CSCI 544 | Homework 3

Name: Sahil Mondal

USC ID: 5092826451

Python version: 3.9.12

```
import pandas as pd
import numpy as np
from sklearn.utils import resample
from sklearn.model_selection import train_test_split
from matplotlib import pyplot as plt
# Uncomment the below if gensim error appears
# !pip install gensim
from gensim.models import Word2Vec
import gensim.downloader as api
from sklearn.linear_model import Perceptron
from sklearn.metrics import accuracy_score
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import LinearSVC
```

1. Dataset Generation

We will use the Amazon reviews dataset used in HW1. Load the dataset and build a balanced dataset of 100K reviews along with their labels through random selection similar to HW1. You can store your dataset after generation and reuse it to reduce the computational load. For your experiments consider a 80%/20% training/testing split.

```
# Load the dataset
data = pd.read_csv('data.tsv', sep='\t', on_bad_lines='skip',
index_col=False)

# Make an independent copy of the DataFrame
df = data.copy()

df.dropna(inplace=True)
```

```
C:\Users\sahil\AppData\Local\Temp\ipykernel_33752\3581478179.py:2:
DtypeWarning: Columns (7) have mixed types. Specify dtype option on
import or set low_memory=False.
  data = pd.read_csv('data.tsv', sep='\t', on_bad_lines='skip',
index_col=False)
```

Explanation of above cell :

It is used to load a dataset from a TSV (Tab-Separated Values) file into a Pandas DataFrame. Here's what it does:

It reads the data from a TSV file named 'data.tsv' into a Pandas DataFrame, using a tab ('\t') as the separator for the values in the file.

The 'on_bad_lines' parameter is set to 'skip,' which means that if there are any lines in the file with formatting issues (bad lines), they will be skipped and not included in the DataFrame.

It creates a new DataFrame 'df' as an independent copy of the loaded data.

It removes rows with missing values (NaN) from 'df' using the dropna() method, effectively cleaning the dataset from rows containing missing data.

The result is 'df,' a cleaned DataFrame ready for further data analysis or processing.

```
# Keep only 'Reviews' and 'Ratings' columns
df = df[['star_rating', 'review_body']]

# Create class labels: 1 for ratings 1, 2, 3; 2 for ratings 4, 5
df['sentiment_class'] = df['star_rating'].apply(lambda x: 1 if x in
[1, 2, 3] else 2)

# Downsample to 50,000 reviews per class
class1_data = resample(df[df['sentiment_class'] == 1],
n_samples=50000, random_state=42)
class2_data = resample(df[df['sentiment_class'] == 2],
n_samples=50000, random_state=42)
compressed_data = pd.concat([class1_data, class2_data])

# full_data = pd.concat([compressed_data, compressed_data],
ignore_index=True)

# Split dataset into training (80%) and testing (20%)
train_df, test_df = train_test_split(compressed_data, test_size=0.2,
random_state=42)
```

Explanation of above cell :

The above code is used for preprocessing and splitting a dataset. Here's a breakdown:

It selects only the 'star_rating' (Ratings) and 'review_body' (Reviews) columns from the DataFrame 'df' and stores them in a new DataFrame 'df'.

It creates a new column 'sentiment_class' based on the 'star_rating' column. Ratings 1, 2, and 3 are assigned the class label 1, while ratings 4 and 5 are assigned the class label 2.

It downsamples the data to have 50,000 samples per class, ensuring a balanced dataset by randomly selecting 50,000 samples for each sentiment class (1 and 2). The result is stored in 'compressed_data'.

Optionally, there's a line that concatenates 'compressed_data' with itself, but it's commented out, so it doesn't affect the code.

It splits the 'compressed_data' into training and testing sets with an 80-20 split ratio, where 80% of the data is used for training and 20% for testing. The training set is stored in 'train_df', and the testing set is stored in 'test_df'. The random_state parameter ensures reproducibility in the data split. This is a common practice for machine learning tasks to evaluate model performance.

```
print(train_df.shape)
```

```
(80000, 3)
```

```
print(train_df.head(10))
```

```
        star_rating                                        review_body
\
807022            5  Great pens.  Work well for finer detail work. ...

1996924           2  The head bifurcates so wide that I have to hav...

451173            1  some of the pen are out of ink, and ink is all...

1474986           1  I bought mine at Wal-Mart about three months a...

1811952           5  My Brother HL-2240 printer loves these labels,...

1160648           5  I purchased this to hang a 100 lb heavy bag fr...

1843674           5  We use it at our radio station and it works gr...

125115            1  Had to return the product twice due to it not ...

791052            5  I need to pull the switch( white dot) inside t...

1920079           3  I had an older version of this phone and was r...


         sentiment_class
807022                 2
1996924                1
451173                 1
1474986                1
1811952                2
1160648                2
1843674                2
125115                 1
791052                 2
1920079                1
```

