.8487
Avg Custom W2V	Percepton	Binary	0.7971
Avg Custom W2V	SVM	Binary	0.8487
Avg Pre-trained W2V	FNN	Binary	0.84985
Avg Custom W2V	FNN	Binary	0.851
Con Pre-trained W2V	FNN	Binary	0.7652
Con Custom W2V	FNN	Binary	0.79105
Avg Pre-trained W2V	FNN	Ternary	0.6428
Avg Custom W2V	FNN	Ternary	0.647633
Con Pre-trained W2V	FNN	Ternary	0.526033
Con Custom W2V	FNN	Ternary	0.545533
Avg Pre-trained W2V	CNN	Binary	0.84155
Avg Custom W2V	CNN	Binary	0.8418
Avg Pre-trained W2V	CNN	Ternary	0.6352
Avg Custom W2V	CNN	Ternary	0.630267