CSCI 544 - Applied Natural Language Processing

CSCI 544 - Assignment 3

Name: Sri Manvith Vaddeboyina

USC ID: 1231409457

1. Data Generation

Importing necessary libraries/packages

In [1]:
```
import re
import sys
import numpy as np
import pandas as pd
import contractions
import tensorflow as tf
from gensim import models
from bs4 import BeautifulSoup
import gensim.downloader as api
from sklearn.svm import LinearSVC
from gensim.models import Word2Vec
from sklearn.metrics import accuracy_score
from sklearn.linear_model import Perceptron
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.feature_extraction.text import TfidfVectorizer

import warnings
warnings.filterwarnings('ignore')
```

Read Data

Reading Amazon US Beauty Reviews (tsv) dataset and retaining only the following two columns:
1. review_body
2. star_rating

In [2]:
```
df = pd.read_csv('amazon_reviews_us_Beauty_v1_00.tsv', on_bad_lines = 'skip', sep='\t')
```

In [3]:
```
df = df[['review_body','star_rating']]
```

Dropping the entire rows where any of the column contains NA value

In [4]:
```
df.dropna(inplace=True)
```

Keep Reviews and Ratings

Create a three-class classification problem according to the ratings.
Ratings:
1 and 2 - class 1
3 - class 2
4 and 5 - class 3

In [5]:
```
 df = df[
         df['star_rating'].eq('1') |
         df['star_rating'].eq('2') |
         df['star_rating'].eq('3') |
         df['star_rating'].eq('4') |
         df['star_rating'].eq('5')
        ]
```

Verifying the datatype of each column and setting them correctly

In [6]:
```
df['star_rating']=df['star_rating'].astype(int)
df['review_body']=df['review_body'].astype(str)
```

Creating a 3-class classification on ratings

In [7]:
```
def condition(x):
    if x==1 or x==2:
        return 1
    elif x==3:
        return 2
    elif x==4 or x==5:
        return 3

df['rating'] = df['star_rating'].apply(condition)
```

We form three classes and select 20000 reviews randomly from each class.

Randomly selecting 20000 reviews from each of class 1,2 and 3.
Total: 60000 reviews

In [8]:
```
df=df.groupby('rating').sample(n=20000)
```

In [9]:
```
df.drop(['star_rating'],inplace=True,axis=1)
```

Data Cleaning Removing the following as part of data cleaning:
1. URLs
2. HTML tags
2. Contractions Expansion
3. Non-alphabetic characters
4. Converting text to lower case
5. Removing extra spaces

In [10]:
```
def lower_case(texts):
    return texts.lower()
```

In [11]:
```
def cleanhtml(texts):
    regex = re.compile('<.*?>|&([a-z0-9]+|#[0-9]{1,6}|#x[0-9a-f]{1,6});')
    cleantext = re.sub(regex, '', texts)

    return cleantext
```

In [12]:
```
def remove_url(texts):
    regex = re.compile('http\S+')
    cleantext = re.sub(regex, '', texts)
    return cleantext
```

In [13]:
```
def non_alphabetical(texts):
    regex = re.compile('[^a-zA-Z]')
    cleantext = re.sub(regex, ' ', texts)

    regex = re.compile('_')
    cleantext = re.sub(regex, ' ', cleantext)

    return cleantext
```

In [14]:
```
def extra_spaces(texts):
    regex = re.compile('[\s]{2,}')
    cleantext = re.sub(regex, ' ', texts)

    return cleantext.rstrip()
```

In [15]:
```python
def contractionfunction(text):
    expanded_words = []
    for word in text.split():
        # using contractions.fix to expand the shotened words
        expanded_words.append(contractions.fix(word))

    expanded_words = ' '.join(expanded_words)
    return expanded_words
```

In [16]:
```python
def corpus_contractions(texts):
    expended_corpus = []
    for text in texts:
        expended_corpus.append(contractionfunction(text))
    return expended_corpus
```

In [17]:
```python
review_body = df.copy(deep = True).review_body.tolist()
labels = df.copy(deep = True).rating.tolist()
clean_review_body = []

for index , sen in enumerate(review_body):
    sen = lower_case(sen)
    sen = cleanhtml(sen)
    sen = remove_url(sen)
    sen = contractionfunction(sen)
    sen = non_alphabetical(sen)
    sen = extra_spaces(sen)
    clean_review_body.append(sen)
```

Cleaning data

In [18]:
```python
df['clean_text'] = clean_review_body
```

2. Word Embedding

(a) word2vec-google-news-300 Word2Vec model

Loading the pretrained "word2vec-google-news-300" Word2Vec model from gensim library.

In [19]:
```python
google_news_word2vec = api.load('word2vec-google-news-300')
```

Checking the semantic similarity of example words

In [20]:
```python
google_news_word2vec.most_similar(positive=["king","woman"],negative=["man"])[0]
```

Out[20]:
```
('queen', 0.7118193507194519)
```

In [21]:
```python
print("Similarity for: [good, better] : ", google_news_word2vec.similarity(w1="good", w2="better"))
print("Similarity for: [neat, clean] : ", google_news_word2vec.similarity(w1="neat", w2="clean"))
print("Similarity for: [big, huge] : ", google_news_word2vec.similarity(w1="big", w2="huge"))
```

```
Similarity for: [good, better] :  0.6120729
Similarity for: [neat, clean] :  0.29077712
Similarity for: [big, huge] :  0.7809856
```

(b) Train a Word2Vec model using your own dataset

Training a Word2Vec model using amazon reviews dataset. Generating the tokens and training the word2vec model.

In [22]:
```
def get_reviews_tokens(reviews):
    reviews_tokens = []
    for rev in reviews:
        reviews_tokens.append(rev.split(" "))
    return reviews_tokens
reviews_tokens = get_reviews_tokens(df['clean_text'])
```

Training a word2vec model with a embedding size as 300, window size as 13 and min word count as 9

In [23]:
```
word2vec = Word2Vec(sentences = reviews_tokens, vector_size = 300, window = 13, min_count = 9)
```

Checking the semantic similarity of example words

In [24]:
```
w1 = word2vec.wv["good"]
w2 = word2vec.wv["better"]
print("Similarity for: [good, better] : ", cosine_similarity(w1.reshape(1,-1),w2.reshape(1,-1))[0][0])

w3 = word2vec.wv["neat"]
w4 = word2vec.wv["clean"]
print("Similarity for: [neat, clean] : ", cosine_similarity(w3.reshape(1,-1),w4.reshape(1,-1))[0][0])

w5 = word2vec.wv["big"]
w6 = word2vec.wv["huge"]
print("Similarity for: [big, huge] : ", cosine_similarity(w5.reshape(1,-1),w6.reshape(1,-1))[0][0])
```

```
Similarity for: [good, better] :  0.35388795
Similarity for: [neat, clean] :  0.19055185
Similarity for: [big, huge] :  0.61514467
```

What do you conclude from comparing vectors generated by yourself and the pretrained model? Which of the Word2Vec models seems to encode semantic similarities between words better?

