HW2-CSCI544

September 29, 2022

1 CSCI544 Homework2 - Ravi Chandra Reddy Basireddy

1.1 Imports

- Pandas: To work with dataframes.
- NLTK: A Natural Language Toolkit used for processing textual data.
- RE: Regular Expressions used for handling word findings & substitutions.
- BS4: BeautifulSoup Library is a parser that can handle HTML Tags and Links.
- Contractions: A library to contract and de-contract Contractions.
- String: A library to handle strings.
- Warning: A library to handle console warnings.

[1]:
```
import pandas as pd
import numpy as np
import nltk
nltk.download('wordnet',quiet=True)
import re
from bs4 import BeautifulSoup
import contractions
import string
import warnings
warnings.filterwarnings(action='ignore')
import torch

#!pip install bs4
#!pip install contractions
#!pip install nltk
#!pip install string
#!pip install pandas
#!pip install warnings
#!pip install sklearn
```

1.2 Read Data

- Read Data from a TSV file where the data is seperated using tabs.
- we are only intrested in Star Rating and Review Body.
- Star Rating: Rating given by the customers in the range of 1 to 5.
- Review Body: Review given by the customers in the textual format.

```
[2]: amazon_reviews=pd.read_csv('data.tsv',␣
      ↪sep='\t',usecols=['star_rating','review_body'],low_memory=False)
```

1.3 Keep Reviews and Ratings

- Already completed at the reading data step.
- Dropping NaN Values which has no meaning to the rating.
- Dropping Duplicates which are repeated.
- Printing out the first five values to know what kind of data is in dataframe.

```
[3]: amazon_reviews=amazon_reviews.dropna()
     amazon_reviews=amazon_reviews.drop_duplicates()
     amazon_reviews.head(5)
```

```
[3]:   star_rating                                        review_body
     0           5  so beautiful even tho clearly not high end …
     1           5  Great product.. I got this set for my mother, …
     2           5  Exactly as pictured and my daughter's friend l…
     3           5  Love it. Fits great. Super comfortable and nea…
     4           5  Got this as a Mother's Day gift for my Mom and…
```

1.4 We select 20000 reviews randomly from each rating class.

- filtering out the data with respective labels.
- sampling 20k reviews from each class.
- Combining all the data from diffrent classes to create a vector of Dimension (100000,2).

```
[4]: star_one=amazon_reviews[amazon_reviews.star_rating=='1']
     star_one=star_one.sample(n=20000)
     star_two=amazon_reviews[amazon_reviews.star_rating=='2']
     star_two=star_two.sample(n=20000)
     star_three=amazon_reviews[amazon_reviews.star_rating=='3']
     star_three=star_three.sample(n=20000)
     star_four=amazon_reviews[amazon_reviews.star_rating=='4']
     star_four=star_four.sample(n=20000)
     star_five=amazon_reviews[amazon_reviews.star_rating=='5']
     star_five=star_five.sample(n=20000)
     sampled_reviews=pd.
      ↪concat([star_one,star_two,star_three,star_four,star_five],ignore_index=True)
```

2 Data Cleaning

- Cleaning the data inorder to make the models better, as better data will always result better Prediction.

3 Pre-processing

1. Removing URL.
2. Removing HTML Tags.
3. Removing All the characters except for A-Z&a-z.
4. Removing any html text left with BeautifulSoup Library.
5. Removing Contractions.
6. Removing Punctuation.
7. Removing extra Spaces.
8. Converting the text to lowecase.

[5]:
```
def remove_punctuation(review):
    return ''.join([words for words in review if words not in string.
 ↪punctuation ])
```

[6]:
```
def clean_review(review):
    review = re.sub(r"http\S+", "", review)
    review = re.sub('<.*?>+', '', review)
    review = re.sub('[^A-Za-z]+', ' ', review)
    review = BeautifulSoup(review, "html.parser").get_text()
    review = contractions.fix(review)
    review = remove_punctuation(review)
    review = re.sub("\S*\d\S*", "", review).strip()
    review = review.lower()
    return review
```

3.0.1 Calculating the Average Length of Reviews by Character

[7]:
```
def average_count(sampled_reviews):
    number_of_sentences=len(sampled_reviews)
    return sum(map(len,sampled_reviews))/number_of_sentences
```

[8]:
```
beforeCleaning=average_count(sampled_reviews['review_body'])
sampled_reviews['review_body']=sampled_reviews['review_body'].apply(lambda␣
 ↪review:clean_review(review))
afterCleaning=average_count(sampled_reviews['review_body'])
print("Average character length of the reviews Before and After␣
 ↪Cleaning",beforeCleaning,',',afterCleaning)
```

```
Average character length of the reviews Before and After Cleaning 197.64722 ,
189.79002
```

[9]:
```
list_of_sentences=[]
for sentence in sampled_reviews['review_body'].values:
    list_of_sentences.append(sentence.split())
```

3.1 Word2Vec

```
[10]: from gensim.models import Word2Vec
      from gensim.models import KeyedVectors
      import pickle
```

```
[11]: import gensim.downloader as api
      word2vec_model = api.load('word2vec-google-news-300')
```

```
[12]: word2vec_model.most_similar(positive = ['king', 'woman'], negative = ['man'])
```

```
[12]: [('queen', 0.7118193507194519),
       ('monarch', 0.6189674735069275),
       ('princess', 0.5902431011199951),
       ('crown_prince', 0.5499460697174072),
       ('prince', 0.5377322435379028),
       ('kings', 0.5236844420433044),
       ('Queen_Consort', 0.5235945582389832),
       ('queens', 0.5181134939193726),
       ('sultan', 0.5098593235015869),
       ('monarchy', 0.5087411403656006)]
```

```
[13]: word2vec_model.most_similar(positive = ['excellent'])
```

```
[13]: [('terrific', 0.7409728765487671),
       ('superb', 0.7062716484069824),
       ('exceptional', 0.681470513343811),
       ('fantastic', 0.6802847385406494),
       ('good', 0.644292950630188),
       ('great', 0.6124600768089294),
       ('Excellent', 0.6091997623443604),
       ('impeccable', 0.5980966687202454),
       ('exemplary', 0.5959650278091431),
       ('marvelous', 0.5829284191131592)]
```

```
[14]: word2vec_model.most_similar(positive = ['time'],negative=['clock'])
```

```
[14]: [('month', 0.34959733486175537),
       ('year', 0.3340516984462738),
       ('months', 0.3266761004924774),
       ('week', 0.3075323700904846),
       ('summer', 0.30571261048316956),
       ('years', 0.3044481873512268),
       ('moment', 0.3007998466491699),
       ('season', 0.3001079261302948),
       ('opportunity', 0.29813849925994873),
       ('weeks', 0.2971378564834595)]
```

```
[15]: word2vec_amazon_model=Word2Vec(list_of_sentences,min_count=10,vector_size=300,window=11)
```

```
[16]: word2vec_amazon_model.wv.most_similar(positive = ['king', 'woman'], negative =␣
      ↪['man'])
```

```
[16]: [('lifetime', 0.5438417196273804),
       ('west', 0.5429157614707947),
       ('world', 0.5113186836242676),
       ('co', 0.5072171688079834),
       ('responsible', 0.4986431300640106),
       ('teenagers', 0.4971093237400055),
       ('future', 0.4936424791812897),
       ('purposes', 0.4877610206604004),
       ('copy', 0.4861401617527008),
       ('pompeii', 0.4840492308139801)]
```

```
[17]: word2vec_amazon_model.wv.most_similar(positive = ['excellent'])
```

```
[17]: [('exceptional', 0.8293458223342896),
       ('outstanding', 0.7952795028686523),
       ('superb', 0.7474648356437683),
       ('inferior', 0.709937334060669),
       ('amazing', 0.676922619342804),
       ('lacking', 0.6761174201965332),
       ('decent', 0.6621876955032349),
       ('poor', 0.6587395668029785),
       ('acceptable', 0.6580824851989746),
       ('fantastic', 0.6437696814537048)]
```

```
[18]: word2vec_amazon_model.wv.most_similar(positive = ['time'],negative=['clock'])
```

```
[18]: [('day', 0.5510172247886658),
       ('once', 0.3180447816848755),
       ('spent', 0.30497655272483826),
       ('waiting', 0.2866770327091217),
       ('chance', 0.2708539664745331),
       ('morning', 0.2696778178215027),
       ('penny', 0.2651486396789551),
       ('twice', 0.262528657913208),
       ('week', 0.26093748211860657),
       ('compliments', 0.25867024064064026)]
```

Conclusion : Word2Vec Model performs better than amazon word2vec model because it is trained with billions of features,

```
[19]: average_wv=[]
      for sentence in list_of_sentences:
```

```
    sentence_vectors=np.zeros(300)
    number_of_words=0
    for words in sentence:
        try:
            vector=word2vec_model[words]
            sentence_vectors=np.add(sentence_vectors,vector)
            number_of_words+=1
        except:
            pass
    if number_of_words != 0:
        sentence_vectors/=number_of_words
    average_wv.append(sentence_vectors)
```

4 TF-IDF Feature Extraction

- Converting Reviews to Count Vectors using a concept known as TF-IDF.
- It is the relation between Term Frequency and Inverse Document Frequency.
- Term Frequency is the frequency of the word in a corpus.
- Inverse Document Frequecy is the Frequency of a Word in that particular Document.
- TF - IDF tells about the Frequency of a Word in that Particular Document With Respect to the Entire Corpus.
- N-Grams are combination of words in that particular document. Bi-gram Example (really-appreciate).

[20]:
```
from sklearn.feature_extraction.text import TfidfVectorizer
tf_idf_vect=TfidfVectorizer(ngram_range=(1,3))
final_tf_idf=tf_idf_vect.fit_transform(sampled_reviews['review_body'].values)
```

4.1 Train Test Split

- We split the data in the split of 80:20 which is 80% of the for Training and 20% of the Data for Testing.
- We use train test split in order to train the model and test performance of the model.

[21]:
```
from sklearn.model_selection import train_test_split
xtrain, xtest, ytrain, ytest =␣
 ↪train_test_split(average_wv,sampled_reviews['star_rating'] , test_size = 0.2)
```

[22]:
```
from sklearn.model_selection import train_test_split
xtrain_tfidf, xtest_tfidf, ytrain_tfidf, ytest_tfidf =␣
 ↪train_test_split(final_tf_idf, sampled_reviews['star_rating'], test_size = 0.
 ↪2)
```

[23]:
```
xtrain=np.array(xtrain)
xtest=np.array(xtest)
ytrain=np.array(ytrain).astype(np.float16)
ytest=np.array(ytest).astype(np.float16)
```

4.2 Classification Metrics

A Function that gives you information on Accuracy, Precision, Recall and F1-score.

[24]:
```
from sklearn.metrics import classification_report, confusion_matrix,␣
↪accuracy_score

def metrics(prediction, actual):
    print('\nAccuracy:', accuracy_score(actual, prediction))
    print('\nclassification_report\n')
    print(classification_report(actual, prediction))
```

5 Perceptron

- Perceptron is a two class classification Model.
- It uses a concept of Neuron, which has an activation function, which activates only when crossing a certain threshold.
- We used random_state to randomize the data.
- We used n_jobs to Run Parallel on All Cores.

5.1 Perceptron with Word2Vec

[25]:
```
from sklearn.linear_model import Perceptron
perceptronModel = Perceptron(random_state=0,n_jobs=-1)
perceptronModel.fit(xtrain, ytrain)
predictions=perceptronModel.predict(xtest)
metrics(predictions, ytest)
```

```

Accuracy: 0.4405

classification_report

              precision    recall  f1-score   support

         1.0       0.60      0.42      0.50      3929
         2.0       0.33      0.53      0.40      4040
         3.0       0.35      0.31      0.33      4010
         4.0       0.44      0.23      0.30      3975
         5.0       0.56      0.71      0.63      4046

    accuracy                           0.44     20000
   macro avg       0.46      0.44      0.43     20000
weighted avg       0.46      0.44      0.43     20000
```

5.2 Perceptron with TF-IDF

[26]:
```
from sklearn.linear_model import Perceptron
perceptronModel = Perceptron(random_state=0,n_jobs=-1)
perceptronModel.fit(xtrain_tfidf, ytrain_tfidf)
predictions=perceptronModel.predict(xtest_tfidf)
metrics(predictions, ytest_tfidf)
```

```
Accuracy: 0.49465

classification_report

              precision    recall  f1-score   support

           1       0.56      0.66      0.60      3990
           2       0.41      0.27      0.32      4020
           3       0.40      0.41      0.40      4042
           4       0.43      0.44      0.44      3958
           5       0.62      0.70      0.66      3990

    accuracy                           0.49     20000
   macro avg       0.48      0.50      0.49     20000
weighted avg       0.48      0.49      0.49     20000
```

Conclusion : Perceptron with TFIDF performs better as we are taking Average Word2Vec which losses lot of information

6 SVM

- SVM uses a concept of boundary, which helps it to detect and avoid outliers.
- SVM have diffent form of Kernel: Linear, Poly and more, which can be used fir different firms of data.
- We used C=0.1 which is a regularization parameter.

6.1 SVM with Word2Vec

[27]:
```
from sklearn.svm import LinearSVC
SVM = LinearSVC(C=0.1)
SVM.fit(xtrain, ytrain)
predictions=SVM.predict(xtest)
metrics(predictions, ytest)
```

```
Accuracy: 0.4894

classification_report
```

```
              precision    recall  f1-score   support

         1.0       0.50      0.73      0.59      3929
         2.0       0.41      0.27      0.32      4040
         3.0       0.41      0.37      0.39      4010
         4.0       0.45      0.32      0.37      3975
         5.0       0.59      0.76      0.67      4046

    accuracy                           0.49     20000
   macro avg       0.47      0.49      0.47     20000
weighted avg       0.47      0.49      0.47     20000
```

6.2 SVM with TF-IDF

[28]:
```python
from sklearn.svm import LinearSVC
SVM = LinearSVC(C=0.1)
SVM.fit(xtrain_tfidf, ytrain_tfidf)
predictions=SVM.predict(xtest_tfidf)
metrics(predictions, ytest_tfidf)
```

```
Accuracy: 0.54945

classification_report

              precision    recall  f1-score   support

           1       0.57      0.73      0.64      3990
           2       0.45      0.33      0.38      4020
           3       0.48      0.43      0.45      4042
           4       0.53      0.44      0.48      3958
           5       0.65      0.82      0.73      3990

    accuracy                           0.55     20000
   macro avg       0.54      0.55      0.54     20000
weighted avg       0.54      0.55      0.54     20000
```

Conclusion : SVM with TFIDF performs better as we are taking Average Word2Vec which losses lot of information.

6.3 Importing Pytorch

- TensorDataset for creating dataset from traing and test data
- Dataloader for loading tensorDataset
- nn module for creating custom Neural Network Model

[29]:
```
import os
import torch
from torch import nn
from torch.utils.data import DataLoader,TensorDataset
```

Finding out whether CPU or GPU in Use

[30]:
```
if torch.cuda.is_available():
    device = "cuda"
elif torch.has_mps:
    device = "mps"
else:
    device ="cpu"
print(f"Using {device} device")
```

```
Using mps device
```

[31]:
```
torch.device(device)
```

[31]:
```
device(type='mps')
```

[32]:
```
def predict(model, dataloader):
    count=0
    for data, target in dataloader:
        outputs = model(data)
        _, predicted = torch.max(outputs, 1)
        for x in range(len(predicted)):
            if(target[x]-1==predicted[x]):
                count+=1
    return count
```

[33]:
```
def predict_with_hidden(model, dataloader):
    count=0
    hidden_state = model.init_hidden(40)
    for data, target in dataloader:
        outputs,_ = model(data,h)
        _, predicted = torch.max(outputs, 1)
        for x in range(len(predicted)):
            if(target[x]-1==predicted[x]):
                count+=1
    return count
```

6.4 Creating TesnorDataset and DataLoader

- TensorDataset for creating dataset from traing and test data
- Dataloader for loading tensorDataset
- we are using batchsize of 40

[34]:
```
from sklearn.model_selection import train_test_split

def creating_loading_tensor(X,y,batch):
    xtrain, xtest, ytrain, ytest = train_test_split(X,y, test_size = 0.2)
    xtrain=np.array(xtrain)
    xtest=np.array(xtest)
    ytrain=np.array(ytrain).astype(np.float16)
    ytest=np.array(ytest).astype(np.float16)
    test_data = TensorDataset(torch.FloatTensor(xtest),  torch.
 ↪LongTensor(ytest))
    train_data = TensorDataset(torch.FloatTensor(xtrain),  torch.
 ↪LongTensor(ytrain))
    batch_size = batch
    train_loader = DataLoader(train_data, shuffle = True, batch_size =␣
 ↪batch_size)
    test_loader = DataLoader(test_data, shuffle = True, batch_size = batch_size)
    return train_loader,test_loader
```

6.5 Feed Forward Network

- Input Layer : 300
- Hidden Layer 1 : 50
- Hidden Layer 2 : 10
- Output Layer : 5
- We are using RELU as Activation Function
- We are using CrossEntropy as loss Function
- We are using SGD for Optimzation
- Reference : https://www.kaggle.com/mishra1993/pytorch-multi-layer-perceptron-mnist

6.5.1 FNN Using Average Word2Vve

[35]:
```
train_loader,test_loader=creating_loading_tensor(average_wv,sampled_reviews['star_rating'],bat
```

[36]:
```
import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        hidden_1 = 50
        hidden_2 = 10
        self.fc1 = nn.Linear(1*300, hidden_1)
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        self.fc3 = nn.Linear(hidden_2, 5)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        return x

model = Net()
print(model)
```

```
Net(
  (fc1): Linear(in_features=300, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
  (fc3): Linear(in_features=10, out_features=5, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

[37]:
```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

[38]:
```
n_epochs = 50
for epoch in range(n_epochs):
    train_loss = 0.0
    model.train()
    for data,target in train_loader:
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target-1)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()*data.size(0)

    train_loss = train_loss/len(train_loader.dataset)
    print('Epoch: {} \tTraining Loss: {:.6f}'.format(
        epoch+1,
        train_loss
        ))
```

```
Epoch: 1        Training Loss: 1.609100
Epoch: 2        Training Loss: 1.604974
Epoch: 3        Training Loss: 1.594845
Epoch: 4        Training Loss: 1.554497
Epoch: 5        Training Loss: 1.455629
```

```
Epoch: 6        Training Loss: 1.381124
Epoch: 7        Training Loss: 1.344667
Epoch: 8        Training Loss: 1.325516
Epoch: 9        Training Loss: 1.311536
Epoch: 10       Training Loss: 1.297184
Epoch: 11       Training Loss: 1.288385
Epoch: 12       Training Loss: 1.280477
Epoch: 13       Training Loss: 1.271495
Epoch: 14       Training Loss: 1.265411
Epoch: 15       Training Loss: 1.257448
Epoch: 16       Training Loss: 1.251293
Epoch: 17       Training Loss: 1.245571
Epoch: 18       Training Loss: 1.240104
Epoch: 19       Training Loss: 1.236047
Epoch: 20       Training Loss: 1.232546
Epoch: 21       Training Loss: 1.228585
Epoch: 22       Training Loss: 1.226145
Epoch: 23       Training Loss: 1.224111
Epoch: 24       Training Loss: 1.221565
Epoch: 25       Training Loss: 1.218959
Epoch: 26       Training Loss: 1.215534
Epoch: 27       Training Loss: 1.215866
Epoch: 28       Training Loss: 1.214642
Epoch: 29       Training Loss: 1.211396
Epoch: 30       Training Loss: 1.210918
Epoch: 31       Training Loss: 1.209289
Epoch: 32       Training Loss: 1.206202
Epoch: 33       Training Loss: 1.205123
Epoch: 34       Training Loss: 1.204540
Epoch: 35       Training Loss: 1.204996
Epoch: 36       Training Loss: 1.201968
Epoch: 37       Training Loss: 1.203151
Epoch: 38       Training Loss: 1.202018
Epoch: 39       Training Loss: 1.200992
Epoch: 40       Training Loss: 1.199943
Epoch: 41       Training Loss: 1.196884
Epoch: 42       Training Loss: 1.197687
Epoch: 43       Training Loss: 1.196309
Epoch: 44       Training Loss: 1.195645
Epoch: 45       Training Loss: 1.196932
Epoch: 46       Training Loss: 1.193906
Epoch: 47       Training Loss: 1.195524
Epoch: 48       Training Loss: 1.193642
Epoch: 49       Training Loss: 1.192101
Epoch: 50       Training Loss: 1.189703
```

[39]:
```
count = predict(model,test_loader)
print(count/len(ytest))
```

```
0.4751
```

The FNN Accuracy with Average Word2Vec is 47.5%

6.5.2 FNN with the first 10 Word2Vec vectors

[40]:
```
ten_wv=[]
for sentence in list_of_sentences:
    sentence_vectors=np.zeros(300)
    sentence=sentence[:10]
    count=0
    for words in sentence:
        try:
            vector=word2vec_model[words]
            sentence_vectors+=vector
            number_of_words+=1
        except:
            pass
    ten_wv.append(sentence_vectors)
```

[41]:
```
train_loader,test_loader=creating_loading_tensor(ten_wv,sampled_reviews['star_rating'],batch=4
```

[42]:
```
model = Net()
print(model)
```

```
Net(
  (fc1): Linear(in_features=300, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=10, bias=True)
  (fc3): Linear(in_features=10, out_features=5, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

[43]:
```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

[45]:
```
n_epochs = 50

for epoch in range(n_epochs):
    train_loss = 0.0
    model.train() # prep model for training
    for data,target in train_loader:
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target-1)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()*data.size(0)
    train_loss = train_loss/len(train_loader.dataset)
    print('Epoch: {} \tTraining Loss: {:.6f}'.format(
        epoch+1,
        train_loss
        ))
```

```
Epoch: 1        Training Loss: 1.454457
Epoch: 2        Training Loss: 1.411800
Epoch: 3        Training Loss: 1.391481
Epoch: 4        Training Loss: 1.378038
Epoch: 5        Training Loss: 1.368624
Epoch: 6        Training Loss: 1.359985
Epoch: 7        Training Loss: 1.355694
Epoch: 8        Training Loss: 1.347346
Epoch: 9        Training Loss: 1.344286
Epoch: 10       Training Loss: 1.339169
Epoch: 11       Training Loss: 1.332864
Epoch: 12       Training Loss: 1.330800
Epoch: 13       Training Loss: 1.325207
Epoch: 14       Training Loss: 1.322960
Epoch: 15       Training Loss: 1.319139
Epoch: 16       Training Loss: 1.317790
Epoch: 17       Training Loss: 1.311468
Epoch: 18       Training Loss: 1.311271
Epoch: 19       Training Loss: 1.309242
Epoch: 20       Training Loss: 1.306897
Epoch: 21       Training Loss: 1.305781
Epoch: 22       Training Loss: 1.304713
Epoch: 23       Training Loss: 1.300655
Epoch: 24       Training Loss: 1.299550
Epoch: 25       Training Loss: 1.296511
Epoch: 26       Training Loss: 1.294990
Epoch: 27       Training Loss: 1.293665
Epoch: 28       Training Loss: 1.290324
Epoch: 29       Training Loss: 1.290615
Epoch: 30       Training Loss: 1.289189
Epoch: 31       Training Loss: 1.288014
Epoch: 32       Training Loss: 1.285990
Epoch: 33       Training Loss: 1.285855
Epoch: 34       Training Loss: 1.285038
Epoch: 35       Training Loss: 1.280020
Epoch: 36       Training Loss: 1.281092
Epoch: 37       Training Loss: 1.280250
Epoch: 38       Training Loss: 1.275528
```

```
Epoch: 39       Training Loss: 1.277442
Epoch: 40       Training Loss: 1.275224
Epoch: 41       Training Loss: 1.274126
Epoch: 42       Training Loss: 1.274759
Epoch: 43       Training Loss: 1.271891
Epoch: 44       Training Loss: 1.270694
Epoch: 45       Training Loss: 1.270272
Epoch: 46       Training Loss: 1.271610
Epoch: 47       Training Loss: 1.270159
Epoch: 48       Training Loss: 1.266529
Epoch: 49       Training Loss: 1.267950
Epoch: 50       Training Loss: 1.265401
```

[46]:
```
count = predict(model,test_loader)
print(count/len(ytest))
```

```
0.41185
```

The FNN Accuracy with 10 Word2Vec is 41.2%

Conclusion : Feed Forward Network with AverageWord2Vec performs better as we are cosidering only 10 words in the 10 Word2Vec Model

6.6 Reccurent Neural Network

- Input Layer : 300
- Hidden Layer 1 : 20
- Output Layer : 5
- We are using Softmax as Activation Function
- We are using CrossEntropy as loss Function
- We are using SGD for Optimzation
- Reference : https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

[47]:
```
class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.i2h = nn.Linear(input_size + hidden_size, hidden_size)
        self.i2o = nn.Linear(input_size + hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, input, hidden):
        combined = torch.cat((input, hidden), 1)
        hidden = self.i2h(combined)
        output = self.i2o(combined)
        output = self.softmax(output)
        return output, hidden

    def initHidden(self,batch):
        return torch.zeros(batch,self.hidden_size,requires_grad=False)

n_hidden = 20
model = RNN(300, n_hidden, 5)
```

[48]:
```
twenty_wv=[]
for sentence in list_of_sentences:
    sentence_vectors=np.zeros(300)
    number_of_words=0
    sentence=sentence[:20]
    while len(sentence) < 20:
        sentence.append(0)

    for words in sentence:
        try:
            vector=word2vec_model[words]
            sentence_vectors=np.add(sentence_vectors,vector)
            number_of_words+=1
        except:
            pass
    twenty_wv.append(sentence_vectors)
```

[49]:
```
train_loader,test_loader=creating_loading_tensor(twenty_wv,sampled_reviews['star_rating'],batc
```

[50]:
```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

[51]:
```
n_epochs = 50

for epoch in range(n_epochs):
    train_loss = 0.0
    hidden_state = model.initHidden(40)
    model.train() # prep model for training
    for data,target in train_loader:
        optimizer.zero_grad()
        output,hidden = model(data,hidden_state)
        loss = criterion(output, target-1)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()*data.size(0)

    train_loss = train_loss/len(train_loader.dataset)
    print('Epoch: {} \tTraining Loss: {:.6f}'.format(
        epoch+1,
        train_loss,
        ))
```

```
Epoch: 1 	Training Loss: 1.364559
Epoch: 2 	Training Loss: 1.304589
Epoch: 3 	Training Loss: 1.293882
Epoch: 4 	Training Loss: 1.288379
Epoch: 5 	Training Loss: 1.284857
Epoch: 6 	Training Loss: 1.282694
Epoch: 7 	Training Loss: 1.281443
Epoch: 8 	Training Loss: 1.280229
Epoch: 9 	Training Loss: 1.278944
Epoch: 10 	Training Loss: 1.279484
Epoch: 11 	Training Loss: 1.278861
Epoch: 12 	Training Loss: 1.279209
Epoch: 13 	Training Loss: 1.277360
Epoch: 14 	Training Loss: 1.277551
Epoch: 15 	Training Loss: 1.278160
Epoch: 16 	Training Loss: 1.277251
Epoch: 17 	Training Loss: 1.277375
Epoch: 18 	Training Loss: 1.276739
Epoch: 19 	Training Loss: 1.276298
Epoch: 20 	Training Loss: 1.276457
Epoch: 21 	Training Loss: 1.276681
Epoch: 22 	Training Loss: 1.276729
Epoch: 23 	Training Loss: 1.276244
Epoch: 24 	Training Loss: 1.275920
Epoch: 25 	Training Loss: 1.276702
Epoch: 26 	Training Loss: 1.275511
Epoch: 27 	Training Loss: 1.276304
Epoch: 28 	Training Loss: 1.275553
Epoch: 29 	Training Loss: 1.276954
Epoch: 30 	Training Loss: 1.275532
Epoch: 31 	Training Loss: 1.275807
Epoch: 32 	Training Loss: 1.276120
Epoch: 33 	Training Loss: 1.276136
Epoch: 34 	Training Loss: 1.276343
Epoch: 35 	Training Loss: 1.275836
Epoch: 36 	Training Loss: 1.276034
Epoch: 37 	Training Loss: 1.276991
Epoch: 38 	Training Loss: 1.276021
Epoch: 39 	Training Loss: 1.276114
Epoch: 40 	Training Loss: 1.276023
Epoch: 41 	Training Loss: 1.275366
Epoch: 42 	Training Loss: 1.276135
```

```
Epoch: 43       Training Loss: 1.276083
Epoch: 44       Training Loss: 1.275954
Epoch: 45       Training Loss: 1.276924
Epoch: 46       Training Loss: 1.276170
Epoch: 47       Training Loss: 1.276346
Epoch: 48       Training Loss: 1.275218
Epoch: 49       Training Loss: 1.276551
Epoch: 50       Training Loss: 1.276341
```

[52]:
```
def predict(model, dataloader):
    count=0
    hidden_state = model.initHidden(40)
    for data, target in dataloader:
        outputs,_ = model(data,hidden_state)
        _, predicted = torch.max(outputs, 1)
        for x in range(len(predicted)):
            if(target[x]-1==predicted[x]):
                count+=1
    return count
```

[53]:
```
count = predict(model,test_loader)
print(count/len(ytest))
```

```
0.44785
```

The RNN Accuracy with 20 Word2Vec is 44.8%

Conculusion RNN Perform Better to FNN when considering 20 Words, But its less than Average Word2Vec FNN as it omits most of the words and sometimes it suffers from Vanishing Gradients

6.7 Gated Recurrent Unit Cell

- Input Layer : 300
- Hidden Layer 1 : 20
- Output Layer : 5
- We are using Softmax as Activation Function
- We are using CrossEntropy as loss Function
- We are using SGD for Optimzation
- GRU has 2 gates Update, Reset which controls the flow in the network
- Reference : https://pytorch.org/docs/stable/generated/torch.nn.GRU.html

[54]:
```
class GRUModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, n_layers):
        super(GRUModel, self).__init__()
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers
        self.gru = nn.GRU(input_dim, hidden_dim, n_layers, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)
        self.relu = nn.ReLU()

    def forward(self, x, h):
        out, h = self.gru(x, h)
        out = self.fc(self.relu(out[:,-1,:]))
        return out, h