1. Word Embedding (25 points) In this part the of the assignment, you will generate Word2Vec features for the dataset you generated. You can use Gensim library for this purpose. A helpful tutorial is available in the following link: https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html

(a) (5 points) Load the pretrained “word2vec-google-news-300” Word2Vec model and learn how to extract word embeddings for your dataset. Try to check semantic similarities of the generated vectors using three examples of your own, e.g., King −M an + W oman = Queen or excellent ∼outstanding.

```
def similarity(w1, w2):
    return np.dot(w1, w2)/(np.linalg.norm(w1)*np.linalg.norm(w2))

word2vec_model = api.load('word2vec-google-news-300')

print('Similarity between (king - man + woman) and queen using the
pre-trained model:',similarity((word2vec_model['king'] -
word2vec_model['man'] + word2vec_model['woman']),
word2vec_model['queen']))

print('Similarity between outstanding and excellent using the pre-
trained model:',
similarity(word2vec_model['outstanding'],word2vec_model['excellent']))
```

```
Similarity between (king - man + woman) and queen using the pre-
trained model: 0.73005176
Similarity between outstanding and excellent using the pre-trained
model: 0.5567487
```

Explanation of the above code :

The code defines a similarity function using cosine similarity and utilizes the 'word2vec-google-news-300' pre-trained word embedding model. It calculates the similarity between the vectors resulting from arithmetic operations on word vectors ('king - man + woman' compared to 'queen') and between individual word vectors ('outstanding' and 'excellent') using the word embeddings.

```
print('Example 1:')

print('Similarity between (paris - france) and berlin using the pre-
trained model:',similarity((word2vec_model['paris'] -
word2vec_model['france']), word2vec_model['berlin']))
```

```
Example 1:
Similarity between (paris - france) and berlin using the pre-trained
model: 0.052674238
```

This example explores the relationship between countries and their capitals using word vectors. It calculates the similarity between the vector (country - capital) and the vector of a word representing a city. This can help you find cities that are similar in context to the relationship between a country and its capital.

```
print('Example 2:')

print('Similarity between (doctor - nurse) and engineer using the pre-
```

```
trained model:',similarity((word2vec_model['doctor'] -
word2vec_model['nurse']), word2vec_model['engineer']))
```

```
Example 2:
Similarity between (doctor - nurse) and engineer using the pre-trained
model: -0.045589708
```

This example explores the gender associations with different professions using word vectors. It calculates the similarity between the vector (male_profession - female_profession) and the vector of a word representing a profession. This can help you identify gender biases in word embeddings.

```
print('Example 3:')

print('Similarity between (king - man + woman) and mother using the
pre-trained model:',similarity((word2vec_model['king'] -
word2vec_model['man'] + word2vec_model['woman']),
word2vec_model['mother']))
```

```
Example 3:
Similarity between (king - man + woman) and mother using the pre-
trained model: 0.3571068
```

This example performs word analogies using word vectors. It finds words that are similar to the result of a word analogy operation such as "king - man + woman." This can help you discover word relationships based on semantic similarities.

(b) (20 points) Train a Word2Vec model using your own dataset. You will use these extracted features in the subsequent questions of this assignment. Set the embedding size to be 300 and the window size to be 13. You can also consider a minimum word count of 9. Check the semantic similarities for the same two examples in part (a). What do you conclude from comparing vectors generated by yourself and the pretrained model? Which of the Word2Vec models seems to encode semantic similarities between words better? For the rest of this assignment, use the pretrained “word2vec-google-news-300” Word2Vec features.

```
embedding = Word2Vec(train_df.review_body.apply(lambda x:x.split()),
vector_size=300,window=13,min_count=9)

try:
    print('Similarity between (king - man + woman) and queen using the
pre-trained model:',similarity((embedding.wv['king'] -
embedding.wv['man'] + embedding.wv['woman']), embedding.wv['queen']))
except:
    print("Key 'queen' not present")

print('Similarity between outstanding and excellent using the self-
trained model:',
similarity(embedding.wv['outstanding'],embedding.wv['excellent']))
```

```
Key 'queen' not present
Similarity between outstanding and excellent using the self-trained
model: 0.7948789
```

Analysis on Self-trained model vs word2vec-google-news-300 pre-trained model :

The pre-trained 'word2vec-google-news-300' is trained on a bigger corpus which makes it more robust to give out word embeddings for a much bigger vocabulary as compared to my own self-trained model which is just trained on the above dataset which makes the vocabulary poor and thus as seen in the first example above we do not find the embedding for the word 'queen'. Thus, the pre-trained model will perform better than the self-trained one.

Explanation of the above code :

In this code, a Word2Vec model is trained on the 'review_body' text data from the 'train_df' DataFrame, converting words into 300-dimensional word embeddings. It then calculates the cosine similarity between vectors resulting from arithmetic operations on word embeddings ('king - man + woman' compared to 'queen') using the self-trained model. If the word 'queen' isn't present in the vocabulary since the vocabulary is only from the reviews that are present, it prints a message indicating so. Additionally, it calculates the similarity between word vectors for 'outstanding' and 'excellent' using the self-trained model. This code demonstrates how to train a Word2Vec model and use it to compute word vector similarities, handling the case where a word might be missing in the vocabulary.

1. Simple models (20 points) Using the Google pre-trained Word2Vec features, train a single perceptron and an SVM model for the classification problem. For this purpose, use the average Word2Vec vectors for each review as the input feature (x =1/N∑Ni=1 Wi for a review with N words). Report your accuracy values on the testing split for these models similar to HW1, i.e., for each of perceptron and SVM models, report two accuracy values Word2Vec and TF-IDF features. What do you conclude from comparing performances for the models trained using the two different feature types (TF-IDF and your trained Word2Vec features) ?

```
# Define a function to compute the average Word2Vec vectors for a list
of reviews
def compute_average_word2vec_vectors(reviews, model):
    vectors = []
    for review in reviews:
        words = review.split()
        review_vectors = [model[word] for word in words if word in
model]
        if review_vectors:
            average_vector = np.mean(review_vectors, axis=0)
            vectors.append(average_vector)
        else:
            # If no words in the review are in the Word2Vec model
vocabulary, use a zero vector
            vectors.append(np.zeros(model.vector_size))
    return np.array(vectors)
```

```
# Compute average Word2Vec vectors for training and testing data
X_train_word2vec =
compute_average_word2vec_vectors(train_df['review_body'],
word2vec_model)
X_test_word2vec =
compute_average_word2vec_vectors(test_df['review_body'],
word2vec_model)

# TF-IDF features (for comparison)
tfidf_vectorizer = TfidfVectorizer()
X_train_tfidf =
tfidf_vectorizer.fit_transform(train_df['review_body'])
X_test_tfidf = tfidf_vectorizer.transform(test_df['review_body'])

# Labels
y_train = train_df['sentiment_class']
y_test = test_df['sentiment_class']

# Train Perceptron models
perceptron_word2vec = Perceptron(eta0=0.001, random_state=42,
max_iter=20000, early_stopping=True)
perceptron_word2vec.fit(X_train_word2vec, y_train)
perceptron_tfidf = Perceptron(eta0=0.001, random_state=42,
max_iter=20000, early_stopping=True)
perceptron_tfidf.fit(X_train_tfidf, y_train)

# Train SVM models
svm_word2vec = LinearSVC(random_state=42, max_iter=20000)
svm_word2vec.fit(X_train_word2vec, y_train)
svm_tfidf = LinearSVC(random_state=42, max_iter=20000)
svm_tfidf.fit(X_train_tfidf, y_train)

# Evaluate models on testing split
y_pred_perceptron_word2vec =
perceptron_word2vec.predict(X_test_word2vec)
y_pred_perceptron_tfidf = perceptron_tfidf.predict(X_test_tfidf)
y_pred_svm_word2vec = svm_word2vec.predict(X_test_word2vec)
y_pred_svm_tfidf = svm_tfidf.predict(X_test_tfidf)

# Calculate accuracy for each model and feature type
accuracy_perceptron_word2vec = accuracy_score(y_test,
y_pred_perceptron_word2vec)
accuracy_perceptron_tfidf = accuracy_score(y_test,
y_pred_perceptron_tfidf)
accuracy_svm_word2vec = accuracy_score(y_test, y_pred_svm_word2vec)
accuracy_svm_tfidf = accuracy_score(y_test, y_pred_svm_tfidf)

# Report the accuracy values
print("Accuracy (Perceptron, Word2Vec):",
accuracy_perceptron_word2vec)
```

```
print("Accuracy (Perceptron, TF-IDF):", accuracy_perceptron_tfidf)
print("Accuracy (SVM, Word2Vec):", accuracy_svm_word2vec)
print("Accuracy (SVM, TF-IDF):", accuracy_svm_tfidf)
```

```
Accuracy (Perceptron, Word2Vec): 0.76895
Accuracy (Perceptron, TF-IDF): 0.81175
Accuracy (SVM, Word2Vec): 0.79665
Accuracy (SVM, TF-IDF): 0.8602
```

Analysis :

1) Perceptron : We observe here that this model performs better using the TF-IDF features instead of the Word2Vec features. 2) SVM : We observe the similar trend here as well where the model performs better on the TF-IDF features.

Explanation of the above code :

A function compute_average_word2vec_vectors is defined to calculate the average Word2Vec vectors for a list of reviews using a Word2Vec model. It processes each review by splitting it into words, converting words into Word2Vec vectors, and then averaging them. If a word is not present in the Word2Vec model's vocabulary, it uses a zero vector.

The function is used to compute average Word2Vec vectors for both the training and testing data, storing them in X_train_word2vec and X_test_word2vec.

Additionally, TF-IDF features are computed using TfidfVectorizer and stored in X_train_tfidf and X_test_tfidf for comparison.

Labels for the data are stored in y_train and y_test.

Two machine learning models, Perceptron and Linear Support Vector Machine (SVM), are trained using Word2Vec vectors (X_train_word2vec) and TF-IDF features (X_train_tfidf) separately. The models are trained to predict the 'sentiment_class' labels.

The trained models are then evaluated on the testing data, and the predictions are stored in y_pred_perceptron_word2vec, y_pred_perceptron_tfidf, y_pred_svm_word2vec, and y_pred_svm_tfidf.

The code calculates the accuracy of each model by comparing the predicted labels with the actual labels and stores the accuracy scores in variables (accuracy_perceptron_word2vec, accuracy_perceptron_tfidf, accuracy_svm_word2vec, accuracy_svm_tfidf).

Finally, it prints the accuracy values for both Perceptron and SVM models, comparing Word2Vec and TF-IDF feature representations for sentiment classification.

1. Feedforward Neural Networks (25 points) Using the Word2Vec features, train a feedforward multilayer perceptron network for classification. Consider a network with two hidden layers, each with 50 and 5 nodes, respectively. You can use cross entropy loss and your own choice for other hyperparamters, e.g., nonlinearity, number of epochs, etc. Part of getting good results is to select suitable values for these hyperparamters. You can also refer to the following tutorial to familiarize yourself: https://www.kaggle.com/mishra1993/pytorch-multi-layer-perceptron-mnist. Although

the above tutorial is for image data but the concept of training an MLP is very similar to what we want to do.

(a) (10 points) To generate the input features, use the average Word2Vec vectors similar to the "Simple models" section and train the neural network. Report accuracy values on the testing split for your MLP.

```
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.metrics import accuracy_score
import numpy as np

# Convert the Word2Vec features to PyTorch tensors
X_train_word2vec_tensor = torch.tensor(X_train_word2vec,
dtype=torch.float32)
X_test_word2vec_tensor = torch.tensor(X_test_word2vec,
dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values - 1, dtype=torch.long)  #
Adjust labels to start from 0
y_test_tensor = torch.tensor(y_test.values - 1, dtype=torch.long)  #
Adjust labels to start from 0

# Define the MLP model
class MLPModel(nn.Module):
    def __init__(self, input_dim, hidden_dim1, hidden_dim2,
output_dim):
        super(MLPModel, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden_dim1)
        self.fc2 = nn.Linear(hidden_dim1, hidden_dim2)
        self.fc3 = nn.Linear(hidden_dim2, output_dim)
        self.relu = nn.ReLU()
        self.softmax = nn.Softmax(dim=1)

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

# Initialize the model
input_dim = X_train_word2vec_tensor.shape[1]
hidden_dim1 = 50
hidden_dim2 = 5
output_dim = 2  # Two classes (sentiment_class 1 and 2)
model = MLPModel(input_dim, hidden_dim1, hidden_dim2, output_dim)

# Define loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)  # You can adjust
```

```
the learning rate

# Training loop
epochs = 1000  # You can adjust the number of epochs
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    y_pred = model(X_train_word2vec_tensor)
    loss = criterion(y_pred, y_train_tensor)
    loss.backward()
    optimizer.step()

    if (epoch+1) % 100 == 0:
        model.eval()
        with torch.no_grad():
            test_logits = model(X_test_word2vec_tensor)
            y_pred = torch.softmax(test_logits, dim=1).argmax(dim=1)
            test_loss = criterion(test_logits, y_test_tensor)
        acc = accuracy_score(y_test_tensor.numpy(), y_pred.numpy())
        print('epoch', epoch+1, 'loss', test_loss.item(), 'test
accuracy', acc)
```

```
epoch 100 loss 0.5721852779388428 test accuracy 0.77165
epoch 200 loss 0.5149409770965576 test accuracy 0.7936
epoch 300 loss 0.5035223364830017 test accuracy 0.80225
epoch 400 loss 0.49898505210876465 test accuracy 0.8047
epoch 500 loss 0.4964416027069092 test accuracy 0.808
epoch 600 loss 0.49414384365081787 test accuracy 0.80945
epoch 700 loss 0.49223682284355164 test accuracy 0.81185
epoch 800 loss 0.49107417464256287 test accuracy 0.81365
epoch 900 loss 0.49020877480506897 test accuracy 0.8146
epoch 1000 loss 0.48938462138175964 test accuracy 0.81535
```

Explanation of the above code :

The code begins by importing necessary libraries, including PyTorch for neural network modeling and scikit-learn for accuracy score calculation.

Word2Vec features and labels are converted into PyTorch tensors for training and testing data.

A Multi-Layer Perceptron (MLP) neural network model is defined using the nn.Module class. It consists of three fully connected layers with ReLU activation functions between them and a softmax activation function in the final layer.

The model is initialized with the specified input dimensions, hidden layer dimensions, and output dimensions (2 for sentiment classes 1 and 2).

The training loop runs for a specified number of epochs, where the model is trained on the training data, and the validation loss and accuracy on the testing data are calculated.

The training loop prints epoch information, test loss, and accuracy for every 100 epoch.

(b) (15 points) To generate the input features, concatenate the first 10 Word2Vec vectors for each review as the input feature (x = [W T1 , ..., W T10]) and train the neural network. Report the accuracy value on the testing split for your MLP model. What do you conclude by comparing accuracy values you obtain with those obtained in the "'Simple Models" section.

```
# Create a function to generate concatenated Word2Vec features
def generate_concatenated_word2vec(reviews, model, max_seq_length=10,
embedding_dim=300):
    concatenated_features = []
    for review in reviews:
        words = review.split()
        feature = np.zeros((max_seq_length * embedding_dim,))
        for i in range(min(len(words), max_seq_length)):
            if words[i] in model:
                feature[i * embedding_dim : (i + 1) * embedding_dim] =
model[words[i]]
        concatenated_features.append(feature)
    return np.array(concatenated_features)

# Use the function to generate concatenated Word2Vec features
X_w2v_concatenated =
generate_concatenated_word2vec(train_df['review_body'],
word2vec_model)

X_w2v_concatenated_test =
generate_concatenated_word2vec(test_df['review_body'], word2vec_model)

# Convert the concatenated Word2Vec features to PyTorch tensors
X_train_word2vec_concat_tensor = torch.tensor(X_w2v_concatenated,
dtype=torch.float32)
X_test_word2vec_concat_tensor = torch.tensor(X_w2v_concatenated_test,
dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values - 1, dtype=torch.long)  #
Adjust labels to start from 0
y_test_tensor = torch.tensor(y_test.values - 1, dtype=torch.long)  #
Adjust labels to start from 0

# Update the input dimension based on the concatenated vectors
input_dim =  X_train_word2vec_concat_tensor.shape[1]  # Flattened
feature dimension

# Initialize the model with the updated input dimension
model = MLPModel(input_dim, hidden_dim1, hidden_dim2, output_dim)

# Define loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)  # You can adjust
the learning rate

# Training loop
```

```
epochs = 300  # You can adjust the number of epochs
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    y_pred = model(X_train_word2vec_concat_tensor)
    loss = criterion(y_pred, y_train_tensor)
    loss.backward()
    optimizer.step()

    if (epoch+1) % 100 == 0:
        model.eval()
        with torch.no_grad():
            test_logits = model(X_test_word2vec_concat_tensor)
            y_pred = torch.softmax(test_logits, dim=1).argmax(dim=1)
            test_loss = criterion(test_logits, y_test_tensor)
        acc = accuracy_score(y_test_tensor.numpy(), y_pred.numpy())
        print('epoch', epoch+1, 'loss', test_loss.item(), 'test
accuracy', acc)
```

```
epoch 100 loss 0.565644383430481 test accuracy 0.73565
epoch 200 loss 0.560996413230896 test accuracy 0.7367
epoch 300 loss 0.5624485015869141 test accuracy 0.7386
```

Analysis of concatenated first 10 Word2Vec vectors vs normal word2vec vectors on MLP :

1) We can see that the MLP model performs better on the average word2vec vectors in comparison to using the first 10 concatenated vectors for each review.

2) In comparison to the simple models such as SVM and Perceptron, the FFN is slightly performing better than those models for the averaged word2vec vectors. A feedforward neural network (FFN) may perform better with Word2Vec features compared to SVM and perceptron because FFNs are highly capable of learning complex, non-linear relationships within high-dimensional feature spaces, which is essential for capturing the rich semantic information encoded in Word2Vec vectors. SVM and perceptron are linear models and may struggle to exploit the full expressiveness of Word2Vec features.

Explanation of the above code :

A function generate_concatenated_word2vec is defined to generate concatenated Word2Vec features for a list of reviews. It concatenates the Word2Vec vectors for each word in a review, up to a specified maximum sequence length. If a word is not present in the Word2Vec model's vocabulary, it uses a zero vector.

The function is used to generate concatenated Word2Vec features for both the training and testing data, stored in X_w2v_concatenated and X_w2v_concatenated_test.

These concatenated Word2Vec features are converted into PyTorch tensors for both training and testing datasets.

The input dimension of the MLP model is updated to match the flattened feature dimension of the concatenated Word2Vec vectors.

The model is reinitialized with the updated input dimension.

The training loop remains similar to the previous code, but now it trains the MLP model with the concatenated Word2Vec features.

1. Recurrent Neural Networks (30 points) Using the Word2Vec features, train a recurrent neural network (RNN) for classification. You can refer to the following tutorial to familiarize yourself: https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html

(a) (10 points) Train a simple RNN for sentiment analysis. You can consider an RNN cell with the hidden state size of 10. To feed your data into our RNN, limit the maximum review length to 10 by truncating longer reviews and padding shorter reviews with a null value (0). Report accuracy values on the testing split for your RNN model. What do you conclude by comparing accuracy values you obtain with those obtained with feedforward neural network models.

```
def generate_concatenated_word2vec(reviews, model, max_seq_length=10,
embedding_dim=300):
    concatenated_features = []
    for review in reviews:
        words = review.split()
        feature = np.zeros((max_seq_length, embedding_dim,))
        for i in range(min(len(words), max_seq_length)):
            if words[i] in model:
                feature[i] = model[words[i]]
        concatenated_features.append(feature)
    return np.array(concatenated_features)

# Use the function to generate concatenated Word2Vec features
X_w2v_concatenated =
generate_concatenated_word2vec(train_df['review_body'],
word2vec_model)

X_w2v_concatenated_test =
generate_concatenated_word2vec(test_df['review_body'], word2vec_model)

# Convert averaged Word2Vec vectors to PyTorch tensor
X_train_word2vec_padded_tensor = torch.tensor(X_w2v_concatenated,
dtype=torch.float32)
X_test_word2vec_padded_tensor = torch.tensor(X_w2v_concatenated_test,
dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values - 1, dtype=torch.long)  #
Adjust labels to start from 0
y_test_tensor = torch.tensor(y_test.values - 1, dtype=torch.long)  #
Adjust labels to start from 0


# Define the RNN model
class RNNModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(RNNModel, self).__init__()
```

```
        self.hidden_dim = hidden_dim
        self.rnn = nn.RNN(input_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        # Initialize hidden state with zeros
        h0 = torch.zeros(1, x.size(0), self.hidden_dim).to(x.device)
        # We need to detach as we are doing truncated backpropagation
through time (BPTT)
        out, _ = self.rnn(x, h0.detach())
        out = self.fc(out[:, -1, :])  # Take the output from the last
time step
        return out

# Initialize the model
input_dim = X_train_word2vec_padded_tensor.shape[2]  # Word2Vec
dimension
hidden_dim = 10
output_dim = 2  # Two classes (sentiment_class 1 and 2)
model = RNNModel(input_dim, hidden_dim, output_dim)

# Define loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)  # You can adjust
the learning rate

# Training loop
epochs = 500  # You can adjust the number of epochs
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    # y_pred = model(X_train_word2vec_padded_tensor.unsqueeze(1))
    y_pred = model(X_train_word2vec_padded_tensor)
    loss = criterion(y_pred, y_train_tensor)
    loss.backward()
    optimizer.step()

    if (epoch+1) % 100 == 0:
        model.eval()
        with torch.no_grad():
            # test_logits =
model(X_test_word2vec_padded_tensor.unsqueeze(1))
            test_logits = model(X_test_word2vec_padded_tensor)
            y_pred = torch.softmax(test_logits, dim=1).argmax(dim=1)
            test_loss = criterion(test_logits, y_test_tensor)
        acc = accuracy_score(y_test_tensor.numpy(), y_pred.numpy())
        print('epoch', epoch+1, 'loss', test_loss.item(), 'test
accuracy', acc)
```

```
epoch 100 loss 0.5763524770736694 test accuracy 0.70925
epoch 200 loss 0.5295952558517456 test accuracy 0.73855
epoch 300 loss 0.5144208073616028 test accuracy 0.7462
epoch 400 loss 0.5057020783424377 test accuracy 0.7513
epoch 500 loss 0.501423716545105 test accuracy 0.7541
```

Analysis of RNN vs FFN:

By looking at the testing accuracies we can say that the RNN performs better than the FFN trained on the data with first 10 concatenated vectors. It could be because RNNs can capture temporal dependencies and contextual information within sequences. Word2Vec features contain semantic relationships that benefit from sequence modeling. FFNs, lacking this sequential context, may struggle to harness the full potential of Word2Vec features.

Explanation of the code:

The code defines a function, generate_concatenated_word2vec, which generates concatenated Word2Vec features for a list of reviews. It concatenates Word2Vec vectors for words in each review, up to a specified maximum sequence length. If a word is not present in the Word2Vec model's vocabulary, it uses a zero vector.

The function is used to generate concatenated Word2Vec features for both the training and testing data, stored in X_w2v_concatenated and X_w2v_concatenated_test.

These concatenated Word2Vec features are converted into PyTorch tensors for both training and testing datasets.

A new RNN model is defined using the nn.Module class. This model consists of an RNN layer followed by a fully connected layer. It is designed for sequence data.

The model is initialized with the specified input dimension, hidden layer dimension, and output dimension (2 for sentiment classes 1 and 2).

Cross-entropy loss and the Adam optimizer are defined for training the RNN model.

The training loop runs for a specified number of epochs, where the model is trained on the training data. The validation loss and accuracy on the testing data are calculated during training.

(b) (10 points) Repeat part (a) by considering a gated recurrent unit cell

```python
# Define the GRU model
class GRUModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(GRUModel, self).__init__()
        self.hidden_dim = hidden_dim
        self.gru = nn.GRU(input_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        # Initialize hidden state with zeros
        h0 = torch.zeros(1, x.size(0), self.hidden_dim).to(x.device)
```

```
        # We need to detach as we are doing truncated backpropagation
through time (BPTT)
        out, _ = self.gru(x, h0.detach())
        out = self.fc(out[:, -1, :])  # Take the output from the last
time step
        return out

# Initialize the GRU model
input_dim = X_train_word2vec_padded_tensor.shape[2]
hidden_dim = 10
output_dim = 2  # Two classes (sentiment_class 1 and 2)
model = GRUModel(input_dim, hidden_dim, output_dim)

# Define loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)  # You can adjust
the learning rate

# Training loop
epochs = 500  # You can adjust the number of epochs
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    y_pred = model(X_train_word2vec_padded_tensor)
    loss = criterion(y_pred, y_train_tensor)
    loss.backward()
    optimizer.step()

    if (epoch+1) % 100 == 0:
        model.eval()
        with torch.no_grad():
            test_logits = model(X_test_word2vec_padded_tensor)
            y_pred = torch.softmax(test_logits, dim=1).argmax(dim=1)
            test_loss = criterion(test_logits, y_test_tensor)
        acc = accuracy_score(y_test_tensor.numpy(), y_pred.numpy())
        print('epoch', epoch+1, 'loss', test_loss.item(), 'test
accuracy', acc)
```

```
epoch 100 loss 0.6227900981903076 test accuracy 0.66375
epoch 200 loss 0.5182005763053894 test accuracy 0.7453
epoch 300 loss 0.49444955587387085 test accuracy 0.76025
epoch 400 loss 0.4836018979549408 test accuracy 0.7665
epoch 500 loss 0.477856308221817 test accuracy 0.7711
```

Explanation of the above code :

A new class, GRUModel, is defined to create a Gated Recurrent Unit (GRU) model for sentiment classification. This model consists of a GRU layer followed by a fully connected layer. The GRU is a type of recurrent neural network (RNN) designed to capture sequential patterns in data.

The model is initialized with the specified input dimension, hidden layer dimension, and output dimension (2 for sentiment classes 1 and 2).

Cross-entropy loss and the Adam optimizer are defined for training the GRU model.

The training loop runs for a specified number of epochs, where the model is trained on the training data. The validation loss and accuracy on the testing data are calculated during training.

(c) (10 points) Repeat part (a) by considering an LSTM unit cell. What do you conclude by comparing accuracy values you obtain by GRU, LSTM, and simple RNN.

```
# Define the LSTM model
class LSTMModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(LSTMModel, self).__init__()
        self.hidden_dim = hidden_dim
        self.lstm = nn.LSTM(input_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        # Initialize hidden state with zeros
        h0 = torch.zeros(1, x.size(0), self.hidden_dim).to(x.device)
        c0 = torch.zeros(1, x.size(0), self.hidden_dim).to(x.device)
        # We need to detach as we are doing truncated backpropagation
through time (BPTT)
        out, _ = self.lstm(x, (h0.detach(), c0.detach()))
        out = self.fc(out[:, -1, :])  # Take the output from the last
time step
        return out

# Initialize the LSTM model
input_dim = X_train_word2vec_padded_tensor.shape[2]
hidden_dim = 10
output_dim = 2  # Two classes (sentiment_class 1 and 2)
model = LSTMModel(input_dim, hidden_dim, output_dim)

# Define loss and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)  # You can adjust
the learning rate

# Training loop
epochs = 500  # You can adjust the number of epochs
for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
    y_pred = model(X_train_word2vec_padded_tensor)
    loss = criterion(y_pred, y_train_tensor)
    loss.backward()
    optimizer.step()
```

```
    if (epoch+1) % 100 == 0:
        model.eval()
        with torch.no_grad():
            test_logits = model(X_test_word2vec_padded_tensor)
            y_pred = torch.softmax(test_logits, dim=1).argmax(dim=1)
            test_loss = criterion(test_logits, y_test_tensor)
        acc = accuracy_score(y_test_tensor.numpy(), y_pred.numpy())
        print('epoch', epoch+1, 'loss', test_loss.item(), 'test
accuracy', acc)
```

```
epoch 100 loss 0.574582576751709 test accuracy 0.71785
epoch 200 loss 0.5062186121940613 test accuracy 0.7553
epoch 300 loss 0.4854887127876282 test accuracy 0.7667
epoch 400 loss 0.4787159562110901 test accuracy 0.77135
epoch 500 loss 0.4766538441181183 test accuracy 0.77255
```

Analysis of RNN vs GRU vs LSTM:

We can see that RNN performs worser than the other 2 models on testing accuracy. While, if we compare GRU and LSTM, they both provide comparable/similar testing accuracies (LSTM performs just slightly better).

The relatively poorer performance of the basic RNN compared to GRU and LSTM with Word2Vec features can be attributed to the vanishing gradient problem. Basic RNNs have difficulty capturing long-range dependencies in sequences due to the vanishing gradient issue, which hampers their ability to leverage the rich semantic relationships embedded in Word2Vec features. GRU and LSTM architectures are designed to mitigate this problem and can better capture the nuanced semantic information, resulting in similar but better testing accuracies. LSTM, with its more sophisticated gating mechanisms, may outperform GRU slightly by capturing long-term dependencies more effectively.

Explanation of the above code :

A new class, LSTMModel, is defined to create a Long Short-Term Memory (LSTM) model for sentiment classification. The LSTM model consists of an LSTM layer followed by a fully connected layer. LSTMs are a type of recurrent neural network (RNN) that can capture long-range dependencies in sequential data.

The model is initialized with the specified input dimension, hidden layer dimension, and output dimension (2 for sentiment classes 1 and 2).

Cross-entropy loss and the Adam optimizer are defined for training the LSTM model.

The training loop runs for a specified number of epochs, where the model is trained on the training data. During training, the validation loss and accuracy on the testing data are calculated.