Reasoning:
The pretrained word2vec model performed well as compared to the custom trained word2vec model. The reason could be:

Pretrained models have been trained on very large datasets, which allow them to capture a wider range of relationships between words. This makes them more effective at capturing the subtle nuances of language. They have been trained on a diverse range of text data, which makes them more robust and adaptable to different contexts and domains.

3. Simple models

TF-IDF

Splitting data into train and test splits 80:20 to be fed for TF-IDF

In [25]:
```
X_train, X_test, y_train, y_test = train_test_split
(
    df['clean_text'], df['rating'], test_size=0.20, random_state=42, stratify = df['rating']
)
```

TF-IDF (term frequency-inverse document frequency) is a statistical measure that evaluates how relevant a word is to a document in a collection of documents.

In [26]:
```
vectorizer = TfidfVectorizer(ngram_range=(1,4))
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)
y_train_tfidf = y_train
y_test_tfidf = y_test
```

Perceptron

Perceptron is a single layer neural network that does certain computations to detect features or business intelligence in the input data.

In [27]:
```
perceptron_text_clf = Perceptron()
perceptron_text_clf.fit(X_train_tfidf, y_train_tfidf)
perceptron_predictions = perceptron_text_clf.predict(X_test_tfidf)
perceptron_accuracy_tfidf = accuracy_score(y_test_tfidf, perceptron_predictions)
print("Accuracy of Perceptron on TF-IDF data : ", perceptron_accuracy_tfidf)
```

```
Accuracy of Perceptron on TF-IDF data :  0.7114166666666667
```

SVM

The objective of the support vector machine algorithm is to find a hyperplane in an N-dimensional space(N — the number of features) that distinctly classifies the data points.

In [28]:
```
svc_text_clf = LinearSVC()
svc_text_clf.fit(X_train_tfidf, y_train_tfidf)
svc_predictions = svc_text_clf.predict(X_test_tfidf)
svc_accuarcy_tfidf = accuracy_score(y_test_tfidf, svc_predictions)
print("Accuracy of SVM on TF-IDF data : ", svc_accuarcy_tfidf)
```

```
Accuracy of SVM on TF-IDF data :  0.7431666666666666
```

Word2Vec

In [29]:
```
# Define a function to compute the word embeddings for a sentence
def get_review_embedding(data):
    words = data.split(" ")
    embeddings = np.array([google_news_word2vec[word] for word in words if word in google_news_word2vec])
    if embeddings.size == 0:
        return np.zeros(300)
    return np.mean(embeddings, axis=0)
```

Creating the train and test data using the get_review_embedding() function. This data is fed to Perceptron and SVM models

In [30]:
```
# Apply the get_sentence_vector function to each sentence in the X_train_df dataframe
train_review_vectors = [get_review_embedding(sentence) for sentence in X_train]

# Stack the resulting vectors into a 2D numpy array
X_train_w2v = np.stack(train_review_vectors, axis=0)

test_review_vectors = [get_review_embedding(sentence) for sentence in X_test]

# Stack the resulting vectors into a 2D numpy array
X_test_w2v = np.stack(test_review_vectors, axis=0)
y_train_w2v = y_train
y_test_w2v = y_test
```

Perceptron

Perceptron is a single layer neural network that does certain computations to detect features or business intelligence in the input data.

In [31]:
```
perceptron_text_clf = Perceptron()
perceptron_text_clf.fit(X_train_w2v, y_train_w2v)
perceptron_predictions = perceptron_text_clf.predict(X_test_w2v)
perceptron_accuracy_w2v = accuracy_score(y_test_w2v, perceptron_predictions)
print("Accuracy of Perceptron on Word2Vec data : ", perceptron_accuracy_w2v)
```

```
Accuracy of Perceptron on Word2Vec data :  0.5465
```

SVM

The objective of the support vector machine algorithm is to find a hyperplane in an N-dimensional space(N — the number of features) that distinctly classifies the data points.

In [32]:
```
svc_text_clf = LinearSVC()
svc_text_clf.fit(X_train_w2v, y_train_w2v)
svc_predictions = svc_text_clf.predict(X_test_w2v)
svc_accuracy_w2v = accuracy_score(y_test_w2v, svc_predictions)
print("Accuracy of SVM on Word2Vec data : ", svc_accuracy_w2v)
```

```
Accuracy of SVM on Word2Vec data :  0.6711666666666667
```

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

Reasoning:
Between the TF-IDF and google pretrained Word2Vec accuracies on Perceptron and SVM, the accuracies of TF-IDF are better compared to pretrained Word2Vec. Reason could be that: While Word2Vec is a powerful technique for capturing semantic and syntactic relationships between words, TF-IDF may be more appropriate for certain tasks that rely on keyword matching or text classification. In our current use-case, it is the lexical similarity that plays a greater role than the semantic similarity.

4. Feedforward Neural Networks

(a) FNN

A Feed Forward Neural Network is an artificial neural network in which the connections between nodes does not form a cycle.

