In [2]:

```
#!pip install regex

### REFERENCES
#https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html
#https://github.com/christianversloot/machine-learning-articles/blob/main/creating-a-mult
```

In [ ]:

```
# Required Dependencies
"""
contractions        NA
gensim              4.3.0
nltk                3.8.1
numpy               1.23.5
pandas              1.5.3
session_info        1.0.0
sklearn             1.2.1
torch               1.12.1
torchvision         0.13.1
"""
```

Importing Libraries

In [3]:

```
# Importing all the necessary Libraries
import pandas as pd
import numpy as np
import gensim.downloader as api
import gensim
import nltk
import os
import html
import re
import contractions
import torch
from nltk.stem import WordNetLemmatizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
from sklearn.linear_model import Perceptron
from sklearn.svm import LinearSVC
from torch import nn
from torch.utils.data import DataLoader
from torchvision import transforms
from torchvision.datasets import CIFAR10
from itertools import chain
#nltk.download('wordnet')
#nltk.download('punkt')
```

```
C:\Users\vatsa\anaconda3\envs\venv\lib\site-packages\tqdm\auto.py:22: TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html (https://ipywidgets.readthedocs.io/en/stable/user_install.html)
  from .autonotebook import tqdm as notebook_tqdm
```

1. Dataset Generation

In [4]:

```
# Reading the data
df = pd.read_table('data.tsv', on_bad_lines='skip')
```

In [5]:

```
# Count the no of null values
df.isna().any(axis=1).sum()

# Dropping NULL rows
df = df.dropna()
```

In [6]:

```
# Converting Star Rating into three categories
df['star_rating'] = df['star_rating'].replace([2,3,4,5,'1','2','3','4','5'],[1,2,3,3,1,1,

# Removing some irrelevant rows in star rating feature
df = df[df['star_rating'].isin([1,2,3])]

#Converting the datatype from Object to INT
df['star_rating'] = df['star_rating'].astype(str).astype(float).astype(int)
```

Out[6]:

```
"\n# Converting Star Rating into three categories\ndf['star_rating'] = df
['star_rating'].replace([2,3,4,5,'1','2','3','4','5'],[1,2,3,3,1,1,2,3,3])
\n\n# Removing some irrelevant rows in star rating feature\ndf = df[df['st
ar_rating'].isin([1,2,3])]\n\n#Converting the datatype from Object to INT
\ndf['star_rating'] = df['star_rating'].astype(str).astype(float).astype(i
nt)\n"
```

In [7]:

```
# Stratified Sampling
df = df.groupby('star_rating', group_keys=False).apply(lambda x: x.sample(20000))
df.index = (np.arange(60000))
```

In [8]:

```
# Storing the Sampled Dataset
df.to_csv('Sampled_data.csv', index=False)
```

In [9]:

```
df = pd.read_csv('Sampled_data.csv')
```

In [10]:

```
def clean(text):
    # convert html escapes like &amp; to characters.
    text = html.unescape(text)
    # tags like <tab>
    text = re.sub(r'<[^<>]*>', ' ', text)
    # markdown URLs like [Some text](https://....)
    text = re.sub(r'\[([^\[\]]*)\]\([^\(\)]*\)', r'\1', text)
    # text or code in brackets like [0]
    text = re.sub(r'\[[^\[\]]*\]', ' ', text)
    # standalone sequences of specials, matches &# but not #cool
    text = re.sub(r'(?:^|\s)[&#<>{}\[\]+|\\:-]{1,}(?:\s|$)', ' ', text)
    # standalone sequences of hyphens like --- or ==
    text = re.sub(r'(?:^|\s)[\-=\+]{2,}(?:\s|$)', ' ', text)
    # <3 to love
    text = re.sub(r'<3', 'love', text)
    # Removing \\"
    text = re.sub(r'\\"', '', text)
    # sequences of white spaces
    text = re.sub(r'\s+', ' ', text)
    return text.strip()
```

In [11]:

```
def lm(word_tokens):
    lemmatizer = WordNetLemmatizer()
    return [lemmatizer.lemmatize(word) for word in word_tokens]
```

In [12]:

```
def remove_puctuations(text):
  # Remove punctuation
  text = re.sub(r'[^\w\s]', '', text)
  return text
```

In [13]:

```
# Cleaning the dataset
df = df[['star_rating', 'review_body']]
df['review_body'] = df['review_body'].map(lambda x: contractions.fix(x))
df['review_body'] = df['review_body'].map(clean)
df['review_body'] = df['review_body'].map(remove_puctuations)
df['review_body'] = df['review_body'].map(lambda x: nltk.word_tokenize(x))
df['review_body'] = df['review_body'].map(lm)

# Splitting the dataset
X = df.drop(columns='star_rating')
y = df['star_rating']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0,
```

2. Word Embedding

a)

In [14]:

```
# Loading the Google Word2Vec Model
wv = api.load('word2vec-google-news-300')
```

In [15]:

```
wv.most_similar(positive=['man', 'lady'], negative=['boy'], topn=1)
```

Out[15]:

```
[('woman', 0.5815584063529968)]
```

In [16]:

```
wv.most_similar(positive=['serum', 'shampoo'], topn=5)
```

Out[16]:

```
[('shampoos', 0.7020988464355469),
 ('moisturizer', 0.6901729106903076),
 ('serums', 0.6792107224464417),
 ('lotion', 0.668178915977478),
 ('shampoo_conditioner', 0.6428859233856201)]
```

In [17]:

```
wv.doesnt_match(['fire', 'water', 'land', 'sea', 'air', 'car'])
```

Out[17]:

```
'car'
```

b)

In [18]:

```
# Training our own Word2Vec Model
model = gensim.models.Word2Vec(sentences=df["review_body"],min_count=9,vector_size=300,wi
```

In [19]:

```
model.wv.most_similar(positive=['man', 'lady'], negative=['boy'], topn=1)
```

Out[19]:

```
[('woman', 0.6743051409721375)]
```

In [20]:

```
model.wv.most_similar(positive=['serum', 'shampoo'], topn=5)
```

Out[20]:

```
[('conditioner', 0.7931665182113647),
 ('Shampoo', 0.7175987958908081),
 ('toner', 0.7115134596824646),
 ('vitamin', 0.667493462562561),
 ('product', 0.6660668849945068)]
```

In [21]:

```
model.wv.doesnt_match(['fire', 'water', 'land', 'sea', 'air', 'car'])
```

Out[21]:

```
'sea'
```

Q: What do you conclude from comparing vectors generated by yourself and the pretrained model? Which of the Word2Vec models seems to encode semantic similarities between words better?
Ans: The vectors generated from pretrained model are more accurate than the model that is trained on the dataset. The pretrained model covers a wide range of words as it is trained on a large corpus. For Example: "King - Man + Woman = Queen" will give a key error in my trained Word2Vec model beacuse it only has a limited amount of words for training. As the pretrained model has many words in its dictionary it is encoding semantic similarities better between words. Another Example is the doesn't match in the list ['fire', 'water', 'land', 'sea', 'air', 'car']. It should be car because all other are natural objects. The pretrained model is able to find the difference but my trained model doesn't.

3. Simple Models

In [22]:

```
def get_mean_word_embeddings(df):
    """
    Function to get the mean Embedding vector
    """
    word_embeddings = []
    for r in df["review_body"]:
        vec = np.zeros(300)
        c = 0
        for word in r:
            try:
                vec += wv[word]
                c += 1
            except KeyError:
                pass
        if c>0:
            word_embeddings.append(vec/c)
        else:
            word_embeddings.append(vec)
    return word_embeddings
```

In [23]:

```
x = get_mean_word_embeddings(X_train)
x_te = get_mean_word_embeddings(X_test)
```

In [24]:

```
# Training and Evaluating Perceptron
perceptron = Perceptron(random_state=25)
perceptron.fit(x,y_train)
y_pred = perceptron.predict(x_te)
y = classification_report(y_pred,y_test,output_dict=True)
print("Accuracy of Word2Vec Perceptron: ",y["accuracy"])
print("Accuracy of TF-IDF Perceptron: 0.6459166666666667")
```

```
Accuracy of Word2Vec Perceptron:  0.6315833333333334
Accuracy of TF-IDF Perceptron: 0.6459166666666667
```

