Import Libraries

All the libraries used are of the latest version. So if any library doesn't exist, the version that comes with a simple "pip install" should suffice

In [1]:
```
import pandas as pd
import numpy as np
import nltk
nltk.download('wordnet')
import re
from bs4 import BeautifulSoup
import contractions
from sklearn.model_selection import train_test_split
import re
from nltk.corpus import stopwords
nltk.download("stopwords")
from nltk.stem import PorterStemmer
from nltk import sent_tokenize,word_tokenize
nltk.download('punkt')
import gensim.downloader
#!pip install gensim
import gensim
import gensim.downloader
from gensim.test.utils import common_texts
from gensim.models import Word2Vec
import random
import torch
import torch.nn as nn
from torch.nn import functional as F
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import Perceptron
from sklearn.svm import LinearSVC
```

```
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\indra\AppData\Roaming\nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\indra\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\indra\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
G:\Pytorch_practice\lib\site-packages\tqdm\auto.py:22: TqdmWarning: IProgress n
ot found. Please update jupyter and ipywidgets. See https://ipywidgets.readthed
ocs.io/en/stable/user_install.html (https://ipywidgets.readthedocs.io/en/stabl
e/user_install.html)
  from .autonotebook import tqdm as notebook_tqdm
```

Import the data and select the review_body and ratings columns

Data is imported from the "data.tsv" file present in the same folder as this ipynb file. Pandas is the library used to read the data file. The "on_bad_lines" parameter skips the lines that have issues and throws and error while reading the data file such as missing values. The columns "review_body" and "star_rating" are taken from the data and used in this assignment.

In [2]:
```
org_df=pd.read_csv("./data.tsv",sep='\t',on_bad_lines="skip")
required_columns=['review_body','star_rating']
df=org_df[required_columns]
new_df=pd.DataFrame({"review_body":[],"star_rating":[]})
for i in range(1,6):
    new_df=pd.concat([new_df,df.loc[df['star_rating']==i].sample(50000)])
new_df.head()
```

```
G:\Pytorch_practice\lib\site-packages\IPython\core\interactiveshell.py:3457: Dt
ypeWarning: Columns (7) have mixed types.Specify dtype option on import or set
low_memory=False.
  exec(code_obj, self.user_global_ns, self.user_ns)
```

Out[2]:

	review_body	star_rating
1067158	very flimsy and cheap but with a cheap price i...	1
481032	The second time I wore these cuff links sadly ...	1
1690418	I didn't realize it was way too small. It is m...	1.0
1146052	The clasp and straps broke on the second day o...	1
1759339	I am displeased with this seller, because I no...	1

Function to preprocess the data

The data is preprocessed here to improve the quality of results. The steps followed in preprocessing the data in the "review_body" column are as follows:-

-> Removing html tags, urls, non-alphabetic characters other than space, extra space characters, are removed from the data using the regex.

-> Stopwords are removed from the data using the stopwords dictionary from the nltk library.

-> The inword2vec parameter is used as a flag which if set to True performs filtration in the data and retains only those words in the sentence that have corresponding word embeddings in the word2vec model.

-> The wordvec parameter takes as input the word2vec model which is used to check if a word is present as a key value in the model. This check is done only if the inword2vec parameter is set to the value True.

->The fixed size parameter is used to filter the sentences based on their length. Only those sentences are allowed to become a part of the final processed data whose number of words are >= fixedsize value.

-> The df parameter takes as input the raw data.

In [3]:
```
## Preprocessing

def preprocess_data(df,inword2vec=False,wordvec=None,fixedsize=None):
    remove_html_tags='<.*?>';
    remove_urls='http\S+';
    remove_non_alpha='[^A-Za-z ]'
    remove_extra_space=' +'
    processed={"review_body":[],"star_rating":[]}
    for i in range(len(new_df)):
        s=str(new_df['review_body'].iloc[i])
        c=new_df['star_rating'].iloc[i]
        if s=="":
            continue
        s=re.sub(remove_html_tags,"",s)
        s=re.sub(remove_urls,"",s)
        s=re.sub(remove_non_alpha,"",s)
        s=re.sub(remove_extra_space," ",s)
        if s=="":
            continue
        processed["review_body"].append(contractions.fix(s.lower()))
        processed["star_rating"].append(int(c))


    stop_words = set(stopwords.words('english'))
    final_processed_text={"input":[],"target":[]}
    count={1:0,2:0,3:0,4:0,5:0}

    for i in range(len(processed['review_body'])):
        s=processed['review_body'][i]
        c=processed['star_rating'][i]
        if count[int(c)]>=20000:
            continue
        s=s.split(" ")
        s=[word for word in s if word not in stop_words]
        if inword2vec:
            s=[word for word in s if word in wordvec.key_to_index]
        if not fixedsize:
            if len(s)<13:
                continue
            s=" ".join(s)
            tokenize_words=word_tokenize(s)
            final_processed_text['input'].append(tokenize_words)
            final_processed_text['target'].append(c)
            count[int(c)]+=1
        else:
            if len(s)>=fixedsize:
                s=" ".join(s)
                tokenize_words=word_tokenize(s)
                final_processed_text['input'].append(tokenize_words)
                final_processed_text['target'].append(c)
                count[int(c)]+=1

    return final_processed_text
```

Function to generate word2vec based word embeddings for the input data

The following function generates word2vec based word embeddings for the input data. The outputs are of three types:-

-> Each training sample is a mean vector embedding of the sentence in the sample. This happens when skipten==False.

-> Each training sample consists of a sublist of wordvectors and the number of words taken from the sentence is fixed. This happens when skipten==True. The fixedsize contains the value which is the required number of words from the sentence. Here the parameter "concat is set to False" as each word vector in the sample is present inside its own list and not concatenated with the rest of the word vectors. Thus each train sample with be list of lists.

-> Each training sample consists of a list that has all the word vectors concatenated with each other and the number of words taken from the sentence is fixed. This happens when skipten==True. The fixedsize contains the value which is the required number of words from the sentence. Here the parameter "concat is set to True" as each word vector is concatenated with the rest of the word vectors.

-> The parameter long when set to true makes the 0's and 1's in the one hot encoding of the labels to be of datatype long and when it is set to false, the datatype of the 0's and 1's is float.

In [4]:
```
### The role of the label encoding function is to generate one hot encoded labels
def label_encoding(arr,skipped,long=False):
    res=[]
    skipped=set(skipped)
    count=0
    for i in arr:
        if count in skipped:
            count+=1
            continue
        temp=[0]*5
        if long:
            temp[i-1]=1
        else:
            temp[i-1]=float(1)
        res.append(temp)
        count+=1
    return res


def get_word2vec_embeddings(Xtrain,Xtest,ytrain,ytest,skipten=False,fixedsize=Nor
    Xtrain_embeddings=[]
    Xtest_embeddings=[]

    skipped_train_sample=[] ### This list contain those training samples in which
    ##- word2vec model and therefore need to be skipped as they become noise in t

    skipped_test_sample=[]  ### This list contain those testing samples in which
    ##- word2vec model and therefore need to be skipped as they become noise in t


    if not skipten: ## This if condition controls whether for each sentence we co
        ## - or we add the vector of each word as a part of the training/testing
        ## -or of a sentence.

        for i in Xtrain:
            temp=[] ### A temporary array to accumulate the vectors of all the wo
            for j in i:
                if j in w2model_pretrained.key_to_index:
                    temp.append(w2model_pretrained.get_vector(j))

            Xtrain_embeddings.append(w2model_pretrained.get_mean_vector(temp)) ##
            ## - the temp array is added to the final train embeddings array.

        ## The same steps as above but for the test data
        for i in Xtest:
            temp=[]
            for j in i:
                if j in w2model_pretrained.key_to_index:
                    temp.append(w2model_pretrained.get_vector(j))
            Xtest_embeddings.append(w2model_pretrained.get_mean_vector(temp))

    else: ### The following code are executed if each word vector has to be a par

        ### The concat parameter controls if the word vectors in a train sample o
        ###- array when it is set as true or each word vector is present as a sep
        ### subarrays containing each word vector.
```

```
    if not concat: ## When concat is set to False

        for i in range(len(Xtrain)):

            steps=fixedsize ### fixedsize parameter controls the no of word w
            temp=[] ### A list containing sublists where each sublist contair

            for j in Xtrain[i]:
                if steps==0: ## when the required number of words is reached
                    break

                if j in w2model_pretrained.key_to_index:
                    temp.append(w2model_pretrained.get_vector(j))
                    steps-=1

            if len(temp)==0: ## if none of the words are present as a key in
                ## the train sample is skipped to avoid noise and is added to
                ## -sponding labels will be skipped as well

                skipped_train_sample.append(i)
                continue


            if len(temp)<fixedsize: ## if the length of the temp list is not
                ## fixed size the rest of the values are added as 0 vectors c
                ## rest of the sublists.

                temp=temp+[[0.0]*300]*(fixedsize-len(temp))
            Xtrain_embeddings.append(temp)

        ## The same steps as above are followed but for the testing data
        for i in range(len(Xtest)):
            steps=fixedsize
            temp=[]
            for j in Xtest[i]:
                if steps==0:
                    break
                if j in w2model_pretrained.key_to_index:
                    temp.append(w2model_pretrained.get_vector(j))
                    steps-=1
            if len(temp)==0:
                skipped_train_sample.append(i)
                continue
            if len(temp)<fixedsize:
                temp=temp+[[0.0]*300]*(fixedsize-len(temp))

            Xtest_embeddings.append(temp)

    else: ## The following code executes when concat is True and therefore th
        ## each other and therefore each training sample is one list having c

        for i in range(len(Xtrain)):
            steps=fixedsize
            temp=np.array([],dtype=np.float32) ## An empty numpy array to whi
            for j in Xtrain[i]:
                if steps==0:## when the required number of words is reached t
```

```
                    break
                if j in w2model_pretrained.key_to_index:
                    temp=np.concatenate([temp,w2model_pretrained.get_vector(j
                    ##rd2vec model, it is concatenated to the temp array
                    steps-=1

            Xtrain_embeddings.append(temp)

        ## The same steps as above are followed but for the testing data
        for i in range(len(Xtest)):
            steps=fixedsize
            temp=np.array([],dtype=np.float32)
            for j in Xtest[i]:
                if steps==0:
                    break
                if j in w2model_pretrained.key_to_index:
                    temp=np.concatenate([temp,w2model_pretrained.get_vector(j
                    steps-=1
            Xtest_embeddings.append(temp)




    ytrain_en=label_encoding(ytrain,skipped_train_sample,long)
    ytest_en=label_encoding(ytest,skipped_test_sample,long)

    return Xtrain_embeddings,Xtest_embeddings,ytrain_en,ytest
```

A function that returns the corresponding tf-idf word embeddings for given data

In [5]:
```
def tf_idf_embeddings(data):
    tfidf = TfidfVectorizer()
    vector_rep=tfidf.fit_transform(data)
    return vector_rep
```

1)

Fetching preprocessed data and splitting it into training and testing set data with a split of 80% and 20% respectively

In [6]:
```
processed_data=preprocess_data(new_df)
Xtrain,Xtest,ytrain,ytest=train_test_split(processed_data['input'],processed_data
```

2a)

Here the pretrained word2vec-google-news-300 model is used to checkout semantic similarities for a few examples

In [7]:
```
## A pretrained word2vec model used to generate the word embeddings required to t

w2model_pretrained=gensim.downloader.load('word2vec-google-news-300')
```

In [8]:
```
## A function that returns the cosine similarity between two vectors
def get_cosine_similarity(a,b):
    return np.dot(a,b)/((np.linalg.norm(a))*np.linalg.norm(b))
```

In [9]:
```
#### Examples
## Cosine similarity between (King-Man+Woman) and (Queen)
a=w2model_pretrained.get_vector('King')-w2model_pretrained.get_vector('Man')+w2mo
b=w2model_pretrained.get_vector('Queen')
get_cosine_similarity(a,b)
```

Out[9]:
```
0.44240144
```

In [10]:
```
## Cosine similarity between the words Cute and Beautiful

a=w2model_pretrained.get_vector('Cute')
b=w2model_pretrained.get_vector('Beautiful')
get_cosine_similarity(a,b)
```

Out[10]:
```
0.28467536
```

In [11]:
```
## Cosine similarity between the vectors from (Daughter-Woman+man) and (Son)

a=w2model_pretrained.get_vector('Daughter')-w2model_pretrained.get_vector('Woman'
b=w2model_pretrained.get_vector('Son')
get_cosine_similarity(a,b)
```

Out[11]:
```
0.5888708
```

2b)

Here a word2vec model is trained on the Amazon review dataset to checkout the semantic similarities learned by this model

In [12]:
```
## Own model
w2model = Word2Vec(min_count=5,window=11,vector_size=300,workers=1)
w2model.build_vocab(processed_data['input'])
w2model.train(Xtrain,total_examples=w2model.corpus_count,epochs=30)
```

Out[12]:
```
(56501161, 64819410)
```

In [13]:
```
## Cosine similarity between (King-Man+Woman) and (Queen)

a=w2model.wv.get_vector('king')-w2model.wv.get_vector('man')+w2model.wv.get_vecto
b=w2model.wv.get_vector('queen')
get_cosine_similarity(a,b)
```

Out[13]: 0.13117443

In [14]:
```
## Cosine similarity between the words Cute and Beautiful

a=w2model.wv.get_vector('cute')
b=w2model.wv.get_vector('beautiful')
get_cosine_similarity(a,b)
```

Out[14]: 0.2585714

In [15]:
```
## Cosine similarity between the vectors from (Daughter-Woman+man) and (Son)

a=w2model.wv.get_vector('daughter')-w2model.wv.get_vector('woman')+w2model.wv.get
b=w2model.wv.get_vector('son')
get_cosine_similarity(a,b)
```

Out[15]: 0.4414207

It is clear from the above comparisons in the semantic similarities that the pretrained word2vec-google-news-300 model performs better than the newly trained model which could be because of the smaller amount of data used to train the latter model. But the similarity values for words that could possibly have a higher frequency in the Amazon reviews training data yields results very close to that of the pretrained word2vec model

3)

Perceptron and SVM prediction results using the word2vec model trained on the given Amazon reviews dataset

In [16]:
```
## Get the word vector embeddings using the word2vec model we trained on the Amaz
## vector embeddings below because it can simply be used in its original integer
Xtrain_embeddings,Xtest_embeddings,ytrain_en,_=get_word2vec_embeddings(Xtrain,Xte
```

In [17]:
```
# Perceptron results
perceptron=Perceptron()
perceptron.fit(Xtrain_embeddings,ytrain)
perceptron.score(Xtest_embeddings,ytest)
```

Out[17]: 0.4023

In [18]:
```
# SVM results
svm=LinearSVC()
svm.fit(Xtrain_embeddings,ytrain)
svm.score(Xtest_embeddings,ytest)
```

```
G:\Pytorch_practice\lib\site-packages\sklearn\svm\_base.py:1208: ConvergenceWar
ning: Liblinear failed to converge, increase the number of iterations.
  ConvergenceWarning,
```

Out[18]: 0.50725

Perceptron and SVM prediction results using the tf-idf embeddings from the given Amazon reviews dataset

In [19]:
```
## Get the word vector embeddings using the tf-idf model based on the Amazon revi
## Here the train_test_split is done again, as the tf-idf model is fit on the ent
## on the train data, then there can be words in test data that are not present i
## when transforming the test data.

processed_data_as_sentence=[" ".join(i) for i in processed_data['input']]
processed_data_tfidf_vectors=tf_idf_embeddings(processed_data_as_sentence)
Xtrain_embeddings,Xtest_embeddings,ytrain,ytest=train_test_split(processed_data_t
```

In [20]:
```
#Perceptron results
perceptron=Perceptron()
perceptron.fit(Xtrain_embeddings,ytrain)
perceptron.score(Xtest_embeddings,ytest)
```

Out[20]: 0.42395

In [21]:
```
#SVM results
svm=LinearSVC()
svm.fit(Xtrain_embeddings,ytrain)
svm.score(Xtest_embeddings,ytest)
```

Out[21]: 0.49755

Its clear from the above results from the comparison between tfidf and word2vec, that word2vec model performs better than the tfidf model. Though the results of the perceptron are very close to each other and the tfidf exceeds in performance only by a very small amount, that can be attributed to reasons like the possible difference in test data. But the svm trained on word2vec embeddings clearly performs better than tf-idf based one as the word2vec model can incorporate semantic relationship between words into its embeddings unlike the tf-idf model.

4a)

A function to create a perceptron model and train it based on the input parameters

In [22]:
```
#Multilayer perceptron

## The multilayer perceptron class
class NeuralNet(nn.Module):
    def __init__(self,inputsize,outputsize,hidden_layer_1,hidden_layer_2):
        super(NeuralNet,self).__init__()
        self.linear1=nn.Linear(inputsize,hidden_layer_1)
        self.relu=nn.ReLU()
        self.linear2=nn.Linear(hidden_layer_1,hidden_layer_2)
        self.linear3=nn.Linear(hidden_layer_2,outputsize)

    def forward(self,x):
        l1=self.linear1(x)
        a1=self.relu(l1)
        l2=self.linear2(a1)
        a2=self.relu(l2)
        l3=self.linear3(a2)
        return l3

def mlp(epochs,batch_size,h1,h2,classes,lr,ipsize,Xtrain_embeddings,Xtest_embeddi
    ## Hyperparameters which are taken as an input to the function
    epochs=epochs
    batch_size=batch_size
    hidden_layer_1=h1
    hidden_layer_2=h2
    num_classes=classes
    learning_rate=lr
    inputsize=ipsize
    Xtrain_embeddings_tensor=torch.tensor(Xtrain_embeddings)
    Xtest_embeddings_tensor=torch.tensor(Xtest_embeddings)
    ytrain_tensor=torch.tensor(ytrain_en)

    model=NeuralNet(inputsize,num_classes,h1,h2)
    loss=nn.CrossEntropyLoss()
    optimizer= torch.optim.Adam(model.parameters(), lr=learning_rate)

    ## Training the model
    batch_size=100
    i=0
    l=None
    for epoch in range(epochs):
        i=0
        while i<80000:
            outputs=model(Xtrain_embeddings_tensor[i:min(i+100,len(Xtrain_embeddi
            l=loss(outputs,ytrain_tensor[i:min(i+100,len(Xtrain_embeddings_tensor
            i+=100
            l.backward()
            optimizer.step()
            optimizer.zero_grad()
        if epoch%50==0:

            print("epoch ",epoch," loss ",l.item())
    return model
```

In [23]:
```
def savemlp(model,name):
    torch.save(model,name)
```

In [24]:
```
def get_accuracy(Xtest,ytest,model,sublist=True):
    Xtest_embeddings=torch.tensor(Xtest)
    corr=0
    with torch.no_grad():
        for sample in range(len(Xtest_embeddings)):
            outputs=model(Xtest_embeddings[sample])
            outputs=outputs.tolist()
            if sublist:
                pred=outputs[0].index(max(outputs[0]))+1
            else:
                pred=outputs.index(max(outputs))+1
            if pred==ytest[sample]:
                corr+=1
    acc=corr/len(Xtest_embeddings)
    print("Accuracy:",acc)
    return acc
```

In [25]:
```
## Get the processed data, split it into a train_test split of 80%/20% and get th

processed_data=preprocess_data(new_df)
Xtrain,Xtest,ytrain,ytest=train_test_split(processed_data['input'],processed_data
Xtrain_embeddings,Xtest_embeddings,ytrain_en,_=get_word2vec_embeddings(Xtrain,Xte
```

In [26]:
```
mlp_model1=mlp(300,100,50,10,5,0.0001,300,Xtrain_embeddings,Xtest_embeddings,ytra
```

```
G:\Pytorch_practice\lib\site-packages\ipykernel_launcher.py:29: UserWarning: Cr
eating a tensor from a list of numpy.ndarrays is extremely slow. Please conside
r converting the list to a single numpy.ndarray with numpy.array() before conve
rting to a tensor. (Triggered internally at  C:\cb\pytorch_1000000000000\work\t
orch\csrc\utils\tensor_new.cpp:204.)

epoch  0  loss  1.4985710382461548
epoch  50  loss  1.1347074508666992
epoch  100  loss  1.110480546951294
epoch  150  loss  1.089257836341858
epoch  200  loss  1.068753957748413
epoch  250  loss  1.0542562007904053
```

In [27]:
```
get_accuracy(Xtest_embeddings,ytest,mlp_model1,sublist=False)
```

```
Accuracy: 0.48785
```

Out[27]:
```
0.48785
```

4b)

In [28]:
```
### MLP with 10 words in each sample whose word vectors are concatenated with eac
processed_data=preprocess_data(new_df,inword2vec=True,wordvec=w2model_pretrained)
Xtrain,Xtest,ytrain,ytest=train_test_split(processed_data['input'],processed_data
```

In [29]:
```
# Get the corresponding word embeddings where in each sentence only 10 words are

Xtrain_embeddings,Xtest_embeddings,ytrain_en,ytest=get_word2vec_embeddings(Xtrain
```

In [30]:
```
mlp_model2=mlp(epochs=300,batch_size=100,h1=50,h2=10,classes=5,lr=0.0001,ipsize=3
```

```
epoch  0  loss  1.4522488117218018
epoch  50  loss  1.167272925376892
epoch  100  loss  0.770887017250061
epoch  150  loss  0.5499396324157715
epoch  200  loss  0.3842123746871948
epoch  250  loss  0.25224852561950684
```

In [31]:
```
get_accuracy(Xtest_embeddings,ytest,mlp_model2,sublist=False)
```

```
Accuracy: 0.32545
```

Out[31]:
```
0.32545
```

Its clear from the above results that, the model in which word vectors in a sample are concatenated with each other performs poorly than the model where each wordvector is present in a separate list in a train sample list. This could be because, concatenating the word vectors could result in potential information loss, whereas not concatenating them could enable the perceptron model to learn possible relations between the words and therefore perform better

5a)

In [32]:
```
## Get the processed data and perform a train test split with a train to test rat
## embeddings are also fetched where the number of words in a sentence is limited
## each other. If 20 words are not present in a train sample, then the remaining
## size 300 as that is the embedding size for all the rest of the words.

processed_data=preprocess_data(new_df,inword2vec=True,wordvec=w2model_pretrained,
Xtrain,Xtest,ytrain,ytest=train_test_split(processed_data['input'],processed_data
Xtrain_embeddings,Xtest_embeddings,ytrain_en,ytest=get_word2vec_embeddings(Xtrain
```

In [33]:
```
## Convert the embeddings list to a torch tensor

Xtrain_tensor=torch.tensor(Xtrain_embeddings)
Xtest_tensor=torch.tensor(Xtest_embeddings)
ytrain_tensor=torch.tensor(ytrain_en)
```

The following function creates and trains a simple RNN over the dataset

In [34]:
```
### RNN

embedding_len=300
hidden_layer=20
classes=5
lr=0.0001

class RNN(nn.Module):
    def __init__(self,embedding_len,hidden_layer,classes,lr,nlayers):
        super(RNN,self).__init__()
        self.embedding_len=embedding_len
        self.hidden_layer=hidden_layer
        self.nlayers=nlayers
        self.rnn=nn.RNN(self.embedding_len,self.hidden_layer,self.nlayers,batch_f
        self.linear=nn.Linear(self.hidden_layer,classes)

    def forward(self,inp):
        hidden=self.init_hidden(inp.shape[0])
        outputs,hidden=self.rnn(inp,hidden)
        outputs=outputs[:,-1,:]
        outputs=self.linear(outputs)
        return outputs


    def init_hidden(self,batchsize):
        return torch.zeros(self.nlayers,batchsize,self.hidden_layer)

model=RNN(embedding_len,hidden_layer,classes,lr,1)
crit=nn.CrossEntropyLoss()
optimizer=torch.optim.Adam(model.parameters(),lr=lr)
epochs=100
batch_size=100
loss=None
for epoch in range(epochs):
    i=0
    while i<len(Xtrain_tensor):
        outputs=model(Xtrain_tensor[i:i+batch_size])
        loss=crit(outputs,ytrain_tensor[i:i+batch_size])
        i+=batch_size
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
    if epoch%10==0:
        print("epoch: ",epoch," loss: ",loss.item())


rnn_model=model
```

```
epoch:  0  loss:  1.6054154634475708
epoch:  10  loss:  1.2880865335464478
epoch:  20  loss:  1.2418627738952637
epoch:  30  loss:  1.1994792222976685
epoch:  40  loss:  1.1601277589797974
epoch:  50  loss:  1.1396654844284058
epoch:  60  loss:  1.1208961009979248
epoch:  70  loss:  1.1042811870574951
```

```
epoch:  80  loss:  1.0896217823028564
epoch:  90  loss:  1.0768266916275024
```

In [35]:
```
def get_seq_accuracy(Xtest,ytest,model,sublist=True):
    Xtest_embeddings=torch.tensor(Xtest)
    corr=0
    with torch.no_grad():
        for sample in range(len(Xtest_embeddings)):
            outputs=model(torch.unsqueeze(torch.tensor(Xtest_embeddings[sample]),
            outputs=outputs.tolist()
            if sublist:
                pred=outputs[0].index(max(outputs[0]))+1
            else:
                pred=outputs.index(max(outputs))+1
            if pred==ytest[sample]:
                corr+=1
    acc=corr/len(Xtest_embeddings)
    print("Accuracy:",acc)
    return acc
get_seq_accuracy(Xtest_embeddings,ytest,rnn_model,sublist=True)
```

```
G:\Pytorch_practice\lib\site-packages\ipykernel_launcher.py:6: UserWarning: To
copy construct from a tensor, it is recommended to use sourceTensor.clone().det
ach() or sourceTensor.clone().detach().requires_grad_(True), rather than torch.
tensor(sourceTensor).


Accuracy: 0.4677
```

Out[35]:
```
0.4677
```

5b)

The following function creates and trains a GRU over the dataset

In [36]:
```
### GRU
embedding_len=300
hidden_layer=20
classes=5
lr=0.0001

class GRU(nn.Module):
    def __init__(self,embedding_len,hidden_layer,classes,lr,nlayers):
        super(GRU,self).__init__()
        self.embedding_len=embedding_len
        self.hidden_layer=hidden_layer
        self.nlayers=nlayers
        self.gru=nn.GRU(self.embedding_len,self.hidden_layer,self.nlayers,batch_f
        self.linear=nn.Linear(self.hidden_layer,classes)

    def forward(self,inp):
        hidden=self.init_hidden(inp.shape[0])
        outputs,hidden=self.gru(inp,hidden)
        outputs=outputs[:,-1,:]
        outputs=self.linear(outputs)
        return outputs


    def init_hidden(self,batchsize):
        return torch.zeros(self.nlayers,batchsize,self.hidden_layer)

model=GRU(embedding_len,hidden_layer,classes,lr,1)
crit=nn.CrossEntropyLoss()
optimizer=torch.optim.Adam(model.parameters(),lr=lr)
epochs=100
batch_size=100
loss=None
for epoch in range(epochs):
    i=0
    while i<len(Xtrain_tensor):
        outputs=model(Xtrain_tensor[i:i+batch_size])
        loss=crit(outputs,ytrain_tensor[i:i+batch_size])
        i+=batch_size
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

    if epoch%10==0:
        print("epoch: ",epoch," loss: ",loss.item())


gru_model=model
```

```
epoch:  0  loss:  1.5757224559783936
epoch:  10  loss:  1.1311002969741821
epoch:  20  loss:  1.0554609298706055
epoch:  30  loss:  1.0270464420318604
epoch:  40  loss:  1.0128775835037231
epoch:  50  loss:  1.0048797130584717
epoch:  60  loss:  0.9994487166404724
```

```
epoch:  70  loss:  0.995318591594696
epoch:  80  loss:  0.9919390082359314
epoch:  90  loss:  0.9889469146728516
```

In [37]:
```
get_seq_accuracy(Xtest_embeddings,ytest,gru_model)
```

```
G:\Pytorch_practice\lib\site-packages\ipykernel_launcher.py:6: UserWarning: To
copy construct from a tensor, it is recommended to use sourceTensor.clone().det
ach() or sourceTensor.clone().detach().requires_grad_(True), rather than torch.
tensor(sourceTensor).


Accuracy: 0.5068
```

Out[37]:
```
0.5068
```

Its clear from the accuracy results obtained in the sections 5a and 5b that the GRU outperforms the RNN over the given dataset. This could especially be because GRU handles the disadvantages that RNN has such as vanishing gradient problem with its two gate based working. This could enable the GRU to learn long term dependencies between the words which the RNN cannot and thus helps the GRU yield better results than the RNN