In [33]:
```
y_train_w2v = np.array(y_train_w2v)
y_test_w2v = np.array(y_test_w2v)
```

Feed forward Neural Network code with two hidden layers, each with 100 and 10 nodes, respectively. relu and softmax activation is used in the code and "y values-1" is taken to have class labels as 0, 1, 2 instead of 1, 2, 3

In [34]:
```
def FNN(x_train, y_train, x_test, y_test, num_features, epochs, batch_size, learning_rate_val):
    model_fnn = tf.keras.Sequential(
                                    [   tf.keras.layers.InputLayer((num_features,)),
                                        tf.keras.layers.Dense(100,activation='relu'),
                                        tf.keras.layers.Dense(10,activation='relu'),
                                        tf.keras.layers.Dense(3,activation='softmax')
                                    ]
                                )

    model_fnn.compile(
                    optimizer = Adam(learning_rate=learning_rate_val),
                    loss='sparse_categorical_crossentropy',
                    metrics=['accuracy']
                )

    print(model_fnn.summary())

    model_fnn.fit(x_train,y_train-1, batch_size = batch_size, epochs = epochs)

    result = model_fnn.evaluate(x_test,y_test-1)
    return result[1]
```

Passing the num_features, epochs, batch size and learning rate parameters to the FNN function

In [35]:
```
fnn_accuracy = FNN(X_train_w2v, y_train_w2v, X_test_w2v, y_test_w2v, 300, 50, 64,0.001)
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 100)               30100

 dense_1 (Dense)             (None, 10)                1010

 dense_2 (Dense)             (None, 3)                 33

=================================================================
Total params: 31,143
Trainable params: 31,143
Non-trainable params: 0
_________________________________________________________________
None
Epoch 1/50

2023-03-01 20:37:34.634487: W tensorflow/stream_executor/platform/default/dso_loader.cc:64] Could not load dynamic library 'libcuda.so.1'; dlerror: libcuda.so.1: cannot open shared object file: No such file or directory; LD_LIBRARY_PATH: /usr/local/cuda/lib64:/usr/local/nccl2/lib:/usr/local/cuda/extras/CUPTI/lib64
2023-03-01 20:37:34.634526: W tensorflow/stream_executor/cuda/cuda_driver.cc:269] failed call to cuInit: UNKNOWN ERROR (303)
2023-03-01 20:37:34.634549: I tensorflow/stream_executor/cuda/cuda_diagnostics.cc:156] kernel driver does not appear to be running on this host (nlp): /proc/driver/nvidia/version does not exist
2023-03-01 20:37:34.634885: I tensorflow/core/platform/cpu_feature_guard.cc:151] This TensorFlow binary is optimized with oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:  AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
```

```
750/750 [==============================] - 2s 2ms/step - loss: 0.8445 - accuracy: 0.5930
Epoch 2/50
750/750 [==============================] - 1s 2ms/step - loss: 0.7510 - accuracy: 0.6677
Epoch 3/50
750/750 [==============================] - 1s 2ms/step - loss: 0.7295 - accuracy: 0.6783
Epoch 4/50
750/750 [==============================] - 1s 2ms/step - loss: 0.7170 - accuracy: 0.6851
Epoch 5/50
750/750 [==============================] - 1s 2ms/step - loss: 0.7064 - accuracy: 0.6913
Epoch 6/50
750/750 [==============================] - 1s 2ms/step - loss: 0.6968 - accuracy: 0.6959
Epoch 7/50
750/750 [==============================] - 1s 2ms/step - loss: 0.6891 - accuracy: 0.6991
Epoch 8/50
750/750 [==============================] - 1s 2ms/step - loss: 0.6812 - accuracy: 0.7022
Epoch 9/50
750/750 [==============================] - 1s 2ms/step - loss: 0.6758 - accuracy: 0.7060
Epoch 10/50
750/750 [==============================] - 1s 2ms/step - loss: 0.6686 - accuracy: 0.7092
Epoch 11/50
750/750 [==============================] - 1s 2ms/step - loss: 0.6618 - accuracy: 0.7128
Epoch 12/50
750/750 [==============================] - 1s 2ms/step - loss: 0.6561 - accuracy: 0.7156
Epoch 13/50
750/750 [==============================] - 1s 2ms/step - loss: 0.6500 - accuracy: 0.7175
Epoch 14/50
750/750 [==============================] - 1s 2ms/step - loss: 0.6447 - accuracy: 0.7204
Epoch 15/50
750/750 [==============================] - 1s 2ms/step - loss: 0.6394 - accuracy: 0.7228
Epoch 16/50
750/750 [==============================] - 1s 2ms/step - loss: 0.6341 - accuracy: 0.7254
Epoch 17/50
750/750 [==============================] - 1s 2ms/step - loss: 0.6277 - accuracy: 0.7284
Epoch 18/50
750/750 [==============================] - 1s 2ms/step - loss: 0.6222 - accuracy: 0.7309
Epoch 19/50
750/750 [==============================] - 1s 2ms/step - loss: 0.6179 - accuracy: 0.7331
Epoch 20/50
750/750 [==============================] - 1s 2ms/step - loss: 0.6120 - accuracy: 0.7364
Epoch 21/50
750/750 [==============================] - 1s 2ms/step - loss: 0.6077 - accuracy: 0.7373
Epoch 22/50
750/750 [==============================] - 1s 2ms/step - loss: 0.6041 - accuracy: 0.7396
Epoch 23/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5993 - accuracy: 0.7414
Epoch 24/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5935 - accuracy: 0.7455
Epoch 25/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5887 - accuracy: 0.7479
Epoch 26/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5845 - accuracy: 0.7498
Epoch 27/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5806 - accuracy: 0.7518
Epoch 28/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5757 - accuracy: 0.7542
Epoch 29/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5718 - accuracy: 0.7571
Epoch 30/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5666 - accuracy: 0.7598
Epoch 31/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5617 - accuracy: 0.7616
Epoch 32/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5575 - accuracy: 0.7619
Epoch 33/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5546 - accuracy: 0.7652
Epoch 34/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5490 - accuracy: 0.7668
Epoch 35/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5448 - accuracy: 0.7693
Epoch 36/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5420 - accuracy: 0.7737
Epoch 37/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5383 - accuracy: 0.7739
Epoch 38/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5350 - accuracy: 0.7756
Epoch 39/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5311 - accuracy: 0.7753
Epoch 40/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5251 - accuracy: 0.7797
Epoch 41/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5216 - accuracy: 0.7819
Epoch 42/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5178 - accuracy: 0.7832
Epoch 43/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5137 - accuracy: 0.7850
Epoch 44/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5098 - accuracy: 0.7856
Epoch 45/50
```

```
750/750 [==============================] - 1s 2ms/step - loss: 0.5050 - accuracy: 0.7882
Epoch 46/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5021 - accuracy: 0.7907
Epoch 47/50
750/750 [==============================] - 1s 2ms/step - loss: 0.5001 - accuracy: 0.7906
Epoch 48/50
750/750 [==============================] - 1s 2ms/step - loss: 0.4968 - accuracy: 0.7929
Epoch 49/50
750/750 [==============================] - 1s 2ms/step - loss: 0.4921 - accuracy: 0.7974
Epoch 50/50
750/750 [==============================] - 1s 2ms/step - loss: 0.4894 - accuracy: 0.7961
375/375 [==============================] - 1s 1ms/step - loss: 0.8172 - accuracy: 0.6727
```

In [36]:
```
print("Accuracy of FNN model: ",fnn_accuracy)
```

```
Accuracy of FNN model:  0.6727499961853027
```

(b) Top 10 FNN

Concatenate the first 10 Word2Vec vectors for each review. Padding 0 values to the vectors if there is no sufficient review length

In [37]:
```
def get_review_top10(reviews):
    top10_embeddings=[]
    count = 0

    for word in reviews.split(" "):
        if word in google_news_word2vec:
            count = count + 1
            if count > 10:
                break
            else:
                word_embedding = google_news_word2vec[word]
                top10_embeddings.extend(word_embedding)

    length = len(top10_embeddings)
    if length == 0:
        return np.zeros(3000)

    if length < 3000:
        less = 3000 - length
        top10_embeddings += less * [0]

    return top10_embeddings
```

In [38]:
```
# Apply the get_sentence_vector function to each sentence in the X_train_df dataframe
train_top10_review_vectors = X_train.apply(get_review_top10)

# Stack the resulting vectors into a 2D numpy array
X_train_top10_w2v = np.stack(train_top10_review_vectors, axis=0)

test_top10_review_vectors = X_test.apply(get_review_top10)

# Stack the resulting vectors into a 2D numpy array
X_test_top10_w2v = np.stack(test_top10_review_vectors, axis=0)
```

In [39]:
```
fnn_top10_accuracy = FNN(X_train_top10_w2v, y_train_w2v, X_test_top10_w2v, y_test_w2v, 3000, 50, 64, 0.001)
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense_3 (Dense)             (None, 100)               300100

 dense_4 (Dense)             (None, 10)                1010

 dense_5 (Dense)             (None, 3)                 33

=================================================================
Total params: 301,143
Trainable params: 301,143
Non-trainable params: 0
_________________________________________________________________
None
Epoch 1/50
750/750 [==============================] - 2s 2ms/step - loss: 0.8462 - accuracy: 0.6017
Epoch 2/50
750/750 [==============================] - 2s 2ms/step - loss: 0.7528 - accuracy: 0.6595
Epoch 3/50
750/750 [==============================] - 2s 2ms/step - loss: 0.6834 - accuracy: 0.6996
Epoch 4/50
750/750 [==============================] - 2s 2ms/step - loss: 0.5994 - accuracy: 0.7458
Epoch 5/50
750/750 [==============================] - 2s 2ms/step - loss: 0.5011 - accuracy: 0.7926
Epoch 6/50
750/750 [==============================] - 2s 2ms/step - loss: 0.4010 - accuracy: 0.8405
Epoch 7/50
750/750 [==============================] - 2s 2ms/step - loss: 0.3152 - accuracy: 0.8798
Epoch 8/50
750/750 [==============================] - 2s 2ms/step - loss: 0.2459 - accuracy: 0.9083
Epoch 9/50
750/750 [==============================] - 2s 2ms/step - loss: 0.1983 - accuracy: 0.9288
Epoch 10/50
750/750 [==============================] - 2s 3ms/step - loss: 0.1648 - accuracy: 0.9416
Epoch 11/50
750/750 [==============================] - 2s 3ms/step - loss: 0.1409 - accuracy: 0.9496
Epoch 12/50
750/750 [==============================] - 2s 2ms/step - loss: 0.1320 - accuracy: 0.9516
Epoch 13/50
750/750 [==============================] - 2s 2ms/step - loss: 0.1233 - accuracy: 0.9562
Epoch 14/50
750/750 [==============================] - 2s 2ms/step - loss: 0.1093 - accuracy: 0.9617
Epoch 15/50
750/750 [==============================] - 2s 2ms/step - loss: 0.1045 - accuracy: 0.9632
Epoch 16/50
750/750 [==============================] - 2s 3ms/step - loss: 0.1039 - accuracy: 0.9629
Epoch 17/50
750/750 [==============================] - 2s 2ms/step - loss: 0.0939 - accuracy: 0.9664
Epoch 18/50
750/750 [==============================] - 2s 2ms/step - loss: 0.0910 - accuracy: 0.9668
Epoch 19/50
750/750 [==============================] - 2s 3ms/step - loss: 0.0994 - accuracy: 0.9644
Epoch 20/50
750/750 [==============================] - 2s 2ms/step - loss: 0.0886 - accuracy: 0.9679
Epoch 21/50
750/750 [==============================] - 2s 2ms/step - loss: 0.0758 - accuracy: 0.9720
Epoch 22/50
750/750 [==============================] - 2s 2ms/step - loss: 0.0830 - accuracy: 0.9694
Epoch 23/50
750/750 [==============================] - 2s 3ms/step - loss: 0.0837 - accuracy: 0.9704
Epoch 24/50
750/750 [==============================] - 2s 2ms/step - loss: 0.0793 - accuracy: 0.9706
Epoch 25/50
750/750 [==============================] - 2s 2ms/step - loss: 0.0743 - accuracy: 0.9721
Epoch 26/50
750/750 [==============================] - 2s 3ms/step - loss: 0.0813 - accuracy: 0.9698
Epoch 27/50
750/750 [==============================] - 2s 3ms/step - loss: 0.0736 - accuracy: 0.9732
Epoch 28/50
750/750 [==============================] - 2s 3ms/step - loss: 0.0781 - accuracy: 0.9712
Epoch 29/50
750/750 [==============================] - 2s 2ms/step - loss: 0.0701 - accuracy: 0.9743
Epoch 30/50
750/750 [==============================] - 2s 2ms/step - loss: 0.0722 - accuracy: 0.9730
Epoch 31/50
750/750 [==============================] - 2s 3ms/step - loss: 0.0815 - accuracy: 0.9703
Epoch 32/50
750/750 [==============================] - 2s 3ms/step - loss: 0.0667 - accuracy: 0.9751
Epoch 33/50
750/750 [==============================] - 2s 2ms/step - loss: 0.0687 - accuracy: 0.9750
Epoch 34/50
750/750 [==============================] - 2s 3ms/step - loss: 0.0705 - accuracy: 0.9737
Epoch 35/50
750/750 [==============================] - 2s 2ms/step - loss: 0.0693 - accuracy: 0.9743
Epoch 36/50
750/750 [==============================] - 2s 2ms/step - loss: 0.0649 - accuracy: 0.9759
```

```
Epoch 37/50
750/750 [==============================] - 2s 2ms/step - loss: 0.0667 - accuracy: 0.9746
Epoch 38/50
750/750 [==============================] - 2s 2ms/step - loss: 0.0695 - accuracy: 0.9740
Epoch 39/50
750/750 [==============================] - 2s 2ms/step - loss: 0.0673 - accuracy: 0.9744
Epoch 40/50
750/750 [==============================] - 2s 2ms/step - loss: 0.0591 - accuracy: 0.9771
Epoch 41/50
750/750 [==============================] - 2s 2ms/step - loss: 0.0579 - accuracy: 0.9783
Epoch 42/50
750/750 [==============================] - 2s 3ms/step - loss: 0.0672 - accuracy: 0.9748
Epoch 43/50
750/750 [==============================] - 2s 3ms/step - loss: 0.0710 - accuracy: 0.9735
Epoch 44/50
750/750 [==============================] - 2s 3ms/step - loss: 0.0600 - accuracy: 0.9769
Epoch 45/50
750/750 [==============================] - 2s 3ms/step - loss: 0.0591 - accuracy: 0.9777
Epoch 46/50
750/750 [==============================] - 2s 2ms/step - loss: 0.0650 - accuracy: 0.9756
Epoch 47/50
750/750 [==============================] - 2s 2ms/step - loss: 0.0613 - accuracy: 0.9767
Epoch 48/50
750/750 [==============================] - 2s 2ms/step - loss: 0.0563 - accuracy: 0.9783
Epoch 49/50
750/750 [==============================] - 2s 2ms/step - loss: 0.0531 - accuracy: 0.9794
Epoch 50/50
750/750 [==============================] - 2s 2ms/step - loss: 0.0650 - accuracy: 0.9758
375/375 [==============================] - 1s 1ms/step - loss: 4.0429 - accuracy: 0.5889
```

In [40]:
```
print("Accuracy of Top 10 FNN model: ",fnn_top10_accuracy)
```

```
Accuracy of Top 10 FNN model:  0.5889166593551636
```

What do you conclude by comparing accuracy values you obtain with those obtained in the "'Simple Models" section.

Reasoning:

The results obtained from simple models are slightly better than the FNN and top 10 FNN. There is a possibility of FNN giving better accuracies if we build a much more complex architecture. The FNN model that uses only the first 10 word embeddings concatenated to represent the entire review has poorer performance compared to the traditional FNN implementation and the simple models.

The reasons for this poor performance: Just using the first 10 words may not be sufficient and concatenating word embeddings may not accurately capture the contextual semantics.

5. Recurrent Neural Networks

Preparing data for RNN, GRU and LSTM code
Limiting the maximum review length to 20 by truncating longer reviews and padding shorter reviews with a null value (0).
Adding a random string value that is not present in word2vec model as part of padding if the length < 20

In [41]:
```
X_train_processed = []
for sentence in X_train.tolist():
    # Split the sentence by space and take the first 20 words
    words = sentence.split(' ')[:20]

    # Pad the rest with random_word if the sentence is less than 20 words
    words += ['random_word'] * (20 - len(words))

    # Replace each word with its corresponding vector or the default vector
    vectorized_words = []
    for word in words:
        if word in google_news_word2vec:
            vectorized_words.append(google_news_word2vec[word])
        else:
            vectorized_words.append(np.zeros((300,)))

    # Append the processed sentence to the output array
    X_train_processed.append(vectorized_words)

# Convert the output array to a numpy array
X_train = np.array(X_train_processed)
```

In [42]:
```python
# Process X_train
X_test_processed = []
for sentence in X_test.tolist():
    # Split the sentence by space and take the first 20 words
    words = sentence.split(' ')[:20]

    # Pad the rest with random_word if the sentence is less than 20 words
    words += ['random_word'] * (20 - len(words))

    # Replace each word with its corresponding vector or the default vector
    vectorized_words = []
    for word in words:
        if word in google_news_word2vec:
            vectorized_words.append(google_news_word2vec[word])
        else:
            vectorized_words.append(np.zeros((300,)))

    # Append the processed sentence to the output array
    X_test_processed.append(vectorized_words)

# Convert the output array to a numpy array
X_test = np.array(X_test_processed)
```

(a) Simple RNN

RNN works on the principle of saving the output of a particular layer and feeding this back to the input in order to predict the output of the layer.

RNN code with input layer dimensions of (20,300) and 1 SimpleRNN layer with hidden state size of 20. "y values-1" is taken to have class labels as 0, 1, 2 instead of 1, 2, 3

In [43]:
```python
def RNN(x_train, y_train, x_test, y_test, epochs, batch_size, learning_rate_val):
    model_rnn = tf.keras.Sequential([ tf.keras.layers.InputLayer((20,300)),
                                    tf.keras.layers.SimpleRNN(20),
                                    tf.keras.layers.Dense(3,activation='softmax')])


    model_rnn.compile (
                        optimizer = Adam(learning_rate = learning_rate_val),
                        loss='sparse_categorical_crossentropy',
                        metrics=['accuracy']
                    )

    print(model_rnn.summary())

    model_rnn.fit(x_train,y_train-1, batch_size = batch_size, epochs = epochs)
    result = model_rnn.evaluate(x_test,y_test-1)
    return result[1]
```

In [44]:
```
rnn_accuracy = RNN(X_train, y_train, X_test, y_test, 50, 64, 0.001)
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn (SimpleRNN)      (None, 20)                6420

 dense_6 (Dense)             (None, 3)                 63

=================================================================
Total params: 6,483
Trainable params: 6,483
Non-trainable params: 0
_________________________________________________________________
None
Epoch 1/50
750/750 [==============================] - 4s 4ms/step - loss: 0.9362 - accuracy: 0.5282
Epoch 2/50
750/750 [==============================] - 3s 4ms/step - loss: 0.8074 - accuracy: 0.6369
Epoch 3/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7865 - accuracy: 0.6488
Epoch 4/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7748 - accuracy: 0.6524
Epoch 5/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7668 - accuracy: 0.6559
Epoch 6/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7613 - accuracy: 0.6589
Epoch 7/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7570 - accuracy: 0.6606
Epoch 8/50
750/750 [==============================] - 6s 8ms/step - loss: 0.7522 - accuracy: 0.6634
Epoch 9/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7477 - accuracy: 0.6662
Epoch 10/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7426 - accuracy: 0.6695
Epoch 11/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7402 - accuracy: 0.6719
Epoch 12/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7374 - accuracy: 0.6708
Epoch 13/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7333 - accuracy: 0.6735
Epoch 14/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7306 - accuracy: 0.6759
Epoch 15/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7265 - accuracy: 0.6795
Epoch 16/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7253 - accuracy: 0.6785
Epoch 17/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7216 - accuracy: 0.6809
Epoch 18/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7214 - accuracy: 0.6788
Epoch 19/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7149 - accuracy: 0.6837
Epoch 20/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7130 - accuracy: 0.6851
Epoch 21/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7113 - accuracy: 0.6852
Epoch 22/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7082 - accuracy: 0.6857
Epoch 23/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7082 - accuracy: 0.6885
Epoch 24/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7046 - accuracy: 0.6902
Epoch 25/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7011 - accuracy: 0.6907
Epoch 26/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6990 - accuracy: 0.6921
Epoch 27/50
750/750 [==============================] - 3s 4ms/step - loss: 0.7003 - accuracy: 0.6938
Epoch 28/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6954 - accuracy: 0.6950
Epoch 29/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6957 - accuracy: 0.6944
Epoch 30/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6942 - accuracy: 0.6963
Epoch 31/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6905 - accuracy: 0.6979
Epoch 32/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6901 - accuracy: 0.6969
Epoch 33/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6866 - accuracy: 0.7010
Epoch 34/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6849 - accuracy: 0.6991
Epoch 35/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6833 - accuracy: 0.6993
Epoch 36/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6817 - accuracy: 0.7028
Epoch 37/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6796 - accuracy: 0.7031
```

```
Epoch 38/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6790 - accuracy: 0.7032
Epoch 39/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6778 - accuracy: 0.7035
Epoch 40/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6763 - accuracy: 0.7047
Epoch 41/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6768 - accuracy: 0.7054
Epoch 42/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6748 - accuracy: 0.7057
Epoch 43/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6743 - accuracy: 0.7074
Epoch 44/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6733 - accuracy: 0.7082
Epoch 45/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6709 - accuracy: 0.7069
Epoch 46/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6718 - accuracy: 0.7082
Epoch 47/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6685 - accuracy: 0.7092
Epoch 48/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6681 - accuracy: 0.7106
Epoch 49/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6684 - accuracy: 0.7087
Epoch 50/50
750/750 [==============================] - 3s 4ms/step - loss: 0.6661 - accuracy: 0.7127
375/375 [==============================] - 1s 2ms/step - loss: 0.8056 - accuracy: 0.6435
```

In [45]:
```
print("Accuracy of RNN model:",rnn_accuracy)
```

```
Accuracy of RNN model: 0.6434999704360962
```

(b) GRU

A gated recurrent unit (GRU) is part of a specific model of recurrent neural network that intends to use connections through a sequence of nodes to perform machine learning tasks associated with memory and clustering, for instance, in speech recognition. Gated recurrent units help to adjust neural network input weights to solve the vanishing gradient problem that is a common issue with recurrent neural networks.

GRU code with input layer dimensions of (20,300) and 1 GRU layer with hidden state size of 20. "y values-1" is taken to have class labels as 0, 1, 2 instead of 1, 2, 3

In [46]:
```
def GRU(x_train, y_train, x_test, y_test, epochs, batch_size, learning_rate_val):
    model_gru = tf.keras.Sequential([ tf.keras.layers.InputLayer((20,300)),
                                tf.keras.layers.GRU(20),
                                tf.keras.layers.Dense(3,activation='softmax')])


    model_gru.compile (
                    optimizer = Adam(learning_rate = learning_rate_val),
                    loss='sparse_categorical_crossentropy',
                    metrics=['accuracy']
                )

    print(model_gru.summary())

    model_gru.fit(x_train,y_train-1, batch_size = batch_size, epochs = epochs)

    result = model_gru.evaluate(x_test,y_test-1)
    return result[1]
```

In [47]:
```
gru_accuracy = GRU(X_train, y_train, X_test, y_test, 50, 64, 0.001)
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 gru (GRU)                   (None, 20)                19320

 dense_7 (Dense)             (None, 3)                 63

=================================================================
Total params: 19,383
Trainable params: 19,383
Non-trainable params: 0
_________________________________________________________________
None
Epoch 1/50
750/750 [==============================] - 6s 7ms/step - loss: 0.8849 - accuracy: 0.5584
Epoch 2/50
750/750 [==============================] - 5s 7ms/step - loss: 0.7413 - accuracy: 0.6682
Epoch 3/50
750/750 [==============================] - 5s 7ms/step - loss: 0.7127 - accuracy: 0.6820
Epoch 4/50
750/750 [==============================] - 5s 7ms/step - loss: 0.6931 - accuracy: 0.6933
Epoch 5/50
750/750 [==============================] - 5s 7ms/step - loss: 0.6808 - accuracy: 0.7004
Epoch 6/50
750/750 [==============================] - 5s 7ms/step - loss: 0.6696 - accuracy: 0.7063
Epoch 7/50
750/750 [==============================] - 5s 7ms/step - loss: 0.6603 - accuracy: 0.7119
Epoch 8/50
750/750 [==============================] - 5s 7ms/step - loss: 0.6503 - accuracy: 0.7169
Epoch 9/50
750/750 [==============================] - 5s 7ms/step - loss: 0.6430 - accuracy: 0.7196
Epoch 10/50
750/750 [==============================] - 5s 7ms/step - loss: 0.6372 - accuracy: 0.7223
Epoch 11/50
750/750 [==============================] - 5s 7ms/step - loss: 0.6306 - accuracy: 0.7265
Epoch 12/50
750/750 [==============================] - 5s 7ms/step - loss: 0.6243 - accuracy: 0.7283
Epoch 13/50
750/750 [==============================] - 5s 7ms/step - loss: 0.6196 - accuracy: 0.7308
Epoch 14/50
750/750 [==============================] - 5s 7ms/step - loss: 0.6130 - accuracy: 0.7341
Epoch 15/50
750/750 [==============================] - 5s 7ms/step - loss: 0.6073 - accuracy: 0.7387
Epoch 16/50
750/750 [==============================] - 5s 7ms/step - loss: 0.6035 - accuracy: 0.7396
Epoch 17/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5986 - accuracy: 0.7418
Epoch 18/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5938 - accuracy: 0.7442
Epoch 19/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5894 - accuracy: 0.7472
Epoch 20/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5841 - accuracy: 0.7488
Epoch 21/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5806 - accuracy: 0.7513
Epoch 22/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5768 - accuracy: 0.7515
Epoch 23/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5717 - accuracy: 0.7547
Epoch 24/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5685 - accuracy: 0.7567
Epoch 25/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5646 - accuracy: 0.7586
Epoch 26/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5601 - accuracy: 0.7610
Epoch 27/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5555 - accuracy: 0.7621
Epoch 28/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5527 - accuracy: 0.7644
Epoch 29/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5482 - accuracy: 0.7669
Epoch 30/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5459 - accuracy: 0.7674
Epoch 31/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5422 - accuracy: 0.7686
Epoch 32/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5390 - accuracy: 0.7718
Epoch 33/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5347 - accuracy: 0.7735
Epoch 34/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5311 - accuracy: 0.7747
Epoch 35/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5283 - accuracy: 0.7753
Epoch 36/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5244 - accuracy: 0.7793
Epoch 37/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5215 - accuracy: 0.7802
```

```
Epoch 38/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5183 - accuracy: 0.7800
Epoch 39/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5140 - accuracy: 0.7823
Epoch 40/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5123 - accuracy: 0.7833
Epoch 41/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5100 - accuracy: 0.7850
Epoch 42/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5055 - accuracy: 0.7875
Epoch 43/50
750/750 [==============================] - 5s 7ms/step - loss: 0.5012 - accuracy: 0.7896
Epoch 44/50
750/750 [==============================] - 5s 7ms/step - loss: 0.4988 - accuracy: 0.7902
Epoch 45/50
750/750 [==============================] - 5s 7ms/step - loss: 0.4960 - accuracy: 0.7921
Epoch 46/50
750/750 [==============================] - 5s 7ms/step - loss: 0.4936 - accuracy: 0.7932
Epoch 47/50
750/750 [==============================] - 5s 7ms/step - loss: 0.4897 - accuracy: 0.7958
Epoch 48/50
750/750 [==============================] - 5s 7ms/step - loss: 0.4885 - accuracy: 0.7937
Epoch 49/50
750/750 [==============================] - 5s 7ms/step - loss: 0.4856 - accuracy: 0.7971
Epoch 50/50
750/750 [==============================] - 5s 7ms/step - loss: 0.4823 - accuracy: 0.7975
375/375 [==============================] - 1s 3ms/step - loss: 0.8137 - accuracy: 0.6848
```

In [48]:
```
print("Accuracy of GRU model:",gru_accuracy)
```

```
Accuracy of GRU model: 0.6847500205039978
```

(c) LSTM

LSTM stands for long short-term memory networks, used in the field of Deep Learning. It is a variety of recurrent neural networks (RNNs) that are capable of learning long-term dependencies, especially in sequence prediction problems.

LSTM code with input layer dimensions of (20,300) and 1 LSTM layer with hidden state size of 20. "y values-1" is taken to have class labels as 0, 1, 2 instead of 1, 2, 3

In [49]:
```
def LSTM(x_train, y_train, x_test, y_test, epochs, batch_size, learning_rate_val):
    model_lstm = tf.keras.Sequential([ tf.keras.layers.InputLayer((20,300)),
                                    tf.keras.layers.LSTM(20),
                                    tf.keras.layers.Dense(3,activation='softmax')])


    model_lstm.compile (
                        optimizer = Adam(learning_rate = learning_rate_val),
                        loss='sparse_categorical_crossentropy',
                        metrics=['accuracy']
                    )

    print(model_lstm.summary())

    model_lstm.fit(x_train,y_train-1, batch_size = batch_size, epochs = epochs)

    result = model_lstm.evaluate(x_test,y_test-1)
    return result[1]
```

In [50]:

```
lstm_accuracy = LSTM(X_train, y_train, X_test, y_test, 50, 64, 0.001)
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm (LSTM)                 (None, 20)                25680

 dense_8 (Dense)             (None, 3)                 63

=================================================================
Total params: 25,743
Trainable params: 25,743
Non-trainable params: 0
_________________________________________________________________
None
Epoch 1/50
750/750 [==============================] - 7s 7ms/step - loss: 0.8411 - accuracy: 0.6051
Epoch 2/50
750/750 [==============================] - 6s 8ms/step - loss: 0.7454 - accuracy: 0.6693
Epoch 3/50
750/750 [==============================] - 6s 8ms/step - loss: 0.7193 - accuracy: 0.6808
Epoch 4/50
750/750 [==============================] - 6s 7ms/step - loss: 0.7024 - accuracy: 0.6901
Epoch 5/50
750/750 [==============================] - 6s 7ms/step - loss: 0.6875 - accuracy: 0.6961
Epoch 6/50
750/750 [==============================] - 6s 8ms/step - loss: 0.6752 - accuracy: 0.7018
Epoch 7/50
750/750 [==============================] - 6s 8ms/step - loss: 0.6628 - accuracy: 0.7084
Epoch 8/50
750/750 [==============================] - 6s 7ms/step - loss: 0.6553 - accuracy: 0.7116
Epoch 9/50
750/750 [==============================] - 6s 7ms/step - loss: 0.6441 - accuracy: 0.7160
Epoch 10/50
750/750 [==============================] - 6s 7ms/step - loss: 0.6361 - accuracy: 0.7232
Epoch 11/50
750/750 [==============================] - 6s 8ms/step - loss: 0.6276 - accuracy: 0.7269
Epoch 12/50
750/750 [==============================] - 6s 8ms/step - loss: 0.6204 - accuracy: 0.7310
Epoch 13/50
750/750 [==============================] - 6s 8ms/step - loss: 0.6132 - accuracy: 0.7344
Epoch 14/50
750/750 [==============================] - 6s 8ms/step - loss: 0.6058 - accuracy: 0.7387
Epoch 15/50
750/750 [==============================] - 6s 7ms/step - loss: 0.5985 - accuracy: 0.7423
Epoch 16/50
750/750 [==============================] - 6s 8ms/step - loss: 0.5943 - accuracy: 0.7437
Epoch 17/50
750/750 [==============================] - 6s 8ms/step - loss: 0.5884 - accuracy: 0.7468
Epoch 18/50
750/750 [==============================] - 6s 8ms/step - loss: 0.5808 - accuracy: 0.7508
Epoch 19/50
750/750 [==============================] - 6s 7ms/step - loss: 0.5740 - accuracy: 0.7534
Epoch 20/50
750/750 [==============================] - 6s 8ms/step - loss: 0.5701 - accuracy: 0.7561
Epoch 21/50
750/750 [==============================] - 6s 8ms/step - loss: 0.5636 - accuracy: 0.7601
Epoch 22/50
750/750 [==============================] - 6s 7ms/step - loss: 0.5587 - accuracy: 0.7633
Epoch 23/50
750/750 [==============================] - 6s 8ms/step - loss: 0.5541 - accuracy: 0.7644
Epoch 24/50
750/750 [==============================] - 6s 8ms/step - loss: 0.5485 - accuracy: 0.7668
Epoch 25/50
750/750 [==============================] - 6s 8ms/step - loss: 0.5447 - accuracy: 0.7706
Epoch 26/50
750/750 [==============================] - 6s 8ms/step - loss: 0.5382 - accuracy: 0.7728
Epoch 27/50
750/750 [==============================] - 6s 8ms/step - loss: 0.5330 - accuracy: 0.7746
Epoch 28/50
750/750 [==============================] - 6s 8ms/step - loss: 0.5276 - accuracy: 0.7782
Epoch 29/50
750/750 [==============================] - 6s 8ms/step - loss: 0.5256 - accuracy: 0.7791
Epoch 30/50
750/750 [==============================] - 6s 7ms/step - loss: 0.5193 - accuracy: 0.7821
Epoch 31/50
750/750 [==============================] - 6s 8ms/step - loss: 0.5139 - accuracy: 0.7841
Epoch 32/50
750/750 [==============================] - 6s 8ms/step - loss: 0.5123 - accuracy: 0.7855
Epoch 33/50
750/750 [==============================] - 6s 8ms/step - loss: 0.5052 - accuracy: 0.7891
Epoch 34/50
750/750 [==============================] - 6s 8ms/step - loss: 0.5022 - accuracy: 0.7905
Epoch 35/50
750/750 [==============================] - 6s 8ms/step - loss: 0.4972 - accuracy: 0.7945
Epoch 36/50
750/750 [==============================] - 6s 8ms/step - loss: 0.4946 - accuracy: 0.7943
Epoch 37/50
750/750 [==============================] - 6s 8ms/step - loss: 0.4899 - accuracy: 0.7968
```

```
Epoch 38/50
750/750 [==============================] - 6s 8ms/step - loss: 0.4857 - accuracy: 0.7992
Epoch 39/50
750/750 [==============================] - 6s 8ms/step - loss: 0.4813 - accuracy: 0.8008
Epoch 40/50
750/750 [==============================] - 6s 8ms/step - loss: 0.4782 - accuracy: 0.8019
Epoch 41/50
750/750 [==============================] - 6s 8ms/step - loss: 0.4744 - accuracy: 0.8045
Epoch 42/50
750/750 [==============================] - 6s 8ms/step - loss: 0.4704 - accuracy: 0.8065
Epoch 43/50
750/750 [==============================] - 6s 8ms/step - loss: 0.4657 - accuracy: 0.8070
Epoch 44/50
750/750 [==============================] - 6s 8ms/step - loss: 0.4626 - accuracy: 0.8101
Epoch 45/50
750/750 [==============================] - 7s 9ms/step - loss: 0.4599 - accuracy: 0.8110
Epoch 46/50
750/750 [==============================] - 6s 9ms/step - loss: 0.4555 - accuracy: 0.8133
Epoch 47/50
750/750 [==============================] - 6s 9ms/step - loss: 0.4512 - accuracy: 0.8153
Epoch 48/50
750/750 [==============================] - 7s 9ms/step - loss: 0.4473 - accuracy: 0.8171
Epoch 49/50
750/750 [==============================] - 6s 9ms/step - loss: 0.4472 - accuracy: 0.8187
Epoch 50/50
750/750 [==============================] - 6s 9ms/step - loss: 0.4437 - accuracy: 0.8205
375/375 [==============================] - 2s 3ms/step - loss: 0.8732 - accuracy: 0.6710
```

In [51]:
```
print("Accuracy of LSTM model:",lstm_accuracy)
```

```
Accuracy of LSTM model: 0.6710000038146973
```

Accuracy values

In [52]:
```
print("Accuracy of Perceptron on TF-IDF data : ", perceptron_accuracy_tfidf)
print("Accuracy of SVM on TF-IDF data : ", svc_accuarcy_tfidf)
print("Accuracy of Perceptron on Word2Vec data : ", perceptron_accuracy_w2v)
print("Accuracy of SVM on Word2Vec data : ", svc_accuracy_w2v)
print("Accuracy of FNN model: ",fnn_accuracy)
print("Accuracy of Top 10 FNN model: ",fnn_top10_accuracy)
print("Accuracy of RNN model:",rnn_accuracy)
print("Accuracy of GRU model:",gru_accuracy)
print("Accuracy of LSTM model:",lstm_accuracy)
```

```
Accuracy of Perceptron on TF-IDF data :  0.7114166666666667
Accuracy of SVM on TF-IDF data :  0.7431666666666666
Accuracy of Perceptron on Word2Vec data :  0.5465
Accuracy of SVM on Word2Vec data :  0.6711666666666667
Accuracy of FNN model:  0.6727499961853027
Accuracy of Top 10 FNN model:  0.5889166593551636
Accuracy of RNN model: 0.6434999704360962
Accuracy of GRU model: 0.6847500205039978
Accuracy of LSTM model: 0.6710000038146973
```

What do you conclude by comparing accuracy values you obtain by GRU, LSTM, and Simple RNN.

Reasoning:
The accuracy values of SimpleRNN, GRU and LSTM are in this order: LSTM >= GRU > SimpleRNN. The reason could be that it is common to observe that GRU and LSTM tend to outperform Simple RNN in tasks that require processing of long-term dependencies or maintaining memory over a longer period of time. This is because GRU and LSTM have more sophisticated gating mechanisms that allow them to selectively forget or store information in their memory cells, while Simple RNN lacks these mechanisms and is prone to the vanishing gradient problem.

Thank You