Libraries which will be used to implement this NLP HW-2

1. As we have used a dataset (.TSV), to read from and it and access the contents in it, we make use of pandas as pd
2. We make use of numpy for our mathematical operations
3. We make use of re for cleaning the text (from the used dataset), as a regular expression
4. We make use of nltk (a NLP tool) for stop words removal, tokenization, etc..
5. gensim library is used to generate two sets of Word2Vec features KeyedVectors is used to load the Google News Word2Vec model
6. sklearn is primarily used for feature extraction, data splitting, evaluation and baseline models.
7. PyTorch is a Deep Learning model construction, used for training and optimization, data handlin, and GPU acceleration (Sadly, could not make GPU work)

```
import pandas as pd
import numpy as np
import re
import random
import gzip

import nltk
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
nltk.download('punkt_tab')
nltk.download('stopwords')
nltk.download('wordnet')

from bs4 import BeautifulSoup

import gensim
import gensim.downloader as api
from gensim.models import Word2Vec, KeyedVectors

from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.svm import LinearSVC
from sklearn.linear_model import Perceptron

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader, TensorDataset
```

```
[nltk_data] Downloading package punkt_tab to /root/nltk_data...
[nltk_data]   Package punkt_tab is already up-to-date!
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

DATASET GENERATION:

STEP 1 OF HW-2:

We are using the same dataset "amazon_reviews_us_Office_Products_v1_00.tsv.gz" that we used in assignment 1 here as well. We will now take this dataset (given) and read it as a dataframe (df var in code) and keep the necessary columns only.

The var. "FILE_PATH" is the path to the dataset with which we manipulate necessary changes Using "df_full", we do the above mentioned changes such as keeping the necessary column(s), review the text and ratings.

Next, we make sure that "Rating" is pure numeric and drop any rows with NaN not a (number)

```
FILE_PATH = "/content/amazon_reviews_us_Office_Products_v1_00.tsv.gz" #/content/amazon_reviews_us_Office_Products_v1_00.tsv.gz


df_full = pd.read_csv(
    FILE_PATH,
    sep='\t',
    compression='gzip',
    on_bad_lines='skip',
    low_memory=False
)


df_full = df_full[['review_body', 'star_rating']].copy()
```

```
df_full.columns = ['Review', 'Rating']

df_full.dropna(subset=['Review', 'Rating'], inplace=True)
df_full['Rating'] = pd.to_numeric(df_full['Rating'], errors='coerce')
df_full.dropna(subset=['Rating'], inplace=True)
```

We start off with "print(df_full.shape)" which prints the original dataframe. With this, we can identify the number of rows and columns before sampling.

Then, we loop through each rating (starting from 1 till 5), and calculate how many rows have what rating, with "rating_count"

We then print the count to see output and used for debugging as well.

The 2nd for loop iterates over every rating again. This then filters the dataframe to keep only rows of the current rating values and then we print it. And then, we randomly take 50,000 samples (rows). This ensures the criteria specified in the assignment file to use exactly 50,000 reviews.

Finally, we concatenate all sampled subsets into a new dataframe. As concatenations can cause discontinous index, the index is reset. The final dataframe is then printed.

```
print(df_full.shape)

for rating_value in [1, 2, 3, 4, 5]:
    rating_count = (df_full['Rating'] == rating_value).sum()
    print(f"Rating {rating_value} has {rating_count} rows")

dfs = []

for rating_value in [1, 2, 3, 4, 5]:
    temp_df = df_full[df_full['Rating'] == rating_value]
    # Print shape of temp_df for debugging
    print(f"temp_df rating={rating_value} shape:", temp_df.shape)

    temp_df_sampled = temp_df.sample(n=50000, random_state=42)
    dfs.append(temp_df_sampled)

df_balanced = pd.concat(dfs, axis=0).reset_index(drop=True)
print("df_balanced shape:", df_balanced.shape)
```

```
(2640080, 2)
Rating 1 has 306967 rows
Rating 2 has 138381 rows
Rating 3 has 193680 rows
Rating 4 has 418348 rows
Rating 5 has 1582704 rows
temp_df rating=1 shape: (306967, 2)
temp_df rating=2 shape: (138381, 2)
temp_df rating=3 shape: (193680, 2)
temp_df rating=4 shape: (418348, 2)
temp_df rating=5 shape: (1582704, 2)
df_balanced shape: (250000, 2)
```

We now define a function "map_rating_to_class" that takes numeric rating 'r' and maps it according to specified fields, being positive(4 or 5), neutral(3) and negative(1 or 2). We then apply the mapping function to the rating column, creating a new column "sentiment" with values 1,2,3.

```
def map_rating_to_class(r):
    if r > 3:
        return 1  # Positive
    elif r < 3:
        return 2  # Negative
    else:
        return 3  # Neutral

df_balanced['Sentiment'] = df_balanced['Rating'].apply(map_rating_to_class)

print("Value counts for Sentiment:")
print(df_balanced['Sentiment'].value_counts())
```

```
Value counts for Sentiment:
Sentiment
2    100000
1    100000
3     50000
Name: count, dtype: int64
```

As mentioned in the pdf, we now split the dataset into training (80%) and testing (20%). We make use of "stratify=df_balanced['Sentiment']" to preserve the class distribution proportionally in both train and test sets. We then print the training and testing dataframes to confirm whether they match the required specifications.

```
train_df, test_df = train_test_split(
    df_balanced,
    test_size=0.2,
    stratify=df_balanced['Sentiment'],
    random_state=42
)

print("Train shape:", train_df.shape)
print("Test shape :", test_df.shape)

# Check distribution of sentiment in each set
print(train_df['Sentiment'].value_counts())
print(test_df['Sentiment'].value_counts())
```

```
Train shape: (200000, 3)
Test shape : (50000, 3)
Sentiment
1    80000
2    80000
3    40000
Name: count, dtype: int64
Sentiment
1    20000
2    20000
3    10000
Name: count, dtype: int64
```

WORD EMBEDDING

STEP 2 OF HW-2:PART-A To try out the api way of using the word2vec model, we have made use of the api command from the "helpful tutorial given in the HW-2 pdf". We have printed before and after for debugging purposes

```
print("Loading 'word2vec-google-news-300'")
wv = api.load('word2vec-google-news-300')
print("Model loaded successfully!")
```

```
Loading 'word2vec-google-news-300'
Model loaded successfully!
```

From the pdf, we can see that we need to test out the dataset and validate the semantic similarities.
The following code does exactly that. We take the sample example from the pdf and one of our own, to compare within the top 5 contenders and thier metric values. Hence, we know that the code works! With this, we complete PART-A of our step 2.

```
print("\nAnalogy: king - man + woman = ? ")
result_analogy = wv.most_similar(positive=['king', 'woman'], negative=['man'], topn=5)
for word, sim in result_analogy:
    print(f"{word}: {sim:.4f}")

sim_score = wv.similarity('excellent', 'outstanding')
print(f"\nSimilarity between 'excellent' and 'outstanding': {sim_score:.4f}")

most_similar_excellent = wv.most_similar('excellent', topn=5)
print("\nMost similar words to 'excellent':")
for word, sim in most_similar_excellent:
    print(f"{word}: {sim:.4f}")
```

```

Analogy: king - man + woman = ?
queen: 0.7118
monarch: 0.6190
princess: 0.5902
crown_prince: 0.5499
prince: 0.5377

Similarity between 'excellent' and 'outstanding': 0.5567

Most similar words to 'excellent':
terrific: 0.7410
superb: 0.7063
exceptional: 0.6815
fantastic: 0.6803
```

```
good: 0.6443
```

The next 2 code blocks are given a part of Step-3 in the pdf. But, improved scores were observed when used in between STEP-2 PART (A) AND PART (B).

This first code block does the Data cleaning (exact same used in HW-1 as mentioned in pdf).

```
df_balanced.dropna(subset=['Review'], inplace=True)
df_balanced['Review'] = df_balanced['Review'].astype(str)

avg_length_before_cleaning = df_balanced['Review'].apply(len).mean()
print("Average length (in characters) before cleaning:", avg_length_before_cleaning)

df_balanced['Review'] = df_balanced['Review'].str.lower()

def remove_html_and_urls(text):
    # Remove HTML tags using BeautifulSoup
    text_no_html = BeautifulSoup(text, "html.parser").get_text(separator=" ")

    text_no_url = re.sub(r'(https?://\S+|www\.\S+)', '', text_no_html)

    return text_no_url

df_balanced['Review'] = df_balanced['Review'].apply(remove_html_and_urls)

df_balanced['Review'] = df_balanced['Review'].str.replace('[^a-z]', ' ', regex=True)

df_balanced['Review'] = df_balanced['Review'].str.split().str.join(' ')

#did my best to add as much as possible
contractions_dict = {
    "won't": "will not",
    "can't": "cannot",
    "don't": "do not",
    "didn't": "did not",
    "i'm": "i am",
    "it's": "it is",
    "he's": "he is",
    "she's": "she is",
    "that's": "that is",
    "aren't": "are not",
    "weren't": "were not",
    "haven't": "have not",
    "hasn't": "has not",
    "shouldn't": "should not",
    "wouldn't": "would not",
    "couldn't": "could not",
    "isn't": "is not",
    "what's": "what is",
    "where's": "where is",
    "who's": "who is",
    "you'd": "you would",
    "you'll": "you will",
    "you're": "you are",
    "they're": "they are",
    "they've": "they have",
    "we're": "we are",
    "we've": "we have",
    "there's": "there is"
}

contractions_pattern = re.compile(r'\b(' + '|'.join(contractions_dict.keys()) + r')\b')

def expand_contractions(text, pattern=contractions_pattern):
    def replace(match):
        return contractions_dict[match.group(0)]
    return pattern.sub(replace, text)

df_balanced['Review'] = df_balanced['Review'].apply(expand_contractions)

avg_length_after_cleaning = df_balanced['Review'].apply(len).mean()
print("Average length (in characters) after cleaning:", avg_length_after_cleaning)
```

```
Average length (in characters) before cleaning: 341.193312
<ipython-input-9-c636221de1a1>:11: MarkupResemblesLocatorWarning: The input passed in on this line looks more like a URL than HTML c

If you meant to use Beautiful Soup to parse the web page found at a certain URL, then something has gone wrong. You should use an Py

However, if you want to parse some data that happens to look like a URL, then nothing has gone wrong: you are using Beautiful Soup c
```

```
    from bs4 import MarkupResemblesLocatorWarning
    import warnings

    warnings.filterwarnings("ignore", category=MarkupResemblesLocatorWarning)

  text_no_html = BeautifulSoup(text, "html.parser").get_text(separator=" ")
Average length (in characters) after cleaning: 323.726712
```

This code block does the pre-processing (Again, as explained in the pdf, this is the code used in HW-1).

```
stop_words = set(stopwords.words('english'))
lemmatizer = WordNetLemmatizer()

def preprocess_text(text):
    tokens = text.split()
    tokens = [word for word in tokens if word not in stop_words]
    tokens = [lemmatizer.lemmatize(word) for word in tokens]
    processed_text = " ".join(tokens)
    return processed_text

sample_indices = df_balanced.sample(3, random_state=42).index

print("SAMPLE REVIEWS BEFORE PREPROCESSING:")
for idx in sample_indices:
    print(f"Review {idx}:\n{df_balanced.loc[idx, 'Review']}")
    print("-"*80)

avg_length_before_preprocessing = df_balanced['Review'].apply(len).mean()
print("Average length (in characters) before preprocessing:", avg_length_before_preprocessing)

df_balanced['Review'] = df_balanced['Review'].apply(preprocess_text)

print("SAMPLE REVIEWS AFTER PREPROCESSING:")
for idx in sample_indices:
    print(f"Review {idx}:\n{df_balanced.loc[idx, 'Review']}")
    print("-"*80)

avg_length_after_preprocessing = df_balanced['Review'].apply(len).mean()
print("Average length (in characters) after preprocessing:", avg_length_after_preprocessing)

print("Average length (in characters) before preprocessing:", avg_length_before_preprocessing)
```

```
SAMPLE REVIEWS BEFORE PREPROCESSING:
Review 38683:
light didn t work so i tried to replace the battery and the entire pen basically fell apart in my hands this is the biggest piece of
--------------------------------------------------------------------------------
Review 64939:
let me say up front that i m going to try to upload a picture of my badge holder for you all to look at if it s there great if not t
--------------------------------------------------------------------------------
Review 3954:
the printer will not print in any color if just one of the ink cartridges are out i ve had terrible problems with the printing skipp
--------------------------------------------------------------------------------
Average length (in characters) before preprocessing: 323.726712
SAMPLE REVIEWS AFTER PREPROCESSING:
Review 38683:
light work tried replace battery entire pen basically fell apart hand biggest piece crap ever bought even working piece cheap cheap
--------------------------------------------------------------------------------
Review 64939:
let say front going try upload picture badge holder look great say three month routine use badge holder developed substantial crack
--------------------------------------------------------------------------------
Review 3954:
printer print color one ink cartridge terrible problem printing skipping around even read read learned maintenence clean head printe
--------------------------------------------------------------------------------
Average length (in characters) after preprocessing: 199.191456
Average length (in characters) before preprocessing: 323.726712
```

```
df_binary = df_balanced[df_balanced['Sentiment'].isin([1, 2])].copy()

# Map sentiment: class 1 (positive) -> label 1, class 2 (negative) -> label 0
df_binary['Label'] = df_binary['Sentiment'].apply(lambda x: 1 if x == 1 else 0)

print("Binary class distribution:")
print(df_binary['Label'].value_counts())
```

```
Binary class distribution:
Label
```

```
0    100000
1    100000
Name: count, dtype: int64
```

STEP 2 OF HW-2:PART-B:

We make use of nltk package (previously seen on HW-1), to implement tokenization process. nltk.download('punkt') is used for word tokenization. We had to use nltk.download('punkt_tab') to run this code due to unexpected error with using just "punkt".
Step 1: Tokenize the input text using NLTK word_tokenize, convert to lowercase, remove non-alphabetic tokens. This is done using the function tokenize(text).
Step 2: Create 'sentences' from df_balanced. Already exists from part 1. df_balanced has columns ['Review', 'Rating', 'Sentiment']. Create a Word2Vec model with desired params: 1.vector_size=300 (given); window=11 (given); min_count=10 (ignore words appearing <10 times) workers=12 (I have a 16 core CPU, hence I have used 12).
We will then build the vocabulary from tokenized sentences. model_own.train() trains my model. To get a better result, I have done 7 epochs and extracted the trained word vectors. Step 4: heck Semantic Similarities. I have started with the previously used example here as well and then compared it.

The answers to asked question:
Q1. What do you conclude from comparing vectors generated by yourself and the pretrained model?
A1. When comparing the pretrained Google News Word2Vec model with a Word2Vec model trained on our own Amazon reviews dataset, we observe some clear differences in the way each model captures and encodes semantic similarities. Pretrained Model:
Trained on a massive, general-purpose corpus (Google News), encompassing a wide variety of everyday topics and vocabulary.
Consequently, it has extensive coverage of general English words, historical/political entities, cultural references, etc.
For words like “king,” “queen,” “woman,” and “man,” it has seen rich contextual usage, so it can correctly perform analogies like “king - man + woman → queen.”

Custom Model (Amazon Reviews):
Trained on domain-specific text (product reviews).
The corpus is skewed toward topics like product features, brand names, shipping, customer service, etc.
General or infrequent words like “king” or “queen” may appear rarely and thus do not develop strong semantic associations.

Domain-Specific Strengths:

While the pretrained model handles general English very well, it may not reflect how specific product terms and brand names are used in Amazon reviews.
The custom model, in contrast, can learn very precise relationships for frequently discussed items (e.g., “printer” ↔ “ink,” “paper” ↔ “quality,” etc.).
If you test words that occur commonly in the Amazon domain (e.g., “shipping,” “refund,” “receipt”), you may find that the custom model produces more coherent or contextually appropriate synonyms and neighbors.

Q2. Which of the Word2Vec models seems to encode semantic similarities between words better?
A2. For General English / Common Words: The pretrained Google News model is more likely to produce high-quality, intuitive analogies and similarity scores. It has seen a broader corpus, so it captures well-known linguistic relationships (like “king → queen,” “excellent → outstanding”).
For Domain-Specific Terms: The custom Amazon reviews model can outperform the pretrained model when dealing with specialized or product-centric vocabulary. If a term is very frequent in your dataset (e.g., “laptop,” “battery,” “warranty”), the custom embeddings might capture nuances that the general model misses.

CONCLUSION:

Pretrained Word2Vec (Google News) is generally stronger at broad, standard semantic relationships, especially for words that are common in general discourse.
Custom Word2Vec is more tailored to the Amazon reviews domain and can excel at capturing domain-specific semantics, but it lacks coverage of rarer general words or contexts not present in the review corpus.

This brings an end to our STEP-2 PART-B.

```
def tokenize(text):

    text = text.lower()
    tokens = nltk.word_tokenize(text)
    tokens = [t for t in tokens if t.isalpha()]

    return tokens

print("Binary class distribution:")
print(df_binary['Label'].value_counts())
print("Tokenizing all reviews")
sentences = [tokenize(review) for review in df_binary['Review']]
print(f"Completed.\nNumber of documents (reviews): {len(sentences)}")


model_own = Word2Vec(
    vector_size=300,
    window=11,
    min_count=10,
    workers=12
)

model_own.build_vocab(sentences)
print(f"Vocabulary size: {len(model_own.wv.key_to_index)} words")

print("Training Word2Vec on dataset..")
model_own.train(
    sentences,
    total_examples=model_own.corpus_count,
    epochs=7
)
print("Training complete!")

word_vectors_own = model_own.wv


print("\n=== Checking Analogy: king - man + woman ===")
try:
    analogy_result = word_vectors_own.most_similar(
        positive=['king', 'woman'],
        negative=['man'],
        topn=5
    )
    for word, similarity in analogy_result:
        print(f"{word}: {similarity:.4f}")
except KeyError as e:
    print(f"Word not in vocabulary: {e}")

print("\n=== Similarity: 'excellent' vs. 'outstanding' ===")
try:
    sim_score = word_vectors_own.similarity('excellent', 'outstanding')
    print(f"Similarity: {sim_score:.4f}")
except KeyError as e:
    print(f"Word not in vocabulary: {e}")
```

```
Binary class distribution:
Label
0    100000
1    100000
Name: count, dtype: int64
Tokenizing all reviews
Completed.
Number of documents (reviews): 200000
Vocabulary size: 12862 words
Training Word2Vec on dataset..
Training complete!

=== Checking Analogy: king - man + woman ===
latin: 0.5140
men: 0.4750
reproductive: 0.4612
gender: 0.4575
cancer: 0.4519

=== Similarity: 'excellent' vs. 'outstanding' ===
Similarity: 0.7871
```

SIMPLE MODELS (STEP-3)

Having imported libraries already (see above), we will follow the pdf to achieve the step by step process to achieve the required model.

1. As mentioned previously, Data cleaning, Pre-processing and filtering into a binary class has been completed in previous steps. We will now continue with creating average word2vec vector computation.
   This code executes the average word vector for a given text using a given Word2Vec model. Uses NLTK's word_tokenize to split text and filters to alphabetic tokens. Returns a zero vector if no tokens are found in the model's vocabulary.
   We then compute average embeddings for each review. for both pre-trained and custom word2, new columns are created with the avg. vector.

We will also do 'AvgPretrained' for df_balanced (FUTURE PURPOSE in step-4)

```
def average_word_vector(text, model, vector_size=300):

    tokens = nltk.word_tokenize(text.lower())

    tokens = [t for t in tokens if t.isalpha()]

    valid_vectors = []
    for token in tokens:
        if token in model.key_to_index:
            valid_vectors.append(model[token])

    if len(valid_vectors) == 0:
        return np.zeros(vector_size)

    return np.mean(valid_vectors, axis=0)

df_binary['AvgPretrained'] = df_binary['Review'].apply(lambda x: average_word_vector(x, wv))
df_binary['AvgCustom'] = df_binary['Review'].apply(lambda x: average_word_vector(x, model_own.wv))
```

```
def average_word_vector(text, model, vector_size=300):
    tokens = nltk.word_tokenize(text.lower())
    tokens = [token for token in tokens if token.isalpha()]

    vectors = [model[token] for token in tokens if token in model.key_to_index]

    if not vectors:
        return np.zeros(vector_size)

    return np.mean(vectors, axis=0)

df_balanced['AvgPretrained'] = df_balanced['Review'].apply(lambda x: average_word_vector(x, wv))
print(df_balanced[['Review', 'AvgPretrained']].head())
```

```
                                              Review  \
0  purchased tab whim put movie poster used four ...
1               returned much garbage involved setup
2  upholstered living room chair particularly big...
3                                              trash
4  hate printer without warning get error message...

                                       AvgPretrained
0  [0.04835728, 0.02457973, 0.009993689, 0.135595...
1  [0.11796875, 0.13583985, -0.06367187, 0.091650...
2  [0.034998577, 0.045771282, 0.032447178, 0.1003...
3  [0.024780273, 0.38476562, 0.092285156, 0.20605...
4  [0.0443637, 0.0371015, -0.03636071, 0.09358855...
```

Now, we will re-use our TF-IDF (FEATURE EXTRACTION) steps made in HW-1 (As mentioned in pdf). The following code block executes that very same.

```
vectorizer = TfidfVectorizer(
    ngram_range=(1, 2),
    min_df=5,
    max_df=0.8,
    sublinear_tf=True
)

X = vectorizer.fit_transform(df_binary['Review'])
y = df_binary['Sentiment'].values

X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    random_state=42
)

print("X_train shape:", X_train.shape)
print("X_test shape :", X_test.shape)
print("y_train shape:", y_train.shape)
print("y_test shape :", y_test.shape)
```

```
X_train shape: (160000, 192491)
X_test shape : (40000, 192491)
y_train shape: (160000,)
y_test shape : (40000,)
```

In this step, we will be creating matrices for word2vec representations bt stacking the average vectors into a 2D matrix.
And then, we will extract the binary labels.(1 = positive; 0 = negative)

For a pre-trained word2vec, each row is a 300-Dim vector.

```
X_pretrained = np.vstack(df_binary['AvgPretrained'].values)
X_custom = np.vstack(df_binary['AvgCustom'].values)

y = df_binary['Label'].values
```

Now, we will execute the train-test split. This will be an 80-20 split (respectively, as mentioned in the pdf).

```
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# For pretrained Word2Vec features
X_train_pre, X_test_pre, _, _ = train_test_split(
    X_pretrained, y, test_size=0.2, random_state=42, stratify=y
)

# For custom Word2Vec features
X_train_custom, X_test_custom, _, _ = train_test_split(
    X_custom, y, test_size=0.2, random_state=42, stratify=y
)
```

Now, we will train classifiers and evaluate them. As mentioned in the pdf, it will the perceptron and SVM.

For each feature type, we train a Perceptron and an SVM model, then report testing accuracy.

The first code blovk will implement perceptron. (Below!)

```
perc_tfidf = Perceptron(random_state=42)
perc_tfidf.fit(X_train, y_train)
acc_tfidf_perc = accuracy_score(y_test, perc_tfidf.predict(X_test))

# Perceptron using pretrained Word2Vec features
perc_pre = Perceptron(random_state=42)
perc_pre.fit(X_train_pre, y_train)
acc_pre_perc = accuracy_score(y_test, perc_pre.predict(X_test_pre))

# Similarly, for the custom Word2Vec features:
perc_custom = Perceptron(random_state=42)
perc_custom.fit(X_train_custom, y_train)
acc_custom_perc = accuracy_score(y_test, perc_custom.predict(X_test_custom))
```

The below code block will implement Linear SVM. As SVM took more than 55 minutes to run!!! (TWICE)

```
linear_svc_tfidf = LinearSVC(random_state=42, max_iter=10000)
linear_svc_tfidf.fit(X_train, y_train)
acc_tfidf_linear = accuracy_score(y_test, linear_svc_tfidf.predict(X_test))

# LinearSVC using pretrained Word2Vec features
linear_svc_pre = LinearSVC(random_state=42, max_iter=10000)
linear_svc_pre.fit(X_train_pre, y_train)
acc_pre_linear = accuracy_score(y_test, linear_svc_pre.predict(X_test_pre))

# LinearSVC using custom Word2Vec features
linear_svc_custom = LinearSVC(random_state=42, max_iter=10000)
linear_svc_custom.fit(X_train_custom, y_train)
acc_custom_linear = accuracy_score(y_test, linear_svc_custom.predict(X_test_custom))
```

The following code block PRINTS the desired values as the computation is now complete. This brings an end to the STEP-3 of our homework.

```
print("Perceptron Testing Accuracy:")
print("TF-IDF:", acc_tfidf_perc)
print("Pretrained Word2Vec:", acc_pre_perc)
print("Custom Word2Vec:", acc_custom_perc)

print("\nSVM Testing Accuracy:")
print("TF-IDF:", acc_tfidf_linear)
print("Pretrained Word2Vec:", acc_pre_linear)
print("Custom Word2Vec:", acc_custom_linear)
```

```
Perceptron Testing Accuracy:
TF-IDF: 0.850075
Pretrained Word2Vec: 0.762025
Custom Word2Vec: 0.7682

SVM Testing Accuracy:
```

```
TF-IDF: 0.8833
Pretrained Word2Vec: 0.817425
Custom Word2Vec: 0.8433
```

QUESTION BASED ON STEP-3 FROM PDF:

Q1. What do you conclude from comparing performances for the models trained using the three different feature types (TF-IDF, pretrained Word2Vec, your trained Word2Vec)?
A1.
TF-IDF Features Are Most Effective:
Both the Perceptron and SVM classifiers achieved the highest testing accuracies when using TF-IDF features (approximately 85.0% for Perceptron and 88.3% for SVM).
This suggests that the sparse, high-dimensional representation from TF-IDF—which directly captures word frequency and discriminative terms—provides strong signals for sentiment classification on this dataset.

Pretrained Word2Vec Features (Google News):
The classifiers using pretrained Word2Vec embeddings performed moderately well, but they consistently lagged behind the TF-IDF baseline (about 76.2% with Perceptron and 81.7% with SVM). One reason for this could be that the Google News model is trained on a very general corpus. While it captures broad semantic relationships, it might not be as finely tuned to the domain-specific vocabulary and context found in Amazon reviews.

Custom Word2Vec Features (Trained on Amazon Reviews):
The custom Word2Vec model, trained directly on the Amazon reviews, shows slightly lower performance with the Perceptron (around 76.8%) but performs better with the SVM (approximately 84.3%), though still not matching TF-IDF. This indicates that while the domain-specific embeddings capture nuances of the review language, they might require further tuning (e.g., more training data, hyperparameter adjustments) to reach the discriminative power of TF-IDF. It also suggests that the linear classifier (SVM) is better at leveraging the semantic information contained in these lower-dimensional embeddings compared to the Perceptron.

Classifier Differences:
Across all feature types, the SVM (or LinearSVC) models outperformed the Perceptron models. This is not unusual in high-dimensional text classification tasks, as SVMs typically handle sparse and high-dimensional data more robustly.

Overall Conclusion:
For this sentiment classification task on Amazon reviews:

TF-IDF stands out as the strongest feature representation, likely because it directly leverages term frequencies and emphasizes discriminative words.
Pretrained Word2Vec offers moderate performance but may not capture domain-specific subtleties as well as TF-IDF.
Custom Word2Vec embeddings, although tailored to the domain, still trail behind TF-IDF—suggesting that further refinement or combination with other features might be necessary to improve performance.

WITH THIS, WE COMPLETE STEP-3

STEP-4: FEED FORWARD NEURAL NETWORKS

part (a)

We have switched to Colab now, as we tried to upgrade pytorch, which made numpy upgrade and failed our gensim, after a day long session of debugging, costed our kernel to DIE!!

So we will be using CPU to compute this neural network, as I cannot pay $10 for colab subscription.

Anyway, the following code converts our numpy arrays to torch tensors and move them to the selected device (CPU in this case) I am following this method so that I can try to fix my CUDA and run on GPU later!

Following the requirements of the pdf, we will slit the dataset, and print the values.

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

X_pretrained = np.vstack(df_binary['AvgPretrained'].values)
y = df_binary['Label'].values

X_train_avg, X_test_avg, y_train, y_test = train_test_split(
    X_pretrained, y, test_size=0.2, random_state=42, stratify=y
)

print("X_train_avg shape:", X_train_avg.shape)
print("X_test_avg shape :", X_test_avg.shape)
print("y_train shape:", y_train.shape)
print("y_test shape :", y_test.shape)
```

```
Using device: cpu
X_train_avg shape: (160000, 300)
X_test_avg shape : (40000, 300)
y_train shape: (160000,)
y_test shape : (40000,)
```

Now, we will Convert Data to PyTorch Tensors and Create DataLoaders

```
X_train_tensor = torch.from_numpy(X_train_avg).float().to(device)
y_train_tensor = torch.from_numpy(y_train).long().to(device)
X_test_tensor = torch.from_numpy(X_test_avg).float().to(device)
y_test_tensor = torch.from_numpy(y_test).long().to(device)

batch_size = 64
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

Now, we will define and execute the MLP model

```
class MLP(nn.Module):
    def __init__(self, input_dim, hidden1, hidden2, output_dim):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden1)
        self.fc2 = nn.Linear(hidden1, hidden2)
        self.fc3 = nn.Linear(hidden2, output_dim)
        self.relu = nn.ReLU()

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

input_dim = 300   # Average Word2Vec vector dimension
hidden1 = 50
hidden2 = 10
output_dim = 2    # For binary classification: 2 classes

model = MLP(input_dim, hidden1, hidden2, output_dim).to(device)
```

Now, we will define loss function and optimizer, and then follow up with a training loop

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

num_epochs = 10
model.train()
for epoch in range(num_epochs):
    running_loss = 0.0
    for batch_X, batch_y in train_loader:
        optimizer.zero_grad()
        outputs = model(batch_X)
        loss = criterion(outputs, batch_y)
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * batch_X.size(0)
    epoch_loss = running_loss / len(train_dataset)
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {epoch_loss:.4f}")
```

```
Epoch [1/10], Loss: 0.4209
Epoch [2/10], Loss: 0.3810
Epoch [3/10], Loss: 0.3690
Epoch [4/10], Loss: 0.3613
Epoch [5/10], Loss: 0.3547
Epoch [6/10], Loss: 0.3496
Epoch [7/10], Loss: 0.3448
Epoch [8/10], Loss: 0.3402
Epoch [9/10], Loss: 0.3370
Epoch [10/10], Loss: 0.3333
```

Finally, for the binary classification, we will evaluate on test dataset

```
model.eval()
all_preds = []
with torch.no_grad():
    for batch_X, _ in test_loader:
        outputs = model(batch_X)
        _, preds = torch.max(outputs, 1)
        all_preds.append(preds)
all_preds = torch.cat(all_preds).cpu().numpy()

mlp_test_accuracy = accuracy_score(y_test, all_preds)
print("MLP Testing Accuracy (Binary, Avg Word2Vec features):", mlp_test_accuracy)
```

```
MLP Testing Accuracy (Binary, Avg Word2Vec features): 0.84435
```

now, let's do this all over again for the ternary dataset preparation, execution and testing!!!

```
df_balanced['Label_Ternary'] = df_balanced['Sentiment'] - 1

# Debug: Print unique labels to confirm
print("Unique ternary labels:", df_balanced['Label_Ternary'].unique())


X_avg = np.vstack(df_balanced['AvgPretrained'].values)
y_ternary = df_balanced['Label_Ternary'].values         # Should have values 0,1,2


X_train_avg, X_test_avg, y_train, y_test = train_test_split(
    X_avg, y_ternary, test_size=0.2, random_state=42, stratify=y_ternary
)

print("X_train_avg shape:", X_train_avg.shape)
print("X_test_avg shape :", X_test_avg.shape)
print("y_train unique labels:", np.unique(y_train))
print("y_test unique labels:", np.unique(y_test))


X_train_tensor = torch.from_numpy(X_train_avg).float().to(device)
y_train_tensor = torch.from_numpy(y_train).long().to(device)
X_test_tensor = torch.from_numpy(X_test_avg).float().to(device)
y_test_tensor = torch.from_numpy(y_test).long().to(device)

batch_size = 64
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)


class MLP(nn.Module):
    def __init__(self, input_dim, hidden1, hidden2, output_dim):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden1)
        self.fc2 = nn.Linear(hidden1, hidden2)
        self.fc3 = nn.Linear(hidden2, output_dim)
        self.relu = nn.ReLU()

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

input_dim = 300
hidden1 = 50
hidden2 = 10
output_dim = 3    # For ternary classification: 3 classes (0,1,2).

model = MLP(input_dim, hidden1, hidden2, output_dim).to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

num_epochs = 10
model.train()
for epoch in range(num_epochs):
    running_loss = 0.0
    for batch_X, batch_y in train_loader:
        optimizer.zero_grad()
        outputs = model(batch_X)
        loss = criterion(outputs, batch_y)
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * batch_X.size(0)
    epoch_loss = running_loss / len(train_dataset)
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {epoch_loss:.4f}")

model.eval()
all_preds = []
with torch.no_grad():
    for batch_X, _ in test_loader:
        outputs = model(batch_X)
        _, preds = torch.max(outputs, 1)
        all_preds.append(preds)
all_preds = torch.cat(all_preds).cpu().numpy()

mlp_test_accuracy = accuracy_score(y_test, all_preds)
print("MLP Testing Accuracy (Ternary, Avg Word2Vec features):", mlp_test_accuracy)
```

```
Unique ternary labels: [1 2 0]
X_train_avg shape: (200000, 300)
X_test_avg shape : (50000, 300)
y_train unique labels: [0 1 2]
y_test unique labels: [0 1 2]
Epoch [1/10], Loss: 0.8018
Epoch [2/10], Loss: 0.7596
Epoch [3/10], Loss: 0.7479
Epoch [4/10], Loss: 0.7392
Epoch [5/10], Loss: 0.7331
Epoch [6/10], Loss: 0.7278
Epoch [7/10], Loss: 0.7239
Epoch [8/10], Loss: 0.7202
Epoch [9/10], Loss: 0.7168
Epoch [10/10], Loss: 0.7140
MLP Testing Accuracy (Ternary, Avg Word2Vec features): 0.68232
```

Now, we move on to PART (B) of STEP-4

We will now define a function to get concatenated Word2Vec feature

This tokenizes the text using NLTK, keeps only alphabetic tokens, selects the first max_tokens tokens, retrieves their word vectors from 'model', and concatenates them into a single vector. Then returns a 1D numpy array of length max_tokens.

```
def get_concatenated_vector(text, model, vector_size=300, max_tokens=10):

    tokens = nltk.word_tokenize(text.lower())
    # Keep only alphabetic tokens
    tokens = [token for token in tokens if token.isalpha()]

    vectors = []
    for token in tokens[:max_tokens]:
        if token in model.key_to_index:
            vectors.append(model[token])
        else:
            vectors.append(np.zeros(vector_size))
    while len(vectors) < max_tokens:
        vectors.append(np.zeros(vector_size))

    # Concatenate all vectors to form a single 1D array
    return np.concatenate(vectors)
```

We will now compute concatenated featured for each review in the binary dataset. We then prepare feature matrix and labels for binary classification, with debug statements (of course), and split the dataset into training and testing.

```
df_binary['ConcatPretrained'] = df_binary['Review'].apply(lambda x: get_concatenated_vector(x, wv))

X_concat = np.vstack(df_binary['ConcatPretrained'].values)
y = df_binary['Label'].values

print("X_concat shape:", X_concat.shape)
print("Unique labels:", np.unique(y))

X_train_concat, X_test_concat, y_train, y_test = train_test_split(
    X_concat, y, test_size=0.2, random_state=42, stratify=y
)

print("X_train_concat shape:", X_train_concat.shape)
print("X_test_concat shape :", X_test_concat.shape)
print("y_train shape:", y_train.shape)
print("y_test shape :", y_test.shape)
```

```
X_concat shape: (200000, 3000)
Unique labels: [0 1]
X_train_concat shape: (160000, 3000)
X_test_concat shape : (40000, 3000)
y_train shape: (160000,)
y_test shape : (40000,)
```

In this next step, we will we will convert data to PyTorch tensors and create dataloaders

```
X_train_tensor = torch.from_numpy(X_train_concat).float().to(device)
y_train_tensor = torch.from_numpy(y_train).long().to(device)
X_test_tensor = torch.from_numpy(X_test_concat).float().to(device)
y_test_tensor = torch.from_numpy(y_test).long().to(device)

batch_size = 64
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

Now, we will define the MLP model for binary classification

```
class MLP(nn.Module):
    def __init__(self, input_dim, hidden1, hidden2, output_dim):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_dim, hidden1)
        self.fc2 = nn.Linear(hidden1, hidden2)
        self.fc3 = nn.Linear(hidden2, output_dim)
        self.relu = nn.ReLU()

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

input_dim = 3000
hidden1 = 50
hidden2 = 10
output_dim = 2

model = MLP(input_dim, hidden1, hidden2, output_dim).to(device)
```

We will now define loss function and optimizer, then create a training loop

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

num_epochs = 10
model.train()
for epoch in range(num_epochs):
    running_loss = 0.0
    for batch_X, batch_y in train_loader:
        optimizer.zero_grad()
        outputs = model(batch_X)
        loss = criterion(outputs, batch_y)
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * batch_X.size(0)
    epoch_loss = running_loss / len(train_dataset)
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {epoch_loss:.4f}")
```

```
Epoch [1/10], Loss: 0.4833
Epoch [2/10], Loss: 0.4309
Epoch [3/10], Loss: 0.3902
Epoch [4/10], Loss: 0.3485
Epoch [5/10], Loss: 0.3049
Epoch [6/10], Loss: 0.2648
Epoch [7/10], Loss: 0.2295
Epoch [8/10], Loss: 0.1980
Epoch [9/10], Loss: 0.1714
Epoch [10/10], Loss: 0.1513
```

Finally, we will now evaluate on the test set, get the test dataset accuracy for comparison.

```
model.eval()
all_preds = []
with torch.no_grad():
    for batch_X, _ in test_loader:
        outputs = model(batch_X)
        _, preds = torch.max(outputs, 1)
        all_preds.append(preds)
all_preds = torch.cat(all_preds).cpu().numpy()

mlp_test_accuracy = accuracy_score(y_test, all_preds)
print("MLP Testing Accuracy (Binary, Concat Pretrained Word2Vec features):", mlp_test_accuracy)
```

```
MLP Testing Accuracy (Binary, Concat Pretrained Word2Vec features): 0.752975
```

With this, we now complete STEP-4 part (a) and part (b), displaying necessary (pdf required) results.

MLP Testing Accuracy: 0.68576

MLP Testing Accuracy: 0.746825

QUESTIONS BASED ON STEP-4:

Q1. What do you conclude by comparing accuracy values you obtain with those obtained in the Simple Models section (note you can compare the accuracy values for binary classification).
A1.
Simple Models (from Step 3):

Simple models using TF-IDF features (with Perceptron and SVM) achieved testing accuracies in the range of approximately 85%–88%.
This indicates that TF-IDF—a high-dimensional, sparse representation capturing word frequency and importance—is very effective for binary sentiment classification in the dataset.

MLP with Word2Vec Features (Step 4):

When using the average Word2Vec vectors as input, your MLP achieved a testing accuracy of about 68.6%. When using the concatenation of the first 10 Word2Vec vectors (resulting in a 3000-dimensional input), the MLP improved to about 74.7% testing accuracy. This shows that concatenating the vectors (thus preserving some order and token-level detail) gives better results than simply averaging—but both methods still underperform compared to the TF-IDF baseline.

Conclusions from the Comparison:

TF-IDF features provide very strong performance for this task, likely because they capture explicit term frequency patterns and discriminative vocabulary, which are highly relevant for sentiment analysis.
Word2Vec embeddings, while they capture semantic relationships, seem to lose some discriminative detail when combined using simple averaging. Concatenation helps to preserve more context and improves performance, but even then, the MLP doesn't match the accuracy obtained by TF-IDF-based simple models.

END OF STEP-4

STEP-5

Sentiment classification using the word embedding (executed previously in step-2)
In this step, we will be converting to lower case and keeping only alpha tokens.

```
def review_to_fixed_vectors(text, model, max_length=50, vector_size=300):
    tokens = nltk.word_tokenize(text.lower())
    tokens = [t for t in tokens if t.isalpha()]

    vectors = []

    for token in tokens[:max_length]:
        if token in model.key_to_index:
            vectors.append(model[token])
        else:
            vectors.append(np.zeros(vector_size))

    while len(vectors) < max_length:
        vectors.append(np.zeros(vector_size))
    return np.stack(vectors)
```

From previous df_binary dataframe usage, we check if there exists a column: 'Review' and 'label' and we verify the status as well.
We then move on to creating and executing the feature matrix and labels for CNN training.

```
df_subset = df_binary.sample(n=50000, random_state=42).copy()

df_subset['FixedVectors'] = df_binary['Review'].apply(lambda x: review_to_fixed_vectors(x, wv, max_length=50, vector_size=300).astype(np
print("Shape of fixed vector for one review:", df_subset['FixedVectors'].iloc[0].shape)

X_fixed = np.stack(df_subset['FixedVectors'].values)
y = df_subset['Label'].values

X_fixed = X_fixed.reshape(X_fixed.shape[0], 1, 50, 300)
print("X_fixed shape:", X_fixed.shape)

X_train, X_test, y_train, y_test = train_test_split(
    X_fixed, y, test_size=0.2, random_state=42, stratify=y
)

print("X_train shape:", X_train.shape)
print("X_test shape :", X_test.shape)
print("y_train shape:", y_train.shape)
print("y_test shape :", y_test.shape)
```

```
Shape of fixed vector for one review: (50, 300)
X_fixed shape: (50000, 1, 50, 300)
X_train shape: (40000, 1, 50, 300)
X_test shape : (10000, 1, 50, 300)
y_train shape: (40000,)
y_test shape : (10000,)
```

We will now convert data to Pytorch tensors and create dataloaders (what we have seen previously)

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

X_train_tensor = torch.from_numpy(X_train).float().to(device)
y_train_tensor = torch.from_numpy(y_train).long().to(device)
X_test_tensor = torch.from_numpy(X_test).float().to(device)
y_test_tensor = torch.from_numpy(y_test).long().to(device)

batch_size = 64
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

```
Using device: cpu
```

In this cide, we define the CNN for binary classification. We use a 2D CNN that processes the input of shape. Finally, we flatten and use a fully-connected layer to produce 2 outputs.

```
class TextCNN(nn.Module):
    def __init__(self, input_dim, max_length, output_dim):
        super(TextCNN, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=1, out_channels=50, kernel_size=(3, input_dim), padding=(1,0))
        self.conv2 = nn.Conv2d(in_channels=50, out_channels=10, kernel_size=(3,1), padding=(1,0))

        self.fc = nn.Linear(10 * max_length, output_dim)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        x = x.view(x.size(0), -1)
        x = self.dropout(x)
        x = self.fc(x)
        return x

input_dim = 300
max_length = 50
output_dim = 2
```

```
model = TextCNN(input_dim, max_length, output_dim).to(device)
```

Here, we define loss function and entropy (also initialize it), and start the training loop.

```
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

num_epochs = 10
model.train()
for epoch in range(num_epochs):
    running_loss = 0.0
    for batch_X, batch_y in train_loader:
        optimizer.zero_grad()
        outputs = model(batch_X)
        loss = criterion(outputs, batch_y)
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * batch_X.size(0)
    epoch_loss = running_loss / len(train_dataset)
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {epoch_loss:.4f}")
```

```
Epoch [1/10], Loss: 0.4434
Epoch [2/10], Loss: 0.3817
Epoch [3/10], Loss: 0.3446
Epoch [4/10], Loss: 0.3188
Epoch [5/10], Loss: 0.2946
Epoch [6/10], Loss: 0.2739
Epoch [7/10], Loss: 0.2469
Epoch [8/10], Loss: 0.2211
Epoch [9/10], Loss: 0.2002
Epoch [10/10], Loss: 0.1778
```

Finally, we evaluate CNN on the test set, and fetch results accordingly.

```
model.eval()
all_preds = []
with torch.no_grad():
    for batch_X, _ in test_loader:
        outputs = model(batch_X)
        _, preds = torch.max(outputs, 1)
        all_preds.append(preds)
all_preds = torch.cat(all_preds).cpu().numpy()

cnn_test_accuracy = accuracy_score(y_test, all_preds)
print("CNN Testing Accuracy (Binary, Concat Word2Vec features):", cnn_test_accuracy)
```

```
CNN Testing Accuracy (Binary, Concat Word2Vec features): 0.8354
```

Here starts the code for TERNARY classification

Below is the same process for what we did above(FOR BINARY, but we change certain aspects of what's required, such as 'output_dim = 3', and so on...

```
def review_to_fixed_vectors(text, model, max_length=50, vector_size=300):

    tokens = nltk.word_tokenize(text.lower())
    tokens = [t for t in tokens if t.isalpha()]

    vectors = []
    for token in tokens[:max_length]:
        if token in model.key_to_index:
            vectors.append(model[token])
        else:
            vectors.append(np.zeros(vector_size))


    while len(vectors) < max_length:
        vectors.append(np.zeros(vector_size))

    return np.stack(vectors)


df_balanced['Label_Ternary'] = df_balanced['Sentiment'] - 1

print("Unique ternary labels:", df_balanced['Label_Ternary'].unique())

df_subset = df_balanced.sample(n=50000, random_state=42).copy()
```

```
df_subset['FixedVectors'] = df_subset['Review'].apply(lambda x: review_to_fixed_vectors(x, wv, max_length=50, vector_size=300).astype(np
print("Shape of fixed vector for one review:", df_subset['FixedVectors'].iloc[0].shape)

X_fixed = np.stack(df_subset['FixedVectors'].values)
y_ternary = df_subset['Label_Ternary'].values

X_fixed = X_fixed.reshape(X_fixed.shape[0], 1, 50, 300)

X_train, X_test, y_train, y_test = train_test_split(
    X_fixed, y_ternary, test_size=0.2, random_state=42, stratify=y_ternary
)

print("X_train shape:", X_train.shape)
print("X_test shape :", X_test.shape)
print("y_train shape:", y_train.shape)
print("y_test shape :", y_test.shape)
print("Unique labels in y_train:", np.unique(y_train))
print("Unique labels in y_test:", np.unique(y_test))
```

```
Unique ternary labels: [1 2 0]
Shape of fixed vector for one review: (50, 300)
X_train shape: (40000, 1, 50, 300)
X_test shape : (10000, 1, 50, 300)
y_train shape: (40000,)
y_test shape : (10000,)
Unique labels in y_train: [0 1 2]
Unique labels in y_test: [0 1 2]
```

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

X_train_tensor = torch.from_numpy(X_train).float().to(device)
y_train_tensor = torch.from_numpy(y_train).long().to(device)
X_test_tensor = torch.from_numpy(X_test).float().to(device)
y_test_tensor = torch.from_numpy(y_test).long().to(device)

batch_size = 64
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
test_dataset = TensorDataset(X_test_tensor, y_test_tensor)

train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

```
Using device: cpu
```

```
class TextCNN(nn.Module):
    def __init__(self, input_dim, max_length, output_dim):
        super(TextCNN, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=1, out_channels=50, kernel_size=(3, input_dim), padding=(1,0))
        self.conv2 = nn.Conv2d(in_channels=50, out_channels=10, kernel_size=(3,1), padding=(1,0))

        self.fc = nn.Linear(10 * max_length, output_dim)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        x = x.view(x.size(0), -1)
        x = self.dropout(x)
        x = self.fc(x)
        return x


input_dim = 300
max_length = 50
output_dim = 3

model = TextCNN(input_dim, max_length, output_dim).to(device)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

num_epochs = 10
model.train()
for epoch in range(num_epochs):
    running_loss = 0.0
    for batch_X, batch_y in train_loader:
        optimizer.zero_grad()
        outputs = model(batch_X)
        loss = criterion(outputs, batch_y)
        loss.backward()
        optimizer.step()
        running_loss += loss.item() * batch_X.size(0)
    epoch_loss = running_loss / len(train_dataset)
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {epoch_loss:.4f}")
```

```
Epoch [1/10], Loss: 0.8652
Epoch [2/10], Loss: 0.7760
Epoch [3/10], Loss: 0.7379
Epoch [4/10], Loss: 0.7031
Epoch [5/10], Loss: 0.6737
Epoch [6/10], Loss: 0.6489
Epoch [7/10], Loss: 0.6204
Epoch [8/10], Loss: 0.5954
Epoch [9/10], Loss: 0.5714
Epoch [10/10], Loss: 0.5396
```

```
model.eval()
all_preds = []
with torch.no_grad():
    for batch_X, _ in test_loader:
        outputs = model(batch_X)
        _, preds = torch.max(outputs, 1)
        all_preds.append(preds)
all_preds = torch.cat(all_preds).cpu().numpy()

cnn_test_accuracy = accuracy_score(y_test, all_preds)
print("CNN Testing Accuracy (Ternary, Concat Word2Vec features):", cnn_test_accuracy)
```

```
CNN Testing Accuracy (Ternary, Concat Word2Vec features): 0.6781
```

As you can see, the accuracy difference is not marginal.

Accuracy of CNN in binary classification: 0.8354 Accuracy of CNN in ternary classification: 0.6781

Due to system RAM issues (run out of storage, I had to reduce the dataset cap to 50,000 and a 32bit int type. 😭 🙏)

WITH THIS, ALL THE SCORES HAVE BEEN REPORTED AND THE HOMEWORK-2 COMES TO AN END