In [25]:

```
# Training and Evaluating SVM
svc = LinearSVC(random_state = 25)
svc.fit(x,y_train)
y_pred = svc.predict(x_te)
y = classification_report(y_pred,y_test,output_dict=True)
print("Accuracy of Word2Vec SVC: ",y["accuracy"])
print("Accuracy of TF-IDF SVC: 0.7014166666666667")
```

```
Accuracy of Word2Vec SVC:  0.6504166666666666
Accuracy of TF-IDF SVC: 0.7014166666666667
```

Q: What do you conclude from comparing performances for the models trained using the two different feature types (TF-IDF and your trained Word2Vec features)?
Ans: TF-IDF features performed better than the Word2Vec feature on both Perceptron and SVM Model on the accuracy score. TF-IDF outperformed SVM by about 1.4% on Perceptron and around 5% on SVM.

4. Feedforward Neural Networks

In [26]:

```
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

In [27]:

```
class MLP(nn.Module):
  '''
    Multilayer Perceptron.
  '''
  def __init__(self):
    super().__init__()
    self.layers = nn.Sequential(
      nn.Linear(300, 100).cuda(),
      nn.ReLU(),
      nn.Linear(100, 10).cuda(),
      nn.ReLU(),
      nn.Linear(10, 3).cuda(),
      #nn.Dropout(0.2)
    )


  def forward(self, x):
    '''Forward pass'''
    return self.layers(x)
```

In [28]:

```
# Set fixed random number seed
torch.manual_seed(42)
```

Out[28]:

```
<torch._C.Generator at 0x21bc80d5970>
```

a)

In [29]:

```
def get_tuple(x,y):
    tmp = []
    for i in range(len(x)):
        tmp.append((x[i],y.iloc[i]-1))
    return tmp
```

In [30]:

```
trainloader = torch.utils.data.DataLoader(get_tuple(x,y_train),batch_size=64, shuffle=Tru
```

In [31]:

```
# Initialize the MLP
mlp = MLP()
mlp = mlp.cuda()
# Define the loss function and optimizer
loss_function = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(mlp.parameters(), lr=0.001)
```

In [147]:

```
# Run the training loop
for epoch in range(0, 75):

    # Print epoch
    print(f'Starting epoch {epoch+1}')

    # Set current loss value
    current_loss = 0.0

    # Iterate over the DataLoader for training data
    for i, data in enumerate(trainloader, 0):

        # Get inputs
        inputs, targets = data
        inputs = inputs.cuda()
        targets = targets.cuda()

        # Zero the gradients
        optimizer.zero_grad()

        # Perform forward pass
        outputs = mlp(inputs.float())

        # Compute loss
        loss = loss_function(outputs, targets)

        # Perform backward pass
        loss.backward()

        # Perform optimization
        optimizer.step()

        # Print statistics
        current_loss += loss.item()
    print("Loss After Epoch",epoch+1," :",current_loss/len(trainloader))

# Process is complete.
print('Training process has finished.')
```

```
Starting epoch 1
Loss After Epoch 0  : 0.885559449672699
Starting epoch 2
Loss After Epoch 1  : 0.7890025011698405
Starting epoch 3
Loss After Epoch 2  : 0.7708496094544729
Starting epoch 4
Loss After Epoch 3  : 0.759677870353063
Starting epoch 5
Loss After Epoch 4  : 0.7490026630560557
Starting epoch 6
Loss After Epoch 5  : 0.7406753311157227
Starting epoch 7
Loss After Epoch 6  : 0.7340897517999013
Starting epoch 8
Loss After Epoch 7  : 0.727361788113912
Starting epoch 9
Loss After Epoch 8  : 0.7204268963734309
Starting epoch 10
```

In [32]:

```
def predict(model, dataloader):
    """
        Function to get the predictions form a model
    """
    predictions = []
    for i, batch in enumerate(dataloader):
        outputs = model(batch.cuda().float())
        _, predicted = torch.max(outputs.data, 1)
        predictions.append(predicted.cpu())
    #predictions = np.array(predictions)
    for i in range(len(predictions)):
        predictions[i] = predictions[i].tolist()
    predictions = list(chain.from_iterable(predictions))
    predictions = [i+1 for i in predictions]
    return predictions
```

In [30]:

```
test_loader = torch.utils.data.DataLoader(x_te,batch_size=64, num_workers=1)
```

In [158]:

```
# Evaluating MLP
predictions = predict(mlp,test_loader)
y = classification_report(predictions,y_test,output_dict=True)
print(y["accuracy"])
```

```
0.6430833333333333
```

b)

In [32]:

```
def get_first_ten_concat_features(df):
    """
    Function to get the first 10 words concatenated Embedding vector
    """
    word_embeddings = []
    for r in df["review_body"]:
        vec = np.empty( shape=(0,) )
        c = 0
        j = 0
        while(c<10):
            try:
                a = wv[r[j]]
                #print("BELLO!!", c, r[j])
                vec = np.concatenate([vec, a])
                c += 1
                j += 1
            except KeyError:
                j += 1
                continue

            except IndexError:
                #print("BELLO!!", c)
                vec = np.concatenate([vec,np.zeros(300)])
                c += 1

        word_embeddings.append(vec)
    return word_embeddings
```

In [33]:

```
class MLP2(nn.Module):
  '''
    Multilayer Perceptron.
  '''
  def __init__(self):
    super().__init__()
    self.layers = nn.Sequential(
      nn.Linear(3000, 100).cuda(),
      nn.LeakyReLU(),
      nn.Linear(100, 10).cuda(),
      nn.LeakyReLU(),
      nn.Linear(10, 3).cuda(),
      #nn.Dropout(0.2)
    )


  def forward(self, x):
    '''Forward pass'''
    return self.layers(x)
```

In [34]:

```
x = get_first_ten_concat_features(X_train)
trainloader = torch.utils.data.DataLoader(get_tuple(x,y_train),batch_size=64, shuffle=Tru
```

In [35]:

```
# Initialize the MLP
mlp = MLP2()
mlp = mlp.cuda()
# Define the loss function and optimizer
loss_function = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(mlp.parameters(), lr=0.001)
```

In [36]:

```
# Run the training loop
for epoch in range(0, 5):

    # Print epoch
    print(f'Starting epoch {epoch+1}')

    # Set current loss value
    current_loss = 0.0

    # Iterate over the DataLoader for training data
    for i, data in enumerate(trainloader, 0):

        # Get inputs
        inputs, targets = data
        inputs = inputs.cuda()
        targets = targets.cuda()

        # Zero the gradients
        optimizer.zero_grad()

        # Perform forward pass
        outputs = mlp(inputs.float())

        # Compute loss
        loss = loss_function(outputs, targets)

        # Perform backward pass
        loss.backward()

        # Perform optimization
        optimizer.step()

        # Print statistics
        current_loss += loss.item()
    print("Loss After Epoch",epoch+1," :",current_loss/len(trainloader))

# Process is complete.
print('Training process has finished.')
```

```
Starting epoch 1
Loss After Epoch 1  : 0.9310969703992208
Starting epoch 2
Loss After Epoch 2  : 0.8373928763071696
Starting epoch 3
Loss After Epoch 3  : 0.7719850514729818
Starting epoch 4
Loss After Epoch 4  : 0.6889080033302307
Starting epoch 5
Loss After Epoch 5  : 0.5877161382834116
Training process has finished.
```

In [37]:

```
test_loader = torch.utils.data.DataLoader(get_first_ten_concat_features(X_test),batch_siz
```

In [38]:

```
# Evaluating the MLP
predictions = predict(mlp,test_loader)
y = classification_report(predictions,y_test,output_dict=True)
print(y["accuracy"])
```

```
0.5789166666666666
```

Q What do you conclude by comparing accuracy values you obtain with those obtained in the “’Simple Models” section.
Ans: The Accuracy is very similar for the mean word embedding MLP. It is outperforming Perceptron by around 1.2% and underperforming to SVM by only 0.7%. On the other hand the second MLP which only takes the first 10 words is underperforming both Perceptron and SVM by 5.3% and 7.2% respectively.

5. Recurrent Neural Networks

In [33]:

```
def get_first_twenty_concat_features(df):
    """
    Function to get the First 20 words Embedding vector
    """
    word_embeddings = []
    for r in df["review_body"]:
        vec = []
        c = 0
        j = 0
        while(c<20):
            try:
                a = wv[r[j]]
                #print("BELLO!!", c, r[j])
                vec.append(a)
                c += 1
                j += 1
            except KeyError:
                j += 1
                continue

            except IndexError:
                #print("BELLO!!", c)
                vec.append(np.zeros(300))
                c += 1

        word_embeddings.append(vec)
    return word_embeddings
```

In [46]:

```
def train_model(model, epochs, trainloader, device, optimizer, loss_function):
    """
        Function to train RNN based models.
    """
    for epoch in range(0, epochs): # 5 epochs at maximum

        # Print epoch
        print(f'Starting epoch {epoch+1}')

        model.train()
        train_loss_seq = 0.0

        for i, (train_data, target) in enumerate(trainloader):
            train_data = train_data.to(device)
            target = target.to(device)
            optimizer.zero_grad()
            output = model(train_data.float())
            loss = loss_function(output, target)
            loss.backward()
            optimizer.step()
            train_loss_seq += loss.item()*train_data.size(0)
        model.eval()

        train_loss_seq = train_loss_seq/len(trainloader.dataset)
        print("Epoch --> "+str(epoch+1)+" :", train_loss_seq)
    # Process is complete.
    print('Training process has finished.')
```

a)

In [71]:

```
class RNN(torch.nn.Module):
    def __init__(self, input_size=300, hidden_size=20,num_layers = 1,num_classes=3):
        super(RNN, self).__init__()
        self.num_layers = num_layers
        self.hidden_size = hidden_size
        self.rnn = torch.nn.RNN(input_size,hidden_size,num_layers,batch_first=True,dropou
        self.fc = torch.nn.Linear(hidden_size,num_classes)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers,x.size(0), self.hidden_size).to(device)

        out,_ = self.rnn(x,h0)
        out = out[:,-1,:]
        out = self.fc(out)
        return out
```

In [72]:

```
# Initializing RNN Model
rnn_model = RNN()
loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(rnn_model.parameters(), lr=1e-3)
rnn_model = rnn_model.cuda()
```

```
C:\Users\vatsa\anaconda3\envs\venv\lib\site-packages\torch\nn\modules\rnn.
py:62: UserWarning: dropout option adds dropout after all but last recurre
nt layer, so non-zero dropout expects num_layers greater than 1, but got d
ropout=0.5 and num_layers=1
  warnings.warn("dropout option adds dropout after all but last "
```

In [73]:

```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(rnn_model):,} trainable parameters')
```

```
The model has 6,503 trainable parameters
```

In [67]:

```
x = get_first_twenty_concat_features(X_train)
x = [np.asarray(i) for i in x]
trainloader = torch.utils.data.DataLoader(get_tuple(x,y_train),batch_size=64, shuffle=Tru
```

In [68]:

```
x_te = get_first_twenty_concat_features(X_test)
x_te = [np.asarray(i) for i in x_te]
test_loader = torch.utils.data.DataLoader(x_te,batch_size=64, num_workers=1)
```

In [74]:

```
train_model(rnn_model, 30, trainloader, device, optimizer, loss_function)
```

```
Starting epoch 1
Epoch --> 1 : 1.0396501942475638
Starting epoch 2
Epoch --> 2 : 0.9036978100935618
Starting epoch 3
Epoch --> 3 : 0.8653938625653584
Starting epoch 4
Epoch --> 4 : 0.8430932471752167
Starting epoch 5
Epoch --> 5 : 0.8271982341607411
Starting epoch 6
Epoch --> 6 : 0.8164633108774821
Starting epoch 7
Epoch --> 7 : 0.8055796077251435
Starting epoch 8
Epoch --> 8 : 0.7996100840568543
Starting epoch 9
Epoch --> 9 : 0.789910701751709
Starting epoch 10
Epoch --> 10 : 0.7850388867060344
Starting epoch 11
Epoch --> 11 : 0.7788948087692261
Starting epoch 12
Epoch --> 12 : 0.7737591408888499
Starting epoch 13
Epoch --> 13 : 0.7696171850363414
Starting epoch 14
Epoch --> 14 : 0.7654237356980642
Starting epoch 15
Epoch --> 15 : 0.7645855046908061
Starting epoch 16
Epoch --> 16 : 0.7593455042044321
Starting epoch 17
Epoch --> 17 : 0.7556989444891612
Starting epoch 18
Epoch --> 18 : 0.7520121556917826
Starting epoch 19
Epoch --> 19 : 0.7495156970818837
Starting epoch 20
Epoch --> 20 : 0.7468056027094523
Starting epoch 21
Epoch --> 21 : 0.7419921885331472
Starting epoch 22
Epoch --> 22 : 0.7386032441457112
Starting epoch 23
Epoch --> 23 : 0.7399375410874685
Starting epoch 24
Epoch --> 24 : 0.7361737392743428
Starting epoch 25
Epoch --> 25 : 0.7331608528693517
Starting epoch 26
Epoch --> 26 : 0.730109917640686
Starting epoch 27
Epoch --> 27 : 0.7317570678393046
Starting epoch 28
Epoch --> 28 : 0.7286343031724294
Starting epoch 29
Epoch --> 29 : 0.7257773742278417
Starting epoch 30
```

```
Epoch --> 30 : 0.7267016449769338
Training process has finished.
```

In [75]:

```
# Evaluating RNN
predictions = predict(rnn_model,test_loader)
y = classification_report(predictions,y_test,output_dict=True)
print(y["accuracy"])
```

```
0.6465833333333333
```

Q: What do you conclude by comparing accuracy values you obtain with those obtained with feedforward neural network models.
Ans: The RNN is outperforming both the Feed Forward neural networks. The first one by 0.3% and the second one by a large 6.8% difference in the accuracy score. As it is considering the dependency of previous words in the review it is able to differentiate better than the feedforward neural network.

b)

In [55]:

```
class GRU(torch.nn.Module):

    def __init__(self, input_size=300, hidden_size=20,num_layers = 1,num_classes=3):
        super(GRU, self).__init__()
        self.num_layers = num_layers
        self.hidden_size = hidden_size
        self.gru = torch.nn.GRU(input_size,hidden_size,num_layers,batch_first=True)
        self.fc = torch.nn.Linear(hidden_size,num_classes,bias=False)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers,x.size(0), self.hidden_size).to(device)

        out,_ = self.gru(x,h0)
        out = out[:,-1,:]
        out = self.fc(out)
        return out
```

In [56]:

```
# Initializing GRU
gru_model = GRU()
loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(gru_model.parameters(), lr=1e-3)
gru_model = gru_model.cuda()
```

In [57]:

```
print(f'The model has {count_parameters(gru_model):,} trainable parameters')
```

```
The model has 19,380 trainable parameters
```

In [58]:

```
train_model(gru_model, 20, trainloader, device, optimizer, loss_function)
```

```
Starting epoch 1
Epoch --> 1 : 0.9434848532676697
Starting epoch 2
Epoch --> 2 : 0.8038084135055542
Starting epoch 3
Epoch --> 3 : 0.7748165690898895
Starting epoch 4
Epoch --> 4 : 0.7536724455356598
Starting epoch 5
Epoch --> 5 : 0.7379129912853241
Starting epoch 6
Epoch --> 6 : 0.7261548719803492
Starting epoch 7
Epoch --> 7 : 0.7139907979170481
Starting epoch 8
Epoch --> 8 : 0.7038073006073634
Starting epoch 9
Epoch --> 9 : 0.6941189201672872
Starting epoch 10
Epoch --> 10 : 0.6851310456593831
Starting epoch 11
Epoch --> 11 : 0.6782764012813568
Starting epoch 12
Epoch --> 12 : 0.6699842751026154
Starting epoch 13
Epoch --> 13 : 0.6639130733410518
Starting epoch 14
Epoch --> 14 : 0.6566061506271362
Starting epoch 15
Epoch --> 15 : 0.6497284882863362
Starting epoch 16
Epoch --> 16 : 0.6450611269474029
Starting epoch 17
Epoch --> 17 : 0.6398731488784154
Starting epoch 18
Epoch --> 18 : 0.6330426509777705
Starting epoch 19
Epoch --> 19 : 0.6271306569973628
Starting epoch 20
Epoch --> 20 : 0.6230615017016728
Training process has finished.
```

In [59]:

```
#Evaluating GRU
predictions = predict(gru_model,test_loader)
y = classification_report(predictions,y_test,output_dict=True)
print(y["accuracy"])
```

```
0.6685
```

c)

In [60]:

```
class LSTM(torch.nn.Module):
    def __init__(self, input_size=300, hidden_size=20,num_layers = 1,num_classes=3):
        super(LSTM, self).__init__()
        self.num_layers = num_layers
        self.hidden_size = hidden_size
        self.lstm = torch.nn.LSTM(input_size,hidden_size,num_layers,batch_first=True)
        self.fc = torch.nn.Linear(hidden_size,num_classes)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers,x.size(0), self.hidden_size).to(device)
        c0 = torch.zeros(self.num_layers,x.size(0), self.hidden_size).to(device)

        out,_ = self.lstm(x,(h0,c0))
        out = out[:,-1,:]
        out = self.fc(out)
        return out
```

In [61]:

```
# Initializing LSTM
lstm_model = LSTM()
loss_function = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(lstm_model.parameters(), lr=1e-3)
lstm_model = lstm_model.cuda()
```

In [62]:

```
print(f'The model has {count_parameters(lstm_model):,} trainable parameters')
```

```
The model has 25,823 trainable parameters
```

In [63]:

```
train_model(lstm_model, 20, trainloader, device, optimizer, loss_function)
```

```
Starting epoch 1
Epoch --> 1 : 0.9605202089945475
Starting epoch 2
Epoch --> 2 : 0.8394982452392579
Starting epoch 3
Epoch --> 3 : 0.7956503780682882
Starting epoch 4
Epoch --> 4 : 0.7715079264640808
Starting epoch 5
Epoch --> 5 : 0.7535925794045131
Starting epoch 6
Epoch --> 6 : 0.74114122402668
Starting epoch 7
Epoch --> 7 : 0.7280267721017202
Starting epoch 8
Epoch --> 8 : 0.7159666108687719
Starting epoch 9
Epoch --> 9 : 0.7065933414697647
Starting epoch 10
Epoch --> 10 : 0.6971026578744253
Starting epoch 11
Epoch --> 11 : 0.6890288486083349
Starting epoch 12
Epoch --> 12 : 0.6797167123953501
Starting epoch 13
Epoch --> 13 : 0.6716538064082463
Starting epoch 14
Epoch --> 14 : 0.6662318698565165
Starting epoch 15
Epoch --> 15 : 0.658796049674352
Starting epoch 16
Epoch --> 16 : 0.6529413114786148
Starting epoch 17
Epoch --> 17 : 0.645723607579867
Starting epoch 18
Epoch --> 18 : 0.6404907112121582
Starting epoch 19
Epoch --> 19 : 0.6333893365462621
Starting epoch 20
Epoch --> 20 : 0.6285951400200526
Training process has finished.
```

In [64]:

```
#Evaluating LSTM
predictions = predict(lstm_model,test_loader)
y = classification_report(predictions,y_test,output_dict=True)
print(y["accuracy"])
```

```
0.6601666666666667
```

Q What do you conclude by comparing accuracy values you obtain by GRU, LSTM, and simple RNN.
Ans: LSTM and GRU both outperforms RNN by 1.4% and 2.2% respectively. These can be because of the more number of parameters that LSTM and GRU has. Both LSTM and GRU are giving similar performances.

In [ ]: