CSCI544: Homework Assignment No3

Krushang Satani

satani@usc.edu

Dependencies Requires:

```
# Install the 'gensim' library if not already installed
%pip install gensim
%pip install torch
%pip install tabulate
# Import necessary libraries and modules
import pandas as pd
import numpy as np
import nltk
import re
from nltk.corpus import stopwords, wordnet
from nltk.tokenize import TweetTokenizer
from nltk.stem import WordNetLemmatizer
import contractions
from statistics import mean
from sklearn.metrics import accuracy_score
from tabulate import tabulate
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import torch
from torch.utils.data import DataLoader, Dataset
import torch.nn as nn
import torch.nn.functional as F
import gensim.downloader as api
from gensim.models import Word2Vec
from sklearn.linear_model import Perceptron
from sklearn.svm import LinearSVC
import warnings
warnings.filterwarnings("ignore")
```

```
Requirement already satisfied: gensim in c:\users\krusa\appdata\local\
packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\
localcache\local-packages\python311\site-packages (4.3.2)
Requirement already satisfied: numpy>=1.18.5 in c:\users\krusa\
appdata\local\packages\
pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-
packages\python311\site-packages (from gensim) (1.25.2)
Requirement already satisfied: scipy>=1.7.0 in c:\users\krusa\appdata\
```

```
local\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\
localcache\local-packages\python311\site-packages (from gensim)
(1.11.2)
Requirement already satisfied: smart-open>=1.8.1 in c:\users\krusa\
appdata\local\packages\
pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-
packages\python311\site-packages (from gensim) (6.4.0)
Note: you may need to restart the kernel to use updated packages.
Requirement already satisfied: torch in c:\users\krusa\appdata\local\
packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\
localcache\local-packages\python311\site-packages (2.1.0)
Requirement already satisfied: filelock in c:\users\krusa\appdata\
local\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\
localcache\local-packages\python311\site-packages (from torch)
(3.12.4)
Requirement already satisfied: typing-extensions in c:\users\krusa\
appdata\local\packages\
pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-
packages\python311\site-packages (from torch) (4.8.0)
Requirement already satisfied: sympy in c:\users\krusa\appdata\local\
packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\
localcache\local-packages\python311\site-packages (from torch) (1.12)
Requirement already satisfied: networkx in c:\users\krusa\appdata\
local\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\
localcache\local-packages\python311\site-packages (from torch) (3.1)
Requirement already satisfied: jinja2 in c:\users\krusa\appdata\local\
packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\
localcache\local-packages\python311\site-packages (from torch) (3.1.2)
Requirement already satisfied: fsspec in c:\users\krusa\appdata\local\
packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\
localcache\local-packages\python311\site-packages (from torch)
(2023.9.2)
Requirement already satisfied: MarkupSafe>=2.0 in c:\users\krusa\
appdata\local\packages\
pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\localcache\local-
packages\python311\site-packages (from jinja2->torch) (2.1.3)
Requirement already satisfied: mpmath>=0.19 in c:\users\krusa\appdata\
local\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\
localcache\local-packages\python311\site-packages (from sympy->torch)
(1.3.0)
Note: you may need to restart the kernel to use updated packages.
Requirement already satisfied: tabulate in c:\users\krusa\appdata\
local\packages\pythonsoftwarefoundation.python.3.11_qbz5n2kfra8p0\
localcache\local-packages\python311\site-packages (0.9.0)
Note: you may need to restart the kernel to use updated packages.
```

Question 1

Load Amazon reviews dataset

```
# data1 =
pd.read_csv('https://s3.amazonaws.com/amazon-reviews-pds/tsv/amazon_re
views_us_Beauty_v1_00.tsv.gz', compression='gzip', sep='\t',
on_bad_lines='skip')
# Define variables for data loading
file_name = 'amazon_reviews_us_Office_Products_v1_00.tsv'
separator = '\t'
on_bad_lines_option = 'skip'

# Load the data using defined variables
data = pd.read_csv(file_name, sep=separator,
on_bad_lines=on_bad_lines_option)

# Define variables for column selection and row limit
selected_columns = ["star_rating", "review_body"]
row_limit = 10000  # You can set this to the desired row limit

# Select the specified columns
data = data[selected_columns]

# Drop rows with missing values
data = data.dropna()

# Reset the index
data = data.reset_index(drop=True)

# If you want to limit the number of rows, you can uncomment and use
the following line:
# data = data.iloc[:row_limit, :]
```

Code Explanation

This code snippet performs data preprocessing on a DataFrame:

1. **Import Pandas**: We start by importing the Pandas library, which is used for data manipulation.
2. **Define Variables**:
   - `selected_columns`: A list of columns to select, such as "star_rating" and "review_body."
   - `row_limit`: An optional limit on the number of rows to retain.
3. **Select Specified Columns**:
   - Extract the columns listed in `selected_columns` from the DataFrame.
4. **Drop Rows with Missing Values**:

- Remove rows with missing values to ensure data quality.

5. **Reset the Index**:
   - Reindex the DataFrame to have a clean, continuous index.
6. **Optional Row Limiting**:
   - You can uncomment and use this line to limit the number of rows based on `row_limit`.

Built a balanced dataset of 60000 reviews and a validation set of 12000 reviews along with their ratings to create labels through random selection

```
# Define the columns to be used
star_rating_column = data['star_rating']

# Define the classifyRatingGroup function
def classifyRatingGroup(x):
    low_ratings = ['1', '2', 1, 2]
    high_ratings = ['4', '5', 4, 5]

    if x in low_ratings:
        return 1
    elif x in high_ratings:
        return 2
    else:
        return None  # Handle other cases if needed

# Apply the function to create the 'rating_group' column
data['rating_group'] = star_rating_column.apply(classifyRatingGroup)
```

Code Explanation

In this code snippet, we define a function and apply it to create a new column in the DataFrame:

1. **Define the Columns**:
   - `star_rating_column`: We extract the 'star_rating' column from the DataFrame `data`.
2. **Define the `classifyRatingGroup` Function**:
   - This function classifies the 'star_rating' values into two groups: low ratings and high ratings.
   - `low_ratings` includes values '1', '2', 1, and 2.
   - `high_ratings` includes values '4', '5', 4, and 5.
   - If the input `x` is in `low_ratings`, it returns 1. If it's in `high_ratings`, it returns 2. Otherwise, it returns `None`.
3. **Apply the Function to Create 'rating_group' Column**:
   - We use the `apply` method on `star_rating_column` to apply the `classifyRatingGroup` function to each value in the 'star_rating' column.

- The results are stored in a new 'rating_group' column in the DataFrame.

This code allows you to categorize 'star_rating' values into two groups: 1 for low ratings and 2 for high ratings, and stores this classification in a new column 'rating_group'.

```
# Define variables for grouping and sampling
group_column = 'rating_group'
group_size = 4000
sample_size = 20000
dry_run_fraction = 0.1

# Group the data by 'rating_group'
rgData = data.groupby(group_column)

# Initialize lists for valid and temp data
valid = []
temp = []

# Split and sample the data
for group, group_data in rgData:
    valid_data = group_data.iloc[:group_size, :]
    group_data.drop(group_data.index[:group_size], inplace=True)
    temp_data = group_data.sample(sample_size, random_state=0)

    valid.append(valid_data)
    temp.append(temp_data)

# Concatenate temporary data
fData = pd.concat(temp)

# Concatenate valid data
validData = pd.concat(valid)

# Shuffle the data
fData = fData.sample(frac=1)
validData = validData.sample(frac=1)

# # Perform a dry run by selecting only 10% of the data
# sample_size = len(data) * dry_run_fraction
# data = data.iloc[:int(sample_size), :]
```

Code Explanation

In this code, we are preparing and organizing the data into different groups and performing data sampling:

1. **Define Variables for Grouping and Sampling**:
   - `group_column`: The column by which you want to group the data, which appears to be 'rating_group.'
   - `group_size`: The desired size for each group.

- `sample_size`: The size to sample from each group.
- `dry_run_fraction`: A fraction for a dry run, which is currently commented out.

2. **Group the Data by 'rating_group'**:
   - We group the DataFrame 'data' by the 'rating_group' using the `groupby` method and store the resulting grouped data in `rgData`.
3. **Initialize Lists for Valid and Temporary Data**:
   - We initialize two lists, `valid` and `temp`, to store the valid and temporary data.
4. **Split and Sample the Data**:
   - For each group in `rgData`, we:
     - Select the first `group_size` rows and store them in `valid_data`.
     - Remove these rows from the group using `drop`.
     - Sample `sample_size` rows from the remaining data in the group and store them in `temp_data`.
     - Append `valid_data` to the `valid` list and `temp_data` to the `temp` list.
5. **Concatenate Temporary Data**:
   - We concatenate all the temporary data in `temp` to create a single DataFrame `fData`.
6. **Concatenate Valid Data**:
   - We concatenate all the valid data in `valid` to create a single DataFrame `validData`.
7. **Shuffle the Data**:
   - We shuffle the rows of both `fData` and `validData` using `sample(frac=1)` to randomize the data order.
8. **Optional Dry Run (Currently Commented Out)**:
   - If needed, you can uncomment the code block at the end to perform a dry run by selecting only 10% of the data based on the `dry_run_fraction`.

This code segments and samples the data into different groups, making it ready for further analysis or machine learning tasks.

```
# Define variables for reset_index parameters
reset_index_params = {
    'drop': True,
    'inplace': True
}

# Reset the index of validData
validData.reset_index(**reset_index_params)

# Define variables for reset_index parameters
reset_index_params = {
    'drop': True,
    'inplace': True
}
```

```
# Reset the index of fData
fData.reset_index(**reset_index_params)
```

Performing data cleaning on train and validation data.

```
import re
import contractions

# Define regular expressions for cleaning
html_tag_pattern = r'<.*>'
url_pattern = r'http[s]?://\S+'

# Define the expandContraction function
def expandContraction(text):
    expanded_words = [contractions.fix(w) for w in text.split()]
    return ' '.join(expanded_words)

# Define the cleanData function
def cleanData(review):
    review = re.sub(html_tag_pattern, '', review)      # Removes HTML
tags
    review = re.sub(url_pattern, '', review)           # Removes URL
tokens
    return review
```

Code Explanation

This code defines functions for text cleaning and contraction expansion:

1. **Import Libraries**:
   - Imports the `re` library for regular expressions and the `contractions` library for expanding contractions.
2. **Define Regular Expressions**:
   - `html_tag_pattern`: Removes HTML tags.
   - `url_pattern`: Removes URLs.
3. **`expandContraction` Function**:
   - Expands contractions in the input text.
4. **`cleanData` Function**:
   - Cleans text by removing HTML tags and URLs.

These functions are useful for text

```
import nltk
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords

# Create a lemmatizer and load stop words
lemmatizer = WordNetLemmatizer()
```

```
stop_words = set(stopwords.words("english"))

# Define the removeStopWords function
def removeStopWords(text):
    filtered_text = [w for w in text if w not in stop_words]
    return filtered_text

# Define the joinList function
def joinList(text):
    return ' '.join(text)

# Define the lemmatizeText function
def lemmatizeText(text):
    lemmatized_words = [lemmatizer.lemmatize(w) for w in text]
    return lemmatized_words

# Define the tokenizeText function
def tokenizeText(text):
    return nltk.word_tokenize(text)
```

Code Explanation

This code uses the NLTK library to perform text preprocessing, including stop word removal and lemmatization:

1. **Import Required Libraries**:
   - Imports the NLTK library for natural language processing tasks.
   - Specifically, it imports the WordNetLemmatizer and stopwords from the NLTK corpus.
2. **Create a Lemmatizer and Load Stop Words**:
   - Initializes a WordNetLemmatizer for lemmatization and loads English stop words.
3. **Define the `removeStopWords` Function**:
   - Removes stop words from a text by filtering out words that are in the set of stop words.
4. **Define the `joinList` Function**:
   - Joins a list of words back into a single string.
5. **Define the `lemmatizeText` Function**:
   - Lemmatizes words in a text.
6. **Define the `tokenizeText` Function**:
   - Tokenizes a text into individual words.

These functions help prepare text data for further natural language processing tasks by removing stop words, lemmatizing words, and tokenizing text.

```
# Define a sequence of text transformations
text_transformations = [
```

```
    lambda x: x.lower(),
    expandContraction,
    cleanData,
    tokenizeText,
    lemmatizeText
]

# Apply the transformations to the 'review_body' column
for transformation in text_transformations:
    fData['review_body'] = fData['review_body'].apply(transformation)

# Define a sequence of text transformations
text_transformations = [
    lambda x: x.lower(),
    expandContraction,
    cleanData,
    tokenizeText,
    lemmatizeText
]

# Apply the transformations to the 'review_body' column in validData
for transformation in text_transformations:
    validData['review_body'] =
validData['review_body'].apply(transformation)
```

Deleting reviews with zero length, i.e., empty reviews

```
delete = []

for i, rv in enumerate(fData['review_body']):
    if len(rv) == 0:
        delete.append(i)
# Remove the rows with empty 'review_body' based on the indices in the
'delete' list
fData = fData.drop(delete).reset_index(drop=True)


delete = []

for i, rv in enumerate(validData['review_body']):
    if len(rv) == 0:
        delete.append(i)

# Drop the rows with specified indices and reset the index
validData = validData.drop(index=delete).reset_index(drop=True)
```

Question 2

Training models using TF-IDF vectors

```
# Define column accessors
source_column = 'review_body'
target_column = 'review_body'  # You can use a different name if
needed

# Create a new DataFrame 'tfIdf'
tfIdf = pd.DataFrame()

# Apply the 'joinList' function to the specified columns
tfIdf[target_column] = fData[source_column].apply(joinList)

# Define column accessors
X_column = 'review_body'
y_column = 'rating_group'

# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    tfIdf[X_column],
    fData[y_column],
    test_size=0.2,
    random_state=0,
    stratify=fData[y_column]  # Keep 'stratify' as the last argument
)
```

Extracting TF-IDF feature

```
# Define the TF-IDF vectorizer parameters
ngram_range = (1, 3)
max_features = 10000

# Create the TF-IDF vectorizer
tfidf = TfidfVectorizer(ngram_range=ngram_range,
max_features=max_features)

# Fit the TF-IDF vectorizer on the 'review_body' values
tfidf.fit(tfIdf['review_body'].values)

# Transform the training and testing data
trainFeatures = tfidf.transform(X_train)
testFeatures = tfidf.transform(X_test)

# Convert the sparse matrices to DataFrames with feature names
trainFeatures_df = pd.DataFrame(trainFeatures.toarray(),
columns=tfidf.get_feature_names_out())
testFeatures_df = pd.DataFrame(testFeatures.toarray(),
columns=tfidf.get_feature_names_out())
```

Code Explanation

In this code, we set up a TF-IDF (Term Frequency-Inverse Document Frequency) vectorizer to convert text data into numerical feature vectors for machine learning:

1. **Define TF-IDF Vectorizer Parameters**:
   - `ngram_range`: Specifies the range of n-grams (in this case, from unigrams to trigrams).
   - `max_features`: Sets the maximum number of features (words) to consider.
2. **Create the TF-IDF Vectorizer**:
   - Instantiate the TF-IDF vectorizer with the specified parameters.
3. **Fit the TF-IDF Vectorizer**:
   - Fit the TF-IDF vectorizer on the 'review_body' values in your dataset.
4. **Transform Training and Testing Data**:
   - Transform the training and testing data into TF-IDF feature vectors using the fitted vectorizer.
5. **Convert to DataFrames**:
   - Convert the sparse matrices to DataFrames for easier manipulation and analysis. The resulting DataFrames represent the TF-IDF features with feature names as columns.

This code is essential for converting text data into numerical features, making it suitable for machine learning models that require numerical input.

Training a single perceptron model on TF-IDF features

```
# Define the random state for the Perceptron model
random_state = 66

# Create the Perceptron model with the specified random state
perceptronModel = Perceptron(random_state=random_state)

# Fit the Perceptron model on the training features and labels
perceptronModel.fit(trainFeatures, y_train)

# Make predictions using the Perceptron model
PPrediction = perceptronModel.predict(testFeatures)

# Calculate the accuracy score
percTf = accuracy_score(PPrediction, y_test)
```

Code Explanation

In this code, a Perceptron model is created and used for classification:

1. **Define the Random State**:
   - `random_state` is set to a specific value (in this case, 66) to ensure reproducibility of the model's behavior.
2. **Create the Perceptron Model**:

   - An instance of the Perceptron model is created with the specified random state.
3. **Fit the Perceptron Model**:
   - The Perceptron model is trained on the training features (`trainFeatures`) and their corresponding labels (`y_train`).
4. **Make Predictions**:
   - The trained model is used to make predictions on the test features (`testFeatures`), and the predictions are stored in `PPrediction`.
5. **Calculate the Accuracy Score**:
   - The accuracy of the Perceptron model is calculated by comparing its predictions (`PPrediction`) with the actual test labels (`y_test`). The result is stored in `percTf`.

This code demonstrates the typical workflow for training and evaluating a classification model, using a Perceptron as an example. The accuracy score provides a measure of the model's performance on the test data.

Training a SVM model on TF-IDF features

```
# Define the Linear Support Vector Classifier (LSVC) model with a specified hyperparameter C
C_value = 0.1
LSVC = LinearSVC(C=C_value)

# Fit the LSVC model on the training features and labels
LSVC.fit(trainFeatures, y_train)

# Make predictions using the LSVC model
LSVCPrediction = LSVC.predict(testFeatures)

# Calculate the accuracy score for the LSVC model
svmTf = accuracy_score(LSVCPrediction, y_test)
```

Code Explanation

In this code, a Linear Support Vector Classifier (LSVC) model is created and used for classification:

1. **Define the Hyperparameter C**:
   - `C_value` is set to a specific value (in this case, 0.1) to control the trade-off between maximizing the margin and minimizing the classification error.
2. **Create the LSVC Model**:
   - An instance of the Linear Support Vector Classifier (LSVC) model is created with the specified hyperparameter C.
3. **Fit the LSVC Model**:
   - The LSVC model is trained on the training features (`trainFeatures`) and their corresponding labels (`y_train`).
4. **Make Predictions**:

- The trained model is used to make predictions on the test features (`testFeatures`), and the predictions are stored in `LSVCPrediction`.

5. **Calculate the Accuracy Score**:
   - The accuracy of the LSVC model is calculated by comparing its predictions (`LSVCPrediction`) with the actual test labels (`y_test`). The result is stored in `svmTf`.

This code demonstrates the typical workflow for training and evaluating a classification model using a Linear Support Vector Classifier (LSVC). The accuracy score provides a measure of the model's performance on the test data, and the hyperparameter C controls the model's behavior.

Question 2 (a)

Loaded the pretrained “word2vec-google-news-300” Word2Vec model

```
# Define the Word2Vec model name
model_name = 'word2vec-google-news-300'

# Load the Word2Vec model using the variable
wv = api.load(model_name)
```

Code Explanation

In this code, you specify and load a pre-trained Word2Vec model:

1. **Define the Word2Vec Model Name**:
   - `model_name` is set to a specific pre-trained Word2Vec model's name, which in this case is 'word2vec-google-news-300.'
2. **Load the Word2Vec Model**:
   - The specified Word2Vec model is loaded using the `api.load()` function from Gensim, and the resulting model is stored in the variable `wv`.

This code allows you to access a pre-trained Word2Vec model, which is a powerful tool for working with word embeddings in natural language processing tasks. The 'word2vec-google-news-300' model is known for its extensive vocabulary and high-dimensional word vectors.

Checking semantic similarities of the generated vectors

```
# Define variables for the word vectors
positive_words = ['camera', 'photography']
negative_words = ['music']
topn_value = 1

# Find the most similar word using Word2Vec
similar_word = wv.most_similar(positive=positive_words,
negative=negative_words, topn=topn_value)
```

```
# Print the result
print(similar_word)
```

```
[('cameras', 0.6290432810783386)]
```

Code Explanation

In this code, you use the loaded Word2Vec model to find the most similar word to a set of positive words while considering negative words:

1. **Define Variables for Word Vectors**:
   - `positive_words`: A list of positive words that you want to find similarities to.
   - `negative_words`: A list of negative words that you want to exclude from the similarity search.
   - `topn_value`: The number of most similar words you want to retrieve.
2. **Find the Most Similar Word Using Word2Vec**:
   - The `most_similar()` function is called on the Word2Vec model (`wv`).
   - It takes the positive and negative word lists into account to find the most similar word(s) and returns the result.
3. **Print the Result**:
   - The result, which is the most similar word(s), is printed to the console using `print(similar_word)`.

This code demonstrates how to leverage the Word2Vec model to find words that are semantically similar to a set of positive words while considering words to be avoided (negative words). It's a useful technique for tasks such as word analogy and similarity calculations in natural language processing.

```
# Define variables for the word vectors
positive_words = ['smartphone', 'camera']
negative_words = ['battery']
topn_value = 1

# Find the most similar word using Word2Vec
similar_word = wv.most_similar(positive=positive_words,
negative=negative_words, topn=topn_value)

# Print the result
print(similar_word)
```

```
[('smartphones', 0.5551924705505371)]
```

Question 2 (b)

Training a Word2Vec model using preprocessed dataset, with embedding size to be 300, the window size to be 13 and minimum word count of 9.

```
# Define variables for Word2Vec model arguments
sentences = fData['review_body']

model_accuracy_table = []
vector_size = 300
window = 13
min_count = 9

# Create the Word2Vec model
model = Word2Vec(sentences=sentences, vector_size=vector_size,
window=window, min_count=min_count)
```

Code Explanation

In this code, you define variables for Word2Vec model arguments and create a Word2Vec model:

1. **Define Variables for Word2Vec Model Arguments**:
   - `sentences`: A list of sentences or text data from the 'review_body' column of `fData`.
   - `model_accuracy_table`: An empty list (likely intended for storing model evaluation results).
   - `vector_size`: The dimensionality of the word vectors to be created, set to 300 in this case.
   - `window`: The maximum distance between the current and predicted word within a sentence, set to 13.
   - `min_count`: Ignores all words with a total frequency lower than this value (set to 9).
2. **Create the Word2Vec Model**:
   - The Word2Vec model is instantiated using the specified arguments, such as `sentences`, `vector_size`, `window`, and `min_count`.

This code initializes a Word2Vec model for training word embeddings from the provided text data. Word embeddings are widely used in natural language processing tasks for capturing semantic relationships between words in a vector space.

Checking semantic similarities for the examples mentioned above on the new model

```
# Define variables for Word2Vec most_similar arguments
positive_words = ['camera', 'photography']
negative_words = ['music']
```

```
model_accuracy_table = []
topn_value = 1
# Find the most similar word using Word2Vec
similar_word = model.wv.most_similar(positive=positive_words,
negative=negative_words, topn=topn_value)

# Print the result
print(similar_word)
```

```
[('reproduction', 0.6471441388130188)]
```

Code Explanation

In this code, you specify variables for the Word2Vec `most_similar` method's arguments and use the Word2Vec model to find the most similar word:

1. **Define Variables for Word2Vec `most_similar` Arguments**:
   - `positive_words`: A list of positive words for which you want to find similar words.
   - `negative_words`: A list of negative words that you want to consider when searching for similar words.
   - `model_accuracy_table`: An empty list (likely intended for storing model evaluation results).
   - `topn_value`: The number of most similar words you want to retrieve.
2. **Find the Most Similar Word Using Word2Vec**:
   - The `most_similar()` function is called on the Word2Vec model (`model.wv`) to find the most similar word(s).
   - It considers the positive and negative words provided and retrieves the most similar word(s).
3. **Print the Result**:
   - The result, which is the most similar word(s), is printed to the console using `print(similar_word)`.

This code demonstrates how to leverage the Word2Vec model to find words that are semantically similar to a set of positive words while considering words to be avoided (negative words). It's a useful technique for word similarity calculations in natural language processing.

```
# Define variables for the word vectors
positive_words = ['smartphone', 'camera']
model_accuracy_table = []
negative_words = ['battery']
topn_value = 1

# Find the most similar word using Word2Vec
similar_word = wv.most_similar(positive=positive_words,
negative=negative_words, topn=topn_value)
```

```
# Print the result
print(similar_word)
```

```
[('smartphones', 0.5551924705505371)]
```

From comparing the vectors generated on my generated dataset and the pretrained model, we can conclude that:

Creating a list of vectors for all the words in each review

```
word2vec = []
word2vec_mean = []
for fd in fData['review_body']:
    vec = []
    model_accuracy_table = []
    for f in fd:
        mkdirectory = []
        try:
            varwv = wv[f]
            vec.append(varwv)
        except:
            continue
    if(len(vec) == 0):
        zerovars = np.zeros(300)
        vec.append(zerovars)
    word2vec.append(vec)
    mkdirectory.append(f)
    meanvar = np.mean(vec, axis=0)
    mkdirectory.append(fd)
    word2vec_mean.append(meanvar)

fData['word2vec'] = word2vec
fData['word2vec_mean'] = word2vec_mean

word2vec = []
word2vec_mean = []
for fd in validData['review_body']:
    vec = []
    model_accuracy_table = []
    for f in fd:
        mkdirectory = []
        try:
            vec.append(wv[f])
        except:
            continue
    if(len(vec) == 0):
        zerovars = np.zeros(300)
        vec.append(zerovars)
    word2vec.append(vec)
```

```
    mkdirectory.append(f)
    mkdirectory.append(fd)
    meanvar2 = np.mean(vec, axis=0)
    word2vec_mean.append(meanvar2)

validData['word2vec'] = word2vec
validData['word2vec_mean'] = word2vec_mean
```

Calculating mean of list of word2vec vectors for each review

```
word2vec_mean = [[] for i in range(300)]
word_directory = []
model_accuracy_table = []
selector = 'word2vec_mean'
for vec in fData[selector]:
    for i in range(300):
        tobeappend = vec[i]
        word2vec_mean[i].append(tobeappend)
word2vec_mean = np.array(word2vec_mean)
```

Code Explanation

In this code, you prepare data for word embeddings by creating a mean representation of word vectors:

1. **Initialize Variables**:
   - `word2vec_mean`: An empty list of 300 empty lists, each for one dimension in the word vectors.
   - `word_directory`: An empty list.
   - `model_accuracy_table`: Likely an empty list for storing model accuracy results.
   - `selector`: A variable with the value 'word2vec_mean,' which appears to be used for data selection.
2. **Iterate Over Data**:
   - For each item in the 'fData' DataFrame under the 'word2vec_mean' column (denoted by `vec`), you perform the following steps.
3. **Extract and Append Values**:
   - For each of the 300 dimensions in the word vectors, you extract the value (`tobeappend`) from `vec[i]` and append it to the corresponding list in `word2vec_mean`.
4. **Convert to NumPy Array**:
   - Finally, you convert `word2vec_mean` into a NumPy array.

This code aims to create a mean representation of word vectors from the 'word2vec_mean' column in the 'fData' DataFrame. It organizes the word vectors' values into an array for further processing or analysis.

```
#create feature vectors for train data
for i in range(300):
    selector = 'vec'+str(i)
    fData[selector] = word2vec_mean[i, :]

word2vec_mean = [[] for i in range(300)]
word_directory = []
selector = 'word2vec_mean'
model_accuracy_table = []
for vec in validData[selector]:
    for i in range(300):
        tobeappend = vec[i]
        word2vec_mean[i].append(tobeappend)
transferarr = word2vec_mean
word2vec_mean = np.array(transferarr)

#create feature vectors for train valid
model_accuracy_table = []
for i in range(300):
    selector = 'vec'+str(i)
    validData[selector] = word2vec_mean[i, :]

# Define the starting column index
start_column = 5

# Select columns from the starting index to the end
X_data = fData.iloc[:, start_column:]

start_column = 5

# Define the target column name
target_column = 'rating_group'
model_accuracy_table = []
# Select columns for validX
validX = validData.iloc[:, start_column:]

# Select the target column for validY
validY = validData[target_column]

X = X_data

# Define the target variable
y = fData['rating_group']
model_accuracy_table = []
# Define the test size
test_size = 0.2

# Define the random state
random_state = 0

# Split the data into train and test sets
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, stratify=y,
test_size=test_size, random_state=random_state)
```

Question 3

Training and evaluating a Single Perceptron model on Google pre-trained Word2Vec features

```
from sklearn.metrics import accuracy_score
from tabulate import tabulate

random_state = 66

x = X_train
y = y_train
# Create and fit the Perceptron model with Word2Vec vectors
perceptronModel = Perceptron(random_state=random_state)
perceptronModel.fit(x,y)

xt = X_test
yt = y_test
model_accuracy_table = []

PPrediction = perceptronModel.predict(xt)
percW2v = accuracy_score(PPrediction,yt)

# Define the results in a table
results = [
    ["Model", "Vector Type", "Accuracy"],
    ["Perceptron", "Word2Vec", f"{percW2v:.2%}"],
    ["Perceptron", "TF-IDF", f"{percTf:.2%}"]
]

# Print the table
print(tabulate(results, headers="firstrow", tablefmt="grid"))
```

```
+------------+---------------+------------+
| Model      | Vector Type   | Accuracy   |
+============+===============+============+
| Perceptron | Word2Vec      | 82.16%     |
+------------+---------------+------------+
| Perceptron | TF-IDF        | 87.88%     |
+------------+---------------+------------+
```

Code Explanation

In this code, you evaluate a Perceptron model's accuracy on two different vector types (Word2Vec and TF-IDF) and display the results in a tabular format:

1. **Import Required Libraries**:
   - Import the necessary libraries, including `accuracy_score` from `sklearn.metrics` and `tabulate` for creating tabulated results.
2. **Set Random State**:
   - Define `random_state` to ensure reproducibility of the model's behavior (set to 66).
3. **Prepare Data for Training**:
   - Assign training data (`X_train` and `y_train`) to variables `x` and `y`.
4. **Create and Fit Perceptron Model with Word2Vec Vectors**:
   - Initialize a Perceptron model with the specified random state and fit it with the Word2Vec vectors.
5. **Prepare Data for Testing**:
   - Assign testing data (`X_test` and `y_test`) to variables `xt` and `yt`.
6. **Calculate Accuracy and Store in `percW2v`**:
   - Use the trained model to make predictions on the test data (`xt`), calculate accuracy, and store it in `percW2v`.
7. **Define Results in a Table**:
   - Create a results table as a list of lists, where each inner list represents a row with columns: "Model," "Vector Type," and "Accuracy."
   - The table includes accuracy results for both Word2Vec (`percW2v`) and TF-IDF (`percTf`) models.
8. **Print the Table**:
   - Use the `tabulate` function to print the results in a grid format with headers.

This code assesses and presents the accuracy of a Perceptron model on two different vector types (Word2Vec and TF-IDF) and displays the results in a tabulated format.

Training and evaluating a Support Vector Classifier on Google pre-trained Word2Vec features

```
from sklearn.svm import LinearSVC
from sklearn.metrics import accuracy_score
from tabulate import tabulate

# Define the C parameter for LinearSVC
C_parameter = 0.1

x = X_train
y = y_train
# Create and fit the LinearSVC model with Word2Vec vectors
LSVC = LinearSVC(C=C_parameter)
LSVC.fit(x,y)

xt = X_test
yt = y_test
model_accuracy_table = []
```

```
LSVCPrediction = LSVC.predict(xt)
svmW2v = accuracy_score(LSVCPrediction, yt)

# Define the results in a table
results = [
    ["Model", "Vector Type", "Accuracy"],
    ["LinearSVC", "Word2Vec", f"{svmW2v:.2%}"],
    ["LinearSVC", "TF-IDF", f"{svmTf:.2%}"]
]

# Print the table
print(tabulate(results, headers="firstrow", tablefmt="grid"))
```

```
+-----------+---------------+------------+
| Model     | Vector Type   | Accuracy   |
+===========+===============+============+
| LinearSVC | Word2Vec      | 85.74%     |
+-----------+---------------+------------+
| LinearSVC | TF-IDF        | 91.26%     |
+-----------+---------------+------------+
```

Code Summary

This code performs the following tasks:

1. **Training and Evaluation**:
   - It trains a Linear Support Vector Classifier (LinearSVC) model with Word2Vec vectors on the training data and evaluates its accuracy on the testing data.
   - The accuracy of the LinearSVC model for Word2Vec vectors is stored in `svmW2v`.
2. **Tabulated Results**:
   - The code creates a table that compares accuracy between two vector types: Word2Vec and TF-IDF.
   - The table displays accuracy results for both LinearSVC models using Word2Vec (`svmW2v`) and TF-IDF (`svmTf`).
   - The `tabulate` library is used to format and print the table in a grid format.
3. **Clear Comparison**:
   - The resulting table allows for a straightforward comparison of the model's accuracy when using different vector representations, aiding in the assessment of model performance with various feature types.

From the above results it is evident that the Perceptron and SVC model with TF-IDF vectors performs better than Perceptron and SVC model with Word2Vec vectors

Question 4

```
batch_size = 64
def trainModel(model, n_epochs, dataLoader, validLoader, validY):
```

```
    lossFn = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=0.0025)
    testing_results = []
    bestAcc = -1
    bestModel = None
    training_results = []
    print("Epoch", "Training Loss", "Validation Accuracy (%)")
    for epoch in range(n_epochs):
        train_loss = 0.0
        model.train()
        testing_results.append({epoch:train_loss})

        for data, target in dataLoader:
            optimizer.zero_grad()
            output = model(data.float())
            loss = lossFn(output, target.long())
            testing_results_last = testing_results[-1]

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

        train_loss = train_loss / len(dataLoader.dataset)
        testing_results_last = testing_results[-1]

        validPred = predict(model, validLoader)
        validPred = convertTensor(validPred)
        validAcc = accuracy_score(validPred, validY - 1)
        testing_results_last = testing_results[-1]

        if bestAcc < validAcc:
            bestAcc = validAcc
            bestModel = model

        training_results.append([epoch + 1, train_loss, validAcc *
100])
        testing_results_last = testing_results[-1]
        print([epoch + 1, train_loss, validAcc * 100])
    # Print the results in a tabular format
    print(tabulate(training_results, headers=["Epoch", "Training
Loss", "Validation Accuracy (%)"], tablefmt="grid"))

    return bestModel
```

Code Summary

This code defines a training function `trainModel` for deep learning models, which includes the following key functionalities:

1. **Training Loop**:

- The code defines a training loop that iterates over a specified number of epochs (`n_epochs`).
- Within each epoch, it trains the model on the training data using the Adam optimizer and records the training loss.

2. **Validation and Accuracy**:
   - After each epoch, the code evaluates the model on a validation dataset, calculates the validation accuracy, and keeps track of the best-performing model based on validation accuracy.
3. **Results Tracking**:
   - The training and validation results, including training loss and validation accuracy, are collected and stored in lists.
   - The best model with the highest validation accuracy is retained.
4. **Tabulated Results**:
   - The training and validation results are printed in tabular format, displaying epoch, training loss, and validation accuracy.

Overall, this code provides a generic training loop for deep learning models, tracks performance, and identifies the best model based on validation accuracy.

```
def predict(model, dataloader):
    model.eval()
    prediction_list = []
    for i, batch in enumerate(dataloader):
        outputs = model(batch.float())
        dvars = outputs.data
        dashvarlabel , predicted = torch.max(dvars, 1)
        cpuvar = predicted.cpu()
        prediction_list.append(cpuvar)
    return prediction_list

def convertTensor(predictions):
    predList = []
    predictions = np.array(predictions)
    for i in range(len(predictions)):
        ithvar = predictions[i]
        appenvar = int(ithvar)
        predList.append(appenvar)
    return np.array(predList)
```

Used mean list of Word2Vec vectors as input feature to train Feedforward Neural Network

```
trainXPya = []
y = y_train
y_tr = np.array(y)
model_accuracy_table = []
batch_size = 64
lenx = len(X_train)
```

```
for i in range(lenx):
    alongbracs = X_train.iloc[i,:].values
    blongbracs = y_tr[i]-1
    longbracs = (alongbracs,blongbracs)
    trainXPya.append(longbracs)

testXPya = []
batch_size = 64
lenxt = len(X_test)
for i in range(lenxt):
    shortbracs = (X_test.iloc[i,:].values)
    testXPya.append(shortbracs)

validXPya = []
batch_size = 64
vallen = len(validX)
for i in range(vallen):
    valbracs = (validX.iloc[i,:].values)
    validXPya.append(valbracs)

DataLoader = torch.utils.data.DataLoader
batch_size = 64
train_loader_f_a = DataLoader(trainXPya, batch_size=batch_size)
test_loader_f_a = DataLoader(testXPya, batch_size=1)
valid_loader_f_a = DataLoader(validXPya, batch_size=1)
```

Question 4 (a)

Train a feedforward neural network for classification using Word2Vec features.

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(300, 100)
        self.fc2 = nn.Linear(100, 10)
        self.fc3 = nn.Linear(10, 2)
        self.dropout = nn.Dropout(0.3)

    def forward(self, x):
        return
self.fc3(self.dropout(F.relu(self.fc2(self.dropout(F.relu(self.fc1(x.v
iew(-1, 300)))))))))

ffn1 = Net()
ffn1 = trainModel(ffn1, 35, train_loader_f_a, valid_loader_f_a,
validY)
```

```
Epoch Training Loss Validation Accuracy (%)
[1, 0.4241669320319151, 87.92349043630455]
[2, 0.3662997441231901, 88.348543567946]
[3, 0.3576300657858799, 88.9236154519315]
[4, 0.3493424248126468, 89.0986373296662]
[5, 0.3413589740017822, 89.27365920740093]
[6, 0.3351279241362821, 89.08613576697087]
[7, 0.33021055007758066, 89.62370296287035]
[8, 0.3269200025964432, 88.94861857732216]
[9, 0.3249513157806229, 89.42367795974496]
[10, 0.3203148532611197, 89.52369046130767]
[11, 0.3131769011151952, 89.71121390173772]
[12, 0.31191348468739116, 89.28616077009626]
[13, 0.3093155787351165, 89.69871233904239]
[14, 0.30894326963722585, 89.81122640330041]
[15, 0.3030084405232583, 89.66120765095637]
[16, 0.2999785941960488, 89.17364670583822]
[17, 0.29688539132152186, 89.8737342167771]
[18, 0.2989917484134176, 89.59869983747969]
[19, 0.29475559473985896, 90.07375921990248]
[20, 0.28891319574978397, 90.12376547068384]
[21, 0.29124688967816975, 90.22377797224654]
[22, 0.2888598168494608, 89.56119514939368]
[23, 0.2846435389763848, 89.83622952869109]
[24, 0.28455070802583265, 89.3986748343543]
[25, 0.28154163486457123, 89.88623577947243]
[26, 0.27905750066720947, 89.74871858982372]
[27, 0.2760360351620374, 89.71121390173772]
[28, 0.27566181030115694, 89.54869358669833]
[29, 0.2706504278831505, 89.42367795974496]
[30, 0.2722688512918252, 89.6362045255657]
[31, 0.27051965108626624, 89.47368421052632]
[32, 0.2689622820306652, 89.57369671208902]
[33, 0.2675942427594837, 89.52369046130767]
[34, 0.26860724375954487, 90.19877484685586]
[35, 0.2632269734137132, 89.11113889236154]
```

Epoch	Training Loss	Validation Accuracy (%)
1	0.424167	87.9235
2	0.3663	88.3485
3	0.35763	88.9236
4	0.349342	89.0986
5	0.341359	89.2737
6	0.335128	89.0861

7	0.330211	89.6237
8	0.32692	88.9486
9	0.324951	89.4237
10	0.320315	89.5237
11	0.313177	89.7112
12	0.311913	89.2862
13	0.309316	89.6987
14	0.308943	89.8112
15	0.303008	89.6612
16	0.299979	89.1736
17	0.296885	89.8737
18	0.298992	89.5987
19	0.294756	90.0738
20	0.288913	90.1238
21	0.291247	90.2238
22	0.28886	89.5612
23	0.284644	89.8362
24	0.284551	89.3987
25	0.281542	89.8862
26	0.279058	89.7487
27	0.276036	89.7112
28	0.275662	89.5487
29	0.27065	89.4237
30	0.272269	89.6362
31	0.27052	89.4737

```
+---------+-----------------+---------------------------+
|      32 |        0.268962 |                   89.5737 |
+---------+-----------------+---------------------------+
|      33 |        0.267594 |                   89.5237 |
+---------+-----------------+---------------------------+
|      34 |        0.268607 |                   90.1988 |
+---------+-----------------+---------------------------+
|      35 |        0.263227 |                   89.1111 |
+---------+-----------------+---------------------------+
```

Code Summary

This code defines a neural network architecture using PyTorch and trains it using a training function:

- **Neural Network Architecture**:
  - A neural network model (`Net`) is defined with three fully connected layers (`nn.Linear`) and dropout layers (`nn.Dropout`).
  - The architecture comprises an input layer of 300 units, followed by two hidden layers with 100 and 10 units, and an output layer with 2 units.
  - ReLU activation functions are applied between layers to introduce non-linearity.
- **Training the Model**:
  - The neural network model (`ffn1`) is instantiated.
  - The `trainModel` function is called to train the model for 35 epochs using training and validation data provided by `train_loader_f_a` and `valid_loader_f_a`, respectively.
  - The validation labels are given as `validY`.

This code creates a feedforward neural network model and trains it using the specified data and number of epochs, enabling the model to learn patterns and make predictions based on the given architecture.

Feedforward Neural Network Architecture

Input (300) -> Hidden Layer 1 (100) -> Hidden Layer 2 (10) -> Output Layer (2)

```
# Print the results in a tabular format
print("+-----------------------------------------+")
print("| Model                         | Accuracy |")
print("+-----------------------------------------+")
print("| Feedforward Neural Network    | {:.2%}
|".format(accuracy_score(convertTensor(predict(ffn1,
test_loader_f_a)), y_test - 1)))
print("+-----------------------------------------+")
```

```
+-----------------------------------------+
| Model                          | Accuracy |
+-----------------------------------------+
| Feedforward Neural Network     | 86.62%   |
+-----------------------------------------+
```

Question 4 (b)

Used a concatenation of first 10 Word2Vec vectors for each review as the input feature to train Feedforward Neural Network

```
def trimData(data):
    howmuchdata = data[:10]
    x = np.array(howmuchdata).ravel()
    zerovarsa = np.zeros(3000-len(x))
    return np.append(x, zerovarsa)

# Selector variables for dataframes and column names
data_frame1 = fData
column_name = 'word2vec'
new_column_name = 'trimData1'

# Apply the function and assign the new column
data_frame1[new_column_name] =
data_frame1[column_name].apply(trimData)

data_frame2 = validData
data_frame2[new_column_name] =
data_frame2[column_name].apply(trimData)

# Selector variables for dataframes and column names
data_frame = validData
column_name_x = 'trimData1'
column_name_y = 'rating_group'

# Extract data from the validData dataframe
validX = data_frame[column_name_x]
validY = data_frame[column_name_y]

# Selector variables for data and column names
data_frame = fData
column_name_x = 'trimData1'
column_name_y = 'rating_group'

# Split the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    data_frame[column_name_x],
    data_frame[column_name_y],
```

```
    stratify=data_frame[column_name_y],
    test_size=0.2,
    random_state=0
)

# Selector variables for data and column names
data_frame_x = X_train
data_frame_y = y_train

# Initialize a list to store (X, y) pairs
trainXPyb = []

# Convert y_train to a numpy array
y_tr = np.array(data_frame_y)
lendata = len(data_frame_x)
# Iterate through the rows of X_train and y_train
for i in range(lendata):
    data_pointa = data_frame_x.iloc[i]
    data_pointb = y_tr[i] - 1
    data_point = (data_pointa,data_pointb)
    trainXPyb.append(data_point)

# Selector variable for X_test
data_frame_x = X_test

counter = 0
length = len(data_frame_x)

# Initialize a list to store (X) pairs
testXPyb = []

# Iterate through the rows of X_test
while counter < length:
    # Create a data point tuple with only X data
    data_point = (data_frame_x.iloc[counter])

    # Append the data point to the testXPyb list
    testXPyb.append(data_point)

    counter += 1

# Selector variable for validX
data_frame_x = validX

counter = 0
length = len(data_frame_x)

# Initialize a list to store (X) pairs
validXPyb = []

# Iterate through the rows of validX
```

```
while counter < length:
    # Create a data point tuple with only X data
    data_point = (data_frame_x.iloc[counter])

    # Append the data point to the validXPyb list
    validXPyb.append(data_point)

    counter += 1

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(3000, 100)
        self.fc2 = nn.Linear(100, 10)
        self.fc3 = nn.Linear(10, 2)
        self.dropout = nn.Dropout(0.3)


    def forward(self, x):
        return
self.fc3(self.dropout(F.relu(self.fc2(self.dropout(F.relu(self.fc1(x.v
iew(-1, 3000)))))))))
```

Neural Network Architecture

- **Input Layer** (3000 units)
  - Fully Connected Layer 1 (fc1) with 100 units
    - Fully Connected Layer 2 (fc2) with 10 units
      - Fully Connected Layer 3 (fc3) with 2 units (Output Layer)
    - Dropout Layer with a dropout probability of 0.3 after fc2
  - Dropout Layer with a dropout probability of 0.3 after fc1

```
train_loader_f_b = DataLoader(trainXPyb, batch_size=batch_size,
shuffle=True)
test_loader_f_b = DataLoader(testXPyb, batch_size=1, shuffle=False)
valid_loader_f_b = DataLoader(validXPyb, batch_size=1, shuffle=False)

# Create the Feedforward Neural Network model for the second dataset
ffn2 = Net()

# Define the number of training epochs
n_epochs = 35

# Train the model with the specified parameters
ffn2 = trainModel(model=ffn2, n_epochs=n_epochs,
dataLoader=train_loader_f_b, validLoader=valid_loader_f_b,
validY=validY)

Epoch Training Loss Validation Accuracy (%)
[1, 0.5118389506244948, 84.53556694586824]
```

```
[2, 0.4412943754329694, 84.86060757594699]
[3, 0.39823100839543574, 84.98562320290036]
[4, 0.3597638241754128, 84.898112264033]
[5, 0.31803188791318837, 84.4230528816102]
[6, 0.2828539933289916, 84.71058882360295]
[7, 0.2493569050970929, 84.57307163395424]
[8, 0.2242531967744472, 83.68546068258532]
[9, 0.2032320777590388, 85.1981497687211]
[10, 0.18395888448892267, 83.67295911989]
[11, 0.17159840905614565, 83.96049506188274]
[12, 0.1512538290596248, 84.54806850856357]
[13, 0.14499252934441825, 84.27303412926615]
[14, 0.13981576793504238, 83.87298412301539]
[15, 0.1308926170833946, 84.13551693961745]
[16, 0.12470263386670957, 83.47293411676459]
[17, 0.116079762506292, 83.77297162145268]
[18, 0.11448851488205807, 83.8479809976247]
[19, 0.11245586549171706, 83.47293411676459]
[20, 0.10455535003673204, 83.36042005250657]
[21, 0.10144983248321005, 83.93549193649206]
[22, 0.09782375141442237, 83.68546068258532]
[23, 0.09823731285259307, 83.69796224528065]
[24, 0.09257788604558079, 83.91048881110139]
[25, 0.09329088699204977, 84.16052006500813]
[26, 0.08735695420199865, 83.82297787223403]
[27, 0.08344046355418966, 84.22302787848481]
[28, 0.08485910725074958, 83.99799974996874]
[29, 0.0799428441187419, 83.91048881110139]
[30, 0.08433699429530697, 83.12289036129516]
[31, 0.08175383745132805, 84.26053256657082]
[32, 0.08185269706721722, 83.87298412301539]
[33, 0.07989802132353215, 83.48543567945993]
[34, 0.0785471548063161, 84.29803725465683]
[35, 0.06928053503484023, 84.17302162770346]
```

Epoch	Training Loss	Validation Accuracy (%)
1	0.511839	84.5356
2	0.441294	84.8606
3	0.398231	84.9856
4	0.359764	84.8981
5	0.318032	84.4231
6	0.282854	84.7106

7	0.249357	84.5731
8	0.224253	83.6855
9	0.203232	85.1981
10	0.183959	83.673
11	0.171598	83.9605
12	0.151254	84.5481
13	0.144993	84.273
14	0.139816	83.873
15	0.130893	84.1355
16	0.124703	83.4729
17	0.11608	83.773
18	0.114489	83.848
19	0.112456	83.4729
20	0.104555	83.3604
21	0.10145	83.9355
22	0.0978238	83.6855
23	0.0982373	83.698
24	0.0925779	83.9105
25	0.0932909	84.1605
26	0.087357	83.823
27	0.0834405	84.223
28	0.0848591	83.998
29	0.0799428	83.9105
30	0.084337	83.1229
31	0.0817538	84.2605

```
+---------+-----------------+--------------------------+
|      32 |       0.0818527 |                   83.873  |
+---------+-----------------+--------------------------+
|      33 |       0.079898  |                   83.4854 |
+---------+-----------------+--------------------------+
|      34 |       0.0785472 |                   84.298  |
+---------+-----------------+--------------------------+
|      35 |       0.0692805 |                   84.173  |
+---------+-----------------+--------------------------+
```

```
# Print the results in a tabular format
print("+-----------------------------------------+")
print("| Model                          | Accuracy |")
print("+-----------------------------------------+")
print("| Feedforward Neural Network     | {:.2%}
|".format(accuracy_score(convertTensor(predict(ffn2,test_loader_f_b)),
y_test-1)))
print("+-----------------------------------------+")
```

```
+-----------------------------------------+
| Model                          | Accuracy |
+-----------------------------------------+
| Feedforward Neural Network     | 78.86%   |
+-----------------------------------------+
```

When we use the concatenation of the first 10 word2vec vectors as an input for Feedforward Neural Network we conclude that the accuracy decreases w.r.t. to normal Feedforward Neural Network that takes mean of vectors of the entire review as input features. On the other hand, Feedforward Neural Network 4a perfomed better than simple models while simple model performed better than 4b version of Feedforward Neural Network .

Question 5

Created a list of word2vec vectors of maximum review of length 20. Longer reviews are truncated and shorter reviews are padded with null values.

```
def trimData(data, target_length=20, vector_length=300):
    trimmed_data = data[:target_length]

    while len(trimmed_data) < target_length:
        trimmed_data.append(np.zeros(vector_length))

    return np.array(trimmed_data)

# Define the column names for the 'word2vec' column and the new
'trimData2' column
```

```
column_name_word2vec = 'word2vec'
column_name_trimData2 = 'trimData2'

# Apply the 'trimData' function to the 'word2vec' column for the
training data
fData[column_name_trimData2] =
fData[column_name_word2vec].apply(trimData)

# Apply the 'trimData' function to the 'word2vec' column for the
validation data
validData[column_name_trimData2] =
validData[column_name_word2vec].apply(trimData)
```

Code Summary

In this code, a `trimData` function is defined and applied to trim and pad data in a specific column:

1. **`trimData` Function**:
   - The `trimData` function takes as input the data to be trimmed (`data`), a target length (`target_length`) set to 20 by default, and the vector length (`vector_length`) set to 300 by default.
   - It trims the data to the specified `target_length` and pads it with zeros if the length is less than the target.
   - The resulting trimmed and padded data is returned as a NumPy array.
2. **Applying `trimData` to Columns**:
   - The code defines column names for the original 'word2vec' column and the new 'trimData2' column.
   - The `trimData` function is applied to the 'word2vec' column for both training and validation data.
   - The results are stored in new columns 'trimData2' in the respective dataframes.

This code is used to prepare and standardize the length of data in the 'word2vec' column, ensuring that it matches the specified target length and vector length for further processing or analysis.

```
validX_column = 'trimData2'
validY_column = 'rating_group'

validX = validData[validX_column]
validY = validData[validY_column]

X_column = 'trimData2'
y_column = 'rating_group'
test_size = 0.2
random_state = 0

X_train, X_test, y_train, y_test = train_test_split(fData[X_column],
```

```
fData[y_column], stratify=fData[y_column], test_size=test_size,
random_state=random_state)

trainXRNb = [(X_train.iloc[i], y_tr[i] - 1) for i in
range(len(X_train))]
trainXRNb.append((np.zeros((20, 300)), 0))

testXRNb = [X_test.iloc[i] for i in range(len(X_test))]

validXRNb = [validX.iloc[i] for i in range(len(validX))]

train_loader_r_a = DataLoader(trainXRNb, batch_size=batch_size)
test_loader_r_a = DataLoader(testXRNb, batch_size=1)
valid_loader_r_a = DataLoader(validXRNb, batch_size=1)
```

Question 5 (a)

```
class RNN(torch.nn.Module):
    def __init__(self, input_size=300, hidden_size=20, num_classes=2):
        super(RNN, self).__init__()
        self.rnn = torch.nn.RNN(input_size, hidden_size,
batch_first=True, nonlinearity='relu', dropout=0.5)
        self.fc = torch.nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        hidden = self.rnn(x)[0][:, -1, :]
        out = self.fc(hidden)
        return out
```

Code Summary

In this code, a simple Recurrent Neural Network (RNN) model is defined using PyTorch:

1. **`RNN` Class**:
   - The `RNN` class is defined as a PyTorch module for building an RNN-based model.
   - It takes three main parameters:
     - `input_size`: The size of input features (set to 300 by default).
     - `hidden_size`: The size of the hidden state (set to 20 by default).
     - `num_classes`: The number of output classes (set to 2 by default).
2. **Model Architecture**:
   - The RNN layer is defined using `torch.nn.RNN`, with parameters for input size, hidden size, and batch processing. It uses ReLU as the nonlinearity and includes dropout.
   - The output of the RNN is passed through a fully connected layer (`torch.nn.Linear`) to produce the final prediction.
3. **Forward Method**:
   - The `forward` method defines the forward pass of the model.

- It processes the input sequence through the RNN layer and extracts the hidden state at the last time step.
- The hidden state is then passed through the fully connected layer to produce the output.

This code encapsulates the architecture and functionality of a simple RNN model for sequence data, making it suitable for various sequential data tasks, including text and time series analysis.

Train a Recurrent Neural Network for classification using Word2Vec features.

```
rnn = RNN()
rnn = trainModel(rnn, 35, train_loader_r_a, valid_loader_r_a, validY)
```

```
Epoch Training Loss Validation Accuracy (%)
[1, 0.5135570085453267, 86.27328416052006]
[2, 0.4119488867184275, 87.68596074509314]
[3, 0.38564642064012106, 87.54844355544444]
[4, 0.3610267941206945, 88.1485185648206]
[5, 0.3462897984394692, 88.81110138767346]
[6, 0.33551108617508096, 88.79859982497813]
[7, 0.3251311015737035, 89.46118264783098]
[8, 0.3184361038088642, 89.72371546443306]
[9, 0.31640140992699956, 89.66120765095637]
[10, 0.30967320727850467, 89.27365920740093]
[11, 0.30281227221674634, 89.84873109138643]
[12, 0.3038971653874423, 89.24865608201024]
[13, 0.29436802555973945, 89.73621702712839]
[14, 0.2964003442902067, 89.59869983747969]
[15, 0.29336514532905095, 89.8737342167771]
[16, 0.2902637459936124, 88.9611201400175]
[17, 0.2876169351355956, 88.63607950993875]
[18, 0.2853324257367789, 88.41105138142268]
[19, 0.2856174113787862, 88.37354669333666]
[20, 0.28416221622146887, 88.6235779472434]
[21, 0.2875613446581832, 88.39854981872735]
[22, 0.28271689394636673, 87.7734716839605]
[23, 0.28054318215908264, 88.73609201150143]
[24, 0.27954214650196196, 87.52344043005375]
[25, 0.28296060683660357, 85.87323415426928]
[26, 0.2875883153296462, 88.51106388298537]
[27, 0.2724619726068013, 89.31116389548693]
[28, 0.26813049061282423, 88.39854981872735]
[29, 0.3001503656060109, 88.39854981872735]
[30, 0.26979077155720155, 87.93599199899987]
[31, 0.2835181901414541, 88.97362170271283]
[32, 0.2629962815304191, 88.77359669958746]
[33, 0.2659459517474353, 88.32354044255531]
[34, 0.28688843088475185, 89.01112639079885]
[35, 0.2652985674789961, 89.21115139392424]
```

Epoch	Training Loss	Validation Accuracy (%)
1	0.513557	86.2733
2	0.411949	87.686
3	0.385646	87.5484
4	0.361027	88.1485
5	0.34629	88.8111
6	0.335511	88.7986
7	0.325131	89.4612
8	0.318436	89.7237
9	0.316401	89.6612
10	0.309673	89.2737
11	0.302812	89.8487
12	0.303897	89.2487
13	0.294368	89.7362
14	0.2964	89.5987
15	0.293365	89.8737
16	0.290264	88.9611
17	0.287617	88.6361
18	0.285332	88.4111
19	0.285617	88.3735
20	0.284162	88.6236
21	0.287561	88.3985
22	0.282717	87.7735
23	0.280543	88.7361

```
|      24 |        0.279542 |                    87.5234 |
+---------+-----------------+----------------------------+
|      25 |        0.282961 |                    85.8732 |
+---------+-----------------+----------------------------+
|      26 |        0.287588 |                    88.5111 |
+---------+-----------------+----------------------------+
|      27 |        0.272462 |                    89.3112 |
+---------+-----------------+----------------------------+
|      28 |        0.26813  |                    88.3985 |
+---------+-----------------+----------------------------+
|      29 |        0.30015  |                    88.3985 |
+---------+-----------------+----------------------------+
|      30 |        0.269791 |                    87.936  |
+---------+-----------------+----------------------------+
|      31 |        0.283518 |                    88.9736 |
+---------+-----------------+----------------------------+
|      32 |        0.262996 |                    88.7736 |
+---------+-----------------+----------------------------+
|      33 |        0.265946 |                    88.3235 |
+---------+-----------------+----------------------------+
|      34 |        0.286888 |                    89.0111 |
+---------+-----------------+----------------------------+
|      35 |        0.265299 |                    89.2112 |
+---------+-----------------+----------------------------+
```

```
# Print the results in a tabular format
print("+-----------------------------------------+")
print("| Model                         | Accuracy |")
print("+-----------------------------------------+")
print("| Feedforward Neural Network    | {:.2%}
|".format(accuracy_score(convertTensor(predict(rnn,test_loader_r_a)),
y_test-1)))
print("+-----------------------------------------+")
```

```
+-----------------------------------------+
| Model                         | Accuracy |
+-----------------------------------------+
| Feedforward Neural Network    | 85.06%   |
+-----------------------------------------+
```

By comparing accuracy values obtained with **Recurrent Neural Network** and **Feedforward Neural Network**, I conclude that the accuracy of Feedforward Neural Network when we pass the mean of entire review as word2vec vectors as input is more than the accuracy obtained by training Recurrent Neural Network model with a limit of twenty words of each review. However, when we use the concatenation of the first 10 word2vec vectors as an input for Feedforward Neural Network the accuracy decreases w.r.t. to normal Feedforward Neural Network and Recurrent Neural Network

Question 5 (b)

```
class GRU(torch.nn.Module):
    def __init__(self, input_size=300, hidden_size=20, num_classes=2):
        super(GRU, self).__init__()
        self.gru = torch.nn.GRU(input_size, hidden_size,
batch_first=True)
        self.fc = torch.nn.Linear(hidden_size, num_classes)

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

Code Summary

This code defines a Gated Recurrent Unit (GRU) model using PyTorch:

1. **`GRU` Class**:
   - The `GRU` class is implemented as a PyTorch module for constructing a GRU-based model.
   - It accepts three main parameters:
     - `input_size`: The size of input features (default is 300).
     - `hidden_size`: The size of the hidden state (default is 20).
     - `num_classes`: The number of output classes (default is 2).
2. **Model Architecture**:
   - The GRU layer is defined using `torch.nn.GRU`. It takes into account the input size, hidden size, and batch processing (set to batch_first=True by default).
   - The output of the GRU is then passed through a fully connected layer (`torch.nn.Linear`) to produce the final prediction.
3. **Forward Method**:
   - The `forward` method defines the forward pass of the model.
   - It initializes the initial hidden state (`h0`) and processes the input sequence through the GRU layer.
   - The final hidden state at the last time step is extracted and passed through the fully connected layer to generate the output.

This code encapsulates the architecture and functionality of a GRU-based model suitable for sequence data analysis. GRU is a variant of RNNs known for handling long sequences and addressing vanishing gradient problems.

Train a Gated Recurrent Unit for classification using Word2Vec features.

```
# Initialize the GRU model
gru = GRU()
```

```
# Train the GRU model using the trainModel function
num_epochs = 35
trained_gru = trainModel(model=gru,
                         n_epochs=num_epochs,
                         dataLoader=train_loader_r_a,
                         validLoader=valid_loader_r_a,
                         validY=validY)

Epoch Training Loss Validation Accuracy (%)
[1, 0.43117506636203146, 88.93611701462683]
[2, 0.33954398397453367, 90.32379047380923]
[3, 0.3117343652016655, 90.8238529816227]
[4, 0.29096352218225513, 91.01137642205276]
[5, 0.2740359654662549, 91.1988998624828]
[6, 0.25908370116433693, 91.0613826728341]
[7, 0.2451509689875294, 91.08638579822478]
[8, 0.23186195564870574, 90.96137017127141]
[9, 0.21918042970974821, 90.83635454431804]
[10, 0.20749981497774214, 90.21127640955119]
[11, 0.19843022503050517, 89.61120140017502]
[12, 0.19814680747143423, 89.47368421052632]
[13, 0.19427104639656076, 90.02375296912113]
[14, 0.18188071808346928, 90.47380922615326]
[15, 0.16918578418789598, 90.4488061007626]
[16, 0.15723740404337985, 90.22377797224654]
[17, 0.14940215345732652, 89.91123890486311]
[18, 0.14034194112405218, 90.06125765720715]
[19, 0.13792175115555613, 89.37367170896363]
[20, 0.1464698591938808, 89.26115764470559]
[21, 0.1294762290410194, 89.46118264783098]
[22, 0.11778487851507359, 89.58619827478435]
[23, 0.11354749134010997, 89.42367795974496]
[24, 0.11572293661890108, 89.43617952244031]
[25, 0.10886226836653663, 89.02362795349418]
[26, 0.10090604366917291, 89.13614201775222]
[27, 0.10103245865882161, 88.69858732341542]
[28, 0.1034784900861961, 89.57369671208902]
[29, 0.09765453941766616, 89.11113889236154]
[30, 0.09580517166735225, 88.9611201400175]
[31, 0.08760254135451635, 88.74859357419678]
[32, 0.08534627295604415, 88.1485185648206]
[33, 0.09101092164783121, 88.56107013376672]
[34, 0.08502213234055436, 88.49856232029005]
[35, 0.08057281898833027, 88.46105763220402]
+---------+-----------------+--------------------------+
|   Epoch |   Training Loss |   Validation Accuracy (%) |
+=========+=================+==========================+
|       1 |       0.431175  |                  88.9361 |
+---------+-----------------+--------------------------+
```

2	0.339544	90.3238
3	0.311734	90.8239
4	0.290964	91.0114
5	0.274036	91.1989
6	0.259084	91.0614
7	0.245151	91.0864
8	0.231862	90.9614
9	0.21918	90.8364
10	0.2075	90.2113
11	0.19843	89.6112
12	0.198147	89.4737
13	0.194271	90.0238
14	0.181881	90.4738
15	0.169186	90.4488
16	0.157237	90.2238
17	0.149402	89.9112
18	0.140342	90.0613
19	0.137922	89.3737
20	0.14647	89.2612
21	0.129476	89.4612
22	0.117785	89.5862
23	0.113547	89.4237
24	0.115723	89.4362
25	0.108862	89.0236
26	0.100906	89.1361

```
+---------+-----------------+--------------------------+
|      27 |       0.101032  |                  88.6986 |
+---------+-----------------+--------------------------+
|      28 |       0.103478  |                  89.5737 |
+---------+-----------------+--------------------------+
|      29 |       0.0976545 |                  89.1111 |
+---------+-----------------+--------------------------+
|      30 |       0.0958052 |                  88.9611 |
+---------+-----------------+--------------------------+
|      31 |       0.0876025 |                  88.7486 |
+---------+-----------------+--------------------------+
|      32 |       0.0853463 |                  88.1485 |
+---------+-----------------+--------------------------+
|      33 |       0.0910109 |                  88.5611 |
+---------+-----------------+--------------------------+
|      34 |       0.0850221 |                  88.4986 |
+---------+-----------------+--------------------------+
|      35 |       0.0805728 |                  88.4611 |
+---------+-----------------+--------------------------+
```

```
# Print the results in a tabular format
print("+-----------------------------------------+")
print("| Model                         | Accuracy |")
print("+-----------------------------------------+")
print("| Feedforward Neural Network    | {:.2%}
|".format(accuracy_score(convertTensor(predict(gru,test_loader_r_a)),
y_test-1)))
print("+-----------------------------------------+")
```

```
+-----------------------------------------+
| Model                         | Accuracy |
+-----------------------------------------+
| Feedforward Neural Network    | 84.30%   |
+-----------------------------------------+
```

By comparing accuracy values obtained with **Gated Recurrent Unit Cell** and **Feedforward Neural Network**, I conclude that the accuracy of Feedforward Neural Network when we pass the mean of entire review as word2vec vectors as input is more than the accuracy obtained by training Gated Recurrent Unit Cell model with a limit of twenty words of each review. However, when we use the concatenation of the first 10 word2vec vectors as an input for Feedforward Neural Network the accuracy decreases w.r.t. to normal Feedforward Neural Network and Gated Recurrent Unit Cell.

Question 5 (c)

```
import torch
import torch.nn as nn
```

```
class LSTM(nn.Module):
    def __init__(self, input_dim=300, hidden_dim=20, output_dim=2,
n_layers=1):
        super(LSTM, self).__init__()
        self.lstm = nn.LSTM(input_dim, hidden_dim, n_layers,
batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        out, _ = self.lstm(x)
        out = self.fc(out[:, -1, :])
        return out
```

Code Summary

This code defines an LSTM (Long Short-Term Memory) model using PyTorch:

1. **`LSTM` Class**:
   - The `LSTM` class is implemented as a PyTorch module for constructing an LSTM-based model.
   - It accepts several parameters:
     - `input_dim`: The size of input features (default is 300).
     - `hidden_dim`: The size of the hidden state (default is 20).
     - `output_dim`: The number of output classes (default is 2).
     - `n_layers`: The number of LSTM layers (default is 1).
2. **Model Architecture**:
   - The LSTM layer is defined using `torch.nn.LSTM`, considering the input dimension, hidden dimension, and the number of layers. It also sets `batch_first=True`.
   - The output of the LSTM is passed through a fully connected layer (`torch.nn.Linear`) to produce the final prediction.
3. **Forward Method**:
   - The `forward` method defines the forward pass of the model.
   - It processes the input sequence through the LSTM layer.
   - The final hidden state at the last time step is extracted and passed through the fully connected layer to generate the output.

This code encapsulates the architecture and functionality of an LSTM-based model suitable for sequence data analysis. LSTM networks are known for their ability to capture long-range dependencies and mitigate vanishing gradient problems, making them well-suited for various sequential data tasks.

Train a Long Short Term Memory model for classification using Word2Vec features.

```
# Create an instance of the LSTM model
lstm = LSTM()
```

```
# Define the number of epochs for training (e.g., 35)
num_epochs = 35

# Load your training data using the 'train_loader_r_a' DataLoader

# Load your validation data using 'valid_loader_r_a'

# Assuming you have your validation labels in 'validY'

# Train the LSTM model for 'num_epochs' using the 'trainModel'
function
lstm = trainModel(lstm, num_epochs, train_loader_r_a,
valid_loader_r_a, validY)
```

```
Epoch Training Loss Validation Accuracy (%)
[1, 0.45776728728099336, 87.51093886735842]
[2, 0.3636235611421167, 89.31116389548693]
[3, 0.332757977381208, 90.13626703337917]
[4, 0.3113558174047673, 90.47380922615326]
[5, 0.29366317659819513, 90.66133266658333]
[6, 0.2770147660824126, 90.84885610701338]
[7, 0.26196181296154786, 91.07388423552945]
[8, 0.24852616363998145, 90.86135766970871]
[9, 0.23505929296661507, 90.99887485935741]
[10, 0.22272519318451992, 90.59882485310663]
[11, 0.21295856309425862, 89.56119514939368]
[12, 0.20452607537952913, 89.12364045505689]
[13, 0.1989097405324929, 89.52369046130767]
[14, 0.18568896097524873, 89.7737217152144]
[15, 0.17987408902475377, 90.01125140642581]
[16, 0.17049766482841344, 89.88623577947243]
[17, 0.16174776708232558, 89.536192024003]
[18, 0.15493293674046252, 89.52369046130767]
[19, 0.14398858687048205, 89.18614826853357]
[20, 0.138104558903781, 88.44855606950868]
[21, 0.13690832515635842, 88.386048256032]
[22, 0.13331766609300172, 88.43605450681335]
[23, 0.11782589554786682, 88.68608576072009]
[24, 0.11260267934286414, 87.88598574821853]
[25, 0.1057090728579033, 87.84848106013253]
[26, 0.1035881754111272, 88.56107013376672]
[27, 0.10376894577599084, 88.97362170271283]
[28, 0.10318745064697413, 88.27353419177398]
[29, 0.0970950971508117, 89.16114514314289]
[30, 0.09040186661767978, 88.73609201150143]
[31, 0.08940422767508409, 89.08613576697087]
[32, 0.08936107645447057, 89.498687335917]
[33, 0.08480111526279824, 88.99862482810352]
[34, 0.07341774804300498, 89.01112639079885]
```

[35, 0.07661858850851291, 88.93611701462683]

Epoch	Training Loss	Validation Accuracy (%)
1	0.457767	87.5109
2	0.363624	89.3112
3	0.332758	90.1363
4	0.311356	90.4738
5	0.293663	90.6613
6	0.277015	90.8489
7	0.261962	91.0739
8	0.248526	90.8614
9	0.235059	90.9989
10	0.222725	90.5988
11	0.212959	89.5612
12	0.204526	89.1236
13	0.19891	89.5237
14	0.185689	89.7737
15	0.179874	90.0113
16	0.170498	89.8862
17	0.161748	89.5362
18	0.154933	89.5237
19	0.143989	89.1861
20	0.138105	88.4486
21	0.136908	88.386
22	0.133318	88.4361
23	0.117826	88.6861

```
+---------+-----------------+---------------------------+
|      24 |       0.112603  |                   87.886  |
+---------+-----------------+---------------------------+
|      25 |       0.105709  |                   87.8485 |
+---------+-----------------+---------------------------+
|      26 |       0.103588  |                   88.5611 |
+---------+-----------------+---------------------------+
|      27 |       0.103769  |                   88.9736 |
+---------+-----------------+---------------------------+
|      28 |       0.103187  |                   88.2735 |
+---------+-----------------+---------------------------+
|      29 |       0.0970951 |                   89.1611 |
+---------+-----------------+---------------------------+
|      30 |       0.0904019 |                   88.7361 |
+---------+-----------------+---------------------------+
|      31 |       0.0894042 |                   89.0861 |
+---------+-----------------+---------------------------+
|      32 |       0.0893611 |                   89.4987 |
+---------+-----------------+---------------------------+
|      33 |       0.0848011 |                   88.9986 |
+---------+-----------------+---------------------------+
|      34 |       0.0734177 |                   89.0111 |
+---------+-----------------+---------------------------+
|      35 |       0.0766186 |                   88.9361 |
+---------+-----------------+---------------------------+
```

```
# Print the results in a tabular format
print("+-----------------------------------------+")
print("| Model                           | Accuracy |")
print("+-----------------------------------------+")
print("| Feedforward Neural Network    | {:.2%}
|".format(accuracy_score(convertTensor(predict(lstm,test_loader_r_a)),
y_test-1)))
print("+-----------------------------------------+")
```

```
+-----------------------------------------+
| Model                           | Accuracy |
+-----------------------------------------+
| Feedforward Neural Network    | 84.60%   |
+-----------------------------------------+
```

Model Comparison and Conclusion

In this analysis, we compared the performance of various neural network models for a specific task and reached the following conclusions:

Feedforward Neural Network vs. LSTM

- When using the mean of the entire review text as Word2Vec vectors for input:
  - The Feedforward Neural Network outperformed the LSTM model in terms of accuracy.
- When using the concatenation of the first 10 Word2Vec vectors as input for the Feedforward Neural Network:
  - The accuracy decreased compared to the standard Feedforward Neural Network and the LSTM model.

Comparison of RNN, GRU, and LSTM

- In the comparison of three different recurrent neural network models (RNN, GRU, and LSTM):
  - The LSTM model achieved the highest accuracy.
  - The GRU model performed better than the RNN model.
  - The order of performance aligns with the complexity and number of parameters in these models.

Overall, the LSTM model demonstrated the best performance for the given task, emphasizing its capability to capture long-range dependencies and mitigate issues like vanishing gradients.

References

- Gensim Word2Vec Tutorial
- PyTorch MLP on MNIST
- PyTorch Character RNN Classification
- PyTorch Seq Classification with RNN
- GRU with PyTorch
- OpenAI ChatGPT

This analysis demonstrates the importance of choosing the right model architecture and input representation for a given task and highlights the significance of LSTM models for sequential data analysis.