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

n_hidden = 20
model = GRUModel(300,n_hidden,5,1)
```

[55]:
```
twenty_wv=[]
for sentence in list_of_sentences:
    sentence_vectors=[]
    number_of_words=0
    sentence=sentence[:20]

    count=0
    for words in sentence:
        try:
            vector=word2vec_model[words]
            sentence_vectors.append(vector)
            number_of_words+=1
            count+=1
        except:
            pass
    while count<20:
        sentence_vectors.append(np.zeros(300))
        count+=1
    twenty_wv.append(sentence_vectors)
```

[56]:
```
train_loader,test_loader=creating_loading_tensor(twenty_wv,sampled_reviews['star_rating'],batc
```

[57]:
```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

[58]:
```
n_epochs = 50

for epoch in range(n_epochs):
    train_loss = 0.0
    init_hidden=model.init_hidden(40)
    model.train() # prep model for training
    for data,target in train_loader:
        h = init_hidden.data
        optimizer.zero_grad()
        output,h = model(data,h)
        loss = criterion(output, target-1)
        loss.backward()
        optimizer.step()
        train_loss += loss.item()*data.size(0)

    train_loss = train_loss/len(train_loader.dataset)

    print('Epoch: {} \tTraining Loss: {:.6f}'.format(
        epoch+1,
        train_loss
        ))
```

```
Epoch: 1        Training Loss: 1.610465
Epoch: 2        Training Loss: 1.608487
Epoch: 3        Training Loss: 1.607531
Epoch: 4        Training Loss: 1.606474
Epoch: 5        Training Loss: 1.605117
Epoch: 6        Training Loss: 1.603292
Epoch: 7        Training Loss: 1.600669
Epoch: 8        Training Loss: 1.595951
Epoch: 9        Training Loss: 1.568398
Epoch: 10       Training Loss: 1.419354
Epoch: 11       Training Loss: 1.373118
Epoch: 12       Training Loss: 1.349225
Epoch: 13       Training Loss: 1.335193
Epoch: 14       Training Loss: 1.324397
Epoch: 15       Training Loss: 1.316847
Epoch: 16       Training Loss: 1.308476
Epoch: 17       Training Loss: 1.301670
Epoch: 18       Training Loss: 1.295073
Epoch: 19       Training Loss: 1.286345
Epoch: 20       Training Loss: 1.277574
Epoch: 21       Training Loss: 1.267182
Epoch: 22       Training Loss: 1.255956
Epoch: 23       Training Loss: 1.249106
Epoch: 24       Training Loss: 1.242698
Epoch: 25       Training Loss: 1.236998
Epoch: 26       Training Loss: 1.231249
Epoch: 27       Training Loss: 1.228434
Epoch: 28       Training Loss: 1.223186
Epoch: 29       Training Loss: 1.219947
Epoch: 30       Training Loss: 1.215416
Epoch: 31       Training Loss: 1.211686
```

```
Epoch: 32       Training Loss: 1.208664
Epoch: 33       Training Loss: 1.206271
Epoch: 34       Training Loss: 1.203801
Epoch: 35       Training Loss: 1.199405
Epoch: 36       Training Loss: 1.197586
Epoch: 37       Training Loss: 1.195033
Epoch: 38       Training Loss: 1.192427
Epoch: 39       Training Loss: 1.190276
Epoch: 40       Training Loss: 1.187385
Epoch: 41       Training Loss: 1.185006
Epoch: 42       Training Loss: 1.184140
Epoch: 43       Training Loss: 1.181240
Epoch: 44       Training Loss: 1.179079
Epoch: 45       Training Loss: 1.177106
Epoch: 46       Training Loss: 1.175705
Epoch: 47       Training Loss: 1.173531
Epoch: 48       Training Loss: 1.171837
Epoch: 49       Training Loss: 1.170525
Epoch: 50       Training Loss: 1.168343
```

[60]:
```
def predict(model, dataloader):
    count=0
    hidden_state = model.init_hidden(40)
    for data, target in dataloader:
        outputs,_ = model(data,h)
        _, predicted = torch.max(outputs, 1)
        for x in range(len(predicted)):
            if(target[x]-1==predicted[x]):
                count+=1
    return count
```

[62]:
```
count = predict(model,test_loader)
print(count/len(ytest))
```

```
0.31525
```

The GRU Accuracy with 20 Word2Vec is 40.5%

Conclusion : The Performance of GRU is comparible to RNN but as we are considering only 20 words it doesnt help much as most of the information is lost

[ ]: