Dependencies

Install

```
!pip install contractions
!pip install ipython-autotime
!pip install fastparquet
```

```
Collecting contractions
  Downloading contractions-0.1.73-py2.py3-none-any.whl (8.7 kB)
Collecting textsearch>=0.0.21 (from contractions)
  Downloading textsearch-0.0.24-py2.py3-none-any.whl (7.6 kB)
Collecting anyascii (from textsearch>=0.0.21->contractions)
  Downloading anyascii-0.3.2-py3-none-any.whl (289 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 289.9/289.9 kB 10.4 MB/s eta 0:00:00
Collecting pyahocorasick (from textsearch>=0.0.21->contractions)
  Downloading pyahocorasick-2.0.0-cp310-cp310-manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_12_x86_64.manylinux2010_x86_64.whl (110 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 110.8/110.8 kB 16.3 MB/s eta 0:00:00
Installing collected packages: pyahocorasick, anyascii, textsearch, contractions
Successfully installed anyascii-0.3.2 contractions-0.1.73 pyahocorasick-2.0.0 textsearch-0.0.24
Collecting ipython-autotime
  Downloading ipython_autotime-0.3.1-py2.py3-none-any.whl (6.8 kB)
Requirement already satisfied: ipython in /usr/local/lib/python3.10/dist-packages (from ipython-autotime) (7.34.0)
Requirement already satisfied: setuptools>=18.5 in /usr/local/lib/python3.10/dist-packages (from ipython->ipython-autotime) (67.7.2)
Collecting jedi>=0.16 (from ipython->ipython-autotime)
  Downloading jedi-0.19.1-py2.py3-none-any.whl (1.6 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.6/1.6 MB 21.8 MB/s eta 0:00:00
Requirement already satisfied: decorator in /usr/local/lib/python3.10/dist-packages (from ipython->ipython-autotime) (4.4.2)
Requirement already satisfied: pickleshare in /usr/local/lib/python3.10/dist-packages (from ipython->ipython-autotime) (0.7.5)
Requirement already satisfied: traitlets>=4.2 in /usr/local/lib/python3.10/dist-packages (from ipython->ipython-autotime) (5.7.1)
Requirement already satisfied: prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from ipython->ipython-autotime) (3.0.39)
Requirement already satisfied: pygments in /usr/local/lib/python3.10/dist-packages (from ipython->ipython-autotime) (2.16.1)
Requirement already satisfied: backcall in /usr/local/lib/python3.10/dist-packages (from ipython->ipython-autotime) (0.2.0)
Requirement already satisfied: matplotlib-inline in /usr/local/lib/python3.10/dist-packages (from ipython->ipython-autotime) (0.1.6)
Requirement already satisfied: pexpect>4.3 in /usr/local/lib/python3.10/dist-packages (from ipython->ipython-autotime) (4.8.0)
Requirement already satisfied: parso<0.9.0,>=0.8.3 in /usr/local/lib/python3.10/dist-packages (from jedi>=0.16->ipython->ipython-autotime) (0.8.3)
Requirement already satisfied: ptyprocess>=0.5 in /usr/local/lib/python3.10/dist-packages (from pexpect>4.3->ipython->ipython-autotime) (0.7.0)
Requirement already satisfied: wcwidth in /usr/local/lib/python3.10/dist-packages (from prompt-toolkit!=3.0.0,!=3.0.1,<3.1.0,>=2.0.0->ipython->ipython-autotime) (0.2.8)
Installing collected packages: jedi, ipython-autotime
Successfully installed ipython-autotime-0.3.1 jedi-0.19.1
Collecting fastparquet
  Downloading fastparquet-2023.8.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (1.7 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.7/1.7 MB 25.4 MB/s eta 0:00:00
Requirement already satisfied: pandas>=1.5.0 in /usr/local/lib/python3.10/dist-packages (from fastparquet) (1.5.3)
Requirement already satisfied: numpy>=1.20.3 in /usr/local/lib/python3.10/dist-packages (from fastparquet) (1.23.5)
Collecting cramjam>=2.3 (from fastparquet)
  Downloading cramjam-2.7.0-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (1.6 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.6/1.6 MB 103.8 MB/s eta 0:00:00
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-packages (from fastparquet) (2023.6.0)
Requirement already satisfied: packaging in /usr/local/lib/python3.10/dist-packages (from fastparquet) (23.2)
Requirement already satisfied: python-dateutil>=2.8.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.5.0->fastparquet) (2.8.2)
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas>=1.5.0->fastparquet) (2023.3.post1)
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.1->pandas>=1.5.0->fastparquet) (1.16.0)
Installing collected packages: cramjam, fastparquet
Successfully installed cramjam-2.7.0 fastparquet-2023.8.0
```

Imports

```python
import os
import re
import shutil
import unicodedata
import multiprocessing

import warnings
warnings.filterwarnings("ignore")

import numpy as np
import pandas as pd
import requests

import nltk
from nltk.corpus import stopwords, wordnet
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize

nltk.download('punkt', quiet=True)
nltk.download('wordnet', quiet=True)
nltk.download('stopwords', quiet=True)
nltk.download('averaged_perceptron_tagger', quiet=True)

import contractions

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

from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score

from sklearn.linear_model import Perceptron
from sklearn.svm import LinearSVC

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data.sampler import RandomSampler, BatchSampler
from torch.utils.data import Dataset, DataLoader

from tqdm.notebook import tqdm

%load_ext autotime
```

```
time: 309 µs (started: 2023-10-20 19:58:20 +00:00)
```

Config

Set up important configuration parameters and file paths for the project, making it easy to manage various settings and paths from one centralized location

Place the `amazon_reviews_us_Office_Products_v1_00.tsv.gz` at the same level as noetbook

```
# os.chdir("/content/drive/MyDrive/Colab Notebooks/CSCI544/HW3")
os.environ['CUDA_LAUNCH_BLOCKING'] = "1"

CURRENT_DIR = os.getcwd()


class DatasetConfig:
    RANDOM_STATE = 34
    TEST_SPLIT = 0.2
    N_SAMPLES_EACH_CLASS = 50000
    DATA_PATH = os.path.join(
        CURRENT_DIR, "amazon_reviews_us_Office_Products_v1_00.tsv.gz"
    )
    PROCESSED_DATA_PATH = os.path.join(
        CURRENT_DIR, "amazon_review_processed_sentiment_analysis.parquet"
    )
    PREPROCESSED_DATA_PATH = os.path.join(
        CURRENT_DIR, "amazon_review_preprocessed_sentiment_analysis.parquet"
    )
    BUILD_NEW = True
    if os.path.exists(PROCESSED_DATA_PATH) and os.path.exists(PREPROCESSED_DATA_PATH):
        BUILD_NEW = False


class Word2VecConfig:
    PRETRAINED_MODEL = "word2vec-google-news-300"
    PRETRAINED_DEFAULT_SAVE_PATH = os.path.join(
        gensim.downloader.BASE_DIR, PRETRAINED_MODEL, f"{PRETRAINED_MODEL}.gz"
    )
    PRETRAINED_MODEL_SAVE_PATH = os.path.join(
        CURRENT_DIR, PRETRAINED_MODEL, f"{PRETRAINED_MODEL}.gz"
    )
    WINDOW_SIZE = 13
    MAX_LENGTH = 300
    EMBEDDING_SIZE = 300
    MIN_WORD_COUNT = 9
    CUSTOM_MODEL_PATH = os.path.join(CURRENT_DIR, "word2vec-custom.model")
```

```
time: 2.43 ms (started: 2023-10-20 22:01:48 +00:00)
```

Helper Functions

Download & Save Pretrained model

- Run the `api.load()` once and copied the model from temporary path to local drive for fast loading of model in memory.

References:

1. Faster way to load word2vec model
2. Tutorial

```
def load_pretrained_model():
    if not os.path.exists(Word2VecConfig.PRETRAINED_MODEL_SAVE_PATH):
        # Create a directory if it doesn't exist
        os.makedirs(Word2VecConfig.PRETRAINED_MODEL, exist_ok=True)
        # Download the model embeddings
        pretrained_model = api.load(Word2VecConfig.PRETRAINED_MODEL, return_path=True)
        # Copy & save the embeddings file
        shutil.copyfile(
            Word2VecConfig.PRETRAINED_DEFAULT_SAVE_PATH, Word2VecConfig.PRETRAINED_MODEL
        )
    else:
        pretrained_model = gensim.models.keyedvectors.KeyedVectors.load_word2vec_format(
            Word2VecConfig.PRETRAINED_MODEL_SAVE_PATH, binary=True
        )
    return pretrained_model


# Load the pretrained model
pretrained_model = load_pretrained_model()
```

```
time: 1min 31s (started: 2023-10-20 06:45:23 +00:00)
```

Accelarator Configuration

```
def get_device():
    if torch.cuda.is_available():
        # Check if GPU is available
        return torch.device("cuda")
    else:
        # Use CPU if no GPU or TPU is available
        return torch.device("cpu")

device = get_device()
device
```

```
device(type='cpu')time: 5.32 ms (started: 2023-10-20 19:58:39 +00:00)
```

Download Data

Checks if a file specified by `DatasetConfig.DATA_PATH` exists. If not, it downloads the file from a given URL and saves it with the same name. If the file already exists, it prints a message indicating so

```
if not os.path.exists(DatasetConfig.DATA_PATH):
    url = (
        "https://web.archive.org/web/20201127142707if_/https://s3.amazonaws.com/amazon-reviews-pds"
        "/tsv/amazon_reviews_us_Office_Products_v1_00.tsv.gz"
    )
    file_name = DatasetConfig.DATA_PATH

    # Stream and download heavy file in chunks
    with requests.get(url, stream=True) as response:
        with open(file_name, "wb") as file:
            for chunk in response.iter_content(chunk_size=8192):
                file.write(chunk)

    print(f"Downloaded file '{os.path.relpath(file_name)}' successfully.")
```

```
else:
```

```
File '/content/drive/MyDrive/Colab Notebooks/CSCI544/HW3/amazon_reviews_us_Office_Products_v1_00.tsv.gz' already exists.
time: 1.93 ms (started: 2023-10-20 19:58:42 +00:00)
```

Dataset Preparation

This code provides a pipeline for processing and preparing a dataset for sentiment analysis:

1. `LoadData` class loads a dataset from a specified path, keeping only relevant columns.
2. `ProcessData` class performs the following tasks:
    - Converts star ratings to numeric values.
    - Classifies sentiments based on star ratings (1 for negative, 2 for positive).
    - Balances the dataset by sampling an equal number of samples for both sentiments.
3. `CleanText` class defines various text cleaning operations:
    - Removing non-ASCII characters.
    - Expanding contractions.
    - Removing email addresses, URLs, and HTML tags.
    - Lowercasing and stripping spaces.
4. `clean_and_process_data` function executes the entire data processing pipeline:
    - Loads the data.
    - Applies basic processing.
    - Balances the dataset.
    - Cleans the text.
    - Tokenizes the reviews.
5. `preprocess_review_body` function generates word embeddings for each word in a review using a pre-trained Word2Vec model.
6. `get_reviews_dataset` function handles the entire data preprocessing and embedding generation process. It checks if the preprocessed data already exists, and if not, it performs the data preprocessing and saves the preprocessed data in Parquet format.

Overall, this pipeline ensures that the dataset is properly loaded, cleaned, processed, balanced, and transformed into embeddings suitable for sentiment analysis.

> Note:
>
> - Parquet format is efficient for storage.
> - Storing data to avoid running the pipeline and embedding generation process all over again.
> - Provides a ready-to-use dataset for sentiment analysis tasks, allowing for quicker experimentation and model training

Read and Process

```
class LoadData:
    @staticmethod
    def load_data(path):
        df = pd.read_csv(
            path,
            sep="\t",
            usecols=["review_headline", "review_body", "star_rating"],
            on_bad_lines="skip",
            memory_map=True,
        )
        return df


class ProcessData:
    @staticmethod
    def filter_columns(df):
        return df.loc[:, ["review_body", "star_rating"]]

    @staticmethod
    def convert_star_rating(df):
        df["star_rating"] = pd.to_numeric(df["star_rating"], errors="coerce")
        df.dropna(subset=["star_rating"], inplace=True)
        return df

    @staticmethod
    def classify_sentiment(df):
        df["sentiment"] = df["star_rating"].apply(lambda x: 1 if x <= 3 else 2)
        return df

    @staticmethod
    def sample_data(df, n_samples, random_state):
        sampled_df = pd.concat(
            [
                df.query("sentiment==1").sample(n=n_samples, random_state=random_state),
                df.query("sentiment==2").sample(n=n_samples, random_state=random_state),
            ],
            ignore_index=True,
        ).sample(frac=1, random_state=random_state, ignore_index=True)

        sampled_df.drop(columns=["star_rating"], inplace=True)
        return sampled_df


class CleanText:
    @staticmethod
    def unicode_to_ascii(s):
        return "".join(
            c for c in unicodedata.normalize("NFD", s) if unicodedata.category(c) != "Mn"
        )

    @staticmethod
    def expand_contractions(text):
        """Expand contraction for eg., wouldn't => would not"""
        return contractions.fix(text)

    @staticmethod
    def remove_email_addresses(text):
        return re.sub(r"[a-zA-Z0-9_\-\.]+@[a-zA-Z0-9_\-\.]+\.[a-zA-Z]{2,5}", "", text)

    @staticmethod
    def remove_urls(text):
        return re.sub(r"\bhttps?:\/\/\S+|www\.\S+", "", text)

    @staticmethod
```

```
    def remove_html_tags(text):
        return re.sub(r"<.*?>", "", text)

    @staticmethod
    def clean_text(text):
        text = text.lower().strip()
        text = CleanText.unicode_to_ascii(text)
        # text = CleanText.remove_email_addresses(text)
        # text = CleanText.remove_urls(text)
        text = CleanText.remove_html_tags(text)
        text = CleanText.expand_contractions(text)

        # creating a space between a word and the punctuation following it
        # text = re.sub(r"([?.!,¿])", r" \1 ", text)
        # text = re.sub(r'[" "]+', " ", text)

        # removes all non-alphabetical characters
        # text = re.sub(r"[^a-zA-Z\s]+", "", text)

        # remove extra spaces
        # text = re.sub(" +", " ", text)
        return text


def clean_and_process_data(path):
    df = LoadData.load_data(path)

    # Basic processing
    df_filtered = ProcessData.filter_columns(df)
    df_filtered = ProcessData.convert_star_rating(df_filtered)
    df_filtered = ProcessData.classify_sentiment(df_filtered)

    balanced_df = ProcessData.sample_data(
        df_filtered, DatasetConfig.N_SAMPLES_EACH_CLASS, DatasetConfig.RANDOM_STATE
    )

    # Clean data
    balanced_df.dropna(inplace=True)
    balanced_df["review_body"] = balanced_df["review_body"].astype(str)
    balanced_df["review_body"] = balanced_df["review_body"].apply(CleanText.clean_text)
    # Drop reviews that are empty
    balanced_df = balanced_df.loc[balanced_df["review_body"].str.strip() != ""]

    # Tokenize Reviews
    balanced_df["review_body"] = balanced_df["review_body"].apply(word_tokenize)
    return balanced_df


def preprocess_review_body(text, word2vec_model, topn=None):
    embeddings = [word2vec_model[word] for word in text if word in word2vec_model]

    if topn is not None:
        embeddings = np.concatenate(embeddings[:topn], axis=0)
    else:
        embeddings = np.mean(embeddings, axis=0)
    return embeddings


def get_reviews_dataset(new=False):
    if new or not os.path.exists(DatasetConfig.DATA_PATH):
        balanced_df = clean_and_process_data(DatasetConfig.DATA_PATH)
        balanced_df.to_parquet(DatasetConfig.PROCESSED_DATA_PATH, index=False)

        # Preprocess data and generate word2vec embeddings Avg and top 10
        balanced_df["embeddings"] = balanced_df["review_body"].apply(
            lambda text: preprocess_review_body(text, pretrained_model, topn=None)
        )
        # Drop rows with NaN embeddings
        balanced_df.dropna(subset=["embeddings"], inplace=True)

        balanced_df["embeddings_top_10"] = balanced_df["review_body"].apply(
            lambda text: preprocess_review_body(text, pretrained_model, topn=10)
        )

        balanced_df.to_parquet(DatasetConfig.PREPROCESSED_DATA_PATH, index=False)
    else:
        balanced_df = pd.read_parquet(
            DatasetConfig.PREPROCESSED_DATA_PATH,
            # engine="fastparquet"
        )
    return balanced_df
```

time: 2.38 ms (started: 2023-10-20 19:59:52 +00:00)

```
balanced_df = get_reviews_dataset(
    new=DatasetConfig.BUILD_NEW
)
print("Total Records:", balanced_df.shape)
balanced_df.head(10)
```

Total Records: (99862, 4)

	review_body	sentiment	embeddings	embeddings_top_10
0	[i, set, up, a, photo, booth, at, my, sister, ...	2	[0.016994974, 0.024544675, -0.010975713, 0.093...	[-0.22558594, -0.01953125, 0.09082031, 0.23730...
1	[like, everyone, else, ,, i, like, saving, mon...	1	[0.044110615, 0.036876563, 0.0371785, 0.113560...	[0.103515625, 0.13769531, -0.0029754639, 0.181...
2	[the, pen, is, perfect, what, i, want, !, howe...	2	[0.026102701, 0.029064532, 0.010800962, 0.0622...	[0.080078125, 0.10498047, 0.049804688, 0.05346...
3	[i, think, they, are, too, expensive, for, the...	1	[-0.0039075767, 0.032967318, 0.02339106, 0.113...	[-0.22558594, -0.01953125, 0.09082031, 0.23730...
4	[black, is, working, wonderfully, ,, and, both...	1	[0.034285888, 0.013478661, 0.041618653, 0.1132...	[0.10498047, 0.018432617, 0.008972168, -0.0128...
5	[i, have, problems, with, the, moveable, tab, ...	1	[0.010405041, 0.026173819, 0.03433373, 0.09698...	[-0.22558594, -0.01953125, 0.09082031, 0.23730...
6	[this, printer, sucks, !, it, started, out, wo...	1	[0.05581854, 0.035414256, 0.047512088, 0.09278...	[0.109375, 0.140625, -0.03173828, 0.16601562, ...

Review Body stats

Mean number of words = 66

Median number of words = 37

Limiting sequence length for RNN based embeddings = 45

```
balanced_df["review_body"].apply(len).describe().round(2)
```

```
count    99862.00
mean        65.94
std        100.17
min          1.00
25%         20.00
50%         37.00
75%         76.00
max       4847.00
Name: review_body, dtype: float64time: 272 ms (started: 2023-10-20 20:00:20 +00:00)
```

Train, Valid and Test Spilts

```
# Create train and temp sets (80% train, 20% valid + test)
train_df, valid_df = train_test_split(
    balanced_df,
    test_size=0.20,
    random_state=DatasetConfig.RANDOM_STATE,
    stratify=balanced_df["sentiment"]
)

# Create valid and test sets (15% valid, 5% test)
valid_df, test_df = train_test_split(
    valid_df,
    test_size=0.25,  # 25% of 20% is 5%
    random_state=DatasetConfig.RANDOM_STATE,
    stratify=valid_df["sentiment"]
)
```

```
time: 83.1 ms (started: 2023-10-20 20:04:14 +00:00)
```

Word Embedding

Semantic similarity examples with pretrained embeddings

```
# Example 1: King - Man + Woman = Queen
result = pretrained_model.most_similar(positive=['woman', 'king'], negative=['man'])
print(f"Semantic Similarity: {result[0][0]}")

# Example 2: excellent ~ outstanding
result = pretrained_model.similarity('excellent', 'outstanding')
print(f"Semantic Similarity: {result}")

# Example 3: Paris - France + Italy = Milan
result = pretrained_model.most_similar(positive=['Italy', 'Paris'], negative=['France'])
print(f"Semantic Similarity: {result[0][0]}")

# Example 4: Car - Wheel + Boat = Yacht
result = pretrained_model.most_similar(positive=['Boat', 'Car'], negative=['Wheel'])
print(f"Semantic Similarity: {result[0][0]}")

# Example 5: Delicious ~ Tasty
result = pretrained_model.similarity('Delicious', 'Tasty')
print(f"Semantic Similarity: {result}")

# Example 6: Computer ~ Plant
result = pretrained_model.similarity('Computer', 'Plant')
print(f"Semantic Similarity: {result}")

# Example 7: Cat ~ Dog
result = pretrained_model.similarity('Cat', 'Dog')
print(f"Semantic Similarity: {result}")
```

```
Semantic Similarity: queen
Semantic Similarity: 0.5567485690116882
Semantic Similarity: Milan
Semantic Similarity: Yacht
Semantic Similarity: 0.5718502402305603
Semantic Similarity: 0.04445184767246246
Semantic Similarity: 0.6061107516288757
time: 9.78 s (started: 2023-10-18 21:26:10 +00:00)
```

```
del pretrained_model
```

```
time: 479 µs (started: 2023-10-20 06:47:24 +00:00)
```

Custom Word2Vec Embeddings Generation

```
sentences=train_df["review_body"].apply(lambda x: x.tolist()).tolist()

# Train Word2Vec model
w2v_model_custom = Word2Vec(
    sentences=sentences,
    vector_size=Word2VecConfig.MAX_LENGTH,
    window=Word2VecConfig.WINDOW_SIZE,
    min_count=Word2VecConfig.MIN_WORD_COUNT,
    workers=multiprocessing.cpu_count()
)

# Save the model
w2v_model_custom.save(Word2VecConfig.CUSTOM_MODEL_PATH)
```

```
time: 1min 30s (started: 2023-10-18 21:36:09 +00:00)
```

Test Custom Embeddings

```
# Load the custom model
w2v_model_custom = Word2Vec.load(Word2VecConfig.CUSTOM_MODEL_PATH)

# Example 1: King - Man + Woman = Queen
res = w2v_model_custom.wv.most_similar(positive=['woman', 'king'], negative=['man'])
print(f"Semantic Similarity (Custom Model): {res[0]}")

# Example 2: excellent ~ outstanding
res = w2v_model_custom.wv.similarity('excellent', 'outstanding')
print(f"Semantic Similarity (Custom Model): {res}")
```

```
Semantic Similarity (Custom Model): ('queen', 0.5723455548286438)
Semantic Similarity (Custom Model): 0.7957370281219482
time: 241 ms (started: 2023-10-18 21:37:47 +00:00)
```

Conclusion

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?

1. **Custom-trained Word2Vec Model:**
    - **Strengths:**
        - Captures domain-specific relationships and nuances as it trained on very specific dataset.
    - **Weaknesses:**
        - It may not perform as well on tasks outside of its training domain.
        - The quality of embeddings heavily depends on the dataset used for training.
        - For example, if the dataset is small or not representative of the overall language, the embeddings may be less reliable.
2. **Pretrained "word2vec-google-news-300" Model:**
    - **Strengths:**
        - This model has been pretrained on a massive corpus of text from various domains, making it highly versatile and capable of capturing a wide range of semantic relationships.
        - It can generalize well to different tasks and domains.
    - **Weaknesses:**
        - While it provides strong generalization, it may not capture domain-specific relationships as effectively as a model trained on domain-specific data.

- The semantic similarity score is higher for the pretrained model compared to the custom model. This indicates that the pretrained model is better at encoding semantic similarities between words.
- The custom Word2Vec model, which was trained on the provided dataset, may not have had access to as diverse and extensive a corpus as the pretrained model. This can lead to limitations in its ability to generalize and capture nuanced semantic relationships.

```python
del w2v_model_custom, res, sentences
```

```
time: 319 µs (started: 2023-10-19 01:33:10 +00:00)
```

Simple Models

```python
def evaluate_model(model, X_test, y_test):
    # Predict on the test set
    y_pred = model.predict(X_test)

    # Calculate evaluation metrics
    precision = precision_score(y_test, y_pred, average="binary")
    recall = recall_score(y_test, y_pred, average="binary")
    f1 = f1_score(y_test, y_pred, average="binary")
    accuracy = accuracy_score(y_test, y_pred)

    return precision, recall, f1, accuracy


def train_and_evaluate_model(model_class, X_train, y_train, X_test, y_test, **model_params):
    # Initialize model
    model = model_class(**model_params)

    # Train the model
    model.fit(X_train, y_train)

    # Evaluate model
    precision, recall, f1, accuracy = evaluate_model(model, X_test, y_test)
    return model, precision, recall, f1, accuracy
```

```
time: 1.21 ms (started: 2023-10-20 06:47:33 +00:00)
```

```python
X_train = np.vstack(train_df["embeddings"])
y_train = train_df["sentiment"]
X_test = np.vstack(test_df["embeddings"])
y_test = test_df["sentiment"]
```

```
time: 338 ms (started: 2023-10-20 20:04:40 +00:00)
```

SVM

Params	Precision	Recall	F1	Accuracy	Features Used
LinearSVC(C=0.1, max_iter=10000)	0.7997	0.8671	0.8320	0.8321	Word2Vec
LinearSVC(max_iter=10000)	0.8045	0.8623	0.8324	0.8262	Word2Vec
LinearSVC(C=0.01, max_iter=15000)	0.7836	0.8835	0.8305	0.8281	Word2Vec

```python
# Train and evaluate LinearSVC model
(
    _,
    precision_svc,
    recall_svc,
    f1_svc,
    acc_svc
) = train_and_evaluate_model(
    LinearSVC,
    X_train, y_train, X_test, y_test,
    max_iter=10000,
    # C=0.1
)

print(f'Precision Recall F1 Accuracy (LinearSVC): {precision_svc:.4f} {recall_svc:.4f} {f1_svc:.4f} {acc_svc:.4f}')
```

```
Precision Recall F1 Accuracy (LinearSVC): 0.8045 0.8623 0.8324 0.8262
time: 28.2 s (started: 2023-10-19 01:31:55 +00:00)
```

Perceptron

Params	Precision	Recall	F1	Accuracy	Features Used
Perceptron(eta0=0.01, max_iter=5000, penalty='elasticnet', warm_start=True)	0.7693	0.8778	0.8200	0.8071	Word2Vec

Params	Precision	Recall	F1	Accuracy	Features Used
Perceptron(max_iter=5000)	0.7786	0.8613	0.8179	0.8110	Word2Vec
Perceptron()	0.7786	0.8613	0.8179	0.8110	Word2Vec
Perceptron(eta0=0.1, max_iter=5000, penalty='elasticnet', warm_start=True)	0.5977	0.9844	0.7438	0.6655	Word2Vec
Perceptron(eta0=0.001, max_iter=10000, penalty='l2')	0.7367	0.9114	0.8148	0.7849	Word2Vec
Perceptron(eta0=0.01, max_iter=10000, penalty='l2', warm_start=True)	0.7653	0.8789	0.8181	0.8002	Word2Vec
Perceptron(eta0=0.01, penalty='l1', warm_start=True)	0.6133	0.9813	0.7548	0.6832	Word2Vec

```python
# Train and evaluate Perceptron model using BoW features
(
    _,
    precision_perceptron,
    recall_perceptron,
    f1_perceptron,
    acc_perceptron
) = train_and_evaluate_model(
    Perceptron,
    X_train, y_train, X_test, y_test,
    max_iter=5000,
    eta0=0.01,
    warm_start=True,
    penalty="elasticnet"
)

print(f'Precision Recall F1 (Perceptron): {precision_perceptron:.4f} {recall_perceptron:.4f} {f1_perceptron:.4f} {acc_perceptron:.4f}')
```

```
Precision Recall F1 (Perceptron): 0.7693 0.8778 0.8200 0.8071
time: 1.1 s (started: 2023-10-19 01:32:24 +00:00)
```

With TFIDF Features

Homework 1 Script Edited

```python
# @title Homework 1 Script Edited

%%writefile HW1-CSCI544-wo-neg-sw.py
# Python Version: 3.10.12

import re
import unicodedata

import warnings

warnings.filterwarnings("ignore")

import numpy as np
import pandas as pd

import nltk
from nltk.corpus import stopwords, wordnet
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize

nltk.download("punkt", quiet=True)
nltk.download("wordnet", quiet=True)
nltk.download("stopwords", quiet=True)
nltk.download("averaged_perceptron_tagger", quiet=True)

import contractions

from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score

from sklearn.linear_model import Perceptron
from sklearn.svm import LinearSVC


class Config:
    RANDOM_STATE = 56
    DATA_PATH = "amazon_reviews_us_Office_Products_v1_00.tsv.gz"
    TEST_SPLIT = 0.2
    N_SAMPLES_EACH_CLASS = 50000
    NUM_TFIDF_FEATURES = 5000
    NUM_BOW_FEATURES = 5000


class DataLoader:
    @staticmethod
    def load_data(path):
        df = pd.read_csv(
            path,
            sep="\t",
            usecols=["review_headline", "review_body", "star_rating"],
            on_bad_lines="skip",
            memory_map=True,
        )
        return df


class DataProcessor:
    @staticmethod
    def filter_columns(df):
        return df.loc[:, ["review_body", "star_rating"]]

    @staticmethod
    def convert_star_rating(df):
        df["star_rating"] = pd.to_numeric(df["star_rating"], errors="coerce")
        df.dropna(subset=["star_rating"], inplace=True)
        return df

    @staticmethod
    def classify_sentiment(df):
        df["sentiment"] = df["star_rating"].apply(lambda x: 1 if x <= 3 else 2)
        return df

    @staticmethod
    def sample_data(df, n_samples, random_state):
        sampled_df = pd.concat(
            [
                df.query("sentiment==1").sample(n=n_samples, random_state=random_state),
                df.query("sentiment==2").sample(n=n_samples, random_state=random_state),
            ],
            ignore_index=True,
```

```
        ).sample(frac=1, random_state=random_state)

        sampled_df.drop(columns=["star_rating"], inplace=True)
        return sampled_df


class TextCleaner:
    @staticmethod
    def unicode_to_ascii(s):
        return "".join(
            c for c in unicodedata.normalize("NFD", s) if unicodedata.category(c) != "Mn"
        )

    @staticmethod
    def expand_contractions(text):
        return contractions.fix(text)

    @staticmethod
    def remove_email_addresses(text):
        return re.sub(r"[a-zA-Z0-9_\-\.]+@[a-zA-Z0-9_\-\.]+\.[a-zA-Z]{2,5}", " ", text)

    @staticmethod
    def remove_urls(text):
        return re.sub(r"\bhttps?:\/\/\S+|www\.\S+", " ", text)

    @staticmethod
    def remove_html_tags(text):
        return re.sub(r"<.*?>", "", text)

    @staticmethod
    def clean_text(text):
        text = TextCleaner.unicode_to_ascii(text.lower().strip())
        # replacing email addresses with empty string
        text = TextCleaner.remove_email_addresses(text)
        # replacing urls with empty string
        text = TextCleaner.remove_urls(text)
        # Remove HTML tags
        text = TextCleaner.remove_html_tags(text)
        # Expand contraction for eg., wouldn't => would not
        text = TextCleaner.expand_contractions(text)
        # creating a space between a word and the punctuation following it
        text = re.sub(r"([?.!,¿])", r" \1 ", text)
        text = re.sub(r'[" "]+', " ", text)
        # removes all non-alphabetical characters
        text = re.sub(r"[^a-zA-Z\s]+", "", text)
        # remove extra spaces
        text = re.sub(" +", " ", text)
        text = text.strip()
        return text


class TextPreprocessor:
    lemmatizer = WordNetLemmatizer()

    @staticmethod
    def get_stopwords_pattern():
        # Stopword list
        og_stopwords = set(stopwords.words("english"))

        # Define a list of negative words to remove
        neg_words = ["no", "not", "nor", "neither", "none", "never", "nobody", "nowhere"]
        custom_stopwords = [word for word in og_stopwords if word not in neg_words]
        pattern = re.compile(r"\b(" + r"|".join(custom_stopwords) + r")\b\s*")
        return pattern

    @staticmethod
    def pos_tagger(tag):
        if tag.startswith("J"):
            return wordnet.ADJ
        elif tag.startswith("V"):
            return wordnet.VERB
        elif tag.startswith("N"):
            return wordnet.NOUN
        elif tag.startswith("R"):
            return wordnet.ADV
        else:
            return None

    @staticmethod
    def lemmatize_text_using_pos_tags(text):
        words = nltk.pos_tag(word_tokenize(text))
        words = map(lambda x: (x[0], TextPreprocessor.pos_tagger(x[1])), words)
        lemmatized_words = [
            TextPreprocessor.lemmatizer.lemmatize(word, tag) if tag else word for word, tag in words
        ]
        return " ".join(lemmatized_words)

    @staticmethod
    def lemmatize_text(text):
        words = word_tokenize(text)
        lemmatized_words = [TextPreprocessor.lemmatizer.lemmatize(word) for word in words]
        return " ".join(lemmatized_words)

    pattern = get_stopwords_pattern()

    @staticmethod
    def preprocess_text(text):
        # replacing all the stopwords
        text = TextPreprocessor.pattern.sub("", text)
        text = TextPreprocessor.lemmatize_text(text)
        return text


clean_text_vect = np.vectorize(TextCleaner.clean_text)
preprocess_text_vect = np.vectorize(TextPreprocessor.preprocess_text)


def clean_and_process_data(path):
    df = DataLoader.load_data(path)
    df_filtered = DataProcessor.filter_columns(df)
    df_filtered = DataProcessor.convert_star_rating(df_filtered)
    df_filtered = DataProcessor.classify_sentiment(df_filtered)

    balanced_df = DataProcessor.sample_data(
        df_filtered, Config.N_SAMPLES_EACH_CLASS, Config.RANDOM_STATE
    )

    balanced_df["review_body"] = balanced_df["review_body"].astype(str)

    # Clean data
```

```python
    # Clean data
    # avg_len_before_clean = balanced_df["review_body"].apply(len).mean()
    balanced_df["review_body"] = balanced_df["review_body"].apply(clean_text_vect)
    # Drop reviews that are empty
    balanced_df = balanced_df.loc[balanced_df["review_body"].str.strip() != ""]
    # avg_len_after_clean = balanced_df["review_body"].apply(len).mean()

    # Preprocess data
    # avg_len_before_preprocess = avg_len_after_clean
    balanced_df["review_body"] = balanced_df["review_body"].apply(preprocess_text_vect)
    # avg_len_after_preprocess = balanced_df["review_body"].apply(len).mean()

    # Print Results
    # print(f"{avg_len_before_clean:.2f}, {avg_len_after_clean:.2f}")
    # print(f"{avg_len_before_preprocess:.2f}, {avg_len_after_preprocess:.2f}")

    return balanced_df


def evaluate_model(model, X_test, y_test):
    # Predict on the test set
    y_pred = model.predict(X_test)

    # Calculate evaluation metrics
    precision = precision_score(y_test, y_pred, average="binary")
    recall = recall_score(y_test, y_pred, average="binary")
    f1 = f1_score(y_test, y_pred, average="binary")
    accuracy = accuracy_score(y_test, y_pred)

    return precision, recall, f1, accuracy


def train_and_evaluate_model(model_class, X_train, y_train, X_test, y_test, **model_params):
    # Initialize model
    model = model_class(**model_params)

    # Train the model
    model.fit(X_train, y_train)

    # Evaluate model
    precision, recall, f1, accuracy = evaluate_model(model, X_test, y_test)
    return model, precision, recall, f1, accuracy


def main():
    balanced_df = clean_and_process_data(Config.DATA_PATH)

    # Splitting the reviews dataset
    X_train, X_test, y_train, y_test = train_test_split(
        balanced_df["review_body"],
        balanced_df["sentiment"],
        test_size=Config.TEST_SPLIT,
        random_state=Config.RANDOM_STATE,
    )

    # Feature Extraction
    tfidf_vectorizer = TfidfVectorizer(max_features=Config.NUM_TFIDF_FEATURES)
    X_train_tfidf = tfidf_vectorizer.fit_transform(X_train)
    X_test_tfidf = tfidf_vectorizer.transform(X_test)

    # Train and evaluate Perceptron model using TF-IDF features
    (
        _,
        precision_perceptron_tfidf,
        recall_perceptron_tfidf,
        f1_perceptron_tfidf,
        acc_perceptron_tfidf
    ) = train_and_evaluate_model(
        Perceptron, X_train_tfidf, y_train, X_test_tfidf, y_test, max_iter=4000
    )

    # Train and evaluate SVM model using TF-IDF features
    (
        _,
        precision_svm_tfidf,
        recall_svm_tfidf,
        f1_svm_tfidf,
        acc_svm_tfidf
    ) = train_and_evaluate_model(
        LinearSVC, X_train_tfidf, y_train, X_test_tfidf, y_test, max_iter=2500
    )

    # Print the results
    print("Precision Recall F1-Score Accuracy")
    print("Perceptron")
    print(
        f"{precision_perceptron_tfidf:.4f} {recall_perceptron_tfidf:.4f} {f1_perceptron_tfidf:.4f} {acc_perceptron_tfidf:.4f}"
    )

    print("SVM: LinearSVC")
    print(f"{precision_svm_tfidf:.4f} {recall_svm_tfidf:.4f} {f1_svm_tfidf:.4f} {acc_svm_tfidf:.4f}")


if __name__ == "__main__":
    main()
```

```
Overwriting HW1-CSCI544-wo-neg-sw.py
time: 50.9 ms (started: 2023-10-19 22:02:47 +00:00)
```

```
!python HW1-CSCI544-wo-neg-sw.py
```

```
Precision Recall F1-Score Accuracy
Perceptron
0.7637 0.8702 0.8135 0.7998
SVM: LinearSVC
0.8573 0.8602 0.8588 0.8581
time: 4min 1s (started: 2023-10-19 22:02:52 +00:00)
```

Conclusion

Best Accuracies

Model	Accuracy	Features Used
Perceptron	0.8110	Word2Vec
LinearSVC	0.8321	Word2Vec
Perceptron	0.7998	TF-IDF

Model	Accuracy	Features Used
LinearSVC	0.8581	TF-IDF

1. LinearSVC outperforms Perceptron for both feature types (Word2Vec and TF-IDF).
    - LinearSVC is better suited for this classification task compared to Perceptron.
2. When using Word2Vec features, both Perceptron and LinearSVC achieve lower accuracy compared to when using TF-IDF features.
    - Word2Vec embeddings might not be as effective for this specific sentiment classification task as compared to TF-IDF vectors.
3. The LinearSVC model performs particularly well with TF-IDF features, achieving an accuracy of 85.81%.
    - TF-IDF vectors are highly effective in capturing important information for sentiment classification in this dataset.

Overall, based on the provided performance metrics, it seems that TF-IDF features are more effective for this sentiment classification task compared to the Word2Vec embeddings. However, it's important to note that the effectiveness of features can vary depending on the specific dataset and task.

```
del balanced_df
del X_train, y_train, X_test, y_test
```

```
time: 372 µs (started: 2023-10-20 20:04:48 +00:00)
```

Create Pytorch Dataset

- Custom pytorch dataset for on-the-fly processing an d efficient resource utilization
- Each sample in this dataset includes embeddings and their corresponding target label. The label is adjusted by subtracting 1 from the label value in the DataFrame
- Using `DataLoader`'s
    - Used to load and manage batches of data during the training process.
    - Handle tasks like shuffling, batching, and parallel data loading, making it easier to feed data to the model.

```
class AmazonReviewsSentimentDataset(Dataset):
    def __init__(
        self,
        df, embeddings_col_name: str, label_col_name: str,
        max_length=None, flatten: bool=True,
        embedding_size: int=None, num_seq: int=None
    ):
        """Dataset class for Amazon Reviews Sentiment Analysis.

        Args:
            df (DataFrame): The input DataFrame containing the data.
            embeddings_col_name (str): The column name for the embeddings.
            label_col_name (str): The column name for the labels.
            max_length (int, optional): Maximum length of embeddings (padding applied if needed).
            flatten (bool, optional): Whether to flatten the embeddings or not.
            embedding_size (int, optional): The size of each embedding.
            num_seq (int, optional): The number of sequences (used when `flatten=False`).

        Returns:
            dict: A dictionary containing the embeddings and the target label.

        Raises:
            IndexError: If the index is out of bounds.
        """

        self.data = df
        self.embeddings_col_name = embeddings_col_name
        self.label_col_name = label_col_name
        self.max_length = max_length
        self.flatten = flatten
        self.num_seq = num_seq
        self.embedding_size = embedding_size

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

    def __getitem__(self, idx):
        if idx >= self.__len__():
            raise IndexError

        label = self.data.iloc[idx][self.label_col_name] - 1
        embeddings = self.data.iloc[idx][self.embeddings_col_name]

        # Pad embeddings to max_length if specified
        if self.max_length is not None:
            if len(embeddings) < self.max_length:
                padding = np.zeros(self.max_length - len(embeddings), dtype=float)
                embeddings = np.concatenate((embeddings, padding))

            # Reshape embeddings if specified and flatten is False
            if not self.flatten and self.num_seq is not None and self.embedding_size is not None:
                embeddings = embeddings.reshape(self.num_seq, self.embedding_size)

        return {
            "embeddings": torch.tensor(embeddings, dtype=torch.float32),
            "target":  torch.tensor(label, dtype=torch.long)
        }
```

```
time: 1.78 ms (started: 2023-10-20 22:55:50 +00:00)
```

```
TRAIN_BATCH_SIZE = 128
VALID_BATCH_SIZE = 64
TEST_BATCH_SIZE = 32
NUM_PARALLEL_WORKERS = multiprocessing.cpu_count()
```

```
time: 693 µs (started: 2023-10-20 21:38:30 +00:00)
```

Training & Evaluation Functions

- `compute_accuracy` calculates the accuracy of model predictions given true labels.
- `train_loop_fn` handles one training epoch, updating the model's weights based on computed gradients.
- `eval_loop_fn` handles one validation epoch, computing the model's performance on the validation set.
- `train_and_evaluate` orchestrates the training process, saving checkpoints if specified. It reports metrics after each epoch. If a final model path is provided, it saves the model at the end.

```python
def compute_accuracy(outputs, labels):
    """
    Computes the accuracy of the model's predictions.

    Args:
        outputs (torch.Tensor): The model's predictions.
        labels (torch.Tensor): The true labels.

    Returns:
        float: The accuracy score.

    """
    predicted = torch.argmax(outputs.data, dim=1)

    predicted = predicted.detach().cpu().numpy()
    labels = labels.detach().cpu().numpy()

    acc = accuracy_score(labels, predicted)
    return acc

def train_loop_fn(data_loader, model, optimizer, loss_fn, device):
    """
    Performs one training epoch.

    Args:
        data_loader (DataLoader): The DataLoader for training data.
        model (nn.Module): The neural network model.
        optimizer (torch.optim): The optimizer for updating model weights.
        loss_fn: The loss function.
        device (torch.device): The device to perform computations.

    Returns:
        tuple: A tuple containing the training loss and accuracy.

    """
    model.train()
    train_loss = 0.0
    acc = []

    for batch in tqdm(data_loader):
        embeddings = batch['embeddings'].to(device, dtype=torch.float32, non_blocking=True)
        labels = batch['target'].to(device, dtype=torch.long, non_blocking=True)

        optimizer.zero_grad()

        outputs = model(embeddings.float())
        loss = loss_fn(outputs, labels)

        loss.backward()
        optimizer.step()

        train_loss += loss.item()*len(labels)
        acc.append(compute_accuracy(outputs, labels))

    acc = sum(acc)/len(acc)
    return train_loss, acc

def eval_loop_fn(data_loader, model, loss_fn, device):
    """
    Performs one evaluation epoch.

    Args:
        data_loader (DataLoader): The DataLoader for validation data.
        model (nn.Module): The neural network model.
        loss_fn: The loss function.
        device (torch.device): The device to perform computations.

    Returns:
        tuple: A tuple containing the validation loss and accuracy.

    """
    valid_loss = 0.0
    acc = []
    model.eval()

    for batch in data_loader:
        embeddings = batch['embeddings'].to(device, dtype=torch.float32, non_blocking=True)
        labels = batch['target'].to(device, dtype=torch.long, non_blocking=True)

        outputs = model(embeddings.float())

        loss = loss_fn(outputs, labels)
        valid_loss += loss.item()*len(labels)

        acc.append(compute_accuracy(outputs, labels))

    acc = sum(acc)/len(acc)

    return valid_loss, acc


def train_and_evaluate(
    model,
    train_data_loader, valid_data_loader,
    optimizer, loss_fn,
    device,
    num_epochs,
    checkpoint=False,
    path="model.pt",
    early_stopping_patience=5
):
    """
    Trains and evaluates the model.

    Args:
        model (nn.Module): The neural network model.
        train_data_loader (DataLoader): The DataLoader for training data.
        valid_data_loader (DataLoader): The DataLoader for validation data.
        optimizer (torch.optim): The optimizer for updating model weights.
        loss_fn: The loss function.
        device (torch.device): The device to perform computations.
        num_epochs (int): The number of epochs.
        checkpoint (bool, optional): Whether to save model checkpoints.
        path (str, optional): The path to save the model.
        early_stopping_patience (int, optional): Number of epochs to wait before early stopping.

    Returns:
        nn.Module: The best model.

```

```
    """

    # Create directory for saving checkpoint model states
    if checkpoint:
        dirname = path.split(".")[0]
        checkpoint_path = os.path.join(dirname)
        if os.path.exists(checkpoint_path):
            shutil.rmtree(checkpoint_path)
        os.makedirs(dirname)

    best_loss = float('inf')
    no_improvement_count = 0
    best_model = None

    for epoch in range(num_epochs):
        # Train Step
        train_loss, train_acc = train_loop_fn(
            train_data_loader, model, optimizer, loss_fn, device
        )

        # Validation Step
        valid_loss, valid_acc = eval_loop_fn(valid_data_loader, model, loss_fn, device)

        train_loss /= len(train_data_loader.dataset)
        valid_loss /= len(valid_data_loader.dataset)

        epoch_log = (
            f"Epoch {epoch+1}/{num_epochs},"
            f" Train Accuracy={train_acc:.4f}, Validation Accuracy={valid_acc:.4f},"
            f" Train Loss={train_loss:.4f}, Validation Loss={valid_loss:.4f}"
        )
        print(epoch_log)

        # Check for improvement in validation loss
        if valid_loss < best_loss:
            # Save checkpoint if needed
            if checkpoint:
                cp = os.path.join(checkpoint_path, f"{dirname}_epoch{epoch}_loss{valid_loss:.4f}.pt")
                torch.save(model.state_dict(), cp)
                print(f"Validation loss improved from {best_loss:.4f}--->{valid_loss:.4f}")
                print(f"Saved Checkpoint to '{cp}'")

            best_loss = valid_loss
            best_model = model
            no_improvement_count = 0
        else:
            no_improvement_count += 1

            # Early stopping condition
            if no_improvement_count >= early_stopping_patience:
                print(f"No improvement for {early_stopping_patience} epochs. Stopping early.")
                break

    if checkpoint:
        # Save the best model
        best_model_path = os.path.join(checkpoint_path, f"{dirname}-best.pt")
        torch.save(best_model.state_dict(), best_model_path)
        print(f"Saved best model to '{os.path.relpath(best_model_path)}'")

    # Save current model
    torch.save(model.state_dict(), path)

    return best_model


def test_model(model, data_loader, device):
    """
    Tests the model on the test set.

    Args:
        model (nn.Module): The neural network model.
        data_loader (DataLoader): The DataLoader for test data.
        device (torch.device): The device to perform computations.

    Returns:
        tuple: A tuple containing the test accuracy and loss.

    """
    test_loss = 0.0
    acc = []
    loss_fn = nn.CrossEntropyLoss()

    model.eval()
    for batch in tqdm(data_loader):
        embeddings = batch['embeddings'].to(device, dtype=torch.float32)
        y_true = batch["target"].to(device, dtype=torch.long)

        with torch.no_grad():
            y_pred = model(embeddings)

        loss = loss_fn(y_pred, y_true)
        test_loss += loss.item()*len(y_true)

        acc.append(compute_accuracy(y_pred, y_true))

    acc = sum(acc)/len(acc)
    test_loss = test_loss/len(data_loader.dataset)
    return acc, test_loss
```

```
time: 2.22 ms (started: 2023-10-20 22:59:06 +00:00)
```

Feedforward Neural Networks

```
class MLP(nn.Module):
    def __init__(self, num_input_features, num_classes):
        """
        Multi-Layer Perceptron (MLP) for classification tasks.

        Args:
            num_input_features (int): Number of input features.
            num_classes (int): Number of output classes.

        """
        super(MLP, self).__init__()
        # Input size is 300 (Word2Vec dimensions)
        self.fc1 = nn.Linear(num_input_features, 50)
```

```
        self.fc2 = nn.Linear(50, 5)
        # Output size is 2 for binary classification
        self.fc3 = nn.Linear(5, num_classes)

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

time: 860 µs (started: 2023-10-20 20:17:54 +00:00)

```
train_dataset = AmazonReviewsSentimentDataset(
    train_df, embeddings_col_name="embeddings", label_col_name="sentiment"
)

valid_dataset = AmazonReviewsSentimentDataset(
    valid_df, embeddings_col_name="embeddings", label_col_name="sentiment"
)

test_dataset = AmazonReviewsSentimentDataset(
    test_df, embeddings_col_name="embeddings", label_col_name="sentiment"
)

train_data_loader = DataLoader(
    train_dataset,
    batch_size=TRAIN_BATCH_SIZE,
    drop_last=True,
    shuffle=True,
    # num_workers=NUM_PARALLEL_WORKERS
)

valid_data_loader = DataLoader(
    valid_dataset,
    batch_size=VALID_BATCH_SIZE,
    drop_last=False,
    shuffle=False,
    # num_workers=NUM_PARALLEL_WORKERS
)

test_data_loader = DataLoader(
    test_dataset,
    batch_size=TEST_BATCH_SIZE,
    drop_last=False,
    shuffle=False,
    # num_workers=NUM_PARALLEL_WORKERS
)
```

time: 924 µs (started: 2023-10-20 21:38:49 +00:00)

With average Word2Vec features

```
net = MLP(num_input_features=Word2VecConfig.MAX_LENGTH, num_classes=2).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.01)

model = train_and_evaluate(
    model=net,
    train_data_loader=train_data_loader,
    valid_data_loader=valid_data_loader,
    optimizer=optimizer,
    loss_fn=criterion,
    device=device,
    num_epochs=25,
    checkpoint=True,
    path="mlp_w_avg_w2v_feat_v3.pt"
)
```

```
100%                                          624/624 [00:18<00:00, 38.44it/s]
Epoch 1/25, Train Accuracy=0.5131, Validation Accuracy=0.5611, Train Loss=0.6950, Validation Loss=0.6919
Validation loss improved from inf--->0.6919
Saving Checkpoint to 'mlp_w_avg_w2v_feat_v3/mlp_w_avg_w2v_feat_v3_epoch0_loss0.6919.pt'
100%                                          624/624 [00:16<00:00, 38.91it/s]
Epoch 2/25, Train Accuracy=0.6316, Validation Accuracy=0.6768, Train Loss=0.6911, Validation Loss=0.6905
Validation loss improved from 0.6919--->0.6905
Saving Checkpoint to 'mlp_w_avg_w2v_feat_v3/mlp_w_avg_w2v_feat_v3_epoch1_loss0.6905.pt'
100%                                          624/624 [00:17<00:00, 37.97it/s]
Epoch 3/25, Train Accuracy=0.6516, Validation Accuracy=0.6777, Train Loss=0.6894, Validation Loss=0.6884
Validation loss improved from 0.6905--->0.6884
Saving Checkpoint to 'mlp_w_avg_w2v_feat_v3/mlp_w_avg_w2v_feat_v3_epoch2_loss0.6884.pt'
100%                                          624/624 [00:17<00:00, 30.38it/s]
Epoch 4/25, Train Accuracy=0.6742, Validation Accuracy=0.6149, Train Loss=0.6868, Validation Loss=0.6851
Validation loss improved from 0.6884--->0.6851
Saving Checkpoint to 'mlp_w_avg_w2v_feat_v3/mlp_w_avg_w2v_feat_v3_epoch3_loss0.6851.pt'
100%                                          624/624 [00:16<00:00, 38.13it/s]
Epoch 5/25, Train Accuracy=0.6457, Validation Accuracy=0.6580, Train Loss=0.6818, Validation Loss=0.6777
Validation loss improved from 0.6851--->0.6777
Saving Checkpoint to 'mlp_w_avg_w2v_feat_v3/mlp_w_avg_w2v_feat_v3_epoch4_loss0.6777.pt'
100%                                          624/624 [00:18<00:00, 36.78it/s]
Epoch 6/25, Train Accuracy=0.6730, Validation Accuracy=0.6678, Train Loss=0.6723, Validation Loss=0.6655
Validation loss improved from 0.6777--->0.6655
Saving Checkpoint to 'mlp_w_avg_w2v_feat_v3/mlp_w_avg_w2v_feat_v3_epoch5_loss0.6655.pt'
100%                                          624/624 [00:17<00:00, 38.92it/s]
Epoch 7/25, Train Accuracy=0.6914, Validation Accuracy=0.6881, Train Loss=0.6564, Validation Loss=0.6448
Validation loss improved from 0.6655--->0.6448
Saving Checkpoint to 'mlp_w_avg_w2v_feat_v3/mlp_w_avg_w2v_feat_v3_epoch6_loss0.6448.pt'
100%                                          624/624 [00:17<00:00, 38.77it/s]
Epoch 8/25, Train Accuracy=0.7160, Validation Accuracy=0.7258, Train Loss=0.6308, Validation Loss=0.6127
Validation loss improved from 0.6448--->0.6127
Saving Checkpoint to 'mlp_w_avg_w2v_feat_v3/mlp_w_avg_w2v_feat_v3_epoch7_loss0.6127.pt'
100%                                          624/624 [00:17<00:00, 38.84it/s]
Epoch 9/25, Train Accuracy=0.7429, Validation Accuracy=0.7512, Train Loss=0.5944, Validation Loss=0.5711
Validation loss improved from 0.6127--->0.5711
Saving Checkpoint to 'mlp_w_avg_w2v_feat_v3/mlp_w_avg_w2v_feat_v3_epoch8_loss0.5711.pt'
100%                                          624/624 [00:17<00:00, 39.25it/s]
Epoch 10/25, Train Accuracy=0.7628, Validation Accuracy=0.7694, Train Loss=0.5528, Validation Loss=0.5292
Validation loss improved from 0.5711--->0.5292
Saving Checkpoint to 'mlp_w_avg_w2v_feat_v3/mlp_w_avg_w2v_feat_v3_epoch9_loss0.5292.pt'
100%                                          624/624 [00:17<00:00, 29.82it/s]
Epoch 11/25, Train Accuracy=0.7751, Validation Accuracy=0.7859, Train Loss=0.5159, Validation Loss=0.4955
Validation loss improved from 0.5292--->0.4955
Saving Checkpoint to 'mlp_w_avg_w2v_feat_v3/mlp_w_avg_w2v_feat_v3_epoch10_loss0.4955.pt'
100%                                          624/624 [00:16<00:00, 37.17it/s]
Epoch 12/25, Train Accuracy=0.7846, Validation Accuracy=0.7918, Train Loss=0.4887, Validation Loss=0.4720
Validation loss improved from 0.4955--->0.4720
Saving Checkpoint to 'mlp_w_avg_w2v_feat_v3/mlp_w_avg_w2v_feat_v3_epoch11_loss0.4720.pt'
100%                                          624/624 [00:17<00:00, 37.59it/s]
Epoch 13/25, Train Accuracy=0.7917, Validation Accuracy=0.7959, Train Loss=0.4699, Validation Loss=0.4560
Validation loss improved from 0.4720--->0.4560
Saving Checkpoint to 'mlp_w_avg_w2v_feat_v3/mlp_w_avg_w2v_feat_v3_epoch12_loss0.4560.pt'
100%                                          624/624 [00:17<00:00, 37.73it/s]
Epoch 14/25, Train Accuracy=0.7967, Validation Accuracy=0.8017, Train Loss=0.4565, Validation Loss=0.4443
Validation loss improved from 0.4560--->0.4443
Saving Checkpoint to 'mlp_w_avg_w2v_feat_v3/mlp_w_avg_w2v_feat_v3_epoch13_loss0.4443.pt'
100%                                          624/624 [00:17<00:00, 38.39it/s]
```

Overall Accuracy on Test Set

```
Saving Checkpoint to 'mlp_w_avg_w2v_feat_v3/mlp_w_avg_w2v_feat_v3_epoch14_loss0.4353.pt'
```

```python
path_to_saved_model = 'mlp_w_avg_w2v_feat_v3.pt'
model = MLP(num_input_features=Word2VecConfig.MAX_LENGTH, num_classes=2)
model.load_state_dict(torch.load(path_to_saved_model, map_location=device))

acc, loss = test_model(model, test_data_loader, device)
print("Accuracy (Test Dataset):", round(acc,4))
```

```
100%                                          157/157 [00:02<00:00, 95.96it/s]
Accuracy (Test Dataset): 0.8298
time: 2.45 s (started: 2023-10-20 21:43:29 +00:00)
```

With top 10 Word2Vec features

- Embeddings are padded for maintaining consistent input dimensions across different samples in a batch.

```
Saving Checkpoint to 'mlp_w_avg_w2v_feat_v3/mlp_w_avg_w2v_feat_v3_epoch18_loss0.4137.pt'
```

```python
train_dataset = AmazonReviewsSentimentDataset(
    train_df,
    embeddings_col_name="embeddings_top_10",
    label_col_name="sentiment",
    max_length=3000,
    flatten=True
)

valid_dataset = AmazonReviewsSentimentDataset(
    valid_df,
    embeddings_col_name="embeddings_top_10",
    label_col_name="sentiment",
    max_length=3000,
    flatten=True
)

test_dataset = AmazonReviewsSentimentDataset(
    test_df,
    embeddings_col_name="embeddings_top_10",
    label_col_name="sentiment",
    max_length=3000,
    flatten=True
)

train_data_loader = DataLoader(
    train_dataset,
    batch_size=TRAIN_BATCH_SIZE,
    drop_last=True,
    shuffle=True,
)

valid_data_loader = DataLoader(
    valid_dataset,
    batch_size=VALID_BATCH_SIZE,
```

```
    drop_last=False,
    shuffle=False,
)

test_data_loader = DataLoader(
    test_dataset,
    batch_size=TEST_BATCH_SIZE,
    drop_last=False,
    shuffle=False,
)
```

```
time: 1.29 ms (started: 2023-10-20 21:46:47 +00:00)
```

```
net2 = MLP(num_input_features=3000, num_classes=2).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net2.parameters(), lr=0.01)


model2 = train_and_evaluate(
    model=net2,
    train_data_loader=train_data_loader,
    valid_data_loader=valid_data_loader,
    optimizer=optimizer,
    loss_fn=criterion,
    device=device,
    num_epochs=30,
    checkpoint=True,
    path="mlp_w_top10_w2v_feat_v2.pt"
)
```

```
100%                                          624/624 [00:18<00:00, 36.25it/s]
Epoch 1/25, Train Accuracy=0.5929, Validation Accuracy=0.6606, Train Loss=0.6864, Validation Loss=0.6760
Validation loss improved from inf--->0.6760
Saved Checkpoint to 'mlp_w_top10_w2v_feat_v2/mlp_w_top10_w2v_feat_v2_epoch0_loss0.6760.pt'
100%                                          624/624 [00:18<00:00, 33.14it/s]
Epoch 2/25, Train Accuracy=0.6798, Validation Accuracy=0.7102, Train Loss=0.6556, Validation Loss=0.6253
Validation loss improved from 0.6760--->0.6253
Saved Checkpoint to 'mlp_w_top10_w2v_feat_v2/mlp_w_top10_w2v_feat_v2_epoch1_loss0.6253.pt'
100%                                          624/624 [00:18<00:00, 34.66it/s]
Epoch 3/25, Train Accuracy=0.7199, Validation Accuracy=0.7278, Train Loss=0.5863, Validation Loss=0.5492
Validation loss improved from 0.6253--->0.5492
Saved Checkpoint to 'mlp_w_top10_w2v_feat_v2/mlp_w_top10_w2v_feat_v2_epoch2_loss0.5492.pt'
100%                                          624/624 [00:18<00:00, 36.23it/s]
```

Overall Accracy on Test Set

```
Saved Checkpoint to 'mlp w top10 w2v feat v2/mlp w top10 w2v feat v2 epoch3 loss0.5210.pt'
```

```
path_to_saved_model = 'mlp_w_top10_w2v_feat_v2.pt'
model = MLP(num_input_features=3000, num_classes=2)
model.load_state_dict(torch.load(path_to_saved_model, map_location=device))

acc, loss = test_model(model, test_data_loader, device)
print("Accuracy (Test Dataset):", round(acc,4))
```

```
100%                                          157/157 [00:03<00:00, 58.79it/s]
Accuracy (Test Dataset): 0.7761
time: 3.41 s (started: 2023-10-20 21:56:52 +00:00)
Validation loss improved from 0.5041--->0.5000
```

Comparision with Simple Model

The LinearSVC model trained on TF-IDF features was the most effective in this scenario, outperforming both simple models and MLP models trained with Word2Vec embeddings.

Conclusion

1. **Feature Importance**:
   - The choice of features significantly impacts model performance.
   - In this case, TF-IDF features proved to be the most informative for sentiment analysis, as evidenced by the high accuracy achieved by LinearSVC with TF-IDF.
2. **Complexity vs. Performance**:
   - Simple models like Perceptron and LinearSVC can sometimes outperform more complex models.
   - This is evident in the case where LinearSVC with TF-IDF outperformed the MLP models.
3. **Embedding Selection**:
   - Not all embeddings are equally effective. The choice of Word2Vec embeddings, particularly using the average vectors, yielded competitive results, showcasing the importance of using quality word embeddings.
4. **Dimensionality Matters**:
   - Using only the top 10 Word2Vec embeddings didn't capture enough information for sentiment analysis.
   - It's important to consider the dimensionality of the embeddings and how well they represent the underlying semantics.

```
Epoch 14/25, Train Accuracy=0.7784, Validation Accuracy=0.7602, Train Loss=0.4664, Validation Loss=0.4965
```

Recurrent Neural Networks

```
class RNNModel(nn.Module):
    def __init__(
        self, input_size, hidden_size, num_layers, output_size, model_type="rnn"
    ):
        """
        Recurrent Neural Network (RNN) model for sequence data processing.

        Args:
            input_size (int): Dimension of the input features.
            hidden_size (int): Number of units in the hidden layers.
            num_layers (int): Number of recurrent layers.
            output_size (int): Number of output classes.
            model_type (str, optional): Type of RNN ('rnn', 'gru', or 'lstm'). Defaults to 'rnn'.

        """
        super(RNNModel, self).__init__()

        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.model_type = model_type

        # Initialize the recurrent layer based on model_type
        if model_type == "gru":
            self.layer = nn.GRU(input_size, hidden_size, num_layers, batch_first=True, dropout=0.3)
        elif model_type == "lstm":
            self.layer = nn.LSTM(input_size, hidden_size, num_layers, batch_first=True, dropout=0.3)
        else:
            self.layer = nn.RNN(input_size, hidden_size, num_layers, batch_first=True, dropout=0.3)

        # dropout layer to prevent overfitting
        self.dropout = nn.Dropout(0.3)

        # Fully connected layer for final prediction
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        batch_size = x.size(0)
        hidden = self.init_hidden(batch_size)

        # Pass input through the recurrent layer
        out, _ = self.layer(x, hidden)

        # Stack up the model output
        # out = out.contiguous().view(-1, self.hidden_size)

        # Use the output from the last time step
        out = out[:, -1, :]

        # out = self.dropout(out)

        # Apply fully connected layer for final prediction
        out = self.fc(out)
        return out

```

```
    def init_hidden(self, batch_size):
        if self.model_type == "lstm":
            hidden = (
                torch.zeros(self.num_layers, batch_size, self.hidden_size).to(device),
                torch.zeros(self.num_layers, batch_size, self.hidden_size).to(device)
            )
        else:
            hidden = torch.zeros(self.num_layers, batch_size, self.hidden_size).to(device)
        return hidden
```

```
time: 1.65 ms (started: 2023-10-20 21:57:09 +00:00)
```

```
train_dataset = AmazonReviewsSentimentDataset(
    train_df,
    embeddings_col_name="embeddings_top_10",
    label_col_name="sentiment",
    max_length=3000,
    flatten=False,
    embedding_size=Word2VecConfig.EMBEDDING_SIZE,
    num_seq=10
)

valid_dataset = AmazonReviewsSentimentDataset(
    valid_df,
    embeddings_col_name="embeddings_top_10",
    label_col_name="sentiment",
    max_length=3000,
    flatten=False,
    embedding_size=Word2VecConfig.EMBEDDING_SIZE,
    num_seq=10
)

test_dataset = AmazonReviewsSentimentDataset(
    test_df,
    embeddings_col_name="embeddings_top_10",
    label_col_name="sentiment",
    max_length=3000,
    flatten=False,
    embedding_size=Word2VecConfig.EMBEDDING_SIZE,
    num_seq=10
)

train_data_loader = DataLoader(
    train_dataset,
    batch_size=TRAIN_BATCH_SIZE,
    drop_last=True,
    shuffle=True,
)

valid_data_loader = DataLoader(
    valid_dataset,
    batch_size=VALID_BATCH_SIZE,
    drop_last=False,
    shuffle=False,
)

test_data_loader = DataLoader(
    test_dataset,
    batch_size=TEST_BATCH_SIZE,
    drop_last=False,
    shuffle=False,
)
```

```
time: 1.11 ms (started: 2023-10-20 23:05:20 +00:00)
```

```
input_size = 300
hidden_size = 10
output_size = 2
num_layers = 10
```

```
time: 473 µs (started: 2023-10-20 23:05:22 +00:00)
```

Simple RNN

```
net3 = RNNModel(input_size, hidden_size, num_layers, output_size, model_type="rnn").to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net3.parameters(), lr=0.001)

model3 = train_and_evaluate(
    model=net3,
    train_data_loader=train_data_loader,
    valid_data_loader=valid_data_loader,
    optimizer=optimizer,
    loss_fn=criterion,
    device=device,
    num_epochs=20,
    checkpoint=True,
    path="simple_rnn_w2v_feat_v2.pt"
)
```

```
100%                                   624/624 [00:35<00:00, 26.98it/s]
Epoch 1/20, Train Accuracy=0.6423, Validation Accuracy=0.7367, Train Loss=0.6211, Validation Loss=0.5484
Validation loss improved from inf--->0.5484
Saved Checkpoint to 'simple_rnn_w2v_feat_v2/simple_rnn_w2v_feat_v2_epoch0_loss0.5484.pt'
100%                                   624/624 [00:28<00:00, 20.17it/s]
Epoch 2/20, Train Accuracy=0.7385, Validation Accuracy=0.7457, Train Loss=0.5413, Validation Loss=0.5386
Validation loss improved from 0.5484--->0.5386
Saved Checkpoint to 'simple_rnn_w2v_feat_v2/simple_rnn_w2v_feat_v2_epoch1_loss0.5386.pt'
100%                                   624/624 [00:26<00:00, 26.29it/s]
Epoch 3/20, Train Accuracy=0.7492, Validation Accuracy=0.7528, Train Loss=0.5260, Validation Loss=0.5217
Validation loss improved from 0.5386--->0.5217
Saved Checkpoint to 'simple_rnn_w2v_feat_v2/simple_rnn_w2v_feat_v2_epoch2_loss0.5217.pt'
100%                                   624/624 [00:37<00:00, 25.14it/s]
Epoch 4/20, Train Accuracy=0.7559, Validation Accuracy=0.7570, Train Loss=0.5182, Validation Loss=0.5154
Validation loss improved from 0.5217--->0.5154
Saved Checkpoint to 'simple_rnn_w2v_feat_v2/simple_rnn_w2v_feat_v2_epoch3_loss0.5154.pt'
100%                                   624/624 [00:29<00:00, 20.49it/s]
Epoch 5/20, Train Accuracy=0.7605, Validation Accuracy=0.7617, Train Loss=0.5119, Validation Loss=0.5129
Validation loss improved from 0.5154--->0.5129
Saved Checkpoint to 'simple_rnn_w2v_feat_v2/simple_rnn_w2v_feat_v2_epoch4_loss0.5129.pt'
100%                                   624/624 [00:26<00:00, 25.89it/s]
Epoch 6/20, Train Accuracy=0.7609, Validation Accuracy=0.7610, Train Loss=0.5079, Validation Loss=0.5058
Validation loss improved from 0.5129--->0.5058
Saved Checkpoint to 'simple_rnn_w2v_feat_v2/simple_rnn_w2v_feat_v2_epoch5_loss0.5058.pt'
100%                                   624/624 [00:26<00:00, 25.14it/s]
Epoch 7/20, Train Accuracy=0.7624, Validation Accuracy=0.7595, Train Loss=0.5046, Validation Loss=0.5097
100%                                   624/624 [00:26<00:00, 18.12it/s]
Epoch 8/20, Train Accuracy=0.7641, Validation Accuracy=0.7689, Train Loss=0.5016, Validation Loss=0.4986
```

Overall Accuracy on Test Set

```
path_to_saved_model = 'simple_rnn_w2v_feat_v2.pt'
model = RNNModel(input_size, hidden_size, num_layers, output_size, model_type="rnn")
model.load_state_dict(torch.load(path_to_saved_model, map_location=device))

acc, loss = test_model(model, test_data_loader, device)
print("Accuracy (Test Dataset):", round(acc,4))
```

```
100%                                   157/157 [00:01<00:00, 113.81it/s]
Accuracy (Test Dataset): 0.7765
time: 1.49 s (started: 2023-10-20 23:23:30 +00:00)
```

GRU

```
net4 = RNNModel(input_size, hidden_size, num_layers, output_size, model_type="gru").to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net4.parameters(), lr=0.001)

model4 = train_and_evaluate(
    model=net4,
    train_data_loader=train_data_loader,
    valid_data_loader=valid_data_loader,
    optimizer=optimizer,
    loss_fn=criterion,
    device=device,
    num_epochs=20,
    checkpoint=True,
    path="gru_w2v_feat_v2.pt"
)
```

```
100%    624/624 [00:42<00:00, 16.88it/s]
Epoch 1/20, Train Accuracy=0.6928, Validation Accuracy=0.7528, Train Loss=0.5734, Validation Loss=0.5106
Validation loss improved from inf--->0.5106
Saved Checkpoint to 'gru_w2v_feat_v2/gru_w2v_feat_v2_epoch0_loss0.5106.pt'
100%    624/624 [00:37<00:00, 15.09it/s]
Epoch 2/20, Train Accuracy=0.7611, Validation Accuracy=0.7681, Train Loss=0.5031, Validation Loss=0.4882
Validation loss improved from 0.5106--->0.4882
Saved Checkpoint to 'gru_w2v_feat_v2/gru_w2v_feat_v2_epoch1_loss0.4882.pt'
100%    624/624 [00:38<00:00, 16.97it/s]
Epoch 3/20, Train Accuracy=0.7744, Validation Accuracy=0.7754, Train Loss=0.4803, Validation Loss=0.4765
Validation loss improved from 0.4882--->0.4765
Saved Checkpoint to 'gru_w2v_feat_v2/gru_w2v_feat_v2_epoch2_loss0.4765.pt'
100%    624/624 [00:38<00:00, 17.81it/s]
Epoch 4/20, Train Accuracy=0.7806, Validation Accuracy=0.7785, Train Loss=0.4677, Validation Loss=0.4810
100%    624/624 [00:38<00:00, 13.10it/s]
Epoch 5/20, Train Accuracy=0.7858, Validation Accuracy=0.7834, Train Loss=0.4601, Validation Loss=0.4533
Validation loss improved from 0.4765--->0.4533
Saved Checkpoint to 'gru_w2v_feat_v2/gru_w2v_feat_v2_epoch4_loss0.4533.pt'
100%    624/624 [00:38<00:00, 17.16it/s]
Epoch 6/20, Train Accuracy=0.7910, Validation Accuracy=0.7794, Train Loss=0.4504, Validation Loss=0.4791
100%    624/624 [00:38<00:00, 17.37it/s]
```

Overall Accuracy on Test Set

```
path_to_saved_model = 'gru_w2v_feat_v2.pt'
model = RNNModel(input_size, hidden_size, num_layers, output_size, model_type="gru")
model.load_state_dict(torch.load(path_to_saved_model, map_location=device))

acc, loss = test_model(model, test_data_loader, device)
print("Accuracy (Test Dataset):", round(acc,4))
```

```
100%    157/157 [00:02<00:00, 61.83it/s]
Accuracy (Test Dataset): 0.8063
time: 2.32 s (started: 2023-10-20 22:36:42 +00:00)
```

LSTM

```
100%    624/624 [00:38<00:00, 16.88it/s]
```

```
net5 = RNNModel(input_size, hidden_size, num_layers, output_size, model_type="lstm").to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(net5.parameters(), lr=0.001)

model5 = train_and_evaluate(
    model=net5,
    train_data_loader=train_data_loader,
    valid_data_loader=valid_data_loader,
    optimizer=optimizer,
    loss_fn=criterion,
    device=device,
    num_epochs=20,
    checkpoint=True,
    path="lstm_w2v_feat_v2.pt"
)
```

```
100%                                            624/624 [00:31<00:00, 23.71it/s]

Epoch 1/20, Train Accuracy=0.6529, Validation Accuracy=0.7387, Train Loss=0.6076, Validation Loss=0.5329
Validation loss improved from inf--->0.5329
Saved Checkpoint to 'lstm_w2v_feat_v2/lstm_w2v_feat_v2_epoch0_loss0.5329.pt'
100%                                            624/624 [00:27<00:00, 23.73it/s]

Epoch 2/20, Train Accuracy=0.7565, Validation Accuracy=0.7735, Train Loss=0.5129, Validation Loss=0.4886
Validation loss improved from 0.5329--->0.4886
Saved Checkpoint to 'lstm_w2v_feat_v2/lstm_w2v_feat_v2_epoch1_loss0.4886.pt'
100%                                            624/624 [00:28<00:00, 23.71it/s]

Epoch 3/20, Train Accuracy=0.7732, Validation Accuracy=0.7785, Train Loss=0.4826, Validation Loss=0.4754
Validation loss improved from 0.4886--->0.4754
Saved Checkpoint to 'lstm_w2v_feat_v2/lstm_w2v_feat_v2_epoch2_loss0.4754.pt'
100%                                            624/624 [00:29<00:00, 23.78it/s]

Epoch 4/20, Train Accuracy=0.7800, Validation Accuracy=0.7858, Train Loss=0.4669, Validation Loss=0.4692
```

Overall Accuracy on Test Set

```
path_to_saved_model = 'lstm_w2v_feat_v2.pt'
model = RNNModel(input_size, hidden_size, num_layers, output_size, model_type="lstm")
model.load_state_dict(torch.load(path_to_saved_model, map_location=device))

acc, loss = test_model(model, test_data_loader, device)
print("Accuracy (Test Dataset):", round(acc,4))
```

```
100%                                            157/157 [00:01<00:00, 101.68it/s]

Accuracy (Test Dataset): 0.8023
time: 1.79 s (started: 2023-10-20 22:52:48 +00:00)

Saved Checkpoint to 'lstm_w2v_feat_v2/lstm_w2v_feat_v2_epoch6_loss0.4518.pt'
```

Conclusion

1. **Feature Representations**:
    - TF-IDF outperforms Word2Vec across all models.
    - Averaged Word2Vec is better than Concatenated Word2Vec.
2. **Model Comparisons**:
    - SVM outperforms Perceptron consistently.
    - MLP with averaged Word2Vec performs better than RNN, GRU, and LSTM with Word2Vec.
3. **Recurrent Models**:
    - RNN, GRU, and LSTM show similar performance with Word2Vec embeddings.
4. **Overall Performance**:
    - Highest accuracy (~87%) is achieved with SVM using TF-IDF.

Other:

- Averaging Word2Vec embeddings seems a more effective representation
- SVM model is better at capturing the non-linear relationships in the data compared to the Perceptron
- TF-IDF may capture important information more effectively than Word2Vec embeddings

Report

- Results from one of the runs

Model	Accuracy	Features
SVM(LinearSVC)	0.8581	TF-IDF
SVM(LinearSVC)	0.8321	mean word2vec
MLP	0.8298	mean word2vec
Perceptron	0.8110	mean word2vec
GRU	0.8063	top 10 word2vec
LSTM	0.8023	top 10 word2vec
Perceptron	0.7998	TF-IDF
Simple RNN	0.7765	top 10 word2vec
MLP	0.7761	top 10 word2vec

References

1. https://www.kaggle.com/code/abhishek/bert-multi-lingual-tpu-training-8-cores
2. https://www.kaggle.com/mishra1993/pytorch-multi-layer-perceptron-mnist
3. https://www.kaggle.com/code/arunmohan003/sentiment-analysis-using-lstm-pytorch
4. https://pytorch.org/docs/stable/generated/torch.nn.RNN.html
5. https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html
6. https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html

THE END