CSCI544: Homework Assignment No 3
Nehal Muthukumar
1677301672

1. Dataset Generation

1) Sampled 60000 datapoints with equal distribution of classes.
2) Stored in a separate file 'data-sampled-60000.csv' and reused it.
3) Train-test split - 80-20.

2. Word Embedding

a) Loaded the pretrained word2vec-google-news-300 and checked sematic similarity for the following examples:

1. **King − Man + Woman = Queen**
2. **excellent ~ outstanding.**
3. **Apple ~ fruit**

b) Trained Word2Vec model using our own dataset (embedding size =300, window size = 13, minimum word count = 9) and checked the semantic similarity for examples as in part a).

Comparison and conclusion:

Eg 1: King − Man + Woman = Queen
Expectation -> Queen

Top 10 similar word to King-Man+Woman

word2vec-google-news-300

Word	Score
king	0.8449392318725586
queen	0.7300517559051514
monarch	0.645466148853302
princess	0.6156251430511475
crown_prince	0.5818676352500916
prince	0.5777117609977722
kings	0.5613663792610168
sultan	0.5376775860786438
Queen_Consort	0.5344247817993164
queens	0.5289887189865112

Trained Word2Vec model

Word	Score
woman	0.5592942833900452
African	0.5352551341056824
caramel	0.5162470936775208
blonde	0.4962203800678253
Asian	0.4925335645675659
American	0.4676726162433624
Cover	0.4488953948020935
tones	0.4302102327346802
brown	0.42785486578941345
gray	0.42594367265701294

Eg 2: excellent ~ outstanding

Similarity score between excellent and outstanding

Model	Similarity score
word2vec-google-news-300	0.5567
Trained Word2Vec model	0.7155

Eg 3: apple ~ fruit

Similarity score between apple and fruit

Model	Similarity score
word2vec-google-news-300	0.6410
Trained Word2Vec model	0.5349

In the first example (King-Man+Woman=Queen), the "word2vec-google-news-300" generated the expected output "Queen" as one of the top similar words, while our trained model did not. Additionally, the "word2vec-google-news-300" model seemed to generate more semantically similar words overall. For the second example (excellent~outstanding), the trained Word2Vec model performed better, with a higher similarity score between the two words compared to the "word2vec-google-news-300" model. In the third example (apple~fruit), the "word2vec-google-news-300" model performed better, with a higher similarity score between the two words compared to the trained Word2Vec model.

However, it's challenging to conclude which Word2Vec model is better at encoding semantic similarities between words based solely on these examples. Each model performed better for different examples, and the quality of the similarity scores can depend on various factors, including the training data's size and quality, the vector space's dimensionality, and the training model's specific parameters.

In general, Word2Vec models are effective at capturing semantic similarities between words, but the performance can vary depending on the training data and parameters. Pretrained Word2Vec models like "word2vec-google-news-300" are often trained on large amounts of high-quality text data and are better at capturing a wide range of semantic similarities between words. On the other hand, Word2Vec models trained on specific domains or datasets can capture domain-specific semantic relationships more effectively but may not generalize well to other domains. Therefore, the performance of a Word2Vec model in capturing semantic similarities between words will depend on the specific use case, training data, and parameters used to train the model.

3. Simple models

Accuracy comparison for the testing split

Model	Perceptron	SVM
Tf-idf	0.6598	0.6887
Pretrained w2v	0.5370	0.6351

Models from HW1 have been attached along in the submission folder. Results can be verified from there. File Name: hw1.py

After comparing the performance of the models trained using TF-IDF and trained Word2Vec features, we can conclude that the models trained with TF-IDF features performed better overall. This suggests that TF-IDF features are more effective in capturing the necessary information for this specific classification task.

However, it's important to note that the difference in accuracy between the two feature types is not significant, indicating that both feature types have the potential to be effective to some extent. While trained Word2Vec features did not perform as well in this specific task, they may be more effective in other classification tasks or domains.

4. Feedforward Neural Networks

1) Dropout regularization was experimented with, and it was found that a model with a dropout rate of 0 gave the best accuracy. Therefore, dropout was set to 0 in the final model.
2) The batch size was set to 32.
3) A validation approach was used, where the training data was split into a 90-10 ratio for training and validation, respectively.

Accuracy comparison for the testing split

Models	Epoch, learning rate	Accuracy
average Word2Vec	25, 0.005	63.90%
concatenate the first 10 Word2Vec	50, 0.001	55.57%

Based on the comparison of accuracy values, we can conclude that the feedforward neural network performed better than the simple models. The average Word2Vec model achieved an accuracy of 63.90%, which is higher

than the accuracy values obtained from the Perceptron and SVM models. However, the concatenate model with the first 10 Word2Vec models had a lower accuracy of only 55.57%, which is worse than the simple models.

In summary, the performance of the feedforward neural network was mixed compared to the simple models, with one model performing significantly better and one model performing worse. This suggests that the effectiveness of different models can vary depending on the specific features and parameters used.

5. Recurrent Neural Networks

batch size = 32
Epochs = 10
Learning rate = 0.001

Accuracy comparison for the testing split

Model	Accuracy
RNN cell	59.48%
GRU cell	64.42%
LSTM cell	65.07%

Comparing the accuracy values obtained with the RNN cell and the feedforward neural network models, we can conclude that the feedforward neural network models performed slightly better. The average Word2Vec feedforward neural network model achieved an accuracy of 63.90%, which is higher than the accuracy of the RNN cell model at 59.48%. However, the concatenate model with the first 10 Word2Vec models did not perform as well, with an accuracy of only 55.57%.

By comparing the accuracy values obtained with the GRU, LSTM, and simple RNN models, we can conclude that the more complex models, GRU and LSTM, outperformed the simple RNN model. The LSTM model achieved the highest accuracy of 65.07%, followed by the GRU model at 64.42%, while the simple RNN model achieved an accuracy of 59.48%. These results suggest that the added complexity of the GRU and LSTM models, with their ability to better handle long-term dependencies, improved their ability to classify the reviews correctly.

Running Environment:

Google Colab

Pytorch - CPU

Genism version should be 4.3.0. Word similarity function has been as per this version, so might throw error for the older version of genism package.
Other requirements are already present in Colab.

To Run the Code:

The data for this project has been sampled as mention in the Dataset Generation section and is provided in the "data-sampled-60000.csv" file, which is included in the submission folder. The result can be reproduced by directly start running the code from the "2. Word Embedding" section in the Jupyter notebook.

Also the accuracy might slightly differ than the one’s reported in the report on rerunning.

hw3

March 1, 2023

```
[ ]: !pip install gensim==4.3.0
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-
wheels/public/simple/
Collecting gensim==4.3.0
  Downloading
gensim-4.3.0-cp38-cp38-manylinux_2_12_x86_64.manylinux2010_x86_64.whl (24.1 MB)
                           24.1/24.1 MB
35.7 MB/s eta 0:00:00
Collecting FuzzyTM>=0.4.0
  Downloading FuzzyTM-2.0.5-py3-none-any.whl (29 kB)
Requirement already satisfied: smart-open>=1.8.1 in
/usr/local/lib/python3.8/dist-packages (from gensim==4.3.0) (6.3.0)
Requirement already satisfied: scipy>=1.7.0 in /usr/local/lib/python3.8/dist-
packages (from gensim==4.3.0) (1.7.3)
Requirement already satisfied: numpy>=1.18.5 in /usr/local/lib/python3.8/dist-
packages (from gensim==4.3.0) (1.22.4)
Requirement already satisfied: pandas in /usr/local/lib/python3.8/dist-packages
(from FuzzyTM>=0.4.0->gensim==4.3.0) (1.3.5)
Collecting pyfume
  Downloading pyFUME-0.2.25-py3-none-any.whl (67 kB)
                           67.1/67.1 KB
8.4 MB/s eta 0:00:00
Requirement already satisfied: python-dateutil>=2.7.3 in
/usr/local/lib/python3.8/dist-packages (from
pandas->FuzzyTM>=0.4.0->gensim==4.3.0) (2.8.2)
Requirement already satisfied: pytz>=2017.3 in /usr/local/lib/python3.8/dist-
packages (from pandas->FuzzyTM>=0.4.0->gensim==4.3.0) (2022.7.1)
Collecting simpful
  Downloading simpful-2.10.0-py3-none-any.whl (31 kB)
Collecting fst-pso
  Downloading fst-pso-1.8.1.tar.gz (18 kB)
  Preparing metadata (setup.py) … done
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.8/dist-
packages (from python-dateutil>=2.7.3->pandas->FuzzyTM>=0.4.0->gensim==4.3.0)
(1.15.0)
Collecting miniful
  Downloading miniful-0.0.6.tar.gz (2.8 kB)
```

```
  Preparing metadata (setup.py) … done
Requirement already satisfied: requests in /usr/local/lib/python3.8/dist-
packages (from simpful->pyfume->FuzzyTM>=0.4.0->gensim==4.3.0) (2.25.1)
Requirement already satisfied: chardet<5,>=3.0.2 in
/usr/local/lib/python3.8/dist-packages (from
requests->simpful->pyfume->FuzzyTM>=0.4.0->gensim==4.3.0) (4.0.0)
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.8/dist-
packages (from requests->simpful->pyfume->FuzzyTM>=0.4.0->gensim==4.3.0) (2.10)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.8/dist-packages (from
requests->simpful->pyfume->FuzzyTM>=0.4.0->gensim==4.3.0) (2022.12.7)
Requirement already satisfied: urllib3<1.27,>=1.21.1 in
/usr/local/lib/python3.8/dist-packages (from
requests->simpful->pyfume->FuzzyTM>=0.4.0->gensim==4.3.0) (1.26.14)
Building wheels for collected packages: fst-pso, miniful
  Building wheel for fst-pso (setup.py) … done
  Created wheel for fst-pso: filename=fst_pso-1.8.1-py3-none-any.whl size=20443
sha256=f3f2b0ada9c17e79cccd860f080dd147200fcc60249c2314bcf89efd984780da
  Stored in directory: /root/.cache/pip/wheels/6a/65/c4/d27eeee9ba3fc150a0dae150
519591103b9e0dbffde3ae77dc
  Building wheel for miniful (setup.py) … done
  Created wheel for miniful: filename=miniful-0.0.6-py3-none-any.whl size=3530
sha256=8e6d3dcb487fbf1e288f68d2ede5c2ac91dafa7525aec46391a338a6051bde3d
  Stored in directory: /root/.cache/pip/wheels/ba/d9/a0/ddd93af16d5855dd9bad4176
23e70948fdac119d1d34fb17c8
Successfully built fst-pso miniful
Installing collected packages: simpful, miniful, fst-pso, pyfume, FuzzyTM,
gensim
  Attempting uninstall: gensim
    Found existing installation: gensim 3.6.0
    Uninstalling gensim-3.6.0:
      Successfully uninstalled gensim-3.6.0
Successfully installed FuzzyTM-2.0.5 fst-pso-1.8.1 gensim-4.3.0 miniful-0.0.6
pyfume-0.2.25 simpful-2.10.0
```

[20]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
```

1 1. DATASET GENERATION

[ ]:
```
data = pd.read_csv('./data.tsv', sep='\t', on_bad_lines='skip',low_memory=False)
data
```

[ ]:
```
#Keep Reviews and Ratings
reviews=data[["review_body","star_rating"]].copy()
```

```
reviews = reviews[reviews["review_body"].notna()]
```

1.0.1 Relabelling Ratings

[ ]:
```
# We form three classes and select 20000 reviews randomly from each class.
reviews["star_rating"]=reviews["star_rating"].replace('1',1)
reviews["star_rating"]=reviews["star_rating"].replace(2,1)
reviews["star_rating"]=reviews["star_rating"].replace('2',1)
reviews["star_rating"]=reviews["star_rating"].replace('3',2)
reviews["star_rating"]=reviews["star_rating"].replace(4,3)
reviews["star_rating"]=reviews["star_rating"].replace('4',3)
reviews["star_rating"]=reviews["star_rating"].replace(5,3)
reviews["star_rating"]=reviews["star_rating"].replace('5',3)


# https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.sample.html
class1_df = reviews[reviews["star_rating"]==1]
sample1=class1_df.sample(n = 20000,random_state=47)
sample1 = sample1.reset_index(drop=True)
class2_df=reviews[reviews["star_rating"]==2]
sample2=class2_df.sample(n = 20000,random_state=47)
sample2 = sample2.reset_index(drop=True)
class3_df = reviews[reviews["star_rating"]==3]
sample3=class3_df.sample(n = 20000,random_state=47)
sample3 = sample3.reset_index(drop=True)

reviews_df=pd.concat([sample1,sample2,sample3],axis=0,ignore_index=True)
```

[ ]:
```
reviews_df
```

[ ]:

	review_body	star_rating
0	rancid smell.. Threw it away, smelled like it …	1
1	This flavor is gross What a nasty flavor!! The…	1
2	I was not a fan of this product. It … I was …	1
3	Not worth the investment I have been using the…	1
4	Wow I don't mean to be rude about it but wow! …	1
…	…	…
59995	Vi-Tae Shea Butter Soap This is my second purc…	3
59996	Four Stars Not working buy how they handled my…	3
59997	The smell is awesome and it leaves my hair so …	3
59998	Very Pretty Hair Really loved this hair. I wou…	3
59999	Great natural product! In past experiences I h…	3

[60000 rows x 2 columns]

[ ]:
```
reviews_df.to_csv('data-sampled-60000.csv', header=True, index=False)
```

[ ]:

2 2. WORD EMBEDDING

2.0.1 Train - Test Split

[21]:
```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
```

[ ]:
```
#red the sampled data here
review_data = pd.read_csv('/content/data-sampled-60000.csv')
```

[22]:
```
train_data, test_data = train_test_split(review_data, test_size=0.2,␣
↪random_state=42)
```

2.1 a) Loading pretrained W2V Model

[4]:
```
import gensim.downloader as api
wv = api.load('word2vec-google-news-300')
```

2.1.1 Example 1 - King - Man + Woman = Queen

[30]:
```
vec_king = wv['king']
vec_man = wv['man']
vec_woman = wv['woman']

isQueen=vec_king-vec_man+vec_woman

similar_words = wv.similar_by_vector(isQueen, topn=10)
print("Top 10 similar words to 'king-man+woman': ", similar_words)
```

```
Top 10 similar words to 'king-man+woman':  [('king', 0.8449392318725586),
('queen', 0.7300517559051514), ('monarch', 0.645466148853302), ('princess',
0.6156251430511475), ('crown_prince', 0.5818676352500916), ('prince',
0.5777117609977722), ('kings', 0.5613663792610168), ('sultan',
0.5376775860786438), ('Queen_Consort', 0.5344247817993164), ('queens',
0.5289887189865112)]
```

[ ]:
```
from prettytable import PrettyTable

# Define the table headers
table = PrettyTable()
table.field_names = ["Word", "Score"]

# Add the data to the table
```

```
for row in similar_words:
    table.add_row(row)

# Print the table
print(table)
```

```
+--------------+--------------------+
|     Word     |       Score        |
+--------------+--------------------+
|     king     | 0.8449392318725586 |
|    queen     | 0.7300517559051514 |
|   monarch    | 0.645466148853302  |
|   princess   | 0.6156251430511475 |
| crown_prince | 0.5818676352500916 |
|    prince    | 0.5777117609977722 |
|    kings     | 0.5613663792610168 |
|    sultan    | 0.5376775860786438 |
| Queen_Consort | 0.5344247817993164 |
|    queens    | 0.5289887189865112 |
+--------------+--------------------+
```

2.1.2 Example 2 - Excellent Outstanding

[ ]:
```
vec_excellent=wv['excellent']
similar_words = wv.similar_by_vector(vec_excellent, topn=10)
print("Top 10 similar words to 'Excellent': ", similar_words)
```

```
Top 10 similar words to 'Excellent':  [('excellent', 1.0), ('terrific',
0.7409726977348328), ('superb', 0.7062715888023376), ('exceptional',
0.681470513343811), ('fantastic', 0.6802847385406494), ('good',
0.6442928910255432), ('great', 0.6124600172042847), ('Excellent',
0.6091997623443604), ('impeccable', 0.5980967283248901), ('exemplary',
0.5959650278091431)]
```

[ ]:
```
# Calculate and print the semantic similarity of "excellent" and "outstanding"
excellent_outstanding_similarity = wv.similarity("excellent", "outstanding")
print("Similarity between 'excellent' and 'outstanding': ",␣
 ↪excellent_outstanding_similarity)
```

```
Similarity between 'excellent' and 'outstanding':  0.55674857
```

2.1.3 Example 3 - Apple Fruit

[ ]:
```
vec_Apple=wv['apple']
similar_words = wv.similar_by_vector(vec_Apple, topn=10)
print("Top 10 similar words to 'Apple': ", similar_words)
```

```
Top 10 similar words to 'Apple':  [('apple', 1.0), ('apples',
0.7203599214553833), ('pear', 0.6450697183609009), ('fruit',
0.6410146951675415), ('berry', 0.6302295327186584), ('pears',
0.613396167755127), ('strawberry', 0.6058261394500732), ('peach',
0.6025872826576233), ('potato', 0.5960935354232788), ('grape',
0.5935864448547363)]
```

```
[ ]: # Calculate and print the semantic similarity of "apple" and "fruit"
apple_fruit_similarity = wv.similarity("apple", "fruit")
print("Similarity between 'apple' and 'fruit': ", apple_fruit_similarity)
```

```
Similarity between 'apple' and 'fruit':  0.6410147
```

2.2 b) Training Word2Vec model using our own dataset

```
[ ]: import nltk
from nltk.tokenize import word_tokenize
nltk.download('punkt')
```

```
[nltk_data] Downloading package punkt to /root/nltk_data…
[nltk_data]   Package punkt is already up-to-date!
```

```
[ ]: True
```

2.2.1 creating training data

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

# Tokenize each review in the reviews_body column
review_tokens = review_data["review_body"].apply(word_tokenize)

# Convert the list of lists to a list of strings
review_strings = [" ".join(tokens) for tokens in review_tokens]

# Convert the list of strings to a list of lists
review_lists = [string.split() for string in review_strings]
```

2.2.2 Custom w2v traning model

```
[ ]: # Train a Word2Vec model on the tokenized sentences
model = Word2Vec(review_lists, vector_size=300, window=13, min_count=9)
```

2.2.3 Example 1 - King - Man + Woman = Queen

[ ]:
```
# Calculate and print the semantic similarity of "king-man+woman" and "queen"

king_vec = model.wv["King"]
man_vec = model.wv["man"]
woman_vec = model.wv["woman"]
queen_vec = king_vec - man_vec + woman_vec
```

[ ]:
```
queen_similarities = model.wv.most_similar(queen_vec, topn=10)
print("Top 10 similar words to 'king-man+woman': ", queen_similarities)
```

```
Top 10 similar words to 'king-man+woman':  [('woman', 0.5592942833900452),
('African', 0.5352551341056824), ('caramel', 0.5162470936775208), ('blonde',
0.4962203800678253), ('Asian', 0.4925335645675659), ('American',
0.4676726162433624), ('Cover', 0.4488953948020935), ('tones',
0.4302102327346802), ('brown', 0.42785486578941345), ('gray',
0.42594367265701294)]
```

[ ]:
```
table = PrettyTable()
table.field_names = ["Word", "Score"]

# Add the data to the table
for row in queen_similarities:
    table.add_row(row)

# Print the table
print(table)
```

```
+----------+---------------------+
|   Word   |        Score        |
+----------+---------------------+
|  woman   |  0.5592942833900452 |
| African  |  0.5352551341056824 |
| caramel  |  0.5162470936775208 |
|  blonde  |  0.4962203800678253 |
|  Asian   |  0.4925335645675659 |
| American |  0.4676726162433624 |
|  Cover   |  0.4488953948020935 |
|  tones   |  0.4302102327346802 |
|  brown   | 0.42785486578941345 |
|   gray   | 0.42594367265701294 |
+----------+---------------------+
```

2.2.4 Example 2 - Excellent Outstanding

```
[ ]: # Calculate and print the semantic similarity of "excellent" and "outstanding"
excellent_outstanding_similarity = model.wv.similarity("excellent",␣
 ↪"outstanding")
print("Similarity between 'excellent' and 'outstanding': ",␣
 ↪excellent_outstanding_similarity)
```

```
Similarity between 'excellent' and 'outstanding':  0.71553516
```

```
[ ]: vec_excellent=model.wv['excellent']
similar_words = model.wv.most_similar(vec_excellent, topn=10)
print("Top 10 similar words to 'Excellent': ", similar_words)
```

```
Top 10 similar words to 'Excellent':  [('excellent', 1.0), ('outstanding',
0.7155351042747498), ('exceptional', 0.7103866338729858), ('awesome',
0.6956540942192078), ('fantastic', 0.6939514875411987), ('incredible',
0.6319559216499329), ('amazing', 0.628711462020874), ('adequate',
0.6269515752792358), ('acceptable', 0.6211880445480347), ('attractive',
0.6057912111282349)]
```

```
[ ]: vec_Apple=model.wv['apple']
similar_words = model.wv.similar_by_vector(vec_Apple, topn=10)
print("Top 10 similar words to 'Apple': ", similar_words)

# Calculate and print the semantic similarity of "apple" and "fruit"
apple_fruit_similarity = model.wv.similarity("apple", "fruit")
print("Similarity between 'apple' and 'fruit': ", apple_fruit_similarity)
```

```
Top 10 similar words to 'Apple':  [('apple', 0.9999998807907104), ('cider',
0.84869784116745), ('vinegar', 0.726173460483551), ('baking',
0.6765809059143066), ('sea', 0.6538668274879456), ('salt', 0.642741858959198),
('bark', 0.6396810412406921), ('milk', 0.6391778588294983), ('eucalyptus',
0.6306906938552856), ('mixed', 0.6265164613723755)]
Similarity between 'apple' and 'fruit':  0.53493583
```

In the first example (King-Man+Woman=Queen), the "word2vec-google-news-300" generated the expected output "Queen" as one of the top similar words, while our trained model did not. Additionally, the "word2vec-google-news-300" model seemed to generate more semantically similar words overall. For the second example (excellent ~ outstanding), the trained Word2Vec model performed better, with a higher similarity score between the two words compared to the "word2vec-google-news-300" model. In the third example (apple ~ fruit), the "word2vec-google-news-300" model performed better, with a higher similarity score between the two words compared to the trained Word2Vec model.

However, it's challenging to conclude which Word2Vec model is better at encoding semantic similarities between words based solely on these examples. Each model performed better for different examples, and the quality of the similarity scores can depend on various factors, including the training data's size and quality, the vector space's dimensionality, and the training model's specific

parameters.

In general, Word2Vec models are effective at capturing semantic similarities between words, but the performance can vary depending on the training data and parameters. Pretrained Word2Vec models like "word2vec-google-news-300" are often trained on large amounts of high-quality text data and are better at capturing a wide range of semantic similarities between words. On the other hand, Word2Vec models trained on specific domains or datasets can capture domain-specific semantic relationships more effectively but may not generalize well to other domains. Therefore, the performance of a Word2Vec model in capturing semantic similarities between words will depend on the specific use case, training data, and parameters used to train the model.

3 3. Simple models

[31]:
```
import numpy as np
from gensim.models import KeyedVectors
from sklearn.linear_model import Perceptron
from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score
```

3.0.1 Averaging Word2Vec vectors for each review

[32]:
```
# Create input features and output labels for training data
X_train_w2v = np.zeros((len(train_data), 300)) # each row represents a review␣
 ↪with 300 dimensions
y_train = train_data['star_rating'].values

# Compute average Word2Vec vectors for each review in training data
for i, review in enumerate(train_data['review_body']):
    words = review.split()
    vectors = [wv[word] for word in words if word in wv]
    if vectors:
        X_train_w2v[i] = np.mean(vectors, axis=0)

# Create input features and output labels for testing data
X_test_w2v = np.zeros((len(test_data), 300)) # each row represents a review␣
 ↪with 300 dimensions
y_test = test_data['star_rating'].values

# Compute average Word2Vec vectors for each review in testing data
for i, review in enumerate(test_data['review_body']):
    words = review.split()
    vectors = [wv[word] for word in words if word in wv]
    if vectors:
        X_test_w2v[i] = np.mean(vectors, axis=0)
```

3.1 Single Perceptron

[33]:
```
# Train and evaluate perceptron model
perceptron = Perceptron()
perceptron.fit(X_train_w2v, y_train)
y_pred = perceptron.predict(X_test_w2v)
print("Perceptron Accuracy:", accuracy_score(y_test, y_pred))
```

```
Perceptron Accuracy: 0.537
```

3.2 SVM

[34]:
```
# Train and evaluate SVM model
svm = LinearSVC()
svm.fit(X_train_w2v, y_train)
y_pred = svm.predict(X_test_w2v)
print("SVM Accuracy:", accuracy_score(y_test, y_pred))
```

```
SVM Accuracy: 0.6350833333333333
```

After comparing the performance of the models trained using TF-IDF and trained Word2Vec features, we can conclude that the models trained with TF-IDF features performed better overall. This suggests that TF-IDF features are more effective in capturing the necessary information for this specific classification task.

However, it's important to note that the difference in accuracy between the two feature types is not significant, indicating that both feature types have the potential to be effective to some extent. While trained Word2Vec features did not perform as well in this specific task, they may be more effective in other classification tasks or domains.

[ ]:
```

```

4 4. Feedforward Neural Networks

4.1 a) the average Word2Vec vectors

4.1.1 loading dataset

[24]:
```
import numpy as np
from gensim.models import KeyedVectors
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader


class ReviewDataset(Dataset):
```

```
    def __init__(self, data):
        self.data = data
        self.X = np.zeros((len(data), 300)) # each row represents a review with␣
↪300 dimensions
        self.y = data['star_rating'].values - 1 # convert to 0-indexed labels
        for i, review in enumerate(data['review_body']):
            words = review.split()
            vectors = [wv[word] for word in words if word in wv]
            if vectors:
                self.X[i] = np.mean(vectors, axis=0)

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return torch.from_numpy(self.X[idx]), torch.tensor(self.y[idx])

# Create datasets and data loaders for training and testing
train_dataset = ReviewDataset(train_data)
test_dataset = ReviewDataset(test_data)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

4.1.2 FNN Model

[23]:
```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(300, 100)
        self.dropout1 = nn.Dropout(0)
        self.fc2 = nn.Linear(100, 10)
        self.dropout2 = nn.Dropout(0)
        self.fc3 = nn.Linear(10, 3)

    def forward(self, x):
        x = self.dropout1(torch.relu(self.fc1(x)))
        x = self.dropout2(torch.relu(self.fc2(x)))
        x = nn.functional.softmax(self.fc3(x), dim=1)
        return x

# Instantiate the network and the optimizer
net = Net()
optimizer = optim.Adam(net.parameters(), lr=0.005)
```

4.1.3 Training

[9]:
```
# Train the network
for epoch in range(25):
    running_loss = 0.0
    correct = 0
    total = 0
    for X, y in train_loader:
        optimizer.zero_grad()
        output = net(X.float())
        loss = nn.functional.cross_entropy(output, y)
        loss.backward()
        optimizer.step()

        # Calculate running loss and accuracy
        running_loss += loss.item()
        _, predicted = torch.max(output.data, 1)
        total += y.size(0)
        correct += (predicted == y).sum().item()

    # Print epoch loss and accuracy
    epoch_loss = running_loss / len(train_loader)
    epoch_acc = 100 * correct / total
    print(f'Epoch {epoch+1}, Loss: {epoch_loss:.4f}, Accuracy: {epoch_acc:.
 ↪2f}%')

# Evaluate the network on the test set
y_pred = []
y_true = []
with torch.no_grad():
    for X, y in test_loader:
        output = net(X.float())
        _, pred = torch.max(output, 1)
        y_pred.extend(pred.numpy())
        y_true.extend(y.numpy())
accuracy = accuracy_score(y_true, y_pred)
print(f'Testing accuracy: {accuracy*100:.2f}%')
```

```
Epoch 1, Loss: 0.9442, Accuracy: 57.95%
Epoch 2, Loss: 0.9100, Accuracy: 62.39%
Epoch 3, Loss: 0.9036, Accuracy: 63.06%
Epoch 4, Loss: 0.8967, Accuracy: 63.91%
Epoch 5, Loss: 0.8948, Accuracy: 64.36%
Epoch 6, Loss: 0.8911, Accuracy: 64.68%
Epoch 7, Loss: 0.8881, Accuracy: 65.05%
Epoch 8, Loss: 0.8878, Accuracy: 64.95%
Epoch 9, Loss: 0.8838, Accuracy: 65.45%
Epoch 10, Loss: 0.8812, Accuracy: 65.69%
```

```
Epoch 11, Loss: 0.8772, Accuracy: 66.22%
Epoch 12, Loss: 0.8767, Accuracy: 66.26%
Epoch 13, Loss: 0.8734, Accuracy: 66.61%
Epoch 14, Loss: 0.8712, Accuracy: 66.86%
Epoch 15, Loss: 0.8680, Accuracy: 67.24%
Epoch 16, Loss: 0.8662, Accuracy: 67.45%
Epoch 17, Loss: 0.8634, Accuracy: 67.71%
Epoch 18, Loss: 0.8633, Accuracy: 67.83%
Epoch 19, Loss: 0.8608, Accuracy: 68.10%
Epoch 20, Loss: 0.8583, Accuracy: 68.40%
Epoch 21, Loss: 0.8563, Accuracy: 68.55%
Epoch 22, Loss: 0.8539, Accuracy: 68.87%
Epoch 23, Loss: 0.8520, Accuracy: 69.12%
Epoch 24, Loss: 0.8515, Accuracy: 69.18%
Epoch 25, Loss: 0.8481, Accuracy: 69.51%
Testing accuracy: 63.90%
```

4.2 b) concatenate the first 10 Word2Vec vectors for each review

4.2.1 data loading

[10]:
```python
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from gensim.models import KeyedVectors
from sklearn.preprocessing import LabelEncoder
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader

class ReviewDataset(Dataset):
    def __init__(self, data):
        self.data = data
        self.X = np.zeros((len(data), 3000))
        self.y = data['star_rating'].values - 1 # convert to 0-indexed labels
        for i, review in enumerate(data['review_body']):
            words = review.split()
            vectors = [wv[word] for word in words if word in wv ][:10]
            if len(vectors) < 10:
                vectors += [np.zeros(300)] * (10 - len(vectors))
            self.X[i] = np.concatenate(vectors)

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return torch.from_numpy(self.X[idx]), torch.tensor(self.y[idx])
```

```
# Create datasets and data loaders for training and testing
train_dataset = ReviewDataset(train_data)
test_dataset = ReviewDataset(test_data)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

4.2.2 MLP Modelling

[11]:
```
# Define the dimensions of the input and output layers
input_dim = 3000
output_dim = 3

hidden_dim1 = 100
hidden_dim2 = 10

dropout_rate1 = dropout_rate2 = 0

# Define the  architecture
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim1)
        self.dropout1 = nn.Dropout(dropout_rate1)
        self.fc2 = nn.Linear(hidden_dim1, hidden_dim2)
        self.dropout2 = nn.Dropout(dropout_rate2)
        self.fc3 = nn.Linear(hidden_dim2, output_dim)

    def forward(self, x):
        x = self.dropout1(torch.relu(self.fc1(x)))
        x = self.dropout2(torch.relu(self.fc2(x)))
        x = nn.functional.softmax(self.fc3(x), dim=1)
        return x

# Instantiate the network and the optimizer
net = Net()
learning_rate = 0.001
optimizer = optim.Adam(net.parameters(), lr=learning_rate)
```

4.2.3 training

[12]:
```
# Train the network
for epoch in range(50):
    running_loss = 0.0
    correct = 0
    total = 0
```

```
    for X, y in train_loader:
        optimizer.zero_grad()
        output = net(X.float())
        loss = nn.functional.cross_entropy(output, y)
        loss.backward()
        optimizer.step()

        # Calculate running loss and accuracy
        running_loss += loss.item()
        _, predicted = torch.max(output.data, 1)
        total += y.size(0)
        correct += (predicted == y).sum().item()

    # Print epoch loss and accuracy
    epoch_loss = running_loss / len(train_loader)
    epoch_acc = 100 * correct / total
    print(f'Epoch {epoch+1}, Loss: {epoch_loss:.4f}, Accuracy: {epoch_acc:.
↪2f}%')

# Evaluate the network on the test set
y_pred = []
y_true = []
with torch.no_grad():
    for X, y in test_loader:
        output = net(X.float())
        _, pred = torch.max(output, 1)
        y_pred.extend(pred.numpy())
        y_true.extend(y.numpy())
accuracy = accuracy_score(y_true, y_pred)
print(f'Testing accuracy: {accuracy*100:.2f}%')
```

```
Epoch 1, Loss: 0.9868, Accuracy: 53.61%
Epoch 2, Loss: 0.9391, Accuracy: 59.25%
Epoch 3, Loss: 0.9102, Accuracy: 62.73%
Epoch 4, Loss: 0.8759, Accuracy: 66.76%
Epoch 5, Loss: 0.8404, Accuracy: 70.67%
Epoch 6, Loss: 0.8079, Accuracy: 74.20%
Epoch 7, Loss: 0.7844, Accuracy: 76.66%
Epoch 8, Loss: 0.7659, Accuracy: 78.49%
Epoch 9, Loss: 0.7539, Accuracy: 79.72%
Epoch 10, Loss: 0.7444, Accuracy: 80.60%
Epoch 11, Loss: 0.7387, Accuracy: 81.22%
Epoch 12, Loss: 0.7309, Accuracy: 82.08%
Epoch 13, Loss: 0.7278, Accuracy: 82.31%
Epoch 14, Loss: 0.7250, Accuracy: 82.60%
Epoch 15, Loss: 0.7210, Accuracy: 82.99%
Epoch 16, Loss: 0.7189, Accuracy: 83.17%
```

```
Epoch 17, Loss: 0.7156, Accuracy: 83.50%
Epoch 18, Loss: 0.7140, Accuracy: 83.63%
Epoch 19, Loss: 0.7106, Accuracy: 84.03%
Epoch 20, Loss: 0.7098, Accuracy: 84.09%
Epoch 21, Loss: 0.7072, Accuracy: 84.34%
Epoch 22, Loss: 0.7060, Accuracy: 84.45%
Epoch 23, Loss: 0.7041, Accuracy: 84.64%
Epoch 24, Loss: 0.7020, Accuracy: 84.88%
Epoch 25, Loss: 0.7018, Accuracy: 84.87%
Epoch 26, Loss: 0.6982, Accuracy: 85.25%
Epoch 27, Loss: 0.6976, Accuracy: 85.34%
Epoch 28, Loss: 0.6987, Accuracy: 85.16%
Epoch 29, Loss: 0.6971, Accuracy: 85.35%
Epoch 30, Loss: 0.6962, Accuracy: 85.42%
Epoch 31, Loss: 0.6942, Accuracy: 85.68%
Epoch 32, Loss: 0.6953, Accuracy: 85.55%
Epoch 33, Loss: 0.6932, Accuracy: 85.75%
Epoch 34, Loss: 0.6936, Accuracy: 85.72%
Epoch 35, Loss: 0.6922, Accuracy: 85.84%
Epoch 36, Loss: 0.6914, Accuracy: 85.91%
Epoch 37, Loss: 0.6901, Accuracy: 86.05%
Epoch 38, Loss: 0.6900, Accuracy: 86.06%
Epoch 39, Loss: 0.6887, Accuracy: 86.17%
Epoch 40, Loss: 0.6885, Accuracy: 86.25%
Epoch 41, Loss: 0.6875, Accuracy: 86.32%
Epoch 42, Loss: 0.6871, Accuracy: 86.37%
Epoch 43, Loss: 0.6857, Accuracy: 86.50%
Epoch 44, Loss: 0.6839, Accuracy: 86.69%
Epoch 45, Loss: 0.6844, Accuracy: 86.64%
Epoch 46, Loss: 0.6853, Accuracy: 86.55%
Epoch 47, Loss: 0.6846, Accuracy: 86.59%
Epoch 48, Loss: 0.6823, Accuracy: 86.82%
Epoch 49, Loss: 0.6830, Accuracy: 86.76%
Epoch 50, Loss: 0.6817, Accuracy: 86.91%
Testing accuracy: 55.57%
```

Based on the comparison of accuracy values, we can conclude that the feedforward neural network performed better than the simple models. The average Word2Vec model achieved an accuracy of 63.90%, which is higher than the accuracy values obtained from the Perceptron and SVM models. However, the concatenate model with the first 10 Word2Vec models had a lower accuracy of only 55.57%, which is worse than the simple models.

In summary, the performance of the feedforward neural network was mixed compared to the simple models, with one model performing significantly better and one model performing worse. This suggests that the effectiveness of different models can vary depending on the specific features and parameters used.

5 5. Recurrent Neural Networks

5.1 RNN Cell

[37]:
```
import numpy as np
import pandas as pd
import gensim.downloader as api
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from typing import List
```

[38]:
```
class ReviewDataset(Dataset):
    def __init__(self, data, max_len=20):
        self.data = data
        self.X = np.zeros((len(data), max_len, 300)) # each row represents a␣
 ↪review with 300 dimensions
        self.y = data['star_rating'].values - 1 # convert to 0-indexed labels
        self.max_len = max_len
        for i, review in enumerate(data['review_body']):
            words = review.split()
            words = [word for word in words if word in wv ][:max_len]
            for j, word in enumerate(words):
              self.X[i][j] = wv[word]

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        return torch.from_numpy(self.X[idx]), torch.tensor(self.y[idx])



# Create datasets and data loaders for training and testing
train_dataset = ReviewDataset(train_data)
test_dataset = ReviewDataset(test_data)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

class RNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNN, self).__init__()
        self.hidden_size = hidden_size
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)
```

```
    def forward(self, x):
        batch_size = x.size(0)
        h0 = torch.zeros(1, batch_size, self.hidden_size).to(x.device)
        out, hidden = self.rnn(x, h0)
        out = self.fc(hidden[-1])
        return out
```

[39]:
```
# Instantiate the network and the optimizer
net = RNN(input_size=300, hidden_size=20, output_size=3)
optimizer = optim.Adam(net.parameters(), lr=0.001)

# Train the network
for epoch in range(10):
    running_loss = 0.0
    correct_predictions = 0
    total_predictions = 0

    for X, y in train_loader:
        optimizer.zero_grad()
        output = net(X.float())
        loss = nn.functional.cross_entropy(output, y)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, pred = torch.max(output, 1)
        correct_predictions += (pred == y).sum().item()
        total_predictions += len(y)

    # Compute the accuracy and loss for this epoch
    epoch_loss = running_loss / len(train_loader)
    epoch_accuracy = 100 * correct_predictions / total_predictions

    # Print the epoch number, accuracy and loss
    print(f"Epoch {epoch + 1}, Loss: {epoch_loss:.4f}, Accuracy:␣
 ↪{epoch_accuracy:.2f}%")

# Evaluate the network on the test set
y_pred = []
y_true = []
with torch.no_grad():
    for X, y in test_loader:
        output = net(X.float())
        _, pred = torch.max(output, 1)
        y_pred.extend(pred.numpy())
        y_true.extend(y.numpy())
```

```
accuracy = accuracy_score(y_true, y_pred)
print(f'Testing accuracy: {accuracy*100:.2f}%')
```

```
Epoch 1, Loss: 1.0175, Accuracy: 46.37%
Epoch 2, Loss: 0.9432, Accuracy: 53.62%
Epoch 3, Loss: 0.9148, Accuracy: 55.78%
Epoch 4, Loss: 0.8976, Accuracy: 57.31%
Epoch 5, Loss: 0.8853, Accuracy: 58.22%
Epoch 6, Loss: 0.8772, Accuracy: 58.84%
Epoch 7, Loss: 0.8702, Accuracy: 59.15%
Epoch 8, Loss: 0.8655, Accuracy: 59.41%
Epoch 9, Loss: 0.8595, Accuracy: 59.84%
Epoch 10, Loss: 0.8531, Accuracy: 60.26%
Testing accuracy: 59.48%
```

5.2 b) Gated Recurrent Unit Cell

[36]:
```
import numpy as np
import pandas as pd
import gensim.downloader as api
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
from typing import List



class GRU(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(GRU, self).__init__()
        self.hidden_size = hidden_size
        self.gru = nn.GRU(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        batch_size = x.size(0)
        h0 = torch.zeros(1, batch_size, self.hidden_size).to(x.device)
        out, hidden = self.gru(x, h0)
        out = self.fc(hidden[-1])
        return out


# Instantiate the network and the optimizer
net = GRU(input_size=300, hidden_size=20, output_size=3)
```

```
optimizer = optim.Adam(net.parameters(), lr=0.001)

# Train the network
for epoch in range(10):
    running_loss = 0.0
    correct_predictions = 0
    total_predictions = 0

    for X, y in train_loader:
        optimizer.zero_grad()
        output = net(X.float())
        loss = nn.functional.cross_entropy(output, y)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, pred = torch.max(output, 1)
        correct_predictions += (pred == y).sum().item()
        total_predictions += len(y)

    # Compute the accuracy and loss for this epoch
    epoch_loss = running_loss / len(train_loader)
    epoch_accuracy = 100 * correct_predictions / total_predictions

    # Print the epoch number, accuracy and loss
    print(f"Epoch {epoch + 1}, Loss: {epoch_loss:.4f}, Accuracy:␣
 ↪{epoch_accuracy:.2f}%")

# Evaluate the network on the test set
y_pred = []
y_true = []
with torch.no_grad():
    for X, y in test_loader:
        output = net(X.float())
        _, pred = torch.max(output, 1)
        y_pred.extend(pred.numpy())
        y_true.extend(y.numpy())
accuracy = accuracy_score(y_true, y_pred)
print(f'Testing accuracy: {accuracy*100:.2f}%')
```

```
Epoch 1, Loss: 0.9353, Accuracy: 52.90%
Epoch 2, Loss: 0.8154, Accuracy: 62.26%
Epoch 3, Loss: 0.7836, Accuracy: 64.06%
Epoch 4, Loss: 0.7650, Accuracy: 65.20%
Epoch 5, Loss: 0.7488, Accuracy: 66.02%
Epoch 6, Loss: 0.7354, Accuracy: 66.84%
Epoch 7, Loss: 0.7247, Accuracy: 67.44%
```

```
Epoch 8, Loss: 0.7132, Accuracy: 68.13%
Epoch 9, Loss: 0.7031, Accuracy: 68.59%
Epoch 10, Loss: 0.6949, Accuracy: 69.24%
Testing accuracy: 64.42%
```

5.3 LSTM Cell

[35]:
```python
class LSTM(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(LSTM, self).__init__()
        self.hidden_size = hidden_size
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        batch_size = x.size(0)
        h0 = torch.zeros(1, batch_size, self.hidden_size).to(x.device)
        c0 = torch.zeros(1, batch_size, self.hidden_size).to(x.device)
        out, (hidden, cell) = self.lstm(x, (h0, c0))
        out = self.fc(hidden[-1])
        return out


# Instantiate the network and the optimizer
net = LSTM(input_size=300, hidden_size=20, output_size=3)
optimizer = optim.Adam(net.parameters(), lr=0.001)

# Train the network
for epoch in range(10):
    running_loss = 0.0
    correct_predictions = 0
    total_predictions = 0

    for X, y in train_loader:
        optimizer.zero_grad()
        output = net(X.float())
        loss = nn.functional.cross_entropy(output, y)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, pred = torch.max(output, 1)
        correct_predictions += (pred == y).sum().item()
        total_predictions += len(y)

    # Compute the accuracy and loss for this epoch
    epoch_loss = running_loss / len(train_loader)
```

```
    epoch_accuracy = 100 * correct_predictions / total_predictions

    # Print the epoch number, accuracy and loss
    print(f"Epoch {epoch + 1}, Loss: {epoch_loss:.4f}, Accuracy:␣
 ↪{epoch_accuracy:.2f}%")

# Evaluate the network on the test set
y_pred = []
y_true = []
with torch.no_grad():
    for X, y in test_loader:
        output = net(X.float())
        _, pred = torch.max(output, 1)
        y_pred.extend(pred.numpy())
        y_true.extend(y.numpy())
accuracy = accuracy_score(y_true, y_pred)
print(f'Testing accuracy: {accuracy*100:.2f}%')
```

```
Epoch 1, Loss: 0.9502, Accuracy: 51.50%
Epoch 2, Loss: 0.8431, Accuracy: 60.82%
Epoch 3, Loss: 0.8030, Accuracy: 63.15%
Epoch 4, Loss: 0.7800, Accuracy: 64.47%
Epoch 5, Loss: 0.7617, Accuracy: 65.37%
Epoch 6, Loss: 0.7450, Accuracy: 66.38%
Epoch 7, Loss: 0.7343, Accuracy: 66.84%
Epoch 8, Loss: 0.7182, Accuracy: 67.89%
Epoch 9, Loss: 0.7071, Accuracy: 68.48%
Epoch 10, Loss: 0.6966, Accuracy: 69.10%
Testing accuracy: 65.07%
```

Comparing the accuracy values obtained with the RNN cell and the feedforward neural network models, we can conclude that the feedforward neural network models performed slightly better. The average Word2Vec feedforward neural network model achieved an accuracy of 63.90%, which is higher than the accuracy of the RNN cell model at 59.48%. However, the concatenate model with the first 10 Word2Vec models did not perform as well, with an accuracy of only 55.57%.

By comparing the accuracy values obtained with the GRU, LSTM, and simple RNN models, we can conclude that the more complex models, GRU and LSTM, outperformed the simple RNN model. The LSTM model achieved the highest accuracy of 65.07%, followed by the GRU model at 64.42%, while the simple RNN model achieved an accuracy of 59.48%. These results suggest that the added complexity of the GRU and LSTM models, with their ability to better handle long-term dependencies, improved their ability to classify the reviews correctly.

[ ]: