HW3-Bhavik-Upadhyay

October 19, 2023

[1]:
```python
from typing import Optional, Callable
from typing_extensions import TypeAlias

import pandas as pd
import numpy as np
import nltk
import re
from bs4 import BeautifulSoup
import contractions

import os
from urllib import request
import gzip

from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_score, recall_score, f1_score,␣
↪accuracy_score

from sklearn.linear_model import Perceptron, LogisticRegression
from sklearn.svm import LinearSVC
from sklearn.naive_bayes import MultinomialNB
```

[2]:
```python
import torch
import random

seed = 42
torch.manual_seed(seed)
random.seed(seed)
np.random.seed(0)
```

1 Task 1: Dataset Generation

[3]:
```python
url = 'https://web.archive.org/web/20201127142707if_/https://s3.amazonaws.com/
↪amazon-reviews-pds/tsv/amazon_reviews_us_Office_Products_v1_00.tsv.gz'

extracted_file = 'data.tsv'
compressed_file = extracted_file + '.gz'
```

```python
# Retrieve the dataset from given url and store it in location specified by␣
↪compressed_file
if not os.path.exists(extracted_file):
    request.urlretrieve(url, compressed_file)

    # extract the dataset from the gzipped file
    with gzip.open(compressed_file, 'rb') as f_in, open(extracted_file, 'wb')␣
↪as f_out:
        for line in f_in:
            f_out.write(line)

    os.remove(compressed_file)


# read the extracted data into pandas dataframe
original_df = pd.read_csv(extracted_file, sep='\t', on_bad_lines='skip',␣
↪low_memory=False)
print(original_df.head())
```

```
  marketplace  customer_id       review_id  product_id  product_parent  \
0          US     43081963  R18RVCKGH1SSI9  B001BM2MAC       307809868
1          US     10951564  R3L4L6LW1PUOFY  B00DZYEXPQ        75004341
2          US     21143145  R2J8AWXWTDX2TF  B00RTMUHDW       529689027
3          US     52782374  R1PR37BR7G3M6A  B00D7H8XB6       868449945
4          US     24045652  R3BDDDZMZBZDPU  B001XCWP34        33521401

                                       product_title product_category  \
0     Scotch Cushion Wrap 7961, 12 Inches x 100 Feet  Office Products
1           Dust-Off Compressed Gas Duster, Pack of 4  Office Products
2  Amram Tagger Standard Tag Attaching Tagging Gu…  Office Products
3  AmazonBasics 12-Sheet High-Security Micro-Cut …  Office Products
4  Derwent Colored Pencils, Inktense Ink Pencils,…  Office Products

  star_rating  helpful_votes  total_votes vine verified_purchase  \
0           5            0.0          0.0    N                 Y
1           5            0.0          1.0    N                 Y
2           5            0.0          0.0    N                 Y
3           1            2.0          3.0    N                 Y
4           4            0.0          0.0    N                 Y

                                     review_headline  \
0                                         Five Stars
1  Phffffffft, Phfffffft. Lots of air, and it's C…
2                        but I am sure I will like it.
3  and the shredder was dirty and the bin was par…
```

```
4                                         Four Stars

                                        review_body review_date
0                                    Great product.  2015-08-31
1  What's to say about this commodity item except…  2015-08-31
2    Haven't used yet, but I am sure I will like it.  2015-08-31
3  Although this was labeled as &#34;new&#34; the…  2015-08-31
4                       Gorgeous colors and easy to use  2015-08-31
```

[4]:
```python
# creating the dataframe by taking only review_body and star_rating columns
df = pd.DataFrame(original_df[['review_body', 'star_rating']])
print(df.head())

# we notice there are some erroneous values for the star_rating column
print(df['star_rating'].unique())

# converting the star_rating to numeric values and dropping erroneous columns
df['star_rating'] = pd.to_numeric(df['star_rating'], errors='coerce')
df.dropna(inplace=True)

print(df['star_rating'].unique())
```

```
                                        review_body star_rating
0                                    Great product.           5
1  What's to say about this commodity item except…           5
2    Haven't used yet, but I am sure I will like it.           5
3  Although this was labeled as &#34;new&#34; the…           1
4                       Gorgeous colors and easy to use           4
['5' '1' '4' '2' '3' '2015-06-05' '2015-02-11' nan '2014-02-14']
[5. 1. 4. 2. 3.]
```

[5]:
```python
# creating the target column: target = 1 if star_rating is 1, 2 or 3. target =
↪2 if star_rating is 4 or 5
df['star_rating'] = df['star_rating'].astype(int)
df['target'] = df['star_rating'].apply(lambda x: 0 if x <= 3 else 1)

sample_size = 50000

# creating a sample dataframe where target = 1 of size 50000 rows
class_1 = df.loc[df['target'] == 0].sample(n=sample_size, random_state=42)

# creating a sample dataframe where target = 2 of size 50000 rows
class_2 = df.loc[df['target'] == 1].sample(n=sample_size, random_state=42)

# merging the two sample dataframes
df_new = pd.concat([class_1, class_2], ignore_index=True)
```

```python
def clean(review):
    """
    convert to lower-case
    remove html and urls
    remove non-alphabetical character
    remove extra spaces
    """

    # converting to lowercase
    review = review.lower()

    # removing htmls
    soup = BeautifulSoup(review, "html.parser")

    for a_tag in soup.find_all("a"):
        a_tag.decompose()

    review = soup.get_text()

    # removing urls
    review = re.sub(r'^https?:\/\/.*[\r\n]*', '', review)

    # removing non-alphabetical characters
    review = re.sub(r'[^a-zA-Z\s]', '', review)

    # removing extra spaces
    review = re.sub(r'\s+', ' ', review).strip()

    return review


df_new['review_body'] = df_new['review_body'].apply(clean)
```

```
C:\Users\bhavi\AppData\Local\Temp\ipykernel_17012\1465821108.py:13:
MarkupResemblesLocatorWarning: The input looks more like a filename than markup.
You may want to open this file and pass the filehandle into Beautiful Soup.
  soup = BeautifulSoup(review, "html.parser")
C:\Users\bhavi\AppData\Local\Temp\ipykernel_17012\1465821108.py:13:
MarkupResemblesLocatorWarning: The input looks more like a URL than markup. You
may want to use an HTTP client like requests to get the document behind the URL,
and feed that document to Beautiful Soup.
  soup = BeautifulSoup(review, "html.parser")
```

2 Task 2: Creating the Word2Vec Models

When comparing a pretrained model with a custom word2vec model, we find a substantial difference in vocabulary size: 3,000,000 unique words in the pretrained model versus 14,994 in the custom model. This suggests the pretrained model may handle out-of-vocabulary words better during testing.

In semantic similarity tests: 1. Outstanding and excellent show lower similarity in the pretrained model. 2. The arithmetic "King - Man + Woman" correctly yields "Queen" in the pretrained model, but not in the custom model. 3. The arithmetic "Doctor - Man + Woman" doesn't produce nurse-related results in the custom model.

Overall, the pretrained model performs better in some semantic tasks but struggles with similarity in specific cases.

2.1 Word2Vec from pretrained

[7]:
```
import gensim.downloader
from gensim.models import KeyedVectors

# downloading the pre-trained model if it is not available
if not os.path.exists('pretrained_w2v.model'):
    pretrained_w2v = gensim.downloader.load('word2vec-google-news-300')
    pretrained_w2v.save('pretrained_w2v.model')
# if available, we load the data from local storage
else:
    pretrained_w2v = KeyedVectors.load('pretrained_w2v.model')
```

[8]:
```
print(len(pretrained_w2v), len(pretrained_w2v[0]))
```

```
3000000 300
```

[9]:
```
# printing the similarity score for outstanding and excellent
print(pretrained_w2v.similarity('outstanding', 'excellent'))
```

```
0.55674857
```

[10]:
```
# printing the most similar words matching the arithmetic king - man + woman
print(pretrained_w2v.most_similar(positive=['king', 'woman'], negative=['man'],␣
↪topn=5))
```

```
[('queen', 0.7118193507194519), ('monarch', 0.6189674139022827), ('princess',
0.5902431011199951), ('crown_prince', 0.5499460697174072), ('prince',
0.5377321839332581)]
```

[11]:
```
# printing the most similar words matching the arithmetic doctor - man + queen
print(pretrained_w2v.most_similar(positive=['doctor', 'woman'],␣
↪negative=['man'], topn=5))
```

```
[('gynecologist', 0.7093892097473145), ('nurse', 0.6477287411689758),
('doctors', 0.6471460461616516), ('physician', 0.6438996195793152),
('pediatrician', 0.6249487996101379)]
```

[12]:
```
# Creating the vocabulary from the pretrained w2v
vocab = list(pretrained_w2v.index_to_key)

# Initialize an embedding matrix with zeros
embedding_dim = pretrained_w2v.vector_size
embedding_matrix = np.zeros((len(vocab), embedding_dim), dtype=np.float32)

# Populate the embedding matrix with Word2Vec vectors
for i, word in enumerate(vocab):
    if word in pretrained_w2v:
        # print(word, i, custom_w2v.wv[word])
        embedding_matrix[i] = pretrained_w2v[word]

print(len(embedding_matrix), len(embedding_matrix[0]))
```

```
3000000 300
```

2.2 From dataset (custom)

[13]:
```
from gensim.models import Word2Vec

# getting the tokens
tokenized = [nltk.word_tokenize(review) for review in df_new['review_body']]

# if the model is not already available, then create from scratch
if not os.path.exists('custom_w2v.model'):
    custom_w2v = Word2Vec(tokenized, vector_size=300, window=13, min_count=9,␣
↪sg=1, workers=1)
    custom_w2v.save('custom_w2v.model')
# if available, load the model from local storage
else:
    custom_w2v = Word2Vec.load('custom_w2v.model')
```

[14]:
```
print(len(custom_w2v.wv), len(custom_w2v.wv[0]))
```

```
14994 300
```

[15]:
```
# printing the similarity score for outstanding and excellent
print(custom_w2v.wv.similarity('outstanding', 'excellent'))
```

```
0.6205609
```

[16]:
```
# printing the most similar words matching the arithmetic king - man + woman
print(custom_w2v.wv.most_similar(positive=['king', 'woman'], negative=['man'],␣
↪topn=5))
```

```
[('Idea', 0.5137808918952942), ('Archival', 0.5026917457580566), ('fine-point',
0.49942412972450256), ('inherited', 0.4969730079174042), ('Flair',
0.4938381314277649)]
```

```
[17]: # printing the most similar words matching the arithmetic doctor - man + queen
print(custom_w2v.wv.most_similar(positive=['doctor', 'woman'],␣
 ↪negative=['man'], topn=5))
```

```
[('assistant', 0.4972769320011139), ('visiting', 0.4651092290878296),
('deadlines', 0.45854732394218445), ('appointment', 0.45726412534713745),
('prayer', 0.4571249485015869)]
```

3 Task 3: Simple Models

- **Perceptron** achieved **80.77% accuracy** on word embeddings, which is slightly better than **79.345%** using TF-IDF.
- **SVM** attained **82.665% accuracy** on word embeddings, slightly lower than **84.865%** using TF-IDF.
- In general, word embeddings exhibit competitive performance, however, in the SVM task, we notice slightly less performance.

Note: The two figures below show the accuracy obtained for perceptron and SVM when training on TF-IDF. These were obtained by modifying the metric of calculation in Homework 1 without any changes as to how the models were trained or features were extracted.

```
In [12]:
# Creating and training the perceptron for bag-of-words
bow_clf = Perceptron(penalty='elasticnet', l1_ratio=0.1, eta0=1e-3, alpha=1e-6, tol=1e-4, random_state=42)
bow_clf.fit(bow_X_train, Y_train)

# making predictions on the bag-of-words test set
bow_Y_pred = bow_clf.predict(bow_X_test)

print('BOW: ', accuracy_score(Y_test, bow_Y_pred))

# Creating and training the perceptron for TF-IDF
tf_idf_clf = Perceptron(penalty='elasticnet', l1_ratio=0.3, eta0=1e-5, max_iter=1000, alpha=1e-6, tol=1e-4, random_state=42)
tf_idf_clf.fit(tf_idf_X_train, Y_train)

# making predictions on the TD-IDF test set
tf_idf_Y_pred = tf_idf_clf.predict(tf_idf_X_test)

print('TF-IDF: ', accuracy_score(Y_test, tf_idf_Y_pred))

BOW:  0.79925
TF-IDF:  0.79345
```

Accuracy of perceptron on TF-IDF

```
In [14]:
# Creating and training SVM model on Bag-of-words training set
bow_svm = LinearSVC(max_iter=1000, penalty='l1', dual=False, C=0.1, random_state=42)
bow_svm.fit(bow_X_train, Y_train)

# making predictions on bag-of-words test set
bow_Y_pred = bow_svm.predict(bow_X_test)

print('BOW:', accuracy_score(Y_test, bow_Y_pred))


# Creating and training SVM model on TFIDF training set
tf_idf_svm = LinearSVC(max_iter=10, dual=False, C=0.1, random_state=42)
tf_idf_svm.fit(tf_idf_X_train, Y_train)

# making predictions on TF-IDF test set
tf_idf_Y_pred = tf_idf_svm.predict(tf_idf_X_test)

print('TF-IDF: ', accuracy_score(Y_test,  tf_idf_Y_pred))

BOW: 0.8429
TF-IDF:  0.84865
```

Accuracy of SVM on TF-IDF

3.0.1 Creating mean sentence embeddings

[18]:
```
# a function to create mean embeddings for a sentence given a w2v model
def create_avg_embeddings(sentence, w2v):
    tokens = nltk.word_tokenize(sentence)
    vectors = [w2v[word] for word in tokens if word in w2v]

    if vectors:
        embedding = np.mean(vectors, axis=0, dtype=np.float32)
    else:
        embedding = np.zeros(w2v.vector_size, dtype=np.float32)

    return embedding
```

[19]:
```
avg_embeddings = df_new['review_body'].apply(lambda x: create_avg_embeddings(x,
 ↪pretrained_w2v))
avg_embeddings = np.array(avg_embeddings.tolist())

targets = df_new['target']
```

[20]:
```
X_train_avg, X_test_avg, Y_train_avg, Y_test_avg = train_test_split(
    avg_embeddings,
    df_new['target'],
    shuffle=True,
    test_size=0.2,
    random_state=42
)

Y_train_avg = np.array(Y_train_avg.tolist())
Y_test_avg = np.array(Y_test_avg.tolist())
```

3.0.2 Perceptron training

[21]:
```
per_clf = Perceptron(penalty='elasticnet', l1_ratio=0.8, alpha=1e-5, tol=1e-4,
 ↪random_state=42)
per_clf.fit(list(X_train_avg), Y_train_avg)

per_Y_preds = per_clf.predict(list(X_test_avg))
per_acc = accuracy_score(per_Y_preds, Y_test_avg)
print(per_acc)
```

```
0.8077
```

3.0.3 SVM training

[22]:
```
svc_clf = LinearSVC(dual=True, loss='hinge', C=0.8, max_iter=10000,␣
↪random_state=42)
svc_clf.fit(list(X_train_avg), Y_train_avg)

svc_Y_preds = svc_clf.predict(list(X_test_avg))
svc_acc = accuracy_score(svc_Y_preds, Y_test_avg)
print(svc_acc)
```

```
0.82665
```

4 Task 4: Feedforward Neural Network

[23]:
```
# general purpose function for training a model with given training data
def train(model, train_data, test_data, criterion, optimizer, n_epochs=10,␣
↪verbose=True, device='cpu'):
    for epoch in range(n_epochs):
        train_loss = 0.
        test_loss = 0.

        model.train()
        for data, target in train_data:
            # shifting data and target to the appropriate device
            data = data.to(device)
            target = target.to(device)

            # setting the gradients to zero
            optimizer.zero_grad()

            # getting the output and calculating the loss
            out = model(data)
            loss = criterion(out, target)

            # performing the backward step and using the optimizer
            loss.backward()
            optimizer.step()

            train_loss += loss.item()

        model.eval()
        with torch.no_grad():
            for data, target in test_data:
                # shifting data and target to appropriate device
                data = data.to(device)
                target = target.to(device)
```

```
                # getting the output and then getting the loss and updating the␣
↪total test loss
                out = model(data)

                loss = criterion(out, target)
                test_loss += loss.item()

        if verbose:
            print(f'Epoch: {epoch+1} / {n_epochs}\tTraining Loss:␣
↪{train_loss}\tTest Loss: {test_loss}')
```

[24]:
```
# a function to calculate accuracy on test data for a given model
def accuracy(model, test_data, device='cpu'):
    correct, total = 0, 0

    with torch.no_grad():
        for data, target in test_data:
            # shifting data and target to appropriate device
            data = data.to(device)
            target = target.to(device)

            # getting the output and predictions
            out = model(data)
            _, preds = torch.max(out.data, 1)

            # updating the total and correct variables
            total += target.size(0)
            correct += (preds == target).sum().item()

    return (100*correct) / total
```

[25]:
```
import torch
from torch.utils.data import TensorDataset, DataLoader
import torch.nn as nn
import torch.optim as optim

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

# converting numpy arrays created for perceptron and svm to torch tensors
X_train_avg = torch.tensor(X_train_avg, dtype=torch.float32)
X_test_avg = torch.tensor(X_test_avg, dtype=torch.float32)

Y_train_avg = torch.tensor(Y_train_avg, dtype=torch.int64)
Y_test_avg = torch.tensor(Y_test_avg, dtype=torch.int64)

# generating dataset from created tensors
train_dataset1 = TensorDataset(X_train_avg, Y_train_avg)
```

```
test_dataset1 = TensorDataset(X_test_avg, Y_test_avg)

batch_size = 256
# creating train and test data loaders
train_loader1 = DataLoader(train_dataset1, batch_size=batch_size, shuffle=True)
test_loader1 = DataLoader(test_dataset1, batch_size=batch_size)

print(device)
```

```
cuda:0
```

4.1 Task 4 (a)

- Here, we train a feedforward neural network on sentence embeddings obtained by calculating the mean of all word embeddings in the sentence.
- We train the neural network using AdamW optimizer with 1e-4 learning rate for 100 epochs with batch size 256.
- The accuracy on test set for this model is roughly between **82% - 84%**.

[26]:
```
class NeuralNetwork1(nn.Module):
    def __init__(self):
        super(NeuralNetwork1, self).__init__()
        self.embedding_dim = 300
        self.hidden1 = 50
        self.hidden2 = 5
        self.out_dim = 2

        self.linear = nn.Sequential(
            nn.Linear(self.embedding_dim, self.hidden1),
            nn.ReLU(),
            nn.Linear(self.hidden1, self.hidden2),
            nn.ReLU(),
            nn.Linear(self.hidden2, self.out_dim),

        )

    def forward(self, x):
        x = self.linear(x)

        return x

model1 = NeuralNetwork1()
model1.to(device)
print(model1)
```

```
NeuralNetwork1(
  (linear): Sequential(
    (0): Linear(in_features=300, out_features=50, bias=True)
    (1): ReLU()
```

```
    (2): Linear(in_features=50, out_features=5, bias=True)
    (3): ReLU()
    (4): Linear(in_features=5, out_features=2, bias=True)
  )
)
```

[27]:
```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.AdamW(model1.parameters(), lr=1e-4)
```

[28]:
```
# training the model
train(model1, train_loader1, test_loader1, criterion, optimizer, n_epochs=100,␣
 ↪device=device)
```

```
Epoch: 1 / 100  Training Loss: 223.27994138002396       Test Loss:
54.716657280921936
Epoch: 2 / 100  Training Loss: 205.70290875434875       Test Loss:
48.29630637168884
Epoch: 3 / 100  Training Loss: 176.52640929818153       Test Loss:
41.207524448633194
Epoch: 4 / 100  Training Loss: 155.0030519068241        Test Loss:
37.60107463598251
Epoch: 5 / 100  Training Loss: 144.89616572856903       Test Loss:
35.95973256230354
Epoch: 6 / 100  Training Loss: 140.06317156553268       Test Loss:
35.0801557302475
Epoch: 7 / 100  Training Loss: 137.1439170241356        Test Loss:
34.57513916492462
Epoch: 8 / 100  Training Loss: 135.18273961544037       Test Loss:
34.08438017964363
Epoch: 9 / 100  Training Loss: 133.62331506609917       Test Loss:
33.74214479327202
Epoch: 10 / 100 Training Loss: 132.32582929730415       Test Loss:
33.46182382106781
Epoch: 11 / 100 Training Loss: 131.22946453094482       Test Loss:
33.21343466639519
Epoch: 12 / 100 Training Loss: 130.28965264558792       Test Loss:
32.998418152332306
Epoch: 13 / 100 Training Loss: 129.46428593993187       Test Loss:
32.80886098742485
Epoch: 14 / 100 Training Loss: 128.77806401252747       Test Loss:
32.64651048183441
Epoch: 15 / 100 Training Loss: 128.14307433366776       Test Loss:
32.497187316417694
Epoch: 16 / 100 Training Loss: 127.5267682671547        Test Loss:
32.354840099811554
Epoch: 17 / 100 Training Loss: 126.88095933198929       Test Loss:
32.239379823207855
Epoch: 18 / 100 Training Loss: 126.27783998847008       Test Loss:
```

32.22042775154114
Epoch: 19 / 100 Training Loss: 125.91750055551529 Test Loss: 31.992936491966248
Epoch: 20 / 100 Training Loss: 125.32829281687737 Test Loss: 31.872982531785965
Epoch: 21 / 100 Training Loss: 124.9611741900444 Test Loss: 31.765440344810486
Epoch: 22 / 100 Training Loss: 124.49292731285095 Test Loss: 31.662551641464233
Epoch: 23 / 100 Training Loss: 124.03318306803703 Test Loss: 31.565906643867493
Epoch: 24 / 100 Training Loss: 123.64501696825027 Test Loss: 31.490920424461365
Epoch: 25 / 100 Training Loss: 123.26009619235992 Test Loss: 31.418159753084183
Epoch: 26 / 100 Training Loss: 122.91045781970024 Test Loss: 31.33314400911331
Epoch: 27 / 100 Training Loss: 122.6216288805008 Test Loss: 31.26932805776596
Epoch: 28 / 100 Training Loss: 122.26745873689651 Test Loss: 31.225706696510315
Epoch: 29 / 100 Training Loss: 121.92679944634438 Test Loss: 31.129334658384323
Epoch: 30 / 100 Training Loss: 121.6411349773407 Test Loss: 31.135849595069885
Epoch: 31 / 100 Training Loss: 121.3110009431839 Test Loss: 31.019828230142593
Epoch: 32 / 100 Training Loss: 121.060700237751 Test Loss: 30.958229959011078
Epoch: 33 / 100 Training Loss: 120.7994986474514 Test Loss: 30.945522993803024
Epoch: 34 / 100 Training Loss: 120.50094133615494 Test Loss: 30.85480245947838
Epoch: 35 / 100 Training Loss: 120.2346733212471 Test Loss: 30.826304495334625
Epoch: 36 / 100 Training Loss: 120.0566368997097 Test Loss: 30.808255553245544
Epoch: 37 / 100 Training Loss: 119.85122066736221 Test Loss: 30.851676404476166
Epoch: 38 / 100 Training Loss: 119.64552560448647 Test Loss: 30.70193523168564
Epoch: 39 / 100 Training Loss: 119.43237388134003 Test Loss: 30.66961708664894
Epoch: 40 / 100 Training Loss: 119.18115195631981 Test Loss: 30.64968267083168
Epoch: 41 / 100 Training Loss: 119.03138810396194 Test Loss: 30.572869837284088
Epoch: 42 / 100 Training Loss: 118.7706771492958 Test Loss: 30.539514303207397

```
Epoch: 43 / 100 Training Loss: 118.66438245773315       Test Loss:
30.56415206193924
Epoch: 44 / 100 Training Loss: 118.44496589899063       Test Loss:
30.49730333685875
Epoch: 45 / 100 Training Loss: 118.27230963110924       Test Loss:
30.445301681756973
Epoch: 46 / 100 Training Loss: 118.13230195641518       Test Loss:
30.425574600696564
Epoch: 47 / 100 Training Loss: 117.95960614085197       Test Loss:
30.442767202854156
Epoch: 48 / 100 Training Loss: 117.84438782930374       Test Loss:
30.37128135561943
Epoch: 49 / 100 Training Loss: 117.6802129149437        Test Loss:
30.35201469063759
Epoch: 50 / 100 Training Loss: 117.48000007867813       Test Loss:
30.369530647993088
Epoch: 51 / 100 Training Loss: 117.35213413834572       Test Loss:
30.313990354537964
Epoch: 52 / 100 Training Loss: 117.1872521340847        Test Loss:
30.272311061620712
Epoch: 53 / 100 Training Loss: 117.05581396818161       Test Loss:
30.287183433771133
Epoch: 54 / 100 Training Loss: 116.89013743400574       Test Loss:
30.219446301460266
Epoch: 55 / 100 Training Loss: 116.75635251402855       Test Loss:
30.19229105114937
Epoch: 56 / 100 Training Loss: 116.64956653118134       Test Loss:
30.226687908172607
Epoch: 57 / 100 Training Loss: 116.44064235687256       Test Loss:
30.174104303121567
Epoch: 58 / 100 Training Loss: 116.2725133895874        Test Loss:
30.144307047128677
Epoch: 59 / 100 Training Loss: 116.18694359064102       Test Loss:
30.14928701519966
Epoch: 60 / 100 Training Loss: 116.06538730859756       Test Loss:
30.069722801446915
Epoch: 61 / 100 Training Loss: 115.90143203735352       Test Loss:
30.04452556371689
Epoch: 62 / 100 Training Loss: 115.8383446931839        Test Loss:
30.03296783566475
Epoch: 63 / 100 Training Loss: 115.65640524029732       Test Loss:
30.038346081972122
Epoch: 64 / 100 Training Loss: 115.46543255448341       Test Loss:
29.980317026376724
Epoch: 65 / 100 Training Loss: 115.36341765522957       Test Loss:
29.972163796424866
Epoch: 66 / 100 Training Loss: 115.20414933562279       Test Loss:
29.93922859430313
```

Epoch: 67 / 100 Training Loss: 115.07365503907204 Test Loss:
29.927978098392487
Epoch: 68 / 100 Training Loss: 114.92494955658913 Test Loss:
29.88445019721985
Epoch: 69 / 100 Training Loss: 114.82510590553284 Test Loss:
29.96846315264702
Epoch: 70 / 100 Training Loss: 114.70393919944763 Test Loss:
29.846114546060562
Epoch: 71 / 100 Training Loss: 114.50519832968712 Test Loss:
29.823754519224167
Epoch: 72 / 100 Training Loss: 114.43318873643875 Test Loss:
29.961664497852325
Epoch: 73 / 100 Training Loss: 114.29965949058533 Test Loss:
29.841556757688522
Epoch: 74 / 100 Training Loss: 114.14315912127495 Test Loss:
29.775625854730606
Epoch: 75 / 100 Training Loss: 114.02090355753899 Test Loss:
29.751518547534943
Epoch: 76 / 100 Training Loss: 113.9025110900402 Test Loss:
29.726153671741486
Epoch: 77 / 100 Training Loss: 113.76714563369751 Test Loss:
29.707972019910812
Epoch: 78 / 100 Training Loss: 113.66112577915192 Test Loss:
29.693700343370438
Epoch: 79 / 100 Training Loss: 113.52604535222054 Test Loss:
29.68906545639038
Epoch: 80 / 100 Training Loss: 113.44295200705528 Test Loss:
29.694132387638092
Epoch: 81 / 100 Training Loss: 113.22596323490143 Test Loss:
29.699741423130035
Epoch: 82 / 100 Training Loss: 113.16738465428352 Test Loss:
29.632641434669495
Epoch: 83 / 100 Training Loss: 113.02845710515976 Test Loss:
29.603986263275146
Epoch: 84 / 100 Training Loss: 112.89375838637352 Test Loss:
29.577137231826782
Epoch: 85 / 100 Training Loss: 112.86698698997498 Test Loss:
29.59031268954277
Epoch: 86 / 100 Training Loss: 112.68282690644264 Test Loss:
29.568561047315598
Epoch: 87 / 100 Training Loss: 112.57322052121162 Test Loss:
29.532892733812332
Epoch: 88 / 100 Training Loss: 112.46350705623627 Test Loss:
29.514625161886215
Epoch: 89 / 100 Training Loss: 112.35060584545135 Test Loss:
29.497875690460205
Epoch: 90 / 100 Training Loss: 112.22641348838806 Test Loss:
29.4856516122818

```
Epoch: 91 / 100 Training Loss: 112.12844929099083       Test Loss:
29.459724247455597
Epoch: 92 / 100 Training Loss: 111.98545953631401       Test Loss:
29.45690569281578
Epoch: 93 / 100 Training Loss: 111.86591139435768       Test Loss:
29.447207391262054
Epoch: 94 / 100 Training Loss: 111.7561075091362        Test Loss:
29.399598449468613
Epoch: 95 / 100 Training Loss: 111.67023974657059       Test Loss:
29.484870731830597
Epoch: 96 / 100 Training Loss: 111.57436427474022       Test Loss:
29.38424116373062
Epoch: 97 / 100 Training Loss: 111.46817779541016       Test Loss:
29.367492109537125
Epoch: 98 / 100 Training Loss: 111.29240453243256       Test Loss:
29.386514335870743
Epoch: 99 / 100 Training Loss: 111.14512953162193       Test Loss:
29.411173075437546
Epoch: 100 / 100        Training Loss: 111.15172135829926       Test Loss:
29.344766169786453
```

[29]:

```python
part_4a_accuracy = accuracy(model1, test_loader1, device)
print(part_4a_accuracy)
```

```
83.965
```

[30]:

```python
def create_pad_embeddings(sentence, w2v, max_len=10):
    tokens = nltk.word_tokenize(sentence)
    vec_size = w2v.vector_size

    embedding = np.zeros((max_len * vec_size), dtype=np.float32)
    for i, word in enumerate(tokens):
        if i >= max_len:
            break

        if word in w2v:
            embedding[i*vec_size: (i+1)*vec_size] = w2v[word]

    return embedding

sentence = "This is a sample sentence"
sample_embedding = create_pad_embeddings(sentence, pretrained_w2v)
print(sample_embedding.shape)

print(create_avg_embeddings(sentence, pretrained_w2v).shape)
```

```
(3000,)
(300,)
```

[31]:
```
pad_embeddings = df_new['review_body'].apply(lambda x: create_pad_embeddings(x,␣
    ↪pretrained_w2v))
pad_embeddings = np.array(pad_embeddings.tolist())
```

[32]:
```
X_train_pad, X_test_pad, Y_train_pad, Y_test_pad = train_test_split(
    pad_embeddings,
    targets,
    shuffle=True,
    test_size=0.2,
    random_state=42
)

Y_train_pad = np.array(Y_train_pad.tolist())
Y_test_pad = np.array(Y_test_pad.tolist())
```

[33]:
```
X_train_pad = torch.tensor(X_train_pad, dtype=torch.float32)
X_test_pad = torch.tensor(X_test_pad, dtype=torch.float32)

Y_train_pad = torch.tensor(Y_train_pad, dtype=torch.int64)
Y_test_pad = torch.tensor(Y_test_pad, dtype=torch.int64)

train_dataset2 = TensorDataset(X_train_pad, Y_train_pad)
test_dataset2 = TensorDataset(X_test_pad, Y_test_pad)

batch_size = 256
train_loader2 = DataLoader(train_dataset2, batch_size=batch_size, shuffle=True)
test_loader2 = DataLoader(test_dataset2, batch_size=batch_size)
```

4.2 Task 4 (b)

- Here, we train a feedforward neural network on sentence embeddings obtained by concatenating the first 10 word embeddings and applying padding if the sentence is smaller than 10 words.
- We use a ReLU activation layer between the linear layers.
- We train the neural network using AdamW optimizer with 1e-4 learning rate and 1e-4 weight decay for 30 epochs with batch size 256.
- Upon re-running the code several times, we observe the accuracy on test set for this model is roughly between **75.5% - 76.6%**.

[34]:
```
class NeuralNetwork2(nn.Module):
    def __init__(self):
        super(NeuralNetwork2, self).__init__()
        self.embedding_dim = 3000
        self.hidden1 = 50
        self.hidden2 = 5
        self.out_dim = 2
```

```
        self.linear = nn.Sequential(
            nn.Linear(self.embedding_dim, self.hidden1),
            nn.ReLU(),
            nn.Linear(self.hidden1, self.hidden2),
            nn.ReLU(),
            nn.Linear(self.hidden2, self.out_dim),
        )

    def forward(self, x):
        x = self.linear(x)

        return x

model2 = NeuralNetwork2()
model2.to(device)
print(model2)
```

```
NeuralNetwork2(
  (linear): Sequential(
    (0): Linear(in_features=3000, out_features=50, bias=True)
    (1): ReLU()
    (2): Linear(in_features=50, out_features=5, bias=True)
    (3): ReLU()
    (4): Linear(in_features=5, out_features=2, bias=True)
  )
)
```

[35]:
```
optimizer = torch.optim.AdamW(model2.parameters(), lr=1e-4, weight_decay=1e-4)
```

[36]:
```
train(model2, train_loader2, test_loader2, criterion=criterion,
 ↪optimizer=optimizer, n_epochs=30, device=device)
```

```
Epoch: 1 / 30   Training Loss: 194.17466282844543       Test Loss:
43.72030174732208
Epoch: 2 / 30   Training Loss: 165.8945385813713        Test Loss:
41.193147748708725
Epoch: 3 / 30   Training Loss: 159.00936275720596       Test Loss:
40.39856415987015
Epoch: 4 / 30   Training Loss: 155.71505045890808       Test Loss:
39.918146044015884
Epoch: 5 / 30   Training Loss: 153.38784858584404       Test Loss:
39.65698781609535
Epoch: 6 / 30   Training Loss: 151.5978156030178        Test Loss:
39.41517451405525
Epoch: 7 / 30   Training Loss: 149.9291484951973        Test Loss:
39.31141784787178
Epoch: 8 / 30   Training Loss: 148.38147443532944       Test Loss:
39.11142221093178
```

```
Epoch: 9 / 30   Training Loss: 146.90436762571335       Test Loss:
39.007114231586456
Epoch: 10 / 30  Training Loss: 145.52796256542206       Test Loss:
38.82473134994507
Epoch: 11 / 30  Training Loss: 144.09228175878525       Test Loss:
38.75255364179611
Epoch: 12 / 30  Training Loss: 142.6417380273342        Test Loss:
38.639875173568726
Epoch: 13 / 30  Training Loss: 141.23964083194733       Test Loss:
38.59825983643532
Epoch: 14 / 30  Training Loss: 140.0570511519909        Test Loss:
38.49330136179924
Epoch: 15 / 30  Training Loss: 138.561135917902 Test Loss: 38.5560596883297
Epoch: 16 / 30  Training Loss: 137.25511133670807       Test Loss:
38.54057967662811
Epoch: 17 / 30  Training Loss: 135.8898992240429        Test Loss:
38.45652109384537
Epoch: 18 / 30  Training Loss: 134.39312362670898       Test Loss:
38.396814465522766
Epoch: 19 / 30  Training Loss: 133.03255796432495       Test Loss:
38.37224414944649
Epoch: 20 / 30  Training Loss: 131.66759631037712       Test Loss:
38.39131438732147
Epoch: 21 / 30  Training Loss: 130.14192607998848       Test Loss:
38.43498423695564
Epoch: 22 / 30  Training Loss: 128.69674035906792       Test Loss:
38.42015391588211
Epoch: 23 / 30  Training Loss: 127.28686335682869       Test Loss:
38.41508102416992
Epoch: 24 / 30  Training Loss: 125.83272141218185       Test Loss:
38.49851316213608
Epoch: 25 / 30  Training Loss: 124.14575991034508       Test Loss:
38.61513492465019
Epoch: 26 / 30  Training Loss: 122.76370960474014       Test Loss:
38.81642150878906
Epoch: 27 / 30  Training Loss: 121.15855580568314       Test Loss:
38.708027839660645
Epoch: 28 / 30  Training Loss: 119.53005722165108       Test Loss:
38.78369262814522
Epoch: 29 / 30  Training Loss: 117.86586755514145       Test Loss:
38.875079065561295
Epoch: 30 / 30  Training Loss: 116.18799751996994       Test Loss:
38.919744431972504
```

[37]:
```
part_4b_accuracy = accuracy(model2, test_loader2, device=device)
print(part_4b_accuracy)
```

```
76.29
```

5 Task 5: Recurrent Neural Networks

[38]:
```
def create_block_embeddings(sentence, w2v, max_len=10):
    tokens = nltk.word_tokenize(sentence)
    vec_size = w2v.vector_size

    embedding = np.zeros((max_len, vec_size), dtype=np.float32)
    for i, word in enumerate(tokens):
        if i >= max_len:
            break

        if word in w2v:
            embedding[i] = w2v[word]

    return embedding

sentence = "This is a sample sentence"
sample_embedding = create_block_embeddings(sentence, pretrained_w2v)
```

[39]:
```
block_embeddings = df_new['review_body'].apply(lambda x:␣
 ↪create_block_embeddings(x, pretrained_w2v))
block_embeddings = np.array(block_embeddings.tolist())
```

[40]:
```
X_train_block, X_test_block, Y_train_block, Y_test_block = train_test_split(
    block_embeddings,
    targets,
    shuffle=True,
    test_size=0.2,
    random_state=42
)

Y_train_block = np.array(Y_train_block.tolist())
Y_test_block = np.array(Y_test_block.tolist())
```

[41]:
```
X_train_block = torch.tensor(X_train_block, dtype=torch.float32)
X_test_block = torch.tensor(X_test_block, dtype=torch.float32)

Y_train_block = torch.tensor(Y_train_block, dtype=torch.int64)
Y_test_block = torch.tensor(Y_test_block, dtype=torch.int64)

train_dataset3 = TensorDataset(X_train_block, Y_train_block)
test_dataset3 = TensorDataset(X_test_block, Y_test_block)

batch_size = 256
train_loader3 = DataLoader(train_dataset3, batch_size=batch_size, shuffle=True)
test_loader3 = DataLoader(test_dataset3, batch_size=batch_size)
```

5.1 Task 5 (a)

- For this task, we train a RNN on sentence embeddings obtained by stacking the first 10 word embeddings and applying padding if the sentence is smaller than 10 words.
- We train the neural network using AdamW optimizer with 1e-3 learning rate with 1e-6 weight decay for 100 epochs with batch size 256.
- Upon re-running the code several times, we observe the accuracy on test set for this model is roughly around **77.7% - 78.5%**.

[42]:
```
class RNNClf(nn.Module):
    def __init__(self):
        super(RNNClf, self).__init__()
        self.embed_size = 300
        self.hidden_size = 10
        self.out_size = 2

        self.rnn = nn.RNN(input_size=self.embed_size, hidden_size=self.
 ↪hidden_size, batch_first=True)
        self.linear = nn.Linear(self.hidden_size, self.out_size)

    def forward(self, x):
        x, hidden = self.rnn(x)
        x = self.linear(x[:, -1, :])

        return x

rnnModel = RNNClf()
rnnModel.to(device)
print(rnnModel)
```

```
RNNClf(
  (rnn): RNN(300, 10, batch_first=True)
  (linear): Linear(in_features=10, out_features=2, bias=True)
)
```

[43]:
```
optimizer = torch.optim.AdamW(rnnModel.parameters(), lr=1e-3, weight_decay=1e-6)
```

[44]:
```
train(rnnModel, train_loader3, test_loader3, criterion, optimizer,␣
 ↪n_epochs=100, device=device)
```

```
Epoch: 1 / 100  Training Loss: 189.24623772501945       Test Loss:
42.86137253046036
Epoch: 2 / 100  Training Loss: 165.2519319653511        Test Loss:
41.2786665558815
Epoch: 3 / 100  Training Loss: 160.46886545419693       Test Loss:
40.46919998526573
Epoch: 4 / 100  Training Loss: 158.11708521842957       Test Loss:
40.11415392160416
Epoch: 5 / 100  Training Loss: 156.1327583193779        Test Loss:
```

```
39.94943976402283
Epoch: 6 / 100  Training Loss: 154.3838656246662        Test Loss:
39.207602590322495
Epoch: 7 / 100  Training Loss: 153.1704162955284        Test Loss:
39.585414320230484
Epoch: 8 / 100  Training Loss: 151.9054266512394        Test Loss:
38.63105762004852
Epoch: 9 / 100  Training Loss: 150.69831427931786       Test Loss:
38.46751955151558
Epoch: 10 / 100 Training Loss: 149.9881165921688        Test Loss:
38.16771939396858
Epoch: 11 / 100 Training Loss: 149.20873829722404       Test Loss:
38.12427684664726
Epoch: 12 / 100 Training Loss: 148.06970876455307       Test Loss:
37.88624459505081
Epoch: 13 / 100 Training Loss: 147.25514441728592       Test Loss:
37.83639848232269
Epoch: 14 / 100 Training Loss: 147.02438268065453       Test Loss:
37.81947907805443
Epoch: 15 / 100 Training Loss: 146.1519265472889        Test Loss:
37.40324741601944
Epoch: 16 / 100 Training Loss: 145.35491436719894       Test Loss:
37.36286148428917
Epoch: 17 / 100 Training Loss: 144.6717321574688        Test Loss:
37.37463703751564
Epoch: 18 / 100 Training Loss: 143.9930343925953        Test Loss:
37.337947338819504
Epoch: 19 / 100 Training Loss: 143.5869961977005        Test Loss:
37.078449696302414
Epoch: 20 / 100 Training Loss: 143.13337874412537       Test Loss:
37.605335503816605
Epoch: 21 / 100 Training Loss: 142.94152796268463       Test Loss:
37.53921481966972
Epoch: 22 / 100 Training Loss: 142.4502227306366        Test Loss:
36.98101082444191
Epoch: 23 / 100 Training Loss: 141.7295179963112        Test Loss:
36.969017028808594
Epoch: 24 / 100 Training Loss: 141.79342359304428       Test Loss:
36.92888504266739
Epoch: 25 / 100 Training Loss: 141.24457421898842       Test Loss:
36.921512484550476
Epoch: 26 / 100 Training Loss: 140.69035521149635       Test Loss:
36.67476940155029
Epoch: 27 / 100 Training Loss: 140.56993076205254       Test Loss:
36.67093303799629
Epoch: 28 / 100 Training Loss: 140.12270125746727       Test Loss:
36.673606127500534
Epoch: 29 / 100 Training Loss: 140.30863592028618       Test Loss:
```

36.74337786436081
Epoch: 30 / 100 Training Loss: 139.8257461488247 Test Loss: 36.717282712459564
Epoch: 31 / 100 Training Loss: 139.38076090812683 Test Loss: 36.759630620479584
Epoch: 32 / 100 Training Loss: 139.37510937452316 Test Loss: 36.75336068868637
Epoch: 33 / 100 Training Loss: 138.59608009457588 Test Loss: 36.559115529060364
Epoch: 34 / 100 Training Loss: 138.47659105062485 Test Loss: 36.69157314300537
Epoch: 35 / 100 Training Loss: 138.35360470414162 Test Loss: 36.672885090112686
Epoch: 36 / 100 Training Loss: 138.4716020822525 Test Loss: 36.34186860918999
Epoch: 37 / 100 Training Loss: 138.0806143283844 Test Loss: 36.49436393380165
Epoch: 38 / 100 Training Loss: 137.92330566048622 Test Loss: 36.363194674253464
Epoch: 39 / 100 Training Loss: 137.7402704358101 Test Loss: 36.36631777882576
Epoch: 40 / 100 Training Loss: 137.59592580795288 Test Loss: 36.46004328131676
Epoch: 41 / 100 Training Loss: 137.2826405465603 Test Loss: 36.5595398247242
Epoch: 42 / 100 Training Loss: 137.12348607182503 Test Loss: 36.4635514318943
Epoch: 43 / 100 Training Loss: 137.01033291220665 Test Loss: 36.34114542603493
Epoch: 44 / 100 Training Loss: 136.74183830618858 Test Loss: 36.264641135931015
Epoch: 45 / 100 Training Loss: 136.3272634446621 Test Loss: 36.134812384843826
Epoch: 46 / 100 Training Loss: 136.4916720688343 Test Loss: 36.68380865454674
Epoch: 47 / 100 Training Loss: 136.05500841140747 Test Loss: 36.26983544230461
Epoch: 48 / 100 Training Loss: 135.8873808979988 Test Loss: 36.39075103402138
Epoch: 49 / 100 Training Loss: 136.05873787403107 Test Loss: 36.15823784470558
Epoch: 50 / 100 Training Loss: 135.63928240537643 Test Loss: 36.31730031967163
Epoch: 51 / 100 Training Loss: 136.06965699791908 Test Loss: 36.068804532289505
Epoch: 52 / 100 Training Loss: 135.45133262872696 Test Loss: 36.76198589801788
Epoch: 53 / 100 Training Loss: 135.2078354358673 Test Loss:

36.18658712506294
Epoch: 54 / 100 Training Loss: 135.38859137892723 Test Loss:
36.101177006959915
Epoch: 55 / 100 Training Loss: 135.46006244421005 Test Loss:
36.34166017174721
Epoch: 56 / 100 Training Loss: 135.55552461743355 Test Loss:
35.95088368654251
Epoch: 57 / 100 Training Loss: 135.10890033841133 Test Loss:
37.04864928126335
Epoch: 58 / 100 Training Loss: 134.89186203479767 Test Loss:
36.29133144021034
Epoch: 59 / 100 Training Loss: 135.04605770111084 Test Loss:
36.06994870305061
Epoch: 60 / 100 Training Loss: 134.73051998019218 Test Loss:
36.03017449378967
Epoch: 61 / 100 Training Loss: 134.77700686454773 Test Loss:
36.26178798079491
Epoch: 62 / 100 Training Loss: 134.56750398874283 Test Loss:
36.36930540204048
Epoch: 63 / 100 Training Loss: 134.3830843269825 Test Loss:
35.89995536208153
Epoch: 64 / 100 Training Loss: 134.22210678458214 Test Loss:
37.44350093603134
Epoch: 65 / 100 Training Loss: 134.31491148471832 Test Loss:
36.42756715416908
Epoch: 66 / 100 Training Loss: 133.98335886001587 Test Loss:
35.9828961789608
Epoch: 67 / 100 Training Loss: 133.91022527217865 Test Loss:
36.17883452773094
Epoch: 68 / 100 Training Loss: 133.51893836259842 Test Loss:
35.96130481362343
Epoch: 69 / 100 Training Loss: 133.90603253245354 Test Loss:
36.07605600357056
Epoch: 70 / 100 Training Loss: 133.3829669356346 Test Loss:
35.91865962743759
Epoch: 71 / 100 Training Loss: 133.3340601027012 Test Loss:
36.70209327340126
Epoch: 72 / 100 Training Loss: 133.73578864336014 Test Loss:
36.085397362709045
Epoch: 73 / 100 Training Loss: 133.52470207214355 Test Loss:
36.064752370119095
Epoch: 74 / 100 Training Loss: 133.1799268424511 Test Loss:
36.026156067848206
Epoch: 75 / 100 Training Loss: 133.26336923241615 Test Loss:
35.974625289440155
Epoch: 76 / 100 Training Loss: 133.26395693421364 Test Loss:
35.977458477020264
Epoch: 77 / 100 Training Loss: 132.8995196223259 Test Loss:

36.6982978284359
Epoch: 78 / 100 Training Loss: 133.10581400990486 Test Loss:
35.85940346121788
Epoch: 79 / 100 Training Loss: 132.6943188905716 Test Loss:
35.95867204666138
Epoch: 80 / 100 Training Loss: 132.88827127218246 Test Loss:
35.84911406040192
Epoch: 81 / 100 Training Loss: 133.28380650281906 Test Loss:
36.01386970281601
Epoch: 82 / 100 Training Loss: 132.37537708878517 Test Loss:
36.04409897327423
Epoch: 83 / 100 Training Loss: 132.8774455487728 Test Loss:
35.851144552230835
Epoch: 84 / 100 Training Loss: 132.53932788968086 Test Loss:
35.89164027571678
Epoch: 85 / 100 Training Loss: 132.59623190760612 Test Loss:
36.02489432692528
Epoch: 86 / 100 Training Loss: 132.02949604392052 Test Loss:
36.67078024148941
Epoch: 87 / 100 Training Loss: 132.04837357997894 Test Loss:
35.88687840104103
Epoch: 88 / 100 Training Loss: 132.11943557858467 Test Loss:
35.89442652463913
Epoch: 89 / 100 Training Loss: 131.82287102937698 Test Loss:
35.74980768561363
Epoch: 90 / 100 Training Loss: 132.0360058248043 Test Loss:
35.946353524923325
Epoch: 91 / 100 Training Loss: 131.82789173722267 Test Loss:
35.85142061114311
Epoch: 92 / 100 Training Loss: 131.64694225788116 Test Loss:
35.84598225355148
Epoch: 93 / 100 Training Loss: 131.85038036108017 Test Loss:
36.01533231139183
Epoch: 94 / 100 Training Loss: 131.46432846784592 Test Loss:
36.32975900173187
Epoch: 95 / 100 Training Loss: 131.63746419548988 Test Loss:
36.03915509581566
Epoch: 96 / 100 Training Loss: 131.45140880346298 Test Loss:
35.91527062654495
Epoch: 97 / 100 Training Loss: 131.1892467737198 Test Loss:
35.95221844315529
Epoch: 98 / 100 Training Loss: 131.4272505044937 Test Loss:
36.182045221328735
Epoch: 99 / 100 Training Loss: 131.37129864096642 Test Loss:
35.74391394853592
Epoch: 100 / 100 Training Loss: 131.15149646997452 Test Loss:
36.11794114112854

```
[45]: rnn_accuracy = accuracy(rnnModel, test_loader3, device)

print(rnn_accuracy)
```

```
77.955
```

5.2 Task 5(b): GRU

- Here, we train a GRU on sentence embeddings obtained by stacking the first 10 word embeddings and applying padding if the sentence is smaller than 10 words.
- We train the neural network using AdamW optimizer with 1e-3 learning rate with 1e-2 weight decay for 30 epochs with batch size 256.
- Upon re-running the code several times, we observe the accuracy on test set for this model is roughly around **79.1% - 79.7%**.

```
[46]: class GRUClf(nn.Module):
    def __init__(self):
        super(GRUClf, self).__init__()
        self.embed_size = 300

        self.hidden_size = 10
        self.out_size = 2

        self.gru = nn.GRU(input_size=self.embed_size, hidden_size=self.
 ↪hidden_size, batch_first=True)
        self.linear = nn.Linear(self.hidden_size, self.out_size)

    def forward(self, x):
        x, hidden = self.gru(x)
        x = self.linear(x[:, -1, :])

        return x

gruModel = GRUClf()
gruModel.to(device)
print(gruModel)
```

```
GRUClf(
  (gru): GRU(300, 10, batch_first=True)
  (linear): Linear(in_features=10, out_features=2, bias=True)
)
```

```
[47]: optimizer = torch.optim.AdamW(gruModel.parameters(), lr=1e-3, weight_decay=1e-2)
```

```
[48]: train(gruModel, train_loader3, test_loader3, criterion, optimizer, n_epochs=30,␣
 ↪device=device)
```

```
Epoch: 1 / 30   Training Loss: 181.33681312203407       Test Loss:
40.29534995555878
```

```
Epoch: 2 / 30   Training Loss: 152.9514371752739        Test Loss:
38.26805377006531
Epoch: 3 / 30   Training Loss: 146.8739361166954        Test Loss:
37.00517484545708
Epoch: 4 / 30   Training Loss: 143.26274248957634       Test Loss:
36.36405465006828
Epoch: 5 / 30   Training Loss: 141.10489463806152       Test Loss:
35.926189661026
Epoch: 6 / 30   Training Loss: 139.08646640181541       Test Loss:
35.67848202586174
Epoch: 7 / 30   Training Loss: 137.7913582623005        Test Loss:
35.40585646033287
Epoch: 8 / 30   Training Loss: 136.35473904013634       Test Loss:
35.08504235744476
Epoch: 9 / 30   Training Loss: 135.20484054088593       Test Loss:
34.93747437000275
Epoch: 10 / 30  Training Loss: 134.1550863981247        Test Loss:
34.81758573651314
Epoch: 11 / 30  Training Loss: 133.29672893881798       Test Loss:
34.710423558950424
Epoch: 12 / 30  Training Loss: 132.54534032940865       Test Loss:
34.62228259444237
Epoch: 13 / 30  Training Loss: 131.5119285285473        Test Loss:
34.5435888171196
Epoch: 14 / 30  Training Loss: 131.05372193455696       Test Loss:
34.433901876211166
Epoch: 15 / 30  Training Loss: 130.31678879261017       Test Loss:
34.589472591876984
Epoch: 16 / 30  Training Loss: 129.95355436205864       Test Loss:
34.305437207221985
Epoch: 17 / 30  Training Loss: 128.96858605742455       Test Loss:
34.44904673099518
Epoch: 18 / 30  Training Loss: 128.68122020363808       Test Loss:
34.321634382009506
Epoch: 19 / 30  Training Loss: 128.16218599677086       Test Loss:
34.297405660152435
Epoch: 20 / 30  Training Loss: 127.77931371331215       Test Loss:
34.212056785821915
Epoch: 21 / 30  Training Loss: 127.33910638093948       Test Loss:
34.577800303697586
Epoch: 22 / 30  Training Loss: 127.14896404743195       Test Loss:
34.326654225587845
Epoch: 23 / 30  Training Loss: 126.58092293143272       Test Loss:
34.400550216436386
Epoch: 24 / 30  Training Loss: 126.18402501940727       Test Loss:
34.0765418112278
Epoch: 25 / 30  Training Loss: 125.64939358830452       Test Loss:
34.106071442365646
```

```
Epoch: 26 / 30  Training Loss: 125.47544345259666       Test Loss:
34.14732027053833
Epoch: 27 / 30  Training Loss: 125.23535743355751       Test Loss:
34.172821909189224
Epoch: 28 / 30  Training Loss: 124.74122029542923       Test Loss:
33.96618315577507
Epoch: 29 / 30  Training Loss: 124.36532750725746       Test Loss:
34.191488176584244
Epoch: 30 / 30  Training Loss: 124.17429465055466       Test Loss:
34.5347506403923
```

[49]:
```
gru_accuracy = accuracy(gruModel, test_loader3, device)

print(gru_accuracy)
```

```
79.275
```

5.3 Task 5(c): LSTM

- Here, we train a LSTM on sentence embeddings obtained by stacking the first 10 word embeddings and applying padding if the sentence is smaller than 10 words.
- We train the neural network using AdamW optimizer with 1e-3 learning rate and 1e-2 weight decay for 30 epochs with batch size 256.
- A relu activation layer between the lstm cell and the linear layer is applied. Also, the output is passed through a tanh layer before being returned by the model.
- Upon re-running the code several times, we observe the accuracy on test set for this model is roughly between **79.2% - 79.8%**.

[50]:
```
class LSTMClf(nn.Module):
    def __init__(self):
        super(LSTMClf, self).__init__()
        self.embed_size = 300
        self.hidden_size = 10
        self.out_size = 2

        self.lstm = nn.LSTM(input_size=self.embed_size, hidden_size=self.
 ↪hidden_size, batch_first=True)
        self.relu = nn.ReLU()
        self.linear = nn.Linear(self.hidden_size, self.out_size)
        self.tanh = nn.Tanh()


    def forward(self, x):
        x, (hidden, cell) = self.lstm(x)
        x = self.relu(x)
        x = self.linear(x[:, -1, :])
        x = self.tanh(x)
```

```
        return x

lstmModel = LSTMClf()
lstmModel.to(device)
print(lstmModel)
```

```
LSTMClf(
  (lstm): LSTM(300, 10, batch_first=True)
  (relu): ReLU()
  (linear): Linear(in_features=10, out_features=2, bias=True)
  (tanh): Tanh()
)
```

[51]:
```
optimizer = torch.optim.AdamW(lstmModel.parameters(), lr=1e-3,␣
↪weight_decay=1e-2)
```

[52]:
```
train(lstmModel, train_loader3, test_loader3, criterion, optimizer,␣
↪n_epochs=30, device=device)
```

```
Epoch: 1 / 30   Training Loss: 188.53236678242683       Test Loss:
42.60245108604431
Epoch: 2 / 30   Training Loss: 161.78118962049484       Test Loss:
40.11137869954109
Epoch: 3 / 30   Training Loss: 154.7561023235321        Test Loss:
39.04612699151039
Epoch: 4 / 30   Training Loss: 150.59949985146523       Test Loss:
38.261914163827896
Epoch: 5 / 30   Training Loss: 147.96168661117554       Test Loss:
37.8667289018631
Epoch: 6 / 30   Training Loss: 145.9180660545826        Test Loss:
37.60482919216156
Epoch: 7 / 30   Training Loss: 144.42314419150352       Test Loss:
37.16939628124237
Epoch: 8 / 30   Training Loss: 142.6743516921997        Test Loss:
36.983817517757416
Epoch: 9 / 30   Training Loss: 141.31478962302208       Test Loss:
36.718128740787506
Epoch: 10 / 30  Training Loss: 140.29806298017502       Test Loss:
36.9502349793911
Epoch: 11 / 30  Training Loss: 139.1070382297039        Test Loss:
36.48337149620056
Epoch: 12 / 30  Training Loss: 138.49619647860527       Test Loss:
36.24354547262192
Epoch: 13 / 30  Training Loss: 137.58472111821175       Test Loss:
36.23565372824669
Epoch: 14 / 30  Training Loss: 136.69559437036514       Test Loss:
36.28344339132309
Epoch: 15 / 30  Training Loss: 136.19343376159668       Test Loss:
```

```
36.044730961322784
Epoch: 16 / 30  Training Loss: 135.28828984498978       Test Loss:
36.178139090538025
Epoch: 17 / 30  Training Loss: 134.70898419618607       Test Loss:
35.870521783828735
Epoch: 18 / 30  Training Loss: 133.9309034049511        Test Loss:
35.881180971860886
Epoch: 19 / 30  Training Loss: 133.92980736494064       Test Loss:
35.94572842121124
Epoch: 20 / 30  Training Loss: 132.89750257134438       Test Loss:
35.94690823554993
Epoch: 21 / 30  Training Loss: 132.76369643211365       Test Loss:
35.764438807964325
Epoch: 22 / 30  Training Loss: 131.8276786506176        Test Loss:
35.75800174474716
Epoch: 23 / 30  Training Loss: 131.56945458054543       Test Loss:
36.201402485370636
Epoch: 24 / 30  Training Loss: 131.06357857584953       Test Loss:
35.585066854953766
Epoch: 25 / 30  Training Loss: 130.93170562386513       Test Loss:
35.64456853270531
Epoch: 26 / 30  Training Loss: 130.27441161870956       Test Loss:
36.144619435071945
Epoch: 27 / 30  Training Loss: 129.86872991919518       Test Loss:
35.68116870522499
Epoch: 28 / 30  Training Loss: 129.67316290736198       Test Loss:
35.78095597028732
Epoch: 29 / 30  Training Loss: 129.42498436570168       Test Loss:
35.652844071388245
Epoch: 30 / 30  Training Loss: 128.7190609574318        Test Loss:
35.79785969853401
```

[53]:
```
lstm_accuracy = accuracy(lstmModel, test_loader3, device)

print(lstm_accuracy)
```

```
79.56
```