Task 1: Reading Data

In a manner like the approach in the first homework assignment, pandas functions was utilized to import the dataset and initial stages of text-based data cleaning were initiated. To streamline the analysis, all columns from the dataset were pruned, retaining only the review label and the text review.

Note: Google Colab was used for this homework, to harness its GPU capabilities.

Task 2: Processing Data

The data undergoes two distinct processing steps:

TFIDF Transformation: This process, as previously executed in Homework 1, involves using the TFIDF vectorizer to convert text data into numerical representations.

Generating Average Word Vectors: We use Google's Word2Vec model through Gensim. Initially, we load the 300-word model using Gensim's API downloader and verify its functionality. Subsequently, we train this model on the dataset and conduct further testing. Below are the results from our experimentation with these models:

```
Pre-trained model: king+woman-man = [('queen', 0.7118193507194519)]
Similarity between excellent and outstanding: 0.5567486

word2vec model trained on the dataset:
king+woman-man = [('(many', 0.55901038646698)]
Similarity between excellent and outstanding: 0.748253
```

From the above results it can be concluded as follows:

1) The pre-trained model appears to perform better in capturing semantic relationships and word similarities in the context of the specific vector arithmetic task (e.g., "king + woman - man").

2) The model trained on the Amazon reviews dataset performs differently depending on the specific domain and data it was trained on. It appears to indicate a higher similarity between "excellent" and "outstanding," which could be a reflection of the language usage patterns within the Amazon reviews dataset.

3) The differences between the models may be attributed to variations in training data, dataset size, and domain-specific language patterns. Pre-trained models benefit from large and diverse datasets, which may contribute to their ability to capture more general semantic relationships. Model's performance is likely influenced by the characteristics of the Amazon reviews dataset.

To prepare the data for tasks 3 and 4a, I determined the vector size for a word2vec model, created a set of words in the model's index, and generated average word vectors for reviews in a training dataset. For each word, I checked if it's in the model's index, used the word's vector if available, and assigned a random vector if not. These average vectors were organized into a DataFrame along with labels from the training dataset.

An 80-20 split was taken from this DataFrame as the training and testing data for the tasks ahead.

Task 3: SVM and Perceptron

Once the training and testing data has been prepared, they are fed to sklearn’s LinearSVM() and Perceptron() to train and evaluate their performance. Simultaneously, the Tf-Idf Data used in the first homework was also fed to the models to compare results.
Following are the results of the experimentation:

	precision	recall	f1-score
1	0.87	0.86	0.87
2	0.86	0.87	0.87
accuracy			0.87
macro avg	0.87	0.87	0.87
weighted avg	0.87	0.87	0.87

SVM with tf-idf data

	precision	recall	f1-score
1	0.84	0.78	0.81
2	0.76	0.83	0.79
accuracy			0.80
macro avg	0.80	0.80	0.80
weighted avg	0.80	0.80	0.80

SVM with Word2Vec data

	precision	recall	f1-score
1	0.90	0.67	0.77
2	0.56	0.86	0.68
accuracy			0.73
macro avg	0.73	0.76	0.72
weighted avg	0.79	0.73	0.74

Perceptron with tf-idf data

	precision	recall	f1-score
1	0.64	0.80	0.71
2	0.84	0.70	0.77
accuracy			0.74
macro avg	0.74	0.75	0.74
weighted avg	0.76	0.74	0.74

Perceptron with Word2Vec data

Observations and Conclusions:

1. The Perceptron classifier achieved a higher accuracy of **74%** when using Word2Vec features compared to the **73%** accuracy achieved with TF-IDF features. This suggests that, for the specific classification task and dataset, Word2Vec features were more effective in capturing relevant information and patterns for the Perceptron classifier.
2. SVM with TF-IDF achieved an accuracy of **87%,** which was the highest among all the models tested.
3. SVM vs. Perceptron: In general, the SVM classifier tends to perform better than the Perceptron classifier in this task, regardless of the feature type used. This could be due to the SVM's ability to handle non-linear relationships in the data and find better decision boundaries.

Task 4: FeedForward Neural Network

In the process of training neural networks on PyTorch, data undergoes distinct preprocessing steps for scenarios 4a) and 4b).

In the initial situation, the data is retained in its original state. The training and testing datasets are adjusted to match the suitable data types that PyTorch's neural network models can work with. Following this adjustment, these datasets are transformed into tensors. These tensors are further structured into tensor datasets and integrated into PyTorch's data loaders, preparing the data for input into the model.

The model used in 4a) is a straightforward network consisting of three linear layers, with Rectified Linear Unit (ReLU) activation functions.

In the second scenario, each review in the dataset goes through a unique transformation. We select the first 10 words from each review that are found in the vocabulary of a pre-trained word2vec model. These chosen words are concatenated to create a comprehensive vector of size 3000 (300 dimensions for each of the 10 words). Following this transformation, the reviews are subjected to the same preprocessing steps as in scenario a) and are then prepared for model training.

The model employed in 4b) is like the one in 4a) but differs in terms of the vector size and incorporates a Softmax activation function in the final layer to improve accuracy.

In each scenario, the models underwent training for different numbers of epochs. The models were trained beyond overfitting till the point where the test loss started increasing with every epoch.

Accuracy 4a) **81.18 %**

```
#calculating accuracy
preds = []
labels=[]
for data,target in test_loader:
    data, target = data.to(device), target.to(device)
    output = model(data)
    preds += list(torch.argmax(output,dim=1).cpu())
    labels += list(target.cpu())
    # preds.append(torch.argmax(output,dim=1).cpu())
    # labels.append(target[0].cpu())
print("Accuracy: ",accuracy_score(preds,labels))
```

```
Accuracy:  0.8118
```

Accuracy 4b) **74.3 %**

```
preds = []
labels=[]
for data,target in test_loader_n:
    # data, target = data.to(device), target.to(device)
    output = model_b(data)
    preds += list(torch.argmax(output,dim=1).cpu())
    labels += list(target.cpu())
    # preds.append(torch.argmax(output,dim=1).cpu())
    # labels.append(target[0].cpu())
print("Accuracy: ",accuracy_score(preds,labels))
```

```
Accuracy:  0.743
```

Conclusion on training MLP:

The MLP models achieved an accuracy of around 82%(averaged word2vec feautures) and around 75% (features of 10 words), both of which are better than the Perceptron and comparable to the SVM's performance. We can conlcude that the MLP generalizes better over averaged Word2Vec features than the sklearn models.

A possible reason why the second MLP accuracy is lower than the first could be that important information about the sentiment of a review might be more in the latter part of the reviews with length > 10, which might be represented more in the averaged vectors (training data of the 4 a). Since potentially better data may have been fed to the first MLP, it was able to perform better.

Task 5: Recurrent Neural Network

For task 5, the data is the same for all of a,b,c. The only difference is in the model architectures. To prepare the data for model training, I've defined a function called vectorize_reviews. It takes a list of text reviews, a pre-trained word2vec model, and a few other parameters to transform the reviews into vectorized representations. I loop through each review, break it into words, and assign word vectors from the word2vec model if they exist. If a word isn't found, I use a predefined random vector. The resulting vectorized reviews are returned. Then, I use this function to vectorize reviews from my training_dataset, followed by a train-test split to create training and testing datasets for my machine learning model.

Accuracies of the following models on test set are as follows:

Accuracy of Simple RNN: **75.2 %**

Accuracy of LSTM unit cell: **78.1 %**

Accuracy of GRU: **78.4 %**

Conclusion:

The simple RNN achieved better accuracy compared to the MLP models, when fed with features of the first 10 words. It could be because the RNN captures generality over Word2Vec features better than the MLP (or simple FNN). The architecture of RNN may be the reason why it performs better.

Of the three models, the GRU performs the best, performing slightly better than the LSTM unit cell. All the models were trained beyond overfitting till the point where the test loss started increasing with every epoch, following prof. Rostami's advice about deep learning models generalize better despite being overfitted on the training data.

The pages that follow showcase the practical execution of the assignment in the Jupyter notebook:

NLP_HW3

October 18, 2023

[ ]:
```
! pip install gensim
```

```
Collecting gensim
  Using cached gensim-4.3.2-cp310-cp310-macosx_11_0_arm64.whl (24.0 MB)
Requirement already satisfied: numpy>=1.18.5 in
/Users/shreyavinaynayak/miniconda3/lib/python3.10/site-packages (from gensim)
(1.25.2)
Collecting smart-open>=1.8.1
  Using cached smart_open-6.4.0-py3-none-any.whl (57 kB)
Requirement already satisfied: scipy>=1.7.0 in
/Users/shreyavinaynayak/miniconda3/lib/python3.10/site-packages (from gensim)
(1.11.2)
Installing collected packages: smart-open, gensim
Successfully installed gensim-4.3.2 smart-open-6.4.0
```

[ ]:
```
!pip install torch torchvision torchaudio
```

```
Collecting torch
  Downloading torch-2.0.1-cp310-none-macosx_11_0_arm64.whl (55.8 MB)
                           55.8/55.8 MB
21.8 MB/s eta 0:00:0000:0100:01
Collecting torchvision
  Downloading torchvision-0.15.2-cp310-cp310-macosx_11_0_arm64.whl (1.4 MB)
                           1.4/1.4 MB
17.5 MB/s eta 0:00:0000:0100:01
Collecting torchaudio
  Downloading torchaudio-2.0.2-cp310-cp310-macosx_11_0_arm64.whl (3.6 MB)
                           3.6/3.6 MB
23.7 MB/s eta 0:00:00a 0:00:01
Collecting filelock
  Downloading filelock-3.12.4-py3-none-any.whl (11 kB)
Collecting typing-extensions
  Downloading typing_extensions-4.8.0-py3-none-any.whl (31 kB)
Collecting networkx
  Downloading networkx-3.1-py3-none-any.whl (2.1 MB)
                           2.1/2.1 MB
20.7 MB/s eta 0:00:0000:0100:01
Collecting jinja2
  Downloading Jinja2-3.1.2-py3-none-any.whl (133 kB)
```

```
                           133.1/133.1
kB 5.4 MB/s eta 0:00:00
Collecting sympy
  Downloading sympy-1.12-py3-none-any.whl (5.7 MB)
                           5.7/5.7 MB
23.1 MB/s eta 0:00:0000:0100:01
Requirement already satisfied: requests in
/Users/shreyavinaynayak/miniconda3/lib/python3.10/site-packages (from
torchvision) (2.28.1)
Requirement already satisfied: numpy in
/Users/shreyavinaynayak/miniconda3/lib/python3.10/site-packages (from
torchvision) (1.25.2)
Collecting pillow!=8.3.*,>=5.3.0
  Downloading Pillow-10.0.1-cp310-cp310-macosx_11_0_arm64.whl (3.3 MB)
                           3.3/3.3 MB
23.3 MB/s eta 0:00:0000:0100:01
Collecting MarkupSafe>=2.0
  Downloading MarkupSafe-2.1.3-cp310-cp310-macosx_10_9_universal2.whl (17 kB)
Requirement already satisfied: idna<4,>=2.5 in
/Users/shreyavinaynayak/miniconda3/lib/python3.10/site-packages (from
requests->torchvision) (3.4)
Requirement already satisfied: certifi>=2017.4.17 in
/Users/shreyavinaynayak/miniconda3/lib/python3.10/site-packages (from
requests->torchvision) (2022.12.7)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in
/Users/shreyavinaynayak/miniconda3/lib/python3.10/site-packages (from
requests->torchvision) (1.26.15)
Requirement already satisfied: charset-normalizer<3,>=2 in
/Users/shreyavinaynayak/miniconda3/lib/python3.10/site-packages (from
requests->torchvision) (2.0.4)
Collecting mpmath>=0.19
  Downloading mpmath-1.3.0-py3-none-any.whl (536 kB)
                          536.2/536.2 kB
12.5 MB/s eta 0:00:0000:01
Installing collected packages: mpmath, typing-extensions, sympy, pillow,
networkx, MarkupSafe, filelock, jinja2, torch, torchvision, torchaudio
Successfully installed MarkupSafe-2.1.3 filelock-3.12.4 jinja2-3.1.2
mpmath-1.3.0 networkx-3.1 pillow-10.0.1 sympy-1.12 torch-2.0.1 torchaudio-2.0.2
torchvision-0.15.2 typing-extensions-4.8.0
```

Python Version: 3.10.12

Library Versions :

torch : 2.0.1+cu118
gensim : 4.3.2
tqdm : 4.66.1

numpy : 1.23.5
pandas : 1.5.3
sklearn : 1.2.2
torchvision : 0.15.2+cu118

[ ]:
```
import pandas as pd
import numpy as np
import nltk
import pickle
nltk.download('wordnet')
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('averaged_perceptron_tagger')
nltk.download('omw-1.4')
import re
from bs4 import BeautifulSoup
import nltk
from tqdm import tqdm
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
from nltk.corpus.reader.wordnet import NOUN, VERB, ADJ, ADV
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from sklearn.linear_model import Perceptron,LogisticRegression
from sklearn.metrics import␣
↪accuracy_score,precision_score,recall_score,f1_score,classification_report
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.svm import SVC,LinearSVC
from sklearn.naive_bayes import MultinomialNB
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV,␣
↪train_test_split
import gensim.downloader as api
import gensim
import torch
from torch.utils.data import DataLoader, Dataset
import torchvision
import torchvision.transforms as transforms
from torch.utils.data.sampler import SubsetRandomSampler
import torch.nn as nn
import torch.nn.functional as F
import gc
```

```
[nltk_data] Downloading package wordnet to /root/nltk_data…
[nltk_data] Downloading package punkt to /root/nltk_data…
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package stopwords to /root/nltk_data…
[nltk_data]   Unzipping corpora/stopwords.zip.
```

```
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /root/nltk_data…
[nltk_data]   Unzipping taggers/averaged_perceptron_tagger.zip.
[nltk_data] Downloading package omw-1.4 to /root/nltk_data…
```

[ ]:
```
# As the notebook was run on colab to utilize GPUs, the following code cell has
↪been commented out to make it executabele locally

# from google.colab import drive
# drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

[ ]:
```
#Reading the tsv file
# the line commented below was the one used for colab, un-commented line added
↪for local execution
# df = pd.read_table('/content/drive/MyDrive/Shreya Data/Shreya NLP/HW3/data.
↪tsv',on_bad_lines='skip')
#assuming that data.tsv is in the same directory as the notebook:
df = pd.read_table('data.tsv',on_bad_lines = 'skip')
```

```
<ipython-input-3-f4a9d796f4e5>:2: DtypeWarning: Columns (7) have mixed types.
Specify dtype option on import or set low_memory=False.
  df = pd.read_table('/content/drive/MyDrive/Shreya Data/Shreya
NLP/HW3/data.tsv',on_bad_lines='skip')
```

[ ]:
```
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
```

0.0.1 Keep Reviews and Ratings

[ ]:
```
data = df[['star_rating','review_body']] #keeping only columns needed
data.dropna(axis=0,inplace=True)
```

```
<ipython-input-5-7d9acd4d4411>:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  data.dropna(axis=0,inplace=True)
```

[ ]:
```
data['star_rating'].value_counts()
```

[ ]:
```
5    1458992
4     389603
1     286072
3     179867
2     129031
5     123770
4      28757
1      20896
3      13819
2       9350
Name: star_rating, dtype: int64
```

[ ]:
```
# making all rows as integers
data['star_rating']=data['star_rating'].astype('int')
data['star_rating'].value_counts()
```

```
<ipython-input-8-7a970350aaa6>:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead
```

```
See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  data['star_rating']=data['star_rating'].astype('int')
```

[ ]:
```
5    1582762
4     418360
1     306968
3     193686
2     138381
Name: star_rating, dtype: int64
```

[ ]:
```
#splitting data into classes

class1 = data[data['star_rating']<=3] #defining class 1 for ratings with values␣
↪1,2,3
labels = [1]*len(class1)
class1['label'] = labels

class2 = data[data['star_rating']>=4]  #defining class 2 for ratings with␣
↪values 4,5
labels = [2]*len(class2)
class2['label'] = labels
```

```
<ipython-input-9-d096d57f7e16>:5: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  class1['label'] = labels
<ipython-input-9-d096d57f7e16>:9: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  class2['label'] = labels
```

0.0.2 We form two classes and select 50000 reviews randomly from each class.

[ ]:
```
# Sampling 50,000 random reviews from each class
sampled_class1 = class1.sample(n=50000, random_state=42)  # Using a fixed␣
↪random state for reproducibility
sampled_class2 = class2.sample(n=50000, random_state=42)
# Concatenating the sampled data to create a balanced dataset
balanced_data = pd.concat([sampled_class1, sampled_class2], ignore_index=True)
# Shuffle the dataset
```

```
training_dataset = balanced_data.sample(frac=1, random_state=42).
    ↪reset_index(drop=True)
```

[ ]:
```
del df
```

[ ]:
```
training_dataset.head()
```

[ ]:
```
   star_rating                                       review_body  label
0            5  the phone case is awesome  I've had other phon…      2
1            5                                           perfect      2
2            1          fast delivery… grand daughter likes it      1
3            5                           This is a great product.      2
4            5  Works great and so much cheaper than buying at…      2
```

0.1 Loading saved training data

[ ]:
```
# commented out as the same was stored in drive
#training_dataset = pd.read_csv('/content/drive/MyDrive/Shreya Data/Shreya NLP/
    ↪HW3/hw3_processed_data.csv')
```

1 TF-IDF and BoW Feature Extraction

[ ]:
```
vectorizer = TfidfVectorizer()
frequency_matrix = vectorizer.fit_transform(training_dataset['review_body'])

count_vectorizer = CountVectorizer()
bow_features = count_vectorizer.fit_transform(training_dataset['review_body'])
```

[ ]:
```
X_train_tf, X_test_tf, y_train_tf, y_test_tf =␣
    ↪train_test_split(frequency_matrix, training_dataset['label'], test_size=0.2,␣
    ↪random_state=42)
```

2 Task 2

2.0.1 Loading Word2Vec model using GenSim

2.0.2 (a)

[ ]:
```
import gensim.downloader as api
# to save notebook memory, model was stored in drive to load directly.␣
    ↪commented out for local execution

# model_path = "/content/drive/MyDrive/Shreya Data/Shreya NLP/HW3/
    ↪word2vec_model.bin"

# # Check if model is already in Drive, if not, download and save
# try:
```

```
#     gen_word_2_vec = gensim.models.KeyedVectors.load(model_path)
#     print("Model loaded from Google Drive.")
# except:
#     print("Downloading model...")
#     gen_word_2_vec = api.load("word2vec-google-news-300")
#     gen_word_2_vec.save(model_path)
#     print("Model saved to Google Drive.")

#Loading word2vec model using gensim
gen_word_2_vec = api.load('word2vec-google-news-300')
```

```
Model loaded from Google Drive.
```

[ ]:
```
gen_word_2_vec
```

[ ]:
```
#Loading the word2vec model to test the vocabulary
words_to_print = 15
vocabulary = gen_word_2_vec.index_to_key
for index, word in enumerate(vocabulary[:words_to_print]):
    print(f"word #{index}/{len(vocabulary)} is {word}")
```

```
word #0/3000000 is </s>
word #1/3000000 is in
word #2/3000000 is for
word #3/3000000 is that
word #4/3000000 is is
word #5/3000000 is on
word #6/3000000 is ##
word #7/3000000 is The
word #8/3000000 is with
word #9/3000000 is said
word #10/3000000 is was
word #11/3000000 is the
word #12/3000000 is at
word #13/3000000 is not
word #14/3000000 is as
```

[ ]:
```
# Example 1: Checking king+woman-man=queen
gen_word_2_vec.most_similar(positive=['woman', 'king'], negative=['man'],␣
 ↪topn=1)
```

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

[ ]:
```
# Example 2: checking similarity score of 2 similar words
gen_word_2_vec.similarity("king","monarch")
```

[ ]:
```
0.64131945
```

[ ]:
```python
# Example 3: checking similarity score of 2 similar words
gen_word_2_vec.similarity("excellent","outstanding")
```

[ ]: 0.55674857

[ ]:
```python
#Extra example
gen_word_2_vec.most_similar(positive=['girl', 'son'], negative=['boy'], topn=1)
```

[ ]: [('daughter', 0.9154544472694397)]

2.0.3 (b) Training a word2vec model on the dataset

[ ]:
```python
#splitting data into sentences to train the new model on the data
sentences = [x.split() for x in training_dataset['review_body']]
model = gensim.models.Word2Vec(sentences, vector_size=300, window=13,␣
 ↪min_count=9)
model.save('trained_model1.model')
```

Check the semantic similarities for the same two examples in part (a)

[ ]:
```python
# testing king+woman-man=queen
model.wv.most_similar(positive=['woman','king'], negative=['man'], topn=1)
```

[ ]: [('(many', 0.55901038646698)]

[ ]:
```python
model.wv.most_similar(positive=['girl','son'], negative=['boy'], topn=1)
```

[ ]: [('sister', 0.762198805809021)]

[ ]:
```python
# same similarity
model.wv.similarity("excellent","outstanding")
```

[ ]: 0.7482453

What do you conclude from comparing vectors generated by yourself and the pre-trained model?

Which of the Word2Vec models seems to encode semantic similarities between words better? Pre-trained model: king+woman-man = [('queen', 0.7118193507194519)]
Similarity between excellent and outstanding: 0.5567486

word2vec model trained on the dataset:
king+woman-man = [('(many', 0.55901038646698)]
Similarity between excellent and outstanding: 0.748253

Conclusion:

1) The pre-trained model appears to perform better in capturing semantic relationships and word similarities in the context of the specific vector arithmetic task (e.g., "king + woman - man").

2) The model trained on the Amazon reviews dataset performs differently depending on the specific domain and data it was trained on. It appears to indicate a higher similarity between "excellent" and "outstanding," which could be a reflection of the language usage patterns within the Amazon reviews dataset.

3) The differences between the models may be attributed to variations in training data, dataset size, and domain-specific language patterns. Pre-trained models benefit from large and diverse datasets, which may contribute to their ability to capture more general semantic relationships. Model's performance is likely influenced by the characteristics of the Amazon reviews dataset.

[ ]:
```
vector_size = gen_word_2_vec.vector_size
index_to_key_set = set(gen_word_2_vec.index_to_key)  # converting to set for␣
↪O(1) lookups

# Generate a random non-zero vector for words not found in index_to_key_set
random_vector = np.random.randn(vector_size)

train_avg_vectors = [
    np.mean([
        gen_word_2_vec[word] if word in index_to_key_set else random_vector
        for word in review.split()
    ], axis=0)
    for review in tqdm(training_dataset['review_body'], desc="Processing␣
↪reviews")
]
```

```
Processing reviews: 100%|      | 100000/100000 [00:15<00:00, 6412.78it/s]
```

[ ]:
```
sum(sum(np.isnan(train_avg_vectors)))
```

[ ]:
```
0
```

[ ]:
```
vectors = pd.DataFrame(np.vstack(train_avg_vectors))
```

[ ]:
```
# To give every avg_vector corresponding label
vectors['label'] = training_dataset['label']
```

[ ]:
```
vectors.shape
```

[ ]:
```
(100000, 301)
```

[ ]:
```
#splitting into training and testing sets
columns = list(vectors.columns)
columns.remove('label')
```

```
X_train,X_test,y_train,y_test =␣
↪train_test_split(vectors[columns],vectors['label'],test_size = 0.2)
```

[ ]:
```
del vectors
```

3 Task 3 : Simple models

SVM

[ ]:
```
#training SVM on TFIDF
svc_mod = LinearSVC() #(C=0.025, intercept_scaling = 1.0, max_iter = 10000000,␣
↪tol = 1e-5)
svc_mod.fit(X_train_tf,y_train_tf)
svc_preds = svc_mod.predict(X_test_tf)
print(classification_report(svc_preds,y_test_tf))
```

```
              precision    recall  f1-score   support

           1       0.87      0.86      0.87     10092
           2       0.86      0.87      0.87      9908

    accuracy                           0.87     20000
   macro avg       0.87      0.87      0.87     20000
weighted avg       0.87      0.87      0.87     20000
```

[ ]:
```
#training SVM on word2vec
svc_mod = LinearSVC()
svc_mod.fit(X_train,y_train)
svc_preds = svc_mod.predict(X_test)
print(classification_report(svc_preds,y_test))
```

```
              precision    recall  f1-score   support

           1       0.84      0.78      0.81     10827
           2       0.76      0.83      0.79      9173

    accuracy                           0.80     20000
   macro avg       0.80      0.80      0.80     20000
weighted avg       0.80      0.80      0.80     20000
```

Perceptron

[ ]:
```
# training perceptron on tfidf
perceptron = Perceptron(alpha = 0.00065,max_iter = 100000, penalty =␣
↪'elasticnet',tol=0.000001, l1_ratio=0.25)
perceptron.fit(X_train_tf,y_train_tf)
percep_preds = perceptron.predict(X_test_tf)
```

```
print(classification_report(percep_preds,y_test_tf))
```

```
              precision    recall  f1-score   support

           1       0.90      0.67      0.77     13404
           2       0.56      0.86      0.68      6596

    accuracy                           0.73     20000
   macro avg       0.73      0.76      0.72     20000
weighted avg       0.79      0.73      0.74     20000
```

[ ]:
```
# training perceptron on word2vec
perceptron = Perceptron(alpha = 0.00065,max_iter = 100000, penalty =␣
↪'elasticnet',tol=0.000001, l1_ratio=0.25) #(alpha = 0.000035,max_iter =␣
↪100000, penalty = 'elasticnet',tol=0.0001, l1_ratio=0.1)
perceptron.fit(X_train,y_train)
percep_preds = perceptron.predict(X_test)
print(classification_report(percep_preds,y_test))
```

```
              precision    recall  f1-score   support

           1       0.64      0.80      0.71      7985
           2       0.84      0.70      0.77     12015

    accuracy                           0.74     20000
   macro avg       0.74      0.75      0.74     20000
weighted avg       0.76      0.74      0.74     20000
```

Accuracy Scores:

Using TFIDF: SVM accuracy =87%
Perceptron accuracy = 73% ##### Using word2vec: SVM accuracy =80%
Perceptron accuracy =74%

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

1) The Perceptron classifier achieved a higher accuracy of 74% when using Word2Vec features compared to the 73% accuracy achieved with TF-IDF features. This suggests that, for the specific classification task and dataset, Word2Vec features were more effective in capturing relevant information and patterns for the Perceptron classifier.

2)SVM with TF-IDF achieved an accuracy of 87%, which was the highest among all the models tested.

3) SVM vs. Perceptron: In general, the SVM classifier tends to perform better than the Perceptron classifier in this task, regardless of the feature type used. This could be due to the SVM's

ability to handle non-linear relationships in the data and find better decision boundaries.

4 Task 4 : Feedforward Neural Networks

4a)

[ ]:
```
#data generator and dataloader from word2vec data
nn_training_data = torch.from_numpy(X_train.astype('float32').to_numpy())␣
 ↪#float32 to match model weight dtypes
nn_testing_data = torch.from_numpy(X_test.astype('float32').to_numpy())
nn_training_label = torch.from_numpy((y_train-1).astype('long').to_numpy())
nn_testing_label = torch.from_numpy((y_test-1).astype('long').to_numpy())
```

[ ]:
```
# Defining data loaders
training_data_td = torch.utils.data.
 ↪TensorDataset(nn_training_data,nn_training_label)
testing_data_td = torch.utils.data.
 ↪TensorDataset(nn_testing_data,nn_testing_label)
batch_size = 32
train_loader = torch.utils.data.DataLoader(training_data_td,batch_size =␣
 ↪batch_size)
test_loader = torch.utils.data.DataLoader(testing_data_td,batch_size =␣
 ↪batch_size)
```

[ ]:
```
#testing the dataloaders
X,y = next(iter(train_loader))
```

[ ]:
```
X.dtype
```

[ ]:
```
torch.float32
```

[ ]:
```
# defining model:
class FNN_model(nn.Module):
    def __init__(self):
        super(FNN_model,self).__init__()
        self.fc1 = nn.Linear(300,50) #input size, hidden 1 size
        self.fc2 = nn.Linear(50,5) #hidden 1 size, hidden 2 size
        self.fc3 = nn.Linear(5,2) #hidden 2 size, output size (since 2 classes)
        self.dropout = nn.Dropout(p=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 = FNN_model()
model
```

[ ]:
```
FNN_model(
  (fc1): Linear(in_features=300, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=5, bias=True)
  (fc3): Linear(in_features=5, out_features=2, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

[ ]:
```
lr = 0.0003
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(),lr)
```

[ ]:
```
# test run:

epochs = 1
for e in range(epochs):
    train_loss = 0.0
    test_loss = 0.0
    model.train() # prep model for training
    for data, target in train_loader:
        optimizer.zero_grad()
        output = model(data)
#           print("got here!")
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        train_loss += loss.item()*data.size(0)
    model.eval() # prep model for evaluation
    for data, target in test_loader:
        output = model(data)
        loss = criterion(output, target)
        test_loss += loss.item()*data.size(0)
    train_loss = train_loss/len(train_loader.dataset)
    test_loss = test_loss/len(test_loader.dataset)
    print('Epoch: {} \tTraining Loss: {:.6f} \tTest Loss: {:.6f}'.format(
        e+1,
        train_loss,
        test_loss
        ))
```

```
Epoch: 1        Training Loss: 0.722175         Test Loss: 0.710218
```

[ ]:
```
import torch
```

```python
# Check if CUDA (GPU) is available and set the device accordingly - utilizing␣
↪colab's GPU if available
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

model = model.to(device)  # Move model to the selected device
#600 so far
epochs = 600
for e in range(epochs):
    train_loss = 0.0
    test_loss = 0.0
    model.train()  # prep model for training

    for data, target in train_loader:
        # Move data and target to the selected device
        data, target = data.to(device), target.to(device)

        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        train_loss += loss.item() * data.size(0)

    model.eval()  # prep model for evaluation

    for data, target in test_loader:
        # Move data and target to the selected device
        data, target = data.to(device), target.to(device)

        output = model(data)
        loss = criterion(output, target)
        test_loss += loss.item() * data.size(0)

    train_loss = train_loss / len(train_loader.dataset)
    test_loss = test_loss / len(test_loader.dataset)

    if e % 10 == 0:
        print('Epoch: {} \tTraining Loss: {:.6f} \tTest Loss: {:.6f}'.format(
            e + 1,
            train_loss,
            test_loss
        ))
```

```
Using device: cuda:0
Epoch: 1        Training Loss: 0.696178         Test Loss: 0.694661
Epoch: 11       Training Loss: 0.692336         Test Loss: 0.692217
```

```
Epoch: 21       Training Loss: 0.691742         Test Loss: 0.691581
Epoch: 31       Training Loss: 0.690905         Test Loss: 0.690775
Epoch: 41       Training Loss: 0.689935         Test Loss: 0.689636
Epoch: 51       Training Loss: 0.688408         Test Loss: 0.687989
Epoch: 61       Training Loss: 0.686348         Test Loss: 0.685560
Epoch: 71       Training Loss: 0.683010         Test Loss: 0.681924
Epoch: 81       Training Loss: 0.678379         Test Loss: 0.676569
Epoch: 91       Training Loss: 0.672584         Test Loss: 0.669058
Epoch: 101      Training Loss: 0.664093         Test Loss: 0.658559
Epoch: 111      Training Loss: 0.651827         Test Loss: 0.643788
Epoch: 121      Training Loss: 0.635033         Test Loss: 0.624615
Epoch: 131      Training Loss: 0.618467         Test Loss: 0.603940
Epoch: 141      Training Loss: 0.601047         Test Loss: 0.581399
Epoch: 151      Training Loss: 0.585286         Test Loss: 0.560688
Epoch: 161      Training Loss: 0.572609         Test Loss: 0.543480
Epoch: 171      Training Loss: 0.563015         Test Loss: 0.529003
Epoch: 181      Training Loss: 0.553458         Test Loss: 0.519381
Epoch: 191      Training Loss: 0.546436         Test Loss: 0.508227
Epoch: 201      Training Loss: 0.539658         Test Loss: 0.502138
Epoch: 211      Training Loss: 0.536443         Test Loss: 0.497476
Epoch: 221      Training Loss: 0.531746         Test Loss: 0.489759
Epoch: 231      Training Loss: 0.529910         Test Loss: 0.484286
Epoch: 241      Training Loss: 0.524126         Test Loss: 0.482563
Epoch: 251      Training Loss: 0.521110         Test Loss: 0.477601
Epoch: 261      Training Loss: 0.519608         Test Loss: 0.473678
Epoch: 271      Training Loss: 0.516708         Test Loss: 0.474068
Epoch: 281      Training Loss: 0.511676         Test Loss: 0.467757
Epoch: 291      Training Loss: 0.511499         Test Loss: 0.465532
Epoch: 301      Training Loss: 0.510427         Test Loss: 0.464054
Epoch: 311      Training Loss: 0.508241         Test Loss: 0.461719
Epoch: 321      Training Loss: 0.504818         Test Loss: 0.459887
Epoch: 331      Training Loss: 0.505548         Test Loss: 0.458676
Epoch: 341      Training Loss: 0.501971         Test Loss: 0.456807
Epoch: 351      Training Loss: 0.503334         Test Loss: 0.454549
Epoch: 361      Training Loss: 0.501377         Test Loss: 0.454236
Epoch: 371      Training Loss: 0.499151         Test Loss: 0.452093
Epoch: 381      Training Loss: 0.498956         Test Loss: 0.452266
Epoch: 391      Training Loss: 0.498021         Test Loss: 0.449631
Epoch: 401      Training Loss: 0.497400         Test Loss: 0.448998
Epoch: 411      Training Loss: 0.495081         Test Loss: 0.447477
Epoch: 421      Training Loss: 0.497334         Test Loss: 0.447458
Epoch: 431      Training Loss: 0.494475         Test Loss: 0.446502
Epoch: 441      Training Loss: 0.494180         Test Loss: 0.445272
Epoch: 451      Training Loss: 0.492050         Test Loss: 0.444436
Epoch: 461      Training Loss: 0.492393         Test Loss: 0.445070
Epoch: 471      Training Loss: 0.490891         Test Loss: 0.443078
Epoch: 481      Training Loss: 0.491048         Test Loss: 0.442484
Epoch: 491      Training Loss: 0.491162         Test Loss: 0.442129
```

```
Epoch: 501      Training Loss: 0.489568        Test Loss: 0.441786
Epoch: 511      Training Loss: 0.489846        Test Loss: 0.440740
Epoch: 521      Training Loss: 0.490746        Test Loss: 0.442886
Epoch: 531      Training Loss: 0.491148        Test Loss: 0.440015
Epoch: 541      Training Loss: 0.485854        Test Loss: 0.439283
Epoch: 551      Training Loss: 0.486144        Test Loss: 0.438758
Epoch: 561      Training Loss: 0.486752        Test Loss: 0.438547
Epoch: 571      Training Loss: 0.486919        Test Loss: 0.440276
Epoch: 581      Training Loss: 0.486404        Test Loss: 0.439822
Epoch: 591      Training Loss: 0.485436        Test Loss: 0.438573
```

[ ]:
```
with open("model1.pkl", "wb") as f:
    pickle.dump(model, f)
```

Accuracy 4 a)

[ ]:
```
#calculating accuracy
preds = []
labels=[]
for data,target in test_loader:
    data, target = data.to(device), target.to(device)
    output = model(data)
    preds += list(torch.argmax(output,dim=1).cpu())
    labels += list(target.cpu())
    # preds.append(torch.argmax(output,dim=1).cpu())
    # labels.append(target[0].cpu())
print("Accuracy: ",accuracy_score(preds,labels))
```

```
Accuracy:  0.8118
```

[ ]:
```
# freeing some memory to avoid memory crashes:
# del train_loader
# del test_loader
# del training_data_td
# del testing_data_td
# del nn_training_data
# del nn_testing_data
# del nn_training_label
# del nn_testing_label
# del train_avg_vectors
# del model
# del svc_mod
# del perceptron
# del vocabulary
# del vectors
# del X_train,X_test,y_train,y_test
# del gen_word_2_vec
# del optimizer,criterion
# del preds,labels
```

4 b)

[ ]:
```
# #processing data again for 4b:

from tqdm import tqdm
import numpy as np

# Assuming vector_size is defined
vector_size = 300  # assuming this size
random_vector = np.random.randn(vector_size)

# Create a set for O(1) lookup complexity
index_to_key_set = set(gen_word_2_vec.index_to_key)

def process_review(review):
    words = review.split()[:10]
    res = np.zeros((10, vector_size))

    for idx, word in enumerate(words):
        res[idx] = gen_word_2_vec[word] if word in index_to_key_set else␣
↪random_vector

    return res.flatten()

# Process all reviews using a list comprehension within np.array which is more␣
↪memory efficient and faster
new_train_vectors = np.array([process_review(review) for review in␣
↪tqdm(training_dataset['review_body'], desc="Processing reviews")])
```

```
Processing reviews: 100%|      | 100000/100000 [00:04<00:00, 24572.93it/s]
```

[ ]:
```
del gen_word_2_vec
```

[ ]:
```
new_train_vectors = np.array(new_train_vectors)
```

[ ]:
```
new_train_vectors.shape
```

[ ]:
```
(100000, 3000)
```

[ ]:
```
#making a DataFrame for the new vectors, and making training and testing sets
n_vectors = pd.DataFrame(new_train_vectors)
n_vectors['label']=training_dataset['label']
columns = list(n_vectors.columns)
columns.remove('label')
X_train_n,X_test_n,y_train_n,y_test_n =␣
↪train_test_split(n_vectors[columns],n_vectors['label'],test_size = 0.2)
```

[ ]:
```
del new_train_vectors
```

```
[ ]: #data generator and dataloader
nn_train_data_n = torch.from_numpy(X_train_n.astype('float32').to_numpy())
nn_test_data_n = torch.from_numpy(X_test_n.astype('float32').to_numpy())
nn_train_label_n = torch.from_numpy((y_train_n-1).astype('long').to_numpy())
nn_test_label_n = torch.from_numpy((y_test_n-1).astype('long').to_numpy())
#making a tensor dataset for data loaders
train_data_td_n = torch.utils.data.
 ↪TensorDataset(nn_train_data_n,nn_train_label_n)
test_data_td_n = torch.utils.data.TensorDataset(nn_test_data_n,nn_test_label_n)
# Defining data loaders
train_loader_n = torch.utils.data.DataLoader(train_data_td_n,batch_size = 32)
test_loader_n = torch.utils.data.DataLoader(test_data_td_n,batch_size = 32)
```

```
[ ]: class FNN_model_b(nn.Module):
    def __init__(self):
        super(FNN_model_b,self).__init__()
        self.fc1 = nn.Linear(3000,50) #input size, hidden 1 size
        self.fc2 = nn.Linear(50,5) #hidden 1 size, hidden 2 size
        self.fc3 = nn.Linear(5,2) #hidden 2 size, output size (since 2 classes)
        self.dropout = nn.Dropout(p=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
```

```
[ ]: # loading the same model:
model_b = FNN_model_b()
model_b
```

```
[ ]: FNN_model_b(
  (fc1): Linear(in_features=3000, out_features=50, bias=True)
  (fc2): Linear(in_features=50, out_features=5, bias=True)
  (fc3): Linear(in_features=5, out_features=2, bias=True)
  (dropout): Dropout(p=0.2, inplace=False)
)
```

```
[ ]: lr = 0.001
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_b.parameters(),lr)
```

```
[ ]: epochs = 200
for e in range(epochs):
```

```
    train_loss = 0.0
    test_loss = 0.0
    model_b.train() # prep model for training
    for data, target in train_loader_n:
        optimizer.zero_grad()
        output = model_b(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        train_loss += loss.item()*data.size(0)
    model_b.eval() # prep model for evaluation
    for data, target in test_loader_n:
        output = model_b(data)
        loss = criterion(output, target)
        test_loss += loss.item()*data.size(0)
    train_loss = train_loss/len(train_loader_n.dataset)
    test_loss = test_loss/len(test_loader_n.dataset)
    if e%10 ==0:
        print('Epoch: {} \tTraining Loss: {:.6f} \tTest Loss: {:.6f}'.format(
        e+1,
        train_loss,
        test_loss
        ))
```

```
Epoch: 1        Training Loss: 0.694562         Test Loss: 0.692958
Epoch: 11       Training Loss: 0.664800         Test Loss: 0.656524
Epoch: 21       Training Loss: 0.578206         Test Loss: 0.556272
Epoch: 31       Training Loss: 0.549619         Test Loss: 0.529982
Epoch: 41       Training Loss: 0.534888         Test Loss: 0.522288
Epoch: 51       Training Loss: 0.522040         Test Loss: 0.513575
Epoch: 61       Training Loss: 0.513433         Test Loss: 0.513215
Epoch: 71       Training Loss: 0.503665         Test Loss: 0.506810
Epoch: 81       Training Loss: 0.494552         Test Loss: 0.506960
Epoch: 91       Training Loss: 0.485505         Test Loss: 0.510946
Epoch: 101      Training Loss: 0.476249         Test Loss: 0.506060
Epoch: 111      Training Loss: 0.465394         Test Loss: 0.502586
Epoch: 121      Training Loss: 0.454896         Test Loss: 0.509016
Epoch: 131      Training Loss: 0.443605         Test Loss: 0.517942
Epoch: 141      Training Loss: 0.434916         Test Loss: 0.515151
Epoch: 151      Training Loss: 0.421908         Test Loss: 0.523124
Epoch: 161      Training Loss: 0.411221         Test Loss: 0.516478
Epoch: 171      Training Loss: 0.401773         Test Loss: 0.528936
Epoch: 181      Training Loss: 0.389829         Test Loss: 0.542731
Epoch: 191      Training Loss: 0.377702         Test Loss: 0.549939
```

[ ]:
```
with open("model2.pkl", "wb") as f:
    pickle.dump(model, f)
```

[ ]:
```
preds = []
labels=[]
for data,target in test_loader_n:
    # data, target = data.to(device), target.to(device)
    output = model_b(data)
    preds += list(torch.argmax(output,dim=1).cpu())
    labels += list(target.cpu())
    # preds.append(torch.argmax(output,dim=1).cpu())
    # labels.append(target[0].cpu())
print("Accuracy: ",accuracy_score(preds,labels))
```

```
Accuracy:  0.743
```

Conclusion on training MLP:
The MLP models achieved an accuracy of around 82%(averaged word2vec feautures) and around 75% (features of 10 words), both of which are better than the Perceptron and comparable to the SVM's performance. We can conlcude that the MLP generalizes better over averaged Word2Vec features than the sklearn models.

A possible reason why the second MLP accuracy is lower than the first could be that important information about the sentiment of a review might be more in the latter part of the reviews with length > 10, which might be represented more in the averaged vectors (training data of the 4 a). Since potentially better data may have been fed to the first MLP, it was able to perform better.

5 Task 5 : Recurrent Neural Networks

5 a)

[ ]:
```
def vectorize_reviews(reviews, word2vec, max_words=10, vector_size=300):
    num_reviews = len(reviews)
    vectors = np.zeros((num_reviews, max_words, vector_size))

    word2vec_keys = set(word2vec.index_to_key)  # Precompute this
    random_vector = np.random.randn(vector_size)  # Predefined random vector

    for i, review in enumerate(tqdm(reviews, desc="Vectorizing reviews")):
        words = review.split()[:max_words]

        # Efficiently assign word vectors or random_vector based on condition
        vectors[i, :len(words)] = [word2vec[word] if word in word2vec_keys else␣
 ↪random_vector for word in words]

    return vectors

new_train_vectors_5 = vectorize_reviews(training_dataset['review_body'],␣
 ↪gen_word_2_vec)
```

```
# Train-test split
X_train_5, X_test_5, y_train_5, y_test_5 = train_test_split(
    new_train_vectors_5,
    training_dataset['label'],
    test_size=0.2
)
```

```
Vectorizing reviews: 100%|      | 100000/100000 [00:05<00:00, 17023.70it/s]
```

[ ]:
```
del gen_word_2_vec
```

[ ]:
```
del training_dataset
```

[ ]:
```
import gc
# Convert to Torch tensors and free up memory immediately after conversion to␣
↪save RAM
nn_train_data_5 = torch.from_numpy(X_train_5.astype('float32'))
del X_train_5
gc.collect()

nn_test_data_5 = torch.from_numpy(X_test_5.astype('float32'))
del X_test_5
gc.collect()

nn_train_label_5 = torch.from_numpy((y_train_5 - 1).astype('long').to_numpy())
del y_train_5
gc.collect()

nn_test_label_5 = torch.from_numpy((y_test_5 - 1).astype('long').to_numpy())
del y_test_5
gc.collect()

train_data_td_5 = torch.utils.data.TensorDataset(nn_train_data_5,␣
↪nn_train_label_5)
test_data_td_5 = torch.utils.data.TensorDataset(nn_test_data_5, nn_test_label_5)

BATCH_SIZE = 32  # Adjusted according to available memory
train_loader_5 = torch.utils.data.DataLoader(train_data_td_5,␣
↪batch_size=BATCH_SIZE, shuffle=True)
test_loader_5 = torch.utils.data.DataLoader(test_data_td_5,␣
↪batch_size=BATCH_SIZE)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-44-d030024f95ab> in <cell line: 15>()
```

```
     13
     14 # Convert to Torch tensors and free up memory immediately after␣
  ↪conversion
---> 15 nn_train_data_5 = torch.from_numpy(X_train_5.astype('float32'))
     16 del X_train_5
     17 gc.collect()

NameError: name 'X_train_5' is not defined
```

[ ]:
```
class RNN_5a(nn.Module):
    def __init__(self,input_size,hidden_size,output_size):
        super(RNN_5a, self).__init__()

        self.hidden_size = hidden_size
        self.input_size = input_size
#         self.embedding = nn.Embedding(input_size, hidden_size)
        self.rnn = nn.RNN(input_size=input_size, hidden_size=hidden_size,␣
↪batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)
#         self.sigmoid = nn.Sigmoid()
        self. softmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
#         x = self.embedding(x)
        # h0 = torch.zeros(1, x.size(0), self.rnn.hidden_size)
        h0 = torch.zeros(1, x.size(0), self.rnn.hidden_size).to(x.device)

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

[ ]:
```
model_task5a = RNN_5a(300,10,2)
model_task5a
```

[ ]:
```
RNN_5a(
  (rnn): RNN(300, 10, batch_first=True)
  (fc): Linear(in_features=10, out_features=2, bias=True)
  (softmax): LogSoftmax(dim=1)
)
```

[ ]:
```
lr = 0.0001
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_task5a.parameters(),lr)
```

[ ]:
```python
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f"Training on {device}")
#1400
# Move the model to the device
model_task5a.to(device)

epochs = 200
for e in range(epochs):
    train_loss = 0.0
    test_loss = 0.0

    model_task5a.train()  # prep model for training

    for data, target in train_loader_5:
        # Move data and target labels to the device
        data, target = data.to(device), target.to(device)

        optimizer.zero_grad()
        output = model_task5a(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        train_loss += loss.item() * data.size(0)

    model_task5a.eval()  # prep model for evaluation

    for data, target in test_loader_5:
        # Move data and target labels to the device
        data, target = data.to(device), target.to(device)

        output = model_task5a(data)
        loss = criterion(output, target)
        test_loss += loss.item() * data.size(0)

    train_loss = train_loss / len(train_loader_5.dataset)
    test_loss = test_loss / len(test_loader_5.dataset)

    if e % 5 == 0:
        print('Epoch: {} \tTraining Loss: {:.6f} \tTest Loss: {:.6f}'.format(
            e + 1,
            train_loss,
            test_loss
        ))
```

```
Training on cuda:0
Epoch: 1        Training Loss: 0.504585         Test Loss: 0.508744
Epoch: 6        Training Loss: 0.504546         Test Loss: 0.509365
```

```
Epoch: 11       Training Loss: 0.504347         Test Loss: 0.508746
Epoch: 16       Training Loss: 0.504238         Test Loss: 0.508798
Epoch: 21       Training Loss: 0.504215         Test Loss: 0.508271
Epoch: 26       Training Loss: 0.504068         Test Loss: 0.508333
Epoch: 31       Training Loss: 0.503989         Test Loss: 0.508120
Epoch: 36       Training Loss: 0.503887         Test Loss: 0.507966
Epoch: 41       Training Loss: 0.503641         Test Loss: 0.508122
Epoch: 46       Training Loss: 0.503587         Test Loss: 0.507697
Epoch: 51       Training Loss: 0.503555         Test Loss: 0.507855
Epoch: 56       Training Loss: 0.503368         Test Loss: 0.508104
Epoch: 61       Training Loss: 0.503274         Test Loss: 0.507600
Epoch: 66       Training Loss: 0.503149         Test Loss: 0.507688
Epoch: 71       Training Loss: 0.503140         Test Loss: 0.507589
Epoch: 76       Training Loss: 0.502985         Test Loss: 0.507588
Epoch: 81       Training Loss: 0.502946         Test Loss: 0.507426
Epoch: 86       Training Loss: 0.502778         Test Loss: 0.507607
Epoch: 91       Training Loss: 0.502755         Test Loss: 0.507629
Epoch: 96       Training Loss: 0.502574         Test Loss: 0.506947
Epoch: 101      Training Loss: 0.502560         Test Loss: 0.506815
Epoch: 106      Training Loss: 0.502551         Test Loss: 0.507210
Epoch: 111      Training Loss: 0.502267         Test Loss: 0.508302
Epoch: 116      Training Loss: 0.502142         Test Loss: 0.506997
Epoch: 121      Training Loss: 0.502177         Test Loss: 0.506985
Epoch: 126      Training Loss: 0.501948         Test Loss: 0.506867
Epoch: 131      Training Loss: 0.501922         Test Loss: 0.506495
Epoch: 136      Training Loss: 0.501804         Test Loss: 0.506812
Epoch: 141      Training Loss: 0.501665         Test Loss: 0.506625
Epoch: 146      Training Loss: 0.501623         Test Loss: 0.506501
Epoch: 151      Training Loss: 0.501572         Test Loss: 0.506403
Epoch: 156      Training Loss: 0.501436         Test Loss: 0.506130
Epoch: 161      Training Loss: 0.501271         Test Loss: 0.506522
Epoch: 166      Training Loss: 0.501219         Test Loss: 0.506462
Epoch: 171      Training Loss: 0.501150         Test Loss: 0.506607
Epoch: 176      Training Loss: 0.501036         Test Loss: 0.506217
Epoch: 181      Training Loss: 0.500897         Test Loss: 0.505907
Epoch: 186      Training Loss: 0.500754         Test Loss: 0.505595
Epoch: 191      Training Loss: 0.500796         Test Loss: 0.505525
Epoch: 196      Training Loss: 0.500676         Test Loss: 0.505571
```

[ ]:
```
# import pickle
# with open("model1_5a.pkl", "wb") as f:
#     pickle.dump(model_task5a.cpu(), f)
torch.save(model_task5a.state_dict(), 'model_task5a.pth')
```

[ ]:
```
#calculating accuracy
preds = []
labels=[]
```

```
for data,target in test_loader_5:
    data, target = data.to(device), target.to(device)
    output = model_task5a(data)
    preds += list(torch.argmax(output,dim=1).cpu())
    labels += list(target.cpu())
    # preds.append(torch.argmax(output,dim=1).cpu())
    # labels.append(target[0].cpu())
print("Accuracy: ",accuracy_score(preds,labels))
```

```
Accuracy:  0.75195
```

Conclusion on training with simple RNN: The RNN achieved better accuracy compared to the MLP models, when fed with features of the first 10 words. It could be because the RNN captures generality over Word2Vec features better than the MLP (or simple FNN). The architecture of RNN may be the reason why it performs better.

5b)

[ ]:
```
class RNN_5b(nn.Module):
    def __init__(self,input_size,hidden_size,output_size):
        super(RNN_5b, self).__init__()

        self.hidden_size = hidden_size
        self.input_size = input_size
        self.gru = nn.GRU(input_size=input_size, hidden_size=hidden_size,␣
 ↪batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        h0 = torch.zeros(1, x.size(0), self.gru.hidden_size).to(x.device)


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

[ ]:
```
model_task5b = RNN_5b(300,10,2)
model_task5b
```

[ ]:
```
RNN_5b(
  (gru): GRU(300, 10, batch_first=True)
  (fc): Linear(in_features=10, out_features=2, bias=True)
  (softmax): LogSoftmax(dim=1)
)
```

[ ]:
```
# loading predefined weights (saved)
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

```
# Load the state dict
state_dict = torch.load('model_task5b(2).pth', map_location=device)

# Load the state dict to the model
model_task5b.load_state_dict(state_dict)
```

[ ]: <All keys matched successfully>

[ ]:
```
lr = 0.001
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_task5b.parameters(),lr)
```

[ ]:
```
#actual training:
#1200 so far
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f"Training on {device}")

# Move the model to the device
model_task5b.to(device)

epochs = 200
for e in range(epochs):
    train_loss = 0.0
    test_loss = 0.0

    model_task5b.train()  # prep model for training

    for data, target in train_loader_5:
        # Move data and target labels to the device
        data, target = data.to(device), target.to(device)

        optimizer.zero_grad()
        output = model_task5b(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        train_loss += loss.item() * data.size(0)

    model_task5b.eval()  # prep model for evaluation

    for data, target in test_loader_5:
        # Move data and target labels to the device
        data, target = data.to(device), target.to(device)

        output = model_task5b(data)
```

```python
            loss = criterion(output, target)
            test_loss += loss.item() * data.size(0)

    train_loss = train_loss / len(train_loader_5.dataset)
    test_loss = test_loss / len(test_loader_5.dataset)

    if e % 5 == 0:
        print('Epoch: {} \tTraining Loss: {:.6f} \tTest Loss: {:.6f}'.format(
            e + 1,
            train_loss,
            test_loss
        ))
```

```
Training on cuda:0
Epoch: 1 	Training Loss: 0.427826 	Test Loss: 0.455237
Epoch: 6 	Training Loss: 0.427878 	Test Loss: 0.453585
Epoch: 11 	Training Loss: 0.427601 	Test Loss: 0.452934
Epoch: 16 	Training Loss: 0.427415 	Test Loss: 0.453379
Epoch: 21 	Training Loss: 0.427180 	Test Loss: 0.453180
Epoch: 26 	Training Loss: 0.427114 	Test Loss: 0.453084
Epoch: 31 	Training Loss: 0.426851 	Test Loss: 0.453103
Epoch: 36 	Training Loss: 0.426395 	Test Loss: 0.454192
Epoch: 41 	Training Loss: 0.426244 	Test Loss: 0.457943
Epoch: 46 	Training Loss: 0.426164 	Test Loss: 0.454048
Epoch: 51 	Training Loss: 0.426178 	Test Loss: 0.459215
Epoch: 56 	Training Loss: 0.425802 	Test Loss: 0.453513
Epoch: 61 	Training Loss: 0.425443 	Test Loss: 0.455512
Epoch: 66 	Training Loss: 0.425507 	Test Loss: 0.455220
Epoch: 71 	Training Loss: 0.425332 	Test Loss: 0.455201
Epoch: 76 	Training Loss: 0.425002 	Test Loss: 0.453309
Epoch: 81 	Training Loss: 0.424859 	Test Loss: 0.453884
Epoch: 86 	Training Loss: 0.424623 	Test Loss: 0.459841
Epoch: 91 	Training Loss: 0.424486 	Test Loss: 0.454053
Epoch: 96 	Training Loss: 0.424290 	Test Loss: 0.453517
Epoch: 101 	Training Loss: 0.424096 	Test Loss: 0.453776
Epoch: 106 	Training Loss: 0.423805 	Test Loss: 0.453453
Epoch: 111 	Training Loss: 0.423922 	Test Loss: 0.454752
Epoch: 116 	Training Loss: 0.423530 	Test Loss: 0.453878
Epoch: 121 	Training Loss: 0.423425 	Test Loss: 0.455472
Epoch: 126 	Training Loss: 0.423303 	Test Loss: 0.456338
Epoch: 131 	Training Loss: 0.422947 	Test Loss: 0.453549
Epoch: 136 	Training Loss: 0.422824 	Test Loss: 0.454112
Epoch: 141 	Training Loss: 0.422594 	Test Loss: 0.454062
Epoch: 146 	Training Loss: 0.422504 	Test Loss: 0.455184
Epoch: 151 	Training Loss: 0.422309 	Test Loss: 0.453673
Epoch: 156 	Training Loss: 0.421925 	Test Loss: 0.458717
Epoch: 161 	Training Loss: 0.421994 	Test Loss: 0.454776
Epoch: 166 	Training Loss: 0.421719 	Test Loss: 0.453877
```

```
Epoch: 171      Training Loss: 0.421462         Test Loss: 0.454111
Epoch: 176      Training Loss: 0.421563         Test Loss: 0.454158
Epoch: 181      Training Loss: 0.421307         Test Loss: 0.456691
Epoch: 186      Training Loss: 0.420986         Test Loss: 0.454429
Epoch: 191      Training Loss: 0.420921         Test Loss: 0.454812
Epoch: 196      Training Loss: 0.420880         Test Loss: 0.454241
```

[ ]:
```
# import pickle
# with open("model1_5b.pkl", "wb") as f:
#     pickle.dump(model_task5b, f)
torch.save(model_task5b.state_dict(), 'model_task5b.pth')
```

[ ]:
```
#calculating accuracy
preds = []
labels=[]
for data,target in test_loader_5:
    data, target = data.to(device), target.to(device)
    output = model_task5b(data)
    preds += list(torch.argmax(output,dim=1).cpu())
    labels += list(target.cpu())
    # preds.append(torch.argmax(output,dim=1).cpu())
    # labels.append(target[0].cpu())
print("Accuracy: ",accuracy_score(preds,labels))
```

```
Accuracy:  0.7837
```

[ ]:
```
class RNN_task5c(nn.Module):
    def __init__(self,input_size,hidden_size,output_size):
        super(RNN_task5c, self).__init__()

        self.hidden_size = hidden_size
        self.input_size = input_size
#         self.embedding = nn.Embedding(input_size, hidden_size)
        self.lstm = nn.LSTM(input_size=input_size, hidden_size=hidden_size,␣
↪batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)
#         self.sigmoid = nn.Sigmoid()
        self.softmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
#         x = self.embedding(x)
        h0 = torch.zeros(1, x.size(0), self.lstm.hidden_size).to(x.device)
        c0 = torch.zeros(1, x.size(0), self.lstm.hidden_size).to(x.device)
        out, _ = self.lstm(x, (h0, c0))
        out = self.fc(out[:, -1, :])
        out = self.softmax(out)
        return out
```

[ ]:
```
model_task5c = RNN_task5c(300,10,2)
model_task5c
```

[ ]:
```
RNN_task5c(
  (lstm): LSTM(300, 10, batch_first=True)
  (fc): Linear(in_features=10, out_features=2, bias=True)
  (softmax): LogSoftmax(dim=1)
)
```

[ ]:
```
lr = 0.0001 # was 0.001 for the past 1500 epochs
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model_task5c.parameters(),lr)
```

[ ]:
```
count = 2
```

[ ]:
```
#1900
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print(f"Training on {device}")

# Move the model to the device
model_task5c.to(device)

epochs = 200
for e in range(epochs):
    train_loss = 0.0
    test_loss = 0.0

    model_task5c.train()  # prep model for training

    for data, target in train_loader_5:
        # Move data and target labels to the device
        data, target = data.to(device), target.to(device)

        optimizer.zero_grad()
        output = model_task5c(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        train_loss += loss.item() * data.size(0)

    model_task5c.eval()  # prep model for evaluation

    for data, target in test_loader_5:
        # Move data and target labels to the device
        data, target = data.to(device), target.to(device)
```

```
            output = model_task5c(data)
            loss = criterion(output, target)
            test_loss += loss.item() * data.size(0)

    train_loss = train_loss / len(train_loader_5.dataset)
    test_loss = test_loss / len(test_loader_5.dataset)

    if e % 5 == 0:
        print('Epoch: {} \tTraining Loss: {:.6f} \tTest Loss: {:.6f}'.format(
            e + 1,
            train_loss,
            test_loss
        ))
```

```
Training on cuda:0
Epoch: 1        Training Loss: 0.412370         Test Loss: 0.458881
Epoch: 6        Training Loss: 0.412301         Test Loss: 0.458963
Epoch: 11       Training Loss: 0.412250         Test Loss: 0.458905
Epoch: 16       Training Loss: 0.412281         Test Loss: 0.458917
Epoch: 21       Training Loss: 0.412252         Test Loss: 0.459147
Epoch: 26       Training Loss: 0.412236         Test Loss: 0.459024
Epoch: 31       Training Loss: 0.412165         Test Loss: 0.458975
Epoch: 36       Training Loss: 0.412203         Test Loss: 0.459495
Epoch: 41       Training Loss: 0.412188         Test Loss: 0.459074
Epoch: 46       Training Loss: 0.412148         Test Loss: 0.459029
Epoch: 51       Training Loss: 0.412112         Test Loss: 0.459152
Epoch: 56       Training Loss: 0.412150         Test Loss: 0.459204
Epoch: 61       Training Loss: 0.412101         Test Loss: 0.458967
Epoch: 66       Training Loss: 0.412097         Test Loss: 0.459279
Epoch: 71       Training Loss: 0.412042         Test Loss: 0.459022
Epoch: 76       Training Loss: 0.412061         Test Loss: 0.459170
Epoch: 81       Training Loss: 0.411979         Test Loss: 0.459350
Epoch: 86       Training Loss: 0.411970         Test Loss: 0.461255
Epoch: 91       Training Loss: 0.411951         Test Loss: 0.459057
Epoch: 96       Training Loss: 0.411928         Test Loss: 0.459012
Epoch: 101      Training Loss: 0.411932         Test Loss: 0.459191
Epoch: 106      Training Loss: 0.411865         Test Loss: 0.459408
Epoch: 111      Training Loss: 0.411903         Test Loss: 0.459250
Epoch: 116      Training Loss: 0.411945         Test Loss: 0.458920
Epoch: 121      Training Loss: 0.411865         Test Loss: 0.459446
Epoch: 126      Training Loss: 0.411861         Test Loss: 0.459080
Epoch: 131      Training Loss: 0.411835         Test Loss: 0.459192
Epoch: 136      Training Loss: 0.411822         Test Loss: 0.459604
Epoch: 141      Training Loss: 0.411808         Test Loss: 0.459138
Epoch: 146      Training Loss: 0.411779         Test Loss: 0.459093
Epoch: 151      Training Loss: 0.411753         Test Loss: 0.459182
Epoch: 156      Training Loss: 0.411764         Test Loss: 0.459163
Epoch: 161      Training Loss: 0.411638         Test Loss: 0.459385
```

```
Epoch: 166      Training Loss: 0.411729       Test Loss: 0.459324
Epoch: 171      Training Loss: 0.411734       Test Loss: 0.459047
Epoch: 176      Training Loss: 0.411703       Test Loss: 0.459108
Epoch: 181      Training Loss: 0.411625       Test Loss: 0.458997
Epoch: 186      Training Loss: 0.411594       Test Loss: 0.459070
Epoch: 191      Training Loss: 0.411612       Test Loss: 0.459010
Epoch: 196      Training Loss: 0.411611       Test Loss: 0.459280
```

[ ]:
```
# import pickle
# with open("model1_5c.pkl", "wb") as f:
#     pickle.dump(model_task5c, f)
torch.save(model_task5c.state_dict(), f'/content/drive/MyDrive/Shreya Data/
↪Shreya NLP/HW3/trained_models/model_task5c_{count}.pth')
count+=1
```

[ ]:
```
#calculating accuracy
preds = []
labels=[]
for data,target in test_loader_5:
    data, target = data.to(device), target.to(device)
    output = model_task5c(data)
    preds += list(torch.argmax(output,dim=1).cpu())
    labels += list(target.cpu())
    # preds.append(torch.argmax(output,dim=1).cpu())
    # labels.append(target[0].cpu())
print("Accuracy: ",accuracy_score(preds,labels))
```

```
Accuracy:  0.7805
```

Conclusion on training models:
Of the three models, the GRU performs the best, performing slightly better than the LSTM unit cell.
All the models were trained beyond overfitting till the point where the test loss started increasing with every epoch, following prof. Rostami's advice about deep learning models generalize better despite being overfitted on the training data
Of the three, a Simple RNN got the least accuracy of 75.2, followed by the LSTM unit cell, which got an accuracy of 78.1, and then the GRU with 78.4% accuracy on the test set.