NLP (CSCI-544) Homework 2 Report

Dataset Generation

Overview

The dataset, derived from Amazon product reviews, was prepared for sentiment analysis. The goal was to create a balanced dataset with equal representation across different rating levels.

Steps

Data Loading: Utilized Python's Pandas to load 'review_body' and 'star_rating' from the Amazon reviews dataset.
Balancing: Randomly selected 50,000 samples for each star rating (1 to 5), using a fixed random state for reproducibility.
Classification: Assigned ternary labels based on the ratings - Class 1 for positive (rating > 3), Class 2 for negative (rating < 3), and Class 3 for neutral (rating = 3).

Word Embedding

Word Embedding with Pretrained Word2Vec Model

Utilizing Pretrained Word Embeddings

We leveraged the 'word2vec-google-news-300' model using Gensim's downloader API to load pretrained word embeddings. This model contains 300-dimensional vectors trained on a substantial corpus of Google News articles.

Semantic Similarity Assessment

To evaluate the quality of the embeddings, we conducted semantic similarity tests:

Vector Arithmetic for Analogies:
We tested the classic analogy "king - man + woman = ?" and correctly received "queen" as the top result with a similarity score of **0.7118,** demonstrating the model's understanding of relational semantics.
Similarity Between Words:
For direct word comparisons, we measured the similarity between "excellent" and "outstanding," which yielded a similarity score of **0.56,** indicating a moderate semantic correlation consistent with human intuition.

These results confirm that the pretrained Word2Vec embeddings capture meaningful semantic relationships and can provide a robust foundation for downstream NLP tasks such as sentiment analysis.

Training Custom Word2Vec Model

Data Preprocessing

Prior to training our Word2Vec model, we performed a series of preprocessing steps to clean and standardize the review text data:

Normalization: Converted all text to lowercase to unify the casing.
Cleaning: Removed HTML tags and non-alphabetic characters to ensure only textual data was retained.
Contractions: Expanded contractions (e.g., "don't" to "do not") for consistency.
Tokenization: Split the text into individual words or tokens.
Stopword Removal: Eliminated common English words that carry less meaning.
Lemmatization: Reduced words to their base or dictionary form.

These preprocessing steps were crucial for reducing noise in the data and ensuring our Word2Vec model learned meaningful word representations.

Custom Word2Vec Training

The custom Word2Vec model was trained using the following specifications:
Vector Size: 300
Window Size: 11
Minimum Word Count: 10
This setup was chosen to capture a broad context around each word while still limiting the feature space to words that occur frequently enough to be considered relevant.

Semantic Similarity Analysis

We then assessed the model's understanding of semantic relationships:
Direct Comparison:
Calculated the similarity between "happy" and "impressed," resulting in a score of 0.5609.
Compared "excellent" and "outstanding," obtaining a similarity score of 0.78, indicating a strong semantic connection.
Model Comparison:
Compared our trained model against the pretrained Google News model using various word pairs. The analysis showed varying degrees of similarity, with some word pairs being more similar in our custom model compared to the pretrained one.

Word Pair	Pretrained Model Similarity	Amazon Review Model Similarity
smartphone, camera	0.32	0.41
laptop, charger	0.47	0.25
headphone, bluetooth	0.49	0.45
novel, author	0.46	0.58
fiction, character	0.25	0.31

The custom Word2Vec model, trained on Amazon review data, showed a robust ability to capture semantic meanings and similarities, demonstrating its potential effectiveness for sentiment analysis tasks.

Simple Models

Feature Generation

Utilized a custom function to compute average Word2Vec vectors from the reviews, resulting in a feature set of 200,000 samples with 300 features each.

Testing Performance Metrics for Perceptron and SVM Models

Feature Type	Model	Accuracy	Precision	Recall	F1-Score
Pretrained Word2Vec	Perceptron	74.38%	89.17%	55.53%	68.43%
Pretrained Word2Vec	SVM	81.69%	83.40%	79.20%	81.23%
Custom Word2Vec	Perceptron	80.34%	79.37%	80.99%	80.47%
Custom Word2Vec	SVM	84.24%	85.29%	82.75%	84.00%
TF-IDF (HW1)	Perceptron	83.36%	85.36%	80.52%	82.87%
TF-IDF (HW1)	SVM	88.64%	88.98%	88.19%	88.55%

Analysis

The SVM consistently outperforms the Perceptron across all feature types, indicating its robustness in handling high-dimensional data.

Models using TF-IDF features achieved the highest accuracy and F1-Score, suggesting that the TF-IDF method captures more relevant information for sentiment analysis compared to Word2Vec.

The performance improvement from the custom Word2Vec model over the pretrained one suggests that domain-specific training can enhance model understanding of context.

While the Perceptron model shows relatively lower performance, it still benefits from the TF-IDF features, as seen by the higher accuracy compared to Word2Vec features.

Feed Forward Networks

Architecture and Hyperparameters of MLP Network

For binary and ternary classification tasks using average Word2Vec vectors, we designed an MLP (Multilayer Perceptron) with the following architecture:
Input Layer: 300 units (matching the dimensionality of the Word2Vec vectors)
Hidden Layer 1: 50 units with ReLU activation
Hidden Layer 2: 10 units with ReLU activation
Output Layer: 2 units for binary classification or 3 units for ternary classification

Key hyperparameters included:
Learning Rate: 0.001
Batch Size: 64
Epochs: Up to 100 with early stopping
Optimizer: Adam
Loss Function: CrossEntropyLoss
Learning Rate Scheduler: ReduceLROnPlateau with a patience of 1.5 epochs (patience//2) and a factor of 0.1 for adjustment

The MLP model was trained with a patience-based early stopping mechanism to prevent overfitting and a learning rate scheduler to adjust the learning rate based on validation loss, aiming to find the best generalizable parameters.

Training Procedure

During training, the model's weights were updated using backpropagation. After each epoch, the model was evaluated on the validation set, and early stopping was employed if no improvement in validation loss was observed for 3 consecutive epochs. The learning rate scheduler adjusted the learning rate if the validation loss did not decrease after 1.5 epochs.

Results

Data Type	Word2Vec Source	Classification Type	Accuracy	Precision	Recall	F1-Score
Average Values	Custom	Binary	86.54%	86.64%	86.40%	86.52%
Average Values	Google	Binary	84.53%	84.16%	85.05%	84.60%
Average Values	Custom	Ternary	70.41%	67.81%	70.41%	68.34%
Average Values	Google	Ternary	70.15%	67.53%	70.15%	68.04%
Concatenated Values	Custom	Binary	79.30%	78.76%	80.22%	79.48%
Concatenated Values	Google	Binary	77.86%	77.38%	78.72%	78.04%
Concatenated Values	Custom	Ternary	60.98%	58.92%	60.98%	59.61%
Concatenated Values	Google	Ternary	59.85%	57.97%	59.85%	58.63%

Analysis and Comparison with Simple Models:

FNN Binary vs Simple Models: The FNN models using average Word2Vec vectors outperformed the simple models' accuracy for binary classification (with previous SVM accuracy using custom Word2Vec at 84.24%).

Average vs Concatenated Vectors: For binary classification, average vectors led to better accuracy than concatenated vectors in FNN models. This suggests that a mean representation preserves more useful information for classification tasks than a concatenation of several word vectors.
Binary vs Ternary Classification: As expected, binary classification resulted in higher accuracy compared to ternary classification, likely due to the increased complexity of distinguishing between three classes instead of two.
Custom vs Google Word2Vec: Custom Word2Vec vectors generally provided better or comparable performance metrics than Google's pretrained vectors, which could be attributed to the domain-specific learning in the custom model.
Performance Drop in Concatenated Ternary: There is a notable performance drop in ternary classification using concatenated vectors, indicating that this method may not capture the nuances required for a more granular classification.

Convolutional Neural Networks

Architecture and Hyperparameters

The CNN model for sentiment analysis was defined with the following architecture:
Convolutional Layer 1: 300 input channels, 50 output channels, kernel size of 5, padding of 2.
Convolutional Layer 2: 50 input channels, 10 output channels, kernel size of 5, padding of 2.
Fully Connected Layer: Dynamically initialized based on the output of the second convolutional layer.
Activation: ReLU (Rectified Linear Unit) after each convolutional layer.
Pooling: Max pooling with a kernel size of 2 and stride of 2 after each convolutional layer.
Output: Depending on the number of classes (binary or ternary classification), the final layer outputs 2 or 3 units.

Hyperparameters for the CNN model included:
Learning Rate: 0.001
Batch Size: 64
Epochs: Up to 100 with early stopping based on validation accuracy.
Optimizer: Adam
Loss Function: CrossEntropyLoss

Data Preparation

Input data for the CNN was prepared by padding the Word2Vec representations of reviews to ensure uniform length:
Vector Size: 300 (aligned with Word2Vec dimensions).
Pad Size: 50 (maximum length of the reviews).
Padding: Reviews shorter than 50 words were padded with zeros; longer reviews were truncated to the first 50 words.
This preprocessing ensured that each input to the CNN had a fixed size, which is necessary for training neural networks.

Training Procedure

The model was trained using a custom training loop with the following steps:
Loss Computation: Cross-entropy loss between the predictions and the true labels.
Backpropagation: Optimization of weights using the calculated gradients.
Accuracy Calculation: Post-epoch evaluation of model accuracy on the training and validation sets.
Early Stopping: Termination of training if validation accuracy did not improve for a set number of consecutive epochs (patience parameter).

Word2Vec Source	**Classification Type**	**Accuracy**	**Precision**	**Recall**	**F1-Score**
Custom	Binary	84.85%	85.04%	84.85%	84.83%
Google	Binary	85.48%	85.48%	85.48%	85.47%
Custom	Ternary	67.44%	64.78%	67.44%	65.57%
Google	Ternary	68.03%	66.35%	68.03%	66.97%

Conclusions and Comparison with Simple Models:

CNN Binary vs Simple Models: CNN models using padded Word2Vec vectors showed high accuracy for binary classification, with Google Word2Vec slightly outperforming Custom Word2Vec, contrary to the trend observed in MLPs.
Binary vs Ternary Classification: Binary classification yielded significantly higher accuracy compared to ternary classification, consistent with the performance of other models due to the increased complexity of the ternary task.
CNN vs FNN Performance: CNNs demonstrated an improvement in accuracy over FNNs for the binary classification task, likely due to CNNs' ability to capture local dependencies in the data. However, the difference in performance is less pronounced in the ternary classification task.
Training vs Testing Discrepancy: There is a notable difference between training and testing accuracy, which could indicate overfitting during training. The high training accuracy suggests that the model may have learned to memorize training data, which did not generalize as well to unseen data.

Data Generation

In [1]:

```
import pandas as pd
import gzip


file_path = './amazon_reviews_us_Office_Products_v1_00.tsv.gz'


df = pd.read_csv(gzip.open(file_path), sep='\t', usecols=['review_body', 'star_rating'])

df = df[pd.to_numeric(df['star_rating'], errors='coerce').notna()]

df['star_rating'] = df['star_rating'].astype(int)
df = df.dropna()

rating_dfs = []
for i in range(1, 6):
    rating_df = df[df['star_rating'] == i].sample(n=50000, random_state=42)
    rating_dfs.append(rating_df)

dataset = pd.concat(rating_dfs, ignore_index=True)

def categorize_rating(rating):
    if rating > 3:
        return 1
    elif rating < 3:
        return 2
    else:
        return 3


dataset['class'] = dataset['star_rating'].apply(categorize_rating)

dataset.to_csv('data.csv', index=False)
```

In [1]:

```
import pandas as pd

df = pd.read_csv('data.csv')
```

Word Embeddings

Google Word2Vec

In [2]:

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

In [4]:

```
# Perform the vector arithmetic indirectly using `most_similar`
result = wv.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)

print(result)
```

```
[('queen', 0.7118192911148071)]
```

In [ ]:

```
w1='excellent'
w2='outstanding'
print('%r\t%r\t%.2f' % (w1, w2, wv.similarity(w1, w2)))
```

```
'excellent' 'outstanding' 0.56
```

Custom Trained Model Word2Vec

Preprocessing the data

In [3]:

```
df
```

Out[3]:

	star_rating	review_body	class
0	1	The photo is deceiving - makes it look like a ...	2
1	1	Worst labels ever! I purchased these labels to...	2
2	1	This product broke in a very short time. It a...	2
3	1	The printer head is malfunctioning since the i...	2
4	1	When this item shipped to me I was very excite...	2
...	...	...	...
249995	5	Produces great prints.	1
249996	5	perfect for my high school student to use in h...	1
249997	5	The product was Excellent! !	1
249998	5	Arrived fast and works great--good buy!	1
249999	5	I am glad I bought these headsets. I can hear ...	1

250000 rows × 3 columns

In [2]:

```
import re
from bs4 import BeautifulSoup
df['review_body'] = df['review_body'].str.lower()

df['review_body'] = df['review_body'].apply(lambda x: BeautifulSoup(x, 'html.parser').ge
t_text())
df['review_body'] = df['review_body'].apply(lambda x: re.sub(r'http\S+', ' ', x))
df['review_body'] = df['review_body'].str.replace('[^a-zA-Z\s]', ' ', regex=True)
df['review_body'] = df['review_body'].str.replace('\s+', ' ', regex=True)


import contractions
df['review_body'] = df['review_body'].apply(contractions.fix)



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

def tokenize_text(text):
    return nltk.word_tokenize(text)

def remove_stopwords(tokens):
    stop_words = set(stopwords.words('english'))
    return [word for word in tokens if word.lower() not in stop_words]
```

```python
def lemmatize_text(tokens):
    lemmatizer = WordNetLemmatizer()
    return [lemmatizer.lemmatize(word) for word in tokens]

def process_text(text):
    # Tokenization
    tokens = tokenize_text(text)
    # Remove stop words
    tokens_no_stopwords = remove_stopwords(tokens)
    # Lemmatization
    lemmatized_tokens = lemmatize_text(tokens_no_stopwords)

    return lemmatized_tokens


tokenized_data = df['review_body'].apply(process_text)
```

```
/var/folders/64/glschqjx3sn28wyvf5wwcx2r0000gn/T/ipykernel_1842/1517175790.py:5: MarkupResemblesLocatorWarning: The input looks more like a filename than markup. You may want to open this file and pass the filehandle into Beautiful Soup.
  df['review_body'] = df['review_body'].apply(lambda x: BeautifulSoup(x, 'html.parser').get_text())
/var/folders/64/glschqjx3sn28wyvf5wwcx2r0000gn/T/ipykernel_1842/1517175790.py:5: MarkupResemblesLocatorWarning: The input looks more like a URL than markup. You may want to use an HTTP client like requests to get the document behind the URL, and feed that document to Beautiful Soup.
  df['review_body'] = df['review_body'].apply(lambda x: BeautifulSoup(x, 'html.parser').get_text())
```

In [3]:

```python
tokenized_data
```

Out[3]:

```
0          [photo, deceiving, make, look, like, set, pen,...
1          [worst, label, ever, purchased, label, try, re...
2          [product, broke, short, time, also, poor, job,...
3          [printer, head, malfunctioning, since, install...
4          [item, shipped, excited, outside, great, quali...
                                ...
249995                               [produce, great, print]
249996     [perfect, high, school, student, use, math, cl...
249997                                  [product, excellent]
249998               [arrived, fast, work, great, good, buy]
249999     [glad, bought, headset, hear, better, ever, un...
Name: review_body, Length: 250000, dtype: object
```

In [5]:

```python
type(tokenized_data)
```

Out[5]:

```
pandas.core.series.Series
```

Training Word2Vec

In [37]:

```python
from gensim.models import Word2Vec

model = Word2Vec(tokenized_data, vector_size=300, window=11, min_count=10)


model.save("word2vec.model")
```

In [4]:

```python
from gensim.models import Word2Vec
# Load model
```

```
model = Word2Vec.load("word2vec.model")
```

In [5]:

```
similarity = model.wv.similarity('happy', 'impressed')
print(similarity)
```

```
0.5609798
```

In [41]:

```
w1='excellent'
w2='outstanding'
print('%r\t%r\t%.2f' % (w1, w2, model.wv.similarity(w1, w2)))
```

```
'excellent' 'outstanding' 0.78
```

In [42]:

```
word_pairs = [
    ('smartphone', 'camera'), ('laptop', 'charger'), ('headphone', 'bluetooth'),
    ('novel', 'author'), ('fiction', 'character'),
]


def compare_similarities(model1, model2, word_pairs):

    results = []

    for word1, word2 in word_pairs:
        similarity_model1 = model1.similarity(word1, word2)
        similarity_model2 = model2.similarity(word1, word2)
        results.append({
            'Word Pair': f'{word1}, {word2}',
            'Pretrained Model Similarity': round(similarity_model1, 2),
            'Amazon Review Model Similarity': round(similarity_model2, 2)
        })


    results_df = pd.DataFrame(results)


    print(results_df)


compare_similarities(wv, model.wv, word_pairs)
```

```
               Word Pair  Pretrained Model Similarity  \
0    smartphone, camera                         0.32
1       laptop, charger                         0.47
2  headphone, bluetooth                         0.49
3         novel, author                         0.46
4    fiction, character                         0.25

   Amazon Review Model Similarity
0                            0.41
1                            0.25
2                            0.45
3                            0.58
4                            0.31
```

Simple Models

Perceptron and SVM with Custom Word2Vec

In [43]:

```
import numpy as np
```

```
def average_word2vec(reviews, word2vec_model, vector_size):
    features = []

    for review in reviews:

        valid_words = [word for word in review if word in word2vec_model.wv.key_to_index
]

        if not valid_words:

            features.append(np.zeros(vector_size))
            continue


        word_vectors = np.array([word2vec_model.wv[word] for word in valid_words])
        avg_vector = word_vectors.mean(axis=0)
        features.append(avg_vector)

    return np.array(features)


avg_features = average_word2vec(tokenized_data, model, vector_size=300)
```

In [44]:

```
df_filtered = df[df['class'] != 3]

filtered_indices = df_filtered.index.to_numpy()

avg_features_filtered = avg_features[filtered_indices]


print("Filtered DataFrame shape:", df_filtered.shape)
print("Filtered avg_features shape:", avg_features_filtered.shape)
```

```
Filtered DataFrame shape: (200000, 3)
Filtered avg_features shape: (200000, 300)
```

In [45]:

```
X=avg_features_filtered
y=df_filtered['class']
```

In [46]:

```
X.shape, y.shape
```

Out[46]:

```
((200000, 300), (200000,))
```

In [47]:

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split

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

In [48]:

```
from sklearn.linear_model import Perceptron
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

perceptron_model = Perceptron()

perceptron_model.fit(X_train, y_train)

y_train_pred = perceptron_model.predict(X_train)
y_test_pred = perceptron_model.predict(X_test)
```

```
accuracy_train = accuracy_score(y_train, y_train_pred)
precision_train = precision_score(y_train, y_train_pred)
recall_train = recall_score(y_train, y_train_pred)
f1_train = f1_score(y_train, y_train_pred)
accuracy_test = accuracy_score(y_test, y_test_pred)
precision_test = precision_score(y_test, y_test_pred)
recall_test = recall_score(y_test, y_test_pred)
f1_test = f1_score(y_test, y_test_pred)


print("\nTraining Metrics For Perceptron:")
print(f"Accuracy: {accuracy_train}")
print(f"Precision: {precision_train}")
print(f"Recall: {recall_train}")
print(f"F1-Score: {f1_train}")
print("Testing Metrics For Perceptron:")
print(f"Accuracy: {accuracy_test}")
print(f"Precision: {precision_test}")
print(f"Recall: {recall_test}")
print(f"F1-Score: {f1_test}")
```

```
Training Metrics For Perceptron:
Accuracy: 0.80206875
Precision: 0.7983036618188103
Recall: 0.8083332291575512
F1-Score: 0.8032871402749222
Testing Metrics For Perceptron:
Accuracy: 0.803375
Precision: 0.7995657751899734
Recall: 0.8099165292147749
F1-Score: 0.8047078687954708
```

In [49]:

```
from sklearn.svm import LinearSVC

svm_model = LinearSVC()

svm_model.fit(X_train, y_train)

y_train_pred = svm_model.predict(X_train)
y_test_pred = svm_model.predict(X_test)

accuracy_train = accuracy_score(y_train, y_train_pred)
precision_train = precision_score(y_train, y_train_pred)
recall_train = recall_score(y_train, y_train_pred)
f1_train = f1_score(y_train, y_train_pred)

# Evaluate the model on testing data
accuracy_test = accuracy_score(y_test, y_test_pred)
precision_test = precision_score(y_test, y_test_pred)
recall_test = recall_score(y_test, y_test_pred)
f1_test = f1_score(y_test, y_test_pred)

# Print the results
print("\nTraining Metrics For SVM:")
print(f"Accuracy: {accuracy_train}")
print(f"Precision: {precision_train}")
print(f"Recall: {recall_train}")
print(f"F1-Score: {f1_train}")
print("Testing Metrics For SVM:")
print(f"Accuracy: {accuracy_test}")
print(f"Precision: {precision_test}")
print(f"Recall: {recall_test}")
print(f"F1-Score: {f1_test}")
```

```
/Users/darshanrao/anaconda3/lib/python3.11/site-packages/sklearn/svm/_classes.py:32: FutureWarning: The default value of `dual` will change from `True` to `'auto'` in 1.5. Set the value of `dual` explicitly to suppress the warning.
  warnings.warn(
```

```
/Users/darshanrao/anaconda3/lib/python3.11/site-packages/sklearn/svm/_base.py:1242: Conve
rgenceWarning: Liblinear failed to converge, increase the number of iterations.
  warnings.warn(
```

```
Training Metrics For SVM:
Accuracy: 0.84153125
Precision: 0.8524577473874339
Recall: 0.8259972747615416
F1-Score: 0.8390189393217906
Testing Metrics For SVM:
Accuracy: 0.84235
Precision: 0.8529184483025088
Recall: 0.8275103713700205
F1-Score: 0.8400223248262214
```

Perceptron and SVM with Google Word2Vec

In [50]:

```python
def average_word2vec_google(reviews,vector_size):
    features = []

    for review in reviews:

        valid_words = [word for word in review if word in wv.key_to_index]

        if not valid_words:

            features.append(np.zeros(vector_size))
            continue


        word_vectors = np.array([wv[word] for word in valid_words])
        avg_vector = word_vectors.mean(axis=0)
        features.append(avg_vector)

    return np.array(features)


avg_features_pretrained = average_word2vec_google(tokenized_data,vector_size=300)
```

In [51]:

```python
# Get the indices of the remaining rows after filtering
filtered_indices = df_filtered.index.to_numpy()

# Now, use these indices to filter the avg_features array
avg_features_filtered_pretrained = avg_features_pretrained[filtered_indices]

# Checking the dimensions to ensure they match
print("Filtered DataFrame shape:", df_filtered.shape)
print("Filtered avg_features shape:", avg_features_filtered_pretrained.shape)
```

```
Filtered DataFrame shape: (200000, 3)
Filtered avg_features shape: (200000, 300)
```

In [52]:

```python
X=avg_features_filtered_pretrained
y=df_filtered['class']

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

In [54]:

```python
from sklearn.linear_model import Perceptron
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

perceptron_model = Perceptron()
```

```python
perceptron_model.fit(X_train, y_train)

y_train_pred = perceptron_model.predict(X_train)
y_test_pred = perceptron_model.predict(X_test)


accuracy_train = accuracy_score(y_train, y_train_pred)
precision_train = precision_score(y_train, y_train_pred)
recall_train = recall_score(y_train, y_train_pred)
f1_train = f1_score(y_train, y_train_pred)
accuracy_test = accuracy_score(y_test, y_test_pred)
precision_test = precision_score(y_test, y_test_pred)
recall_test = recall_score(y_test, y_test_pred)
f1_test = f1_score(y_test, y_test_pred)


print("\nTraining Metrics For Perceptron:")
print(f"Accuracy: {accuracy_train}")
print(f"Precision: {precision_train}")
print(f"Recall: {recall_train}")
print(f"F1-Score: {f1_train}")
print("Testing Metrics For Perceptron:")
print(f"Accuracy: {accuracy_test}")
print(f"Precision: {precision_test}")
print(f"Recall: {recall_test}")
print(f"F1-Score: {f1_test}")
```

```
Training Metrics For Perceptron:
Accuracy: 0.7436875
Precision: 0.8928074807037343
Recall: 0.5538234595527108
F1-Score: 0.6835989939358403
Testing Metrics For Perceptron:
Accuracy: 0.7438
Precision: 0.8917074737095609
Recall: 0.5552056780126956
F1-Score: 0.6843272548053229
```

In [55]:

```python
from sklearn.svm import LinearSVC

svm_model = LinearSVC()

svm_model.fit(X_train, y_train)

y_train_pred = svm_model.predict(X_train)
y_test_pred = svm_model.predict(X_test)

accuracy_train = accuracy_score(y_train, y_train_pred)
precision_train = precision_score(y_train, y_train_pred)
recall_train = recall_score(y_train, y_train_pred)
f1_train = f1_score(y_train, y_train_pred)

# Evaluate the model on testing data
accuracy_test = accuracy_score(y_test, y_test_pred)
precision_test = precision_score(y_test, y_test_pred)
recall_test = recall_score(y_test, y_test_pred)
f1_test = f1_score(y_test, y_test_pred)

# Print the results
print("\nTraining Metrics For SVM:")
print(f"Accuracy: {accuracy_train}")
print(f"Precision: {precision_train}")
print(f"Recall: {recall_train}")
print(f"F1-Score: {f1_train}")
print("Testing Metrics For SVM:")
print(f"Accuracy: {accuracy_test}")
print(f"Precision: {precision_test}")
print(f"Recall: {recall_test}")
print(f"F1-Score: {f1_test}")
```

```
/Users/darshanrao/anaconda3/lib/python3.11/site-packages/sklearn/svm/_classes.py:32: FutureWarning: The default value of `dual` will change from `True` to `'auto'` in 1.5. Set the value of `dual` explicitly to suppress the warning.
  warnings.warn(
```

```
Training Metrics For SVM:
Accuracy: 0.81735625
Precision: 0.8353191376941773
Recall: 0.7905316715212581
F1-Score: 0.8123085223222029
Testing Metrics For SVM:
Accuracy: 0.816875
Precision: 0.8335963804713805
Recall: 0.7919728095166692
F1-Score: 0.8122516980648469
```

Feedforward Neural Networks

Feedforward Neural Networks Average Values Custom Word2Vec Binary

In [57]:

```
import torch
from torch import nn
from torch.utils.data import TensorDataset, DataLoader
```

In [58]:

```
X=avg_features_filtered
y=df_filtered['class']

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

In [61]:

```
from sklearn.model_selection import train_test_split
from torch.optim import lr_scheduler

# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.long)
X_val_tensor = torch.tensor(X_test, dtype=torch.float32)
y_val_tensor = torch.tensor(y_test.values, dtype=torch.long)
y_train_tensor = y_train_tensor - 1
y_val_tensor = y_val_tensor - 1
# Create datasets and dataloaders
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
val_dataset = TensorDataset(X_val_tensor, y_val_tensor)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)

# Define the MLP model
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(300, 50),
            nn.ReLU(),
            nn.Linear(50, 10),
            nn.ReLU(),
            nn.Linear(10, 2)
        )

    def forward(self, x):
        return self.layers(x)
```

```python
# Initialize the model, loss function, and optimizer
mlp_model = MLP()
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(mlp_model.parameters(), lr=0.001)

def train_model(model, criterion, optimizer, train_loader, val_loader, epochs=10, patien
ce=3):
    # Initialize early stopping variables
    best_val_loss = float('inf')
    epochs_no_improve = 0
    early_stop = False

    # Scheduler for learning rate decay
    scheduler = lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=patience // 2,
factor=0.1, verbose=True)

    for epoch in range(epochs):
        model.train()
        for data, target in train_loader:
            optimizer.zero_grad() # clears old gradients,
            output = model(data)
            loss = criterion(output, target) # Calculate the pred-actual Loss
            loss.backward() # Back proprogation (Calculating the gradient)
            optimizer.step() # Weight updates

        # Validation phase
        model.eval() # Switches the into evaluation mode
        val_loss = 0
        correct = 0
        with torch.no_grad(): # Ensures gradient is not calculated(Saves memory and compu
tation)
            for data, target in val_loader:
                output = model(data) ## Output in the form of probabitilities
                val_loss += criterion(output, target).item()
                pred = output.argmax(dim=1, keepdim=True) # Probabitlity with the max va
lue is the output
                correct += pred.eq(target.view_as(pred)).sum().item() # counts the numbe
r of correct predictions

        val_loss /= len(val_loader.dataset)
        accuracy = 100. * correct / len(val_loader.dataset)
        print(f'Epoch: {epoch+1}, Validation Loss: {val_loss:.4f}, Accuracy: {accuracy:.
2f}%')

        # Early stopping logic
        if val_loss < best_val_loss: # if the  current loss is less the best loss
            best_val_loss = val_loss
            epochs_no_improve = 0
        else: # no improvement in loss
            epochs_no_improve += 1
            if epochs_no_improve >= patience:
                print('Early stopping triggered. Training stopped.')
                early_stop = True
                break

        # Learning rate scheduler step
        scheduler.step(val_loss)

        if early_stop:
            print("Stopped early at epoch:", epoch+1)
            break


train_model(mlp_model, criterion, optimizer, train_loader, val_loader, epochs=100, patie
nce=3)
```

```
/Users/darshanrao/anaconda3/lib/python3.11/site-packages/torch/optim/lr_scheduler.py:28:
UserWarning: The verbose parameter is deprecated. Please use get_last_lr() to access the
learning rate.
  warnings.warn("The verbose parameter is deprecated. Please use get_last_lr() "
```

```
Epoch: 1, Validation Loss: 0.0054, Accuracy: 85.55%
```

```
Epoch: 1, Validation Loss: 0.0054, Accuracy: 85.55%
Epoch: 2, Validation Loss: 0.0052, Accuracy: 85.98%
Epoch: 3, Validation Loss: 0.0052, Accuracy: 85.82%
Epoch: 4, Validation Loss: 0.0051, Accuracy: 86.12%
Epoch: 5, Validation Loss: 0.0051, Accuracy: 86.17%
Epoch: 6, Validation Loss: 0.0051, Accuracy: 86.22%
Epoch: 7, Validation Loss: 0.0051, Accuracy: 86.22%
Epoch: 8, Validation Loss: 0.0050, Accuracy: 86.59%
Epoch: 9, Validation Loss: 0.0050, Accuracy: 86.60%
Epoch: 10, Validation Loss: 0.0050, Accuracy: 86.58%
Epoch: 11, Validation Loss: 0.0050, Accuracy: 86.53%
Epoch: 12, Validation Loss: 0.0050, Accuracy: 86.57%
Epoch: 13, Validation Loss: 0.0050, Accuracy: 86.56%
Epoch: 14, Validation Loss: 0.0050, Accuracy: 86.54%
Early stopping triggered. Training stopped.
```

In [62]:

```
# Evaluate the model on training data
y_train_pred = mlp_model(X_train_tensor).argmax(dim=1).numpy()
accuracy_train = accuracy_score(y_train_tensor.numpy(), y_train_pred)
precision_train = precision_score(y_train_tensor.numpy(), y_train_pred)
recall_train = recall_score(y_train_tensor.numpy(), y_train_pred)
f1_train = f1_score(y_train_tensor.numpy(), y_train_pred)

# Evaluate the model on testing data
y_test_pred = mlp_model(X_val_tensor).argmax(dim=1).numpy()
accuracy_test = accuracy_score(y_val_tensor.numpy(), y_test_pred)
precision_test = precision_score(y_val_tensor.numpy(), y_test_pred)
recall_test = recall_score(y_val_tensor.numpy(), y_test_pred)
f1_test = f1_score(y_val_tensor.numpy(), y_test_pred)

# Print the results
print("Training Metrics For MLP:")
print(f"Accuracy: {accuracy_train:.4f}")
print(f"Precision: {precision_train:.4f}")
print(f"Recall: {recall_train:.4f}")
print(f"F1-Score: {f1_train:.4f}")
print("\nTesting Metrics For MLP:")
print(f"Accuracy: {accuracy_test:.4f}")
print(f"Precision: {precision_test:.4f}")
print(f"Recall: {recall_test:.4f}")
print(f"F1-Score: {f1_test:.4f}")
```

```
Training Metrics For MLP:
Accuracy: 0.8796
Precision: 0.8803
Recall: 0.8787
F1-Score: 0.8795

Testing Metrics For MLP:
Accuracy: 0.8654
Precision: 0.8664
Recall: 0.8640
F1-Score: 0.8652
```

Feedforward Neural Networks Average Values Google Word2Vec Binary

In [63]:

```
X=avg_features_filtered_pretrained
y=df_filtered['class']

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

In [64]:

```
X.shape
```

Out[64]:

```
(200000, 300)
```

In [65]:

```
from sklearn.model_selection import train_test_split
from torch.optim import lr_scheduler

# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.long)
X_val_tensor = torch.tensor(X_test, dtype=torch.float32)
y_val_tensor = torch.tensor(y_test.values, dtype=torch.long)
y_train_tensor = y_train_tensor - 1
y_val_tensor = y_val_tensor - 1
# Create datasets and dataloaders
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
val_dataset = TensorDataset(X_val_tensor, y_val_tensor)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)

# Define the MLP model
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(300, 50),
            nn.ReLU(),
            nn.Linear(50, 10),
            nn.ReLU(),
            nn.Linear(10, 2)
        )

    def forward(self, x):
        return self.layers(x)

# Initialize the model, loss function, and optimizer
mlp_model = MLP()
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(mlp_model.parameters(), lr=0.001)

def train_model(model, criterion, optimizer, train_loader, val_loader, epochs=10, patien
ce=3):
    # Initialize early stopping variables
    best_val_loss = float('inf')
    epochs_no_improve = 0
    early_stop = False

    # Scheduler for learning rate decay
    scheduler = lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=patience // 2,
factor=0.1, verbose=True)

    for epoch in range(epochs):
        model.train()
        for data, target in train_loader:
            optimizer.zero_grad() # clears old gradients,
            output = model(data)
            loss = criterion(output, target) # Calculate the pred-actual Loss
            loss.backward() # Back proprogation (Calculating the gradient)
            optimizer.step() # Weight updates

        # Validation phase
        model.eval() # Switches the into evaluation mode
        val_loss = 0
        correct = 0
        with torch.no_grad(): # Ensures gradient is not calculated(Saves memory and compu
tation)
            for data, target in val_loader:
                output = model(data) ## Output in the form of probabitilities
                val_loss += criterion(output, target).item()
```

```
            pred = output.argmax(dim=1, keepdim=True) # Probabitlity with the max va
lue is the output
            correct += pred.eq(target.view_as(pred)).sum().item() # counts the numbe
r of correct predictions

    val_loss /= len(val_loader.dataset)
    accuracy = 100. * correct / len(val_loader.dataset)
    print(f'Epoch: {epoch+1}, Validation Loss: {val_loss:.4f}, Accuracy: {accuracy:.
2f}%')

    # Early stopping logic
    if val_loss < best_val_loss: # if the  current loss is less the best loss
        best_val_loss = val_loss
        epochs_no_improve = 0
    else: # no improvement in loss
        epochs_no_improve += 1
        if epochs_no_improve >= patience:
            print('Early stopping triggered. Training stopped.')
            early_stop = True
            break

    # Learning rate scheduler step
    scheduler.step(val_loss)

    if early_stop:
        print("Stopped early at epoch:", epoch+1)
        break


train_model(mlp_model, criterion, optimizer, train_loader, val_loader, epochs=100, patie
nce=3)
```

```
/Users/darshanrao/anaconda3/lib/python3.11/site-packages/torch/optim/lr_scheduler.py:28:
UserWarning: The verbose parameter is deprecated. Please use get_last_lr() to access the
learning rate.
  warnings.warn("The verbose parameter is deprecated. Please use get_last_lr() "
```

```
Epoch: 1, Validation Loss: 0.0061, Accuracy: 82.96%
Epoch: 2, Validation Loss: 0.0059, Accuracy: 83.64%
Epoch: 3, Validation Loss: 0.0058, Accuracy: 83.66%
Epoch: 4, Validation Loss: 0.0058, Accuracy: 83.50%
Epoch: 5, Validation Loss: 0.0057, Accuracy: 84.24%
Epoch: 6, Validation Loss: 0.0056, Accuracy: 84.35%
Epoch: 7, Validation Loss: 0.0057, Accuracy: 84.30%
Epoch: 8, Validation Loss: 0.0056, Accuracy: 84.14%
Epoch: 9, Validation Loss: 0.0055, Accuracy: 84.54%
Epoch: 10, Validation Loss: 0.0055, Accuracy: 84.57%
Epoch: 11, Validation Loss: 0.0055, Accuracy: 84.64%
Epoch: 12, Validation Loss: 0.0055, Accuracy: 84.43%
Epoch: 13, Validation Loss: 0.0055, Accuracy: 84.53%
Epoch: 14, Validation Loss: 0.0055, Accuracy: 84.55%
Epoch: 15, Validation Loss: 0.0055, Accuracy: 84.54%
Epoch: 16, Validation Loss: 0.0055, Accuracy: 84.54%
Epoch: 17, Validation Loss: 0.0055, Accuracy: 84.53%
Early stopping triggered. Training stopped.
```

In [66]:

```
# Evaluate the model on training data
y_train_pred = mlp_model(X_train_tensor).argmax(dim=1).numpy()
accuracy_train = accuracy_score(y_train_tensor.numpy(), y_train_pred)
precision_train = precision_score(y_train_tensor.numpy(), y_train_pred)
recall_train = recall_score(y_train_tensor.numpy(), y_train_pred)
f1_train = f1_score(y_train_tensor.numpy(), y_train_pred)

# Evaluate the model on testing data
y_test_pred = mlp_model(X_val_tensor).argmax(dim=1).numpy()
accuracy_test = accuracy_score(y_val_tensor.numpy(), y_test_pred)
precision_test = precision_score(y_val_tensor.numpy(), y_test_pred)
recall_test = recall_score(y_val_tensor.numpy(), y_test_pred)
f1_test = f1_score(y_val_tensor.numpy(), y_test_pred)
```

```
# Print the results
print("Training Metrics For MLP:")
print(f"Accuracy: {accuracy_train:.4f}")
print(f"Precision: {precision_train:.4f}")
print(f"Recall: {recall_train:.4f}")
print(f"F1-Score: {f1_train:.4f}")
print("\nTesting Metrics For MLP:")
print(f"Accuracy: {accuracy_test:.4f}")
print(f"Precision: {precision_test:.4f}")
print(f"Recall: {recall_test:.4f}")
print(f"F1-Score: {f1_test:.4f}")
```

```
Training Metrics For MLP:
Accuracy: 0.8624
Precision: 0.8586
Recall: 0.8676
F1-Score: 0.8631

Testing Metrics For MLP:
Accuracy: 0.8453
Precision: 0.8416
Recall: 0.8505
F1-Score: 0.8460
```

Feedforward Neural Networks Average Values Custom Word2Vec Ternary

In [69]:

```
X=avg_features
y=df['class']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42
)
```

In [70]:

```
X.shape, y.shape
```

Out[70]:

```
((250000, 300), (250000,))
```

In [72]:

```
# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.long)
X_val_tensor = torch.tensor(X_test, dtype=torch.float32)
y_val_tensor = torch.tensor(y_test.values, dtype=torch.long)
y_train_tensor = y_train_tensor - 1
y_val_tensor = y_val_tensor - 1
# Create datasets and dataloaders
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
val_dataset = TensorDataset(X_val_tensor, y_val_tensor)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)

# Define the MLP model
class MLP_2(nn.Module):
    def __init__(self):
        super(MLP_2, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(300, 50),
            nn.ReLU(),
            nn.Linear(50, 10),
            nn.ReLU(),
            nn.Linear(10, 3)
        )

    def forward(self, x):
```

```
        return self.layers(x)

# Initialize the model, loss function, and optimizer
mlp_model_2 = MLP_2()
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(mlp_model_2.parameters(), lr=0.001)

def train_model(model, criterion, optimizer, train_loader, val_loader, epochs=10, patien
ce=3):
    # Initialize early stopping variables
    best_val_loss = float('inf')
    epochs_no_improve = 0
    early_stop = False

    # Scheduler for learning rate decay
    scheduler = lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=patience // 2,
factor=0.1, verbose=True)

    for epoch in range(epochs):
        model.train()
        for data, target in train_loader:
            optimizer.zero_grad() # clears old gradients,
            output = model(data)
            loss = criterion(output, target) # Calculate the pred-actual Loss
            loss.backward() # Back proprogation (Calculating the gradient)
            optimizer.step() # Weight updates

        # Validation phase
        model.eval() # Switches the into evaluation mode
        val_loss = 0
        correct = 0
        with torch.no_grad(): # Ensures gradient is not calculated(Saves memory and compu
tation)
            for data, target in val_loader:
                output = model(data) ## Output in the form of probabitilities
                val_loss += criterion(output, target).item()
                pred = output.argmax(dim=1, keepdim=True) # Probabitlity with the max va
lue is the output
                correct += pred.eq(target.view_as(pred)).sum().item() # counts the numbe
r of correct predictions

        val_loss /= len(val_loader.dataset)
        accuracy = 100. * correct / len(val_loader.dataset)
        print(f'Epoch: {epoch+1}, Validation Loss: {val_loss:.4f}, Accuracy: {accuracy:.
2f}%')

        # Early stopping logic
        if val_loss < best_val_loss: # if the  current loss is less the best loss
            best_val_loss = val_loss
            epochs_no_improve = 0
        else: # no improvement in loss
            epochs_no_improve += 1
            if epochs_no_improve >= patience:
                print('Early stopping triggered. Training stopped.')
                early_stop = True
                break

        # Learning rate scheduler step
        scheduler.step(val_loss)

        if early_stop:
            print("Stopped early at epoch:", epoch+1)
            break


train_model(mlp_model_2, criterion, optimizer, train_loader, val_loader, epochs=100, pat
ience=10)
```

```
/Users/darshanrao/anaconda3/lib/python3.11/site-packages/torch/optim/lr_scheduler.py:28:
UserWarning: The verbose parameter is deprecated. Please use get_last_lr() to access the
learning rate.
  warnings.warn("The verbose parameter is deprecated. Please use get_last_lr() "
```

```
Epoch: 1, Validation Loss: 0.0114, Accuracy: 68.62%
Epoch: 2, Validation Loss: 0.0111, Accuracy: 69.70%
Epoch: 3, Validation Loss: 0.0110, Accuracy: 69.88%
Epoch: 4, Validation Loss: 0.0111, Accuracy: 69.75%
Epoch: 5, Validation Loss: 0.0110, Accuracy: 69.82%
Epoch: 6, Validation Loss: 0.0109, Accuracy: 70.02%
Epoch: 7, Validation Loss: 0.0110, Accuracy: 70.02%
Epoch: 8, Validation Loss: 0.0109, Accuracy: 70.06%
Epoch: 9, Validation Loss: 0.0109, Accuracy: 70.16%
Epoch: 10, Validation Loss: 0.0109, Accuracy: 70.12%
Epoch: 11, Validation Loss: 0.0109, Accuracy: 70.30%
Epoch: 12, Validation Loss: 0.0109, Accuracy: 70.40%
Epoch: 13, Validation Loss: 0.0110, Accuracy: 70.00%
Epoch: 14, Validation Loss: 0.0109, Accuracy: 70.03%
Epoch: 15, Validation Loss: 0.0110, Accuracy: 69.65%
Epoch: 16, Validation Loss: 0.0109, Accuracy: 70.18%
Epoch: 17, Validation Loss: 0.0109, Accuracy: 69.96%
Epoch: 18, Validation Loss: 0.0109, Accuracy: 70.24%
Epoch: 19, Validation Loss: 0.0108, Accuracy: 70.39%
Epoch: 20, Validation Loss: 0.0108, Accuracy: 70.39%
Epoch: 21, Validation Loss: 0.0108, Accuracy: 70.47%
Epoch: 22, Validation Loss: 0.0108, Accuracy: 70.42%
Epoch: 23, Validation Loss: 0.0109, Accuracy: 70.42%
Epoch: 24, Validation Loss: 0.0109, Accuracy: 70.39%
Epoch: 25, Validation Loss: 0.0109, Accuracy: 70.43%
Epoch: 26, Validation Loss: 0.0109, Accuracy: 70.44%
Epoch: 27, Validation Loss: 0.0109, Accuracy: 70.45%
Epoch: 28, Validation Loss: 0.0109, Accuracy: 70.41%
Epoch: 29, Validation Loss: 0.0109, Accuracy: 70.41%
Early stopping triggered. Training stopped.
```

In [75]:

```python
# Evaluate the model on training data
y_train_pred = mlp_model_2(X_train_tensor).argmax(dim=1).numpy()
accuracy_train = accuracy_score(y_train_tensor.numpy(), y_train_pred)
precision_train = precision_score(y_train_tensor.numpy(), y_train_pred, average='weighted
')
recall_train = recall_score(y_train_tensor.numpy(), y_train_pred, average='weighted')
f1_train = f1_score(y_train_tensor.numpy(), y_train_pred, average='weighted')

# Evaluate the model on testing data
y_test_pred = mlp_model_2(X_val_tensor).argmax(dim=1).numpy()
accuracy_test = accuracy_score(y_val_tensor.numpy(), y_test_pred)
precision_test = precision_score(y_val_tensor.numpy(), y_test_pred, average='weighted')
recall_test = recall_score(y_val_tensor.numpy(), y_test_pred, average='weighted')
f1_test = f1_score(y_val_tensor.numpy(), y_test_pred, average='weighted')

# Print the results
print("Training Metrics For MLP:")
print(f"Accuracy: {accuracy_train:.4f}")
print(f"Precision: {precision_train:.4f}")
print(f"Recall: {recall_train:.4f}")
print(f"F1-Score: {f1_train:.4f}")
print("\nTesting Metrics For MLP:")
print(f"Accuracy: {accuracy_test:.4f}")
print(f"Precision: {precision_test:.4f}")
print(f"Recall: {recall_test:.4f}")
print(f"F1-Score: {f1_test:.4f}")
```

```
Training Metrics For MLP:
Accuracy: 0.7287
Precision: 0.7059
Recall: 0.7287
F1-Score: 0.7091

Testing Metrics For MLP:
Accuracy: 0.7041
Precision: 0.6781
Recall: 0.7041
F1-Score: 0.6834
```

Feedforward Neural Networks Average Values Google Word2Vec Ternary

In [ ]:

```
X=avg_features_pretrained
y=df['class']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42
)
```

In [76]:

```
# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.long)
X_val_tensor = torch.tensor(X_test, dtype=torch.float32)
y_val_tensor = torch.tensor(y_test.values, dtype=torch.long)
y_train_tensor = y_train_tensor - 1
y_val_tensor = y_val_tensor - 1
# Create datasets and dataloaders
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
val_dataset = TensorDataset(X_val_tensor, y_val_tensor)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)

# Define the MLP model
class MLP_2(nn.Module):
    def __init__(self):
        super(MLP_2, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(300, 50),
            nn.ReLU(),
            nn.Linear(50, 10),
            nn.ReLU(),
            nn.Linear(10, 3)
        )

    def forward(self, x):
        return self.layers(x)

# Initialize the model, loss function, and optimizer
mlp_model_2 = MLP_2()
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(mlp_model_2.parameters(), lr=0.001)

def train_model(model, criterion, optimizer, train_loader, val_loader, epochs=10, patien
ce=3):
    # Initialize early stopping variables
    best_val_loss = float('inf')
    epochs_no_improve = 0
    early_stop = False

    # Scheduler for learning rate decay
    scheduler = lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=patience // 2,
factor=0.1, verbose=True)

    for epoch in range(epochs):
        model.train()
        for data, target in train_loader:
            optimizer.zero_grad() # clears old gradients,
            output = model(data)
            loss = criterion(output, target) # Calculate the pred-actual Loss
            loss.backward() # Back proprogation (Calculating the gradient)
            optimizer.step() # Weight updates

        # Validation phase
        model.eval() # Switches the into evaluation mode
        val_loss = 0
        correct = 0
```

```
        with torch.no_grad(): # Ensures gradient is not calculated(Saves memory and compu
tation)
            for data, target in val_loader:
                output = model(data) ## Output in the form of probabitilities
                val_loss += criterion(output, target).item()
                pred = output.argmax(dim=1, keepdim=True) # Probabitlity with the max va
lue is the output
                correct += pred.eq(target.view_as(pred)).sum().item() # counts the numbe
r of correct predictions

        val_loss /= len(val_loader.dataset)
        accuracy = 100. * correct / len(val_loader.dataset)
        print(f'Epoch: {epoch+1}, Validation Loss: {val_loss:.4f}, Accuracy: {accuracy:.
2f}%')

        # Early stopping logic
        if val_loss < best_val_loss: # if the  current loss is less the best loss
            best_val_loss = val_loss
            epochs_no_improve = 0
        else: # no improvement in loss
            epochs_no_improve += 1
            if epochs_no_improve >= patience:
                print('Early stopping triggered. Training stopped.')
                early_stop = True
                break

        # Learning rate scheduler step
        scheduler.step(val_loss)

        if early_stop:
            print("Stopped early at epoch:", epoch+1)
            break


train_model(mlp_model_2, criterion, optimizer, train_loader, val_loader, epochs=100, pat
ience=10)
```

```
/Users/darshanrao/anaconda3/lib/python3.11/site-packages/torch/optim/lr_scheduler.py:28:
UserWarning: The verbose parameter is deprecated. Please use get_last_lr() to access the
learning rate.
  warnings.warn("The verbose parameter is deprecated. Please use get_last_lr() "
```

```
Epoch: 1, Validation Loss: 0.0113, Accuracy: 69.02%
Epoch: 2, Validation Loss: 0.0111, Accuracy: 69.71%
Epoch: 3, Validation Loss: 0.0111, Accuracy: 69.80%
Epoch: 4, Validation Loss: 0.0111, Accuracy: 69.51%
Epoch: 5, Validation Loss: 0.0110, Accuracy: 69.90%
Epoch: 6, Validation Loss: 0.0109, Accuracy: 69.96%
Epoch: 7, Validation Loss: 0.0109, Accuracy: 70.13%
Epoch: 8, Validation Loss: 0.0109, Accuracy: 70.10%
Epoch: 9, Validation Loss: 0.0109, Accuracy: 70.01%
Epoch: 10, Validation Loss: 0.0109, Accuracy: 70.29%
Epoch: 11, Validation Loss: 0.0109, Accuracy: 70.06%
Epoch: 12, Validation Loss: 0.0109, Accuracy: 70.03%
Epoch: 13, Validation Loss: 0.0109, Accuracy: 70.26%
Epoch: 14, Validation Loss: 0.0109, Accuracy: 70.15%
Epoch: 15, Validation Loss: 0.0109, Accuracy: 70.01%
Epoch: 16, Validation Loss: 0.0109, Accuracy: 70.07%
Epoch: 17, Validation Loss: 0.0108, Accuracy: 70.21%
Epoch: 18, Validation Loss: 0.0108, Accuracy: 70.23%
Epoch: 19, Validation Loss: 0.0109, Accuracy: 70.19%
Epoch: 20, Validation Loss: 0.0109, Accuracy: 70.10%
Epoch: 21, Validation Loss: 0.0109, Accuracy: 70.19%
Epoch: 22, Validation Loss: 0.0109, Accuracy: 70.17%
Epoch: 23, Validation Loss: 0.0109, Accuracy: 70.16%
Epoch: 24, Validation Loss: 0.0109, Accuracy: 70.12%
Epoch: 25, Validation Loss: 0.0109, Accuracy: 70.16%
Epoch: 26, Validation Loss: 0.0109, Accuracy: 70.17%
Epoch: 27, Validation Loss: 0.0109, Accuracy: 70.14%
Epoch: 28, Validation Loss: 0.0109, Accuracy: 70.15%
Early stopping triggered. Training stopped.
```

In [77]:

```
# Evaluate the model on training data
y_train_pred = mlp_model_2(X_train_tensor).argmax(dim=1).numpy()
accuracy_train = accuracy_score(y_train_tensor.numpy(), y_train_pred)
precision_train = precision_score(y_train_tensor.numpy(), y_train_pred, average='weighted
')
recall_train = recall_score(y_train_tensor.numpy(), y_train_pred, average='weighted')
f1_train = f1_score(y_train_tensor.numpy(), y_train_pred, average='weighted')

# Evaluate the model on testing data
y_test_pred = mlp_model_2(X_val_tensor).argmax(dim=1).numpy()
accuracy_test = accuracy_score(y_val_tensor.numpy(), y_test_pred)
precision_test = precision_score(y_val_tensor.numpy(), y_test_pred, average='weighted')
recall_test = recall_score(y_val_tensor.numpy(), y_test_pred, average='weighted')
f1_test = f1_score(y_val_tensor.numpy(), y_test_pred, average='weighted')

# Print the results
print("Training Metrics For MLP:")
print(f"Accuracy: {accuracy_train:.4f}")
print(f"Precision: {precision_train:.4f}")
print(f"Recall: {recall_train:.4f}")
print(f"F1-Score: {f1_train:.4f}")
print("\nTesting Metrics For MLP:")
print(f"Accuracy: {accuracy_test:.4f}")
print(f"Precision: {precision_test:.4f}")
print(f"Recall: {recall_test:.4f}")
print(f"F1-Score: {f1_test:.4f}")
```

```
Training Metrics For MLP:
Accuracy: 0.7270
Precision: 0.7048
Recall: 0.7270
F1-Score: 0.7074

Testing Metrics For MLP:
Accuracy: 0.7015
Precision: 0.6753
Recall: 0.7015
F1-Score: 0.6804
```

Feedforward Neural Networks Concatenated Values Custom Word2Vec Binary

In [78]:

```
def concatenated_word2vec(reviews, word2vec_model, vector_size, concat_size=10):
    features = []

    for review in reviews:
        valid_words = [word for word in review if word in word2vec_model.wv.key_to_index
]

        if len(valid_words) >= concat_size:
            # Take the embeddings of the first 'concat_size' valid words
            word_vectors = np.array([word2vec_model.wv[word] for word in valid_words[:co
ncat_size]])
            concat_vector = word_vectors.flatten()
        else:
            # If there aren't enough valid words, pad the rest with zeros
            word_vectors = np.array([word2vec_model.wv[word] for word in valid_words] +
                                    [np.zeros(vector_size) for _ in range(concat_size -
len(valid_words))])
            concat_vector = word_vectors.flatten()

        features.append(concat_vector)

    return np.array(features)

# Assuming 'tokenized_data' is your list of tokenized reviews and 'model' is your trained
```

```
Word2Vec model
concat_features = concatenated_word2vec(tokenized_data, model, vector_size=300)
```

In [79]:

```
concat_features.shape
```

Out[79]:

```
(250000, 3000)
```

In [81]:

```
# Get the indices of the remaining rows after filtering
filtered_indices = df_filtered.index.to_numpy()

# Now, use these indices to filter the avg_features array
concat_features_filtered = concat_features[filtered_indices]

# Checking the dimensions to ensure they match
print("Filtered DataFrame shape:", df_filtered.shape)
print("Filtered concat_features shape:", concat_features_filtered.shape)
```

```
Filtered DataFrame shape: (200000, 3)
Filtered concat_features shape: (200000, 3000)
```

In [82]:

```
X=concat_features_filtered
y=df_filtered['class']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42
)
```

In [83]:

```
X.shape,y.shape
```

Out[83]:

```
((200000, 3000), (200000,))
```

In [84]:

```
# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.long)
X_val_tensor = torch.tensor(X_test, dtype=torch.float32)
y_val_tensor = torch.tensor(y_test.values, dtype=torch.long)
y_train_tensor = y_train_tensor - 1
y_val_tensor = y_val_tensor - 1
# Create datasets and dataloaders
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
val_dataset = TensorDataset(X_val_tensor, y_val_tensor)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)

# Define the MLP model
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(3000, 50),
            nn.ReLU(),
            nn.Linear(50, 10),
            nn.ReLU(),
            nn.Linear(10, 2)
        )

    def forward(self, x):
        return self.layers(x)

# Initialize the model, loss function, and optimizer
```

```python
mlp_model = MLP()
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(mlp_model.parameters(), lr=0.001)

def train_model(model, criterion, optimizer, train_loader, val_loader, epochs=10, patience=3):
    # Initialize early stopping variables
    best_val_loss = float('inf')
    epochs_no_improve = 0
    early_stop = False

    # Scheduler for learning rate decay
    scheduler = lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=patience // 2, factor=0.1, verbose=True)

    for epoch in range(epochs):
        model.train()
        for data, target in train_loader:
            optimizer.zero_grad() # clears old gradients,
            output = model(data)
            loss = criterion(output, target) # Calculate the pred-actual Loss
            loss.backward() # Back proprogation (Calculating the gradient)
            optimizer.step() # Weight updates

        # Validation phase
        model.eval() # Switches the into evaluation mode
        val_loss = 0
        correct = 0
        with torch.no_grad(): # Ensures gradient is not calculated(Saves memory and computation)
            for data, target in val_loader:
                output = model(data) ## Output in the form of probabitilities
                val_loss += criterion(output, target).item()
                pred = output.argmax(dim=1, keepdim=True) # Probabitlity with the max value is the output
                correct += pred.eq(target.view_as(pred)).sum().item() # counts the number of correct predictions

        val_loss /= len(val_loader.dataset)
        accuracy = 100. * correct / len(val_loader.dataset)
        print(f'Epoch: {epoch+1}, Validation Loss: {val_loss:.4f}, Accuracy: {accuracy:.2f}%')

        # Early stopping logic
        if val_loss < best_val_loss: # if the  current loss is less the best loss
            best_val_loss = val_loss
            epochs_no_improve = 0
        else: # no improvement in loss
            epochs_no_improve += 1
            if epochs_no_improve >= patience:
                print('Early stopping triggered. Training stopped.')
                early_stop = True
                break

        # Learning rate scheduler step
        scheduler.step(val_loss)

        if early_stop:
            print("Stopped early at epoch:", epoch+1)
            break


train_model(mlp_model, criterion, optimizer, train_loader, val_loader, epochs=100, patience=3)
```

```
/Users/darshanrao/anaconda3/lib/python3.11/site-packages/torch/optim/lr_scheduler.py:28:
UserWarning: The verbose parameter is deprecated. Please use get_last_lr() to access the
learning rate.
  warnings.warn("The verbose parameter is deprecated. Please use get_last_lr() "
```

```
Epoch: 1, Validation Loss: 0.0069, Accuracy: 79.24%
Epoch: 2, Validation Loss: 0.0068, Accuracy: 79.69%
Epoch: 3, Validation Loss: 0.0070, Accuracy: 79.29%
```

```
Epoch: 3, Validation Loss: 0.0070, Accuracy: 79.29%
Epoch: 4, Validation Loss: 0.0071, Accuracy: 79.18%
Epoch: 5, Validation Loss: 0.0076, Accuracy: 79.30%
Early stopping triggered. Training stopped.
```

In [85]:

```
# Evaluate the model on training data
y_train_pred = mlp_model(X_train_tensor).argmax(dim=1).numpy()
accuracy_train = accuracy_score(y_train_tensor.numpy(), y_train_pred)
precision_train = precision_score(y_train_tensor.numpy(), y_train_pred)
recall_train = recall_score(y_train_tensor.numpy(), y_train_pred)
f1_train = f1_score(y_train_tensor.numpy(), y_train_pred)

# Evaluate the model on testing data
y_test_pred = mlp_model(X_val_tensor).argmax(dim=1).numpy()
accuracy_test = accuracy_score(y_val_tensor.numpy(), y_test_pred)
precision_test = precision_score(y_val_tensor.numpy(), y_test_pred)
recall_test = recall_score(y_val_tensor.numpy(), y_test_pred)
f1_test = f1_score(y_val_tensor.numpy(), y_test_pred)

# Print the results
print("Training Metrics For MLP:")
print(f"Accuracy: {accuracy_train:.4f}")
print(f"Precision: {precision_train:.4f}")
print(f"Recall: {recall_train:.4f}")
print(f"F1-Score: {f1_train:.4f}")
print("\nTesting Metrics For MLP:")
print(f"Accuracy: {accuracy_test:.4f}")
print(f"Precision: {precision_test:.4f}")
print(f"Recall: {recall_test:.4f}")
print(f"F1-Score: {f1_test:.4f}")
```

```
Training Metrics For MLP:
Accuracy: 0.8921
Precision: 0.8866
Recall: 0.8992
F1-Score: 0.8929

Testing Metrics For MLP:
Accuracy: 0.7930
Precision: 0.7876
Recall: 0.8022
F1-Score: 0.7948
```

Feedforward Neural Networks Concatenated Values Google Word2Vec Binary

In [86]:

```
def concatenated_word2vec_google(reviews,vector_size, concat_size=10):
    features = []

    for review in reviews:
        valid_words = [word for word in review if word in wv.key_to_index]

        if len(valid_words) >= concat_size:
            # Take the embeddings of the first 'concat_size' valid words
            word_vectors = np.array([wv[word] for word in valid_words[:concat_size]])
            concat_vector = word_vectors.flatten()
        else:
            # If there aren't enough valid words, pad the rest with zeros
            word_vectors = np.array([wv[word] for word in valid_words] +
                                    [np.zeros(vector_size) for _ in range(concat_size -
len(valid_words))])
            concat_vector = word_vectors.flatten()

        features.append(concat_vector)

    return np.array(features)
```

```
# Assuming 'tokenized_data' is your list of tokenized reviews and 'model' is your trained
Word2Vec model
concat_features_pretrained = concatenated_word2vec_google(tokenized_data,vector_size=300)
```

In [87]:

```
concat_features_pretrained.shape
```

Out[87]:

```
(250000, 3000)
```

In [88]:

```
# Get the indices of the remaining rows after filtering
filtered_indices = df_filtered.index.to_numpy()

# Now, use these indices to filter the avg_features array
concat_features_filtered_pretrained = concat_features_pretrained[filtered_indices]

# Checking the dimensions to ensure they match
print("Filtered DataFrame shape:", df_filtered.shape)
print("Filtered concat_features shape:", concat_features_filtered_pretrained.shape)
```

```
Filtered DataFrame shape: (200000, 3)
Filtered concat_features shape: (200000, 3000)
```

In [89]:

```
X=concat_features_filtered_pretrained
y=df_filtered['class']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42
)
```

In [90]:

```
X.shape,y.shape
```

Out[90]:

```
((200000, 3000), (200000,))
```

In [91]:

```
# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.long)
X_val_tensor = torch.tensor(X_test, dtype=torch.float32)
y_val_tensor = torch.tensor(y_test.values, dtype=torch.long)
y_train_tensor = y_train_tensor - 1
y_val_tensor = y_val_tensor - 1
# Create datasets and dataloaders
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
val_dataset = TensorDataset(X_val_tensor, y_val_tensor)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)

# Define the MLP model
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(3000, 50),
            nn.ReLU(),
            nn.Linear(50, 10),
            nn.ReLU(),
            nn.Linear(10, 2)
        )

    def forward(self, x):
        return self.layers(x)
```

```python
# Initialize the model, loss function, and optimizer
mlp_model = MLP()
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(mlp_model.parameters(), lr=0.001)

def train_model(model, criterion, optimizer, train_loader, val_loader, epochs=10, patien
ce=3):
    # Initialize early stopping variables
    best_val_loss = float('inf')
    epochs_no_improve = 0
    early_stop = False

    # Scheduler for learning rate decay
    scheduler = lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=patience // 2,
factor=0.1, verbose=True)

    for epoch in range(epochs):
        model.train()
        for data, target in train_loader:
            optimizer.zero_grad() # clears old gradients,
            output = model(data)
            loss = criterion(output, target) # Calculate the pred-actual Loss
            loss.backward() # Back proprogation (Calculating the gradient)
            optimizer.step() # Weight updates

        # Validation phase
        model.eval() # Switches the into evaluation mode
        val_loss = 0
        correct = 0
        with torch.no_grad(): # Ensures gradient is not calculated(Saves memory and compu
tation)
            for data, target in val_loader:
                output = model(data) ## Output in the form of probabitilities
                val_loss += criterion(output, target).item()
                pred = output.argmax(dim=1, keepdim=True) # Probabitlity with the max va
lue is the output
                correct += pred.eq(target.view_as(pred)).sum().item() # counts the numbe
r of correct predictions

        val_loss /= len(val_loader.dataset)
        accuracy = 100. * correct / len(val_loader.dataset)
        print(f'Epoch: {epoch+1}, Validation Loss: {val_loss:.4f}, Accuracy: {accuracy:.
2f}%')

        # Early stopping logic
        if val_loss < best_val_loss: # if the  current loss is less the best loss
            best_val_loss = val_loss
            epochs_no_improve = 0
        else: # no improvement in loss
            epochs_no_improve += 1
            if epochs_no_improve >= patience:
                print('Early stopping triggered. Training stopped.')
                early_stop = True
                break

        # Learning rate scheduler step
        scheduler.step(val_loss)

        if early_stop:
            print("Stopped early at epoch:", epoch+1)
            break


train_model(mlp_model, criterion, optimizer, train_loader, val_loader, epochs=100, patie
nce=3)
```

```
/Users/darshanrao/anaconda3/lib/python3.11/site-packages/torch/optim/lr_scheduler.py:28:
UserWarning: The verbose parameter is deprecated. Please use get_last_lr() to access the
learning rate.
  warnings.warn("The verbose parameter is deprecated. Please use get_last_lr() "
```

```
Epoch: 1, Validation Loss: 0.0072, Accuracy: 77.88%
```

```
Epoch: 1, Validation Loss: 0.0072, Accuracy: 77.88%
Epoch: 2, Validation Loss: 0.0071, Accuracy: 78.58%
Epoch: 3, Validation Loss: 0.0072, Accuracy: 78.42%
Epoch: 4, Validation Loss: 0.0075, Accuracy: 77.86%
Epoch: 5, Validation Loss: 0.0081, Accuracy: 77.86%
Early stopping triggered. Training stopped.
```

In [92]:

```
# Evaluate the model on training data
y_train_pred = mlp_model(X_train_tensor).argmax(dim=1).numpy()
accuracy_train = accuracy_score(y_train_tensor.numpy(), y_train_pred)
precision_train = precision_score(y_train_tensor.numpy(), y_train_pred)
recall_train = recall_score(y_train_tensor.numpy(), y_train_pred)
f1_train = f1_score(y_train_tensor.numpy(), y_train_pred)

# Evaluate the model on testing data
y_test_pred = mlp_model(X_val_tensor).argmax(dim=1).numpy()
accuracy_test = accuracy_score(y_val_tensor.numpy(), y_test_pred)
precision_test = precision_score(y_val_tensor.numpy(), y_test_pred)
recall_test = recall_score(y_val_tensor.numpy(), y_test_pred)
f1_test = f1_score(y_val_tensor.numpy(), y_test_pred)

# Print the results
print("Training Metrics For MLP:")
print(f"Accuracy: {accuracy_train:.4f}")
print(f"Precision: {precision_train:.4f}")
print(f"Recall: {recall_train:.4f}")
print(f"F1-Score: {f1_train:.4f}")
print("\nTesting Metrics For MLP:")
print(f"Accuracy: {accuracy_test:.4f}")
print(f"Precision: {precision_test:.4f}")
print(f"Recall: {recall_test:.4f}")
print(f"F1-Score: {f1_test:.4f}")
```

```
Training Metrics For MLP:
Accuracy: 0.8966
Precision: 0.8920
Recall: 0.9025
F1-Score: 0.8972

Testing Metrics For MLP:
Accuracy: 0.7786
Precision: 0.7738
Recall: 0.7872
F1-Score: 0.7804
```

Feedforward Neural Networks Concatenated Values Custom Word2Vec Ternary

In [93]:

```
X=concat_features
y=df['class']

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

In [94]:

```
X.shape,y.shape
```

Out[94]:

```
((250000, 3000), (250000,))
```

In [95]:

```
# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
```

```python
y_train_tensor = torch.tensor(y_train.values, dtype=torch.long)
X_val_tensor = torch.tensor(X_test, dtype=torch.float32)
y_val_tensor = torch.tensor(y_test.values, dtype=torch.long)
y_train_tensor = y_train_tensor - 1
y_val_tensor = y_val_tensor - 1
# Create datasets and dataloaders
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
val_dataset = TensorDataset(X_val_tensor, y_val_tensor)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)

# Define the MLP model
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(3000, 50),
            nn.ReLU(),
            nn.Linear(50, 10),
            nn.ReLU(),
            nn.Linear(10, 3)
        )

    def forward(self, x):
        return self.layers(x)

# Initialize the model, loss function, and optimizer
mlp_model = MLP()
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(mlp_model.parameters(), lr=0.001)

def train_model(model, criterion, optimizer, train_loader, val_loader, epochs=10, patien
ce=3):
    # Initialize early stopping variables
    best_val_loss = float('inf')
    epochs_no_improve = 0
    early_stop = False

    # Scheduler for learning rate decay
    scheduler = lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=patience // 2,
factor=0.1, verbose=True)

    for epoch in range(epochs):
        model.train()
        for data, target in train_loader:
            optimizer.zero_grad() # clears old gradients,
            output = model(data)
            loss = criterion(output, target) # Calculate the pred-actual Loss
            loss.backward() # Back proprogation (Calculating the gradient)
            optimizer.step() # Weight updates

        # Validation phase
        model.eval() # Switches the into evaluation mode
        val_loss = 0
        correct = 0
        with torch.no_grad(): # Ensures gradient is not calculated(Saves memory and compu
tation)
            for data, target in val_loader:
                output = model(data) ## Output in the form of probabitilities
                val_loss += criterion(output, target).item()
                pred = output.argmax(dim=1, keepdim=True) # Probabitlity with the max va
lue is the output
                correct += pred.eq(target.view_as(pred)).sum().item() # counts the numbe
r of correct predictions

        val_loss /= len(val_loader.dataset)
        accuracy = 100. * correct / len(val_loader.dataset)
        print(f'Epoch: {epoch+1}, Validation Loss: {val_loss:.4f}, Accuracy: {accuracy:.
2f}%')

        # Early stopping logic
        if val_loss < best_val_loss: # if the  current loss is less the best loss
```

```
            best_val_loss = val_loss
            epochs_no_improve = 0
        else: # no improvement in loss
            epochs_no_improve += 1
            if epochs_no_improve >= patience:
                print('Early stopping triggered. Training stopped.')
                early_stop = True
                break

        # Learning rate scheduler step
        scheduler.step(val_loss)

        if early_stop:
            print("Stopped early at epoch:", epoch+1)
            break


train_model(mlp_model, criterion, optimizer, train_loader, val_loader, epochs=100, patie
nce=10)
```

```
/Users/darshanrao/anaconda3/lib/python3.11/site-packages/torch/optim/lr_scheduler.py:28:
UserWarning: The verbose parameter is deprecated. Please use get_last_lr() to access the
learning rate.
  warnings.warn("The verbose parameter is deprecated. Please use get_last_lr() "
```

```
Epoch: 1, Validation Loss: 0.0128, Accuracy: 63.80%
Epoch: 2, Validation Loss: 0.0127, Accuracy: 64.32%
Epoch: 3, Validation Loss: 0.0128, Accuracy: 63.93%
Epoch: 4, Validation Loss: 0.0130, Accuracy: 63.47%
Epoch: 5, Validation Loss: 0.0132, Accuracy: 63.21%
Epoch: 6, Validation Loss: 0.0136, Accuracy: 62.46%
Epoch: 7, Validation Loss: 0.0141, Accuracy: 62.21%
Epoch: 8, Validation Loss: 0.0146, Accuracy: 61.88%
Epoch: 9, Validation Loss: 0.0157, Accuracy: 61.53%
Epoch: 10, Validation Loss: 0.0164, Accuracy: 61.32%
Epoch: 11, Validation Loss: 0.0168, Accuracy: 61.00%
Epoch: 12, Validation Loss: 0.0173, Accuracy: 60.98%
Early stopping triggered. Training stopped.
```

In [96]:

```
# Evaluate the model on training data
y_train_pred = mlp_model(X_train_tensor).argmax(dim=1).numpy()
accuracy_train = accuracy_score(y_train_tensor.numpy(), y_train_pred)
precision_train = precision_score(y_train_tensor.numpy(), y_train_pred, average='weighted
')
recall_train = recall_score(y_train_tensor.numpy(), y_train_pred, average='weighted')
f1_train = f1_score(y_train_tensor.numpy(), y_train_pred, average='weighted')

# Evaluate the model on testing data
y_test_pred = mlp_model(X_val_tensor).argmax(dim=1).numpy()
accuracy_test = accuracy_score(y_val_tensor.numpy(), y_test_pred)
precision_test = precision_score(y_val_tensor.numpy(), y_test_pred, average='weighted')
recall_test = recall_score(y_val_tensor.numpy(), y_test_pred, average='weighted')
f1_test = f1_score(y_val_tensor.numpy(), y_test_pred, average='weighted')

# Print the results
print("Training Metrics For MLP:")
print(f"Accuracy: {accuracy_train:.4f}")
print(f"Precision: {precision_train:.4f}")
print(f"Recall: {recall_train:.4f}")
print(f"F1-Score: {f1_train:.4f}")
print("\nTesting Metrics For MLP:")
print(f"Accuracy: {accuracy_test:.4f}")
print(f"Precision: {precision_test:.4f}")
print(f"Recall: {recall_test:.4f}")
print(f"F1-Score: {f1_test:.4f}")
```

```
Training Metrics For MLP:
Accuracy: 0.8026
Precision: 0.7943
Recall: 0.8026
F1-Score: 0.7936
```

```
F1-Score: 0.7926

Testing Metrics For MLP:
Accuracy: 0.6098
Precision: 0.5892
Recall: 0.6098
F1-Score: 0.5961
```

Feedforward Neural Networks Concatenated Values Google Word2Vec Ternary

In [97]:

```
X=concat_features_pretrained
y=df['class']

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

In [98]:

```
X.shape,y.shape
```

Out[98]:

```
((250000, 3000), (250000,))
```

In [99]:

```
# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.long)
X_val_tensor = torch.tensor(X_test, dtype=torch.float32)
y_val_tensor = torch.tensor(y_test.values, dtype=torch.long)
y_train_tensor = y_train_tensor - 1
y_val_tensor = y_val_tensor - 1
# Create datasets and dataloaders
train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
val_dataset = TensorDataset(X_val_tensor, y_val_tensor)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=64, shuffle=False)

# Define the MLP model
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.layers = nn.Sequential(
            nn.Linear(3000, 50),
            nn.ReLU(),
            nn.Linear(50, 10),
            nn.ReLU(),
            nn.Linear(10, 3)
        )

    def forward(self, x):
        return self.layers(x)

# Initialize the model, loss function, and optimizer
mlp_model = MLP()
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(mlp_model.parameters(), lr=0.001)

def train_model(model, criterion, optimizer, train_loader, val_loader, epochs=10, patien
ce=3):
    # Initialize early stopping variables
    best_val_loss = float('inf')
    epochs_no_improve = 0
    early_stop = False

    # Scheduler for learning rate decay
```

```
    scheduler = lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patience=patience // 2,
factor=0.1, verbose=True)

    for epoch in range(epochs):
        model.train()
        for data, target in train_loader:
            optimizer.zero_grad() # clears old gradients,
            output = model(data)
            loss = criterion(output, target) # Calculate the pred-actual Loss
            loss.backward() # Back proprogation (Calculating the gradient)
            optimizer.step() # Weight updates

        # Validation phase
        model.eval() # Switches the into evaluation mode
        val_loss = 0
        correct = 0
        with torch.no_grad(): # Ensures gradient is not calculated(Saves memory and compu
tation)
            for data, target in val_loader:
                output = model(data) ## Output in the form of probabitilities
                val_loss += criterion(output, target).item()
                pred = output.argmax(dim=1, keepdim=True) # Probabitlity with the max va
lue is the output
                correct += pred.eq(target.view_as(pred)).sum().item() # counts the numbe
r of correct predictions

        val_loss /= len(val_loader.dataset)
        accuracy = 100. * correct / len(val_loader.dataset)
        print(f'Epoch: {epoch+1}, Validation Loss: {val_loss:.4f}, Accuracy: {accuracy:.
2f}%')

        # Early stopping logic
        if val_loss < best_val_loss: # if the  current loss is less the best loss
            best_val_loss = val_loss
            epochs_no_improve = 0
        else: # no improvement in loss
            epochs_no_improve += 1
            if epochs_no_improve >= patience:
                print('Early stopping triggered. Training stopped.')
                early_stop = True
                break

        # Learning rate scheduler step
        scheduler.step(val_loss)

        if early_stop:
            print("Stopped early at epoch:", epoch+1)
            break


train_model(mlp_model, criterion, optimizer, train_loader, val_loader, epochs=100, patie
nce=10)
```

```
/Users/darshanrao/anaconda3/lib/python3.11/site-packages/torch/optim/lr_scheduler.py:28:
UserWarning: The verbose parameter is deprecated. Please use get_last_lr() to access the
learning rate.
  warnings.warn("The verbose parameter is deprecated. Please use get_last_lr() "
```

```
Epoch: 1, Validation Loss: 0.0131, Accuracy: 62.76%
Epoch: 2, Validation Loss: 0.0130, Accuracy: 63.27%
Epoch: 3, Validation Loss: 0.0131, Accuracy: 63.06%
Epoch: 4, Validation Loss: 0.0132, Accuracy: 62.97%
Epoch: 5, Validation Loss: 0.0136, Accuracy: 62.24%
Epoch: 6, Validation Loss: 0.0141, Accuracy: 61.62%
Epoch: 7, Validation Loss: 0.0146, Accuracy: 61.20%
Epoch: 8, Validation Loss: 0.0154, Accuracy: 60.87%
Epoch: 9, Validation Loss: 0.0163, Accuracy: 60.69%
Epoch: 10, Validation Loss: 0.0169, Accuracy: 60.24%
Epoch: 11, Validation Loss: 0.0174, Accuracy: 60.02%
Epoch: 12, Validation Loss: 0.0178, Accuracy: 59.85%
Early stopping triggered. Training stopped.
```

In [100]:

```
# Evaluate the model on training data
y_train_pred = mlp_model(X_train_tensor).argmax(dim=1).numpy()
accuracy_train = accuracy_score(y_train_tensor.numpy(), y_train_pred)
precision_train = precision_score(y_train_tensor.numpy(), y_train_pred, average='weighted
')
recall_train = recall_score(y_train_tensor.numpy(), y_train_pred, average='weighted')
f1_train = f1_score(y_train_tensor.numpy(), y_train_pred, average='weighted')

# Evaluate the model on testing data
y_test_pred = mlp_model(X_val_tensor).argmax(dim=1).numpy()
accuracy_test = accuracy_score(y_val_tensor.numpy(), y_test_pred)
precision_test = precision_score(y_val_tensor.numpy(), y_test_pred, average='weighted')
recall_test = recall_score(y_val_tensor.numpy(), y_test_pred, average='weighted')
f1_test = f1_score(y_val_tensor.numpy(), y_test_pred, average='weighted')

# Print the results
print("Training Metrics For MLP:")
print(f"Accuracy: {accuracy_train:.4f}")
print(f"Precision: {precision_train:.4f}")
print(f"Recall: {recall_train:.4f}")
print(f"F1-Score: {f1_train:.4f}")
print("\nTesting Metrics For MLP:")
print(f"Accuracy: {accuracy_test:.4f}")
print(f"Precision: {precision_test:.4f}")
print(f"Recall: {recall_test:.4f}")
print(f"F1-Score: {f1_test:.4f}")
```

```
Training Metrics For MLP:
Accuracy: 0.8237
Precision: 0.8166
Recall: 0.8237
F1-Score: 0.8156

Testing Metrics For MLP:
Accuracy: 0.5985
Precision: 0.5797
Recall: 0.5985
F1-Score: 0.5863
```

Convolutional Neural Network

Convolutional Neural Network Padded Value Custom Word2Vec Binary

In [6]:

```
import numpy as np

def padded_word2vec(reviews, word2vec_model, vector_size, pad_size=50):
    features = []

    for review in reviews:
        valid_words = [word for word in review if word in word2vec_model.wv.key_to_index
]
        review_features = np.zeros((vector_size, pad_size), dtype=np.float32)

        for i, word in enumerate(valid_words[:pad_size]):
            review_features[:, i] = word2vec_model.wv[word]

        features.append(review_features)

    return np.array(features)

# Assuming 'tokenized_data' is your list of tokenized reviews and 'model' is your trained
Word2Vec model
padded_features = padded_word2vec(tokenized_data, model, vector_size=300)
```

In [7]:

```
padded_features.shape
```

Out[7]:

```
(250000, 300, 50)
```

In [8]:

```
# Filter out rows from the dataframe where 'class' is not equal to 3
df = df[df['class'] != 3]

# Get the indices of the remaining rows after filtering
filtered_indices = df.index.to_numpy()

# Now, use these indices to filter the avg_features array
padded_features = padded_features[filtered_indices]
```

In [9]:

```
# Checking the dimensions to ensure they match
print("Filtered DataFrame shape:", df.shape)
print("Filtered padded_features shape:", padded_features.shape)
```

```
Filtered DataFrame shape: (200000, 3)
Filtered padded_features shape: (200000, 300, 50)
```

In [10]:

```
X=padded_features
y=df['class']

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42
)
```

In [11]:

```
X_train.shape,y_train.shape
```

Out[11]:

```
((160000, 300, 50), (160000,))
```

In [12]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import TensorDataset, DataLoader

# Define the CNN model
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=3):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=300, out_channels=50, kernel_size=5, padding=
2)
        self.conv2 = nn.Conv1d(in_channels=50, out_channels=10, kernel_size=5, padding=2
)
        self.fc = None  # Will be initialized after the first forward pass

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool1d(x, kernel_size=2, stride=2)
        x = F.relu(self.conv2(x))
        x = F.max_pool1d(x, kernel_size=2, stride=2)

        # Check if the fc layer has been initialized, if not, do it dynamically
        if self.fc is None:
            # Calculate the correct input feature size
```

```
            n_size = x.view(x.size(0), -1).size(1)
            self.fc = nn.Linear(n_size, 2).to(x.device)

        x = x.view(x.size(0), -1)  # Flatten the tensor
        x = self.fc(x)
        return x

# Initialize the model
model = SimpleCNN(num_classes=2)
```

In [13]:

```
# Print the model architecture
print(model)
```

```
SimpleCNN(
  (conv1): Conv1d(300, 50, kernel_size=(5,), stride=(1,), padding=(2,))
  (conv2): Conv1d(50, 10, kernel_size=(5,), stride=(1,), padding=(2,))
)
```

In [16]:

```
# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.long)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test.values, dtype=torch.long)


y_train_tensor = y_train_tensor - 1
y_test_tensor = y_test_tensor - 1

# Create datasets and dataloaders
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=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

In [19]:

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

# Function to calculate accuracy
def calculate_accuracy(y_pred, y_true):
    _, predicted = torch.max(y_pred.data, 1)
    correct = (predicted == y_true).sum().item()
    return correct / y_true.size(0)

def train_model(model, train_loader, test_loader, criterion, optimizer, epochs=10, patie
nce=3):
    best_val_acc = 0.0  # Track the best validation accuracy
    patience_counter = 0  # Counter for how many epochs without improvement

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        running_corrects = 0

        for inputs, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item() * inputs.size(0)
            running_corrects += calculate_accuracy(outputs, labels)

        epoch_loss = running_loss / len(train_loader.dataset)
```

```
        epoch_acc = running_corrects / len(train_loader)

        # Validation phase
        model.eval()
        val_running_corrects = 0
        with torch.no_grad():
            for inputs, labels in test_loader:
                outputs = model(inputs)
                val_running_corrects += calculate_accuracy(outputs, labels)

        val_epoch_acc = val_running_corrects / len(test_loader)
        print(f'Epoch {epoch+1}/{epochs} : Training loss: {epoch_loss:.4f} | Training Ac
curacy: {epoch_acc:.4f} | Val Accuracy: {val_epoch_acc:.4f}')

        # Early Stopping Check
        if val_epoch_acc > best_val_acc:
            best_val_acc = val_epoch_acc
            patience_counter = 0  # Reset patience
        else:
            patience_counter += 1  # Increment patience

        if patience_counter >= patience:
            print("Early stopping triggered")
            break

# Train the model
train_model(model, train_loader, test_loader, criterion, optimizer, epochs=100, patience
=5)
```

```
Epoch 1/100 : Training loss: 0.3755 | Training Accuracy: 0.8373 | Val Accuracy: 0.8532
Epoch 2/100 : Training loss: 0.3260 | Training Accuracy: 0.8627 | Val Accuracy: 0.8575
Epoch 3/100 : Training loss: 0.2996 | Training Accuracy: 0.8741 | Val Accuracy: 0.8591
Epoch 4/100 : Training loss: 0.2748 | Training Accuracy: 0.8866 | Val Accuracy: 0.8601
Epoch 5/100 : Training loss: 0.2554 | Training Accuracy: 0.8949 | Val Accuracy: 0.8521
Epoch 6/100 : Training loss: 0.2358 | Training Accuracy: 0.9040 | Val Accuracy: 0.8555
Epoch 7/100 : Training loss: 0.2182 | Training Accuracy: 0.9120 | Val Accuracy: 0.8534
Epoch 8/100 : Training loss: 0.2035 | Training Accuracy: 0.9179 | Val Accuracy: 0.8478
Epoch 9/100 : Training loss: 0.1892 | Training Accuracy: 0.9242 | Val Accuracy: 0.8485
Early stopping triggered
```

In [20]:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import torch

def evaluate_model(model, dataloader):
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for inputs, labels in dataloader:
            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)
            all_preds.extend(preds.tolist())
            all_labels.extend(labels.tolist())
    return all_preds, all_labels

def calculate_metrics(y_true, y_pred):
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred, average='weighted')
    recall = recall_score(y_true, y_pred, average='weighted')
    f1 = f1_score(y_true, y_pred, average='weighted')
    return accuracy, precision, recall, f1


y_train_pred, y_train_true = evaluate_model(model, train_loader)
y_test_pred, y_test_true = evaluate_model(model, test_loader)

accuracy_train, precision_train, recall_train, f1_train = calculate_metrics(y_train_true
, y_train_pred)
accuracy_test, precision_test, recall_test, f1_test = calculate_metrics(y_test_true, y_t
est_pred)
```

```
# Print the results
print("Training Metrics:")
print(f"Accuracy: {accuracy_train:.4f}")
print(f"Precision: {precision_train:.4f}")
print(f"Recall: {recall_train:.4f}")
print(f"F1-Score: {f1_train:.4f}")
print("\nTesting Metrics:")
print(f"Accuracy: {accuracy_test:.4f}")
print(f"Precision: {precision_test:.4f}")
print(f"Recall: {recall_test:.4f}")
print(f"F1-Score: {f1_test:.4f}")
```

```
Training Metrics:
Accuracy: 0.9358
Precision: 0.9372
Recall: 0.9358
F1-Score: 0.9358

Testing Metrics:
Accuracy: 0.8485
Precision: 0.8504
Recall: 0.8485
F1-Score: 0.8483
```

Convolutional Neural Network Padded Value Google Word2Vec Binary

In [4]:

```
import numpy as np

def padded_word2vec_google(reviews,vector_size, pad_size=50):
    features = []

    for review in reviews:
        valid_words = [word for word in review if word in wv.key_to_index]
        review_features = np.zeros((vector_size, pad_size), dtype=np.float32)

        for i, word in enumerate(valid_words[:pad_size]):
            review_features[:, i] = wv[word]

        features.append(review_features)

    return np.array(features)

# Assuming 'tokenized_data' is your list of tokenized reviews and 'model' is your trained Word2Vec model
padded_features_pretrained = padded_word2vec_google(tokenized_data,vector_size=300)
```

In [9]:

```
padded_features_pretrained.shape
```

Out[9]:

```
(250000, 300, 50)
```

In [5]:

```
# Filter out rows from the dataframe where 'class' is not equal to 3
df = df[df['class'] != 3]

# Get the indices of the remaining rows after filtering
filtered_indices = df.index.to_numpy()

# Now, use these indices to filter the avg_features array
padded_features_pretrained = padded_features_pretrained[filtered_indices]
```

In [6]:

```
# Checking the dimensions to ensure they match
print("Filtered DataFrame shape:", df.shape)
print("Filtered padded_features shape:", padded_features_pretrained.shape)
```

```
Filtered DataFrame shape: (200000, 3)
Filtered padded_features shape: (200000, 300, 50)
```

In [7]:

```
X=padded_features_pretrained
y=df['class']

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42
)
```

In [8]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import TensorDataset, DataLoader

# Define the CNN model
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=3):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=300, out_channels=50, kernel_size=5, padding=
2)
        self.conv2 = nn.Conv1d(in_channels=50, out_channels=10, kernel_size=5, padding=2
)
        self.fc = None

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool1d(x, kernel_size=2, stride=2)
        x = F.relu(self.conv2(x))
        x = F.max_pool1d(x, kernel_size=2, stride=2)


        if self.fc is None:
            # Calculate the correct input feature size
            n_size = x.view(x.size(0), -1).size(1)
            self.fc = nn.Linear(n_size, 2).to(x.device)

        x = x.view(x.size(0), -1)  # Flatten the tensor
        x = self.fc(x)
        return x

# Initialize the model
model = SimpleCNN(num_classes=2)
```

In [9]:

```
# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.long)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test.values, dtype=torch.long)


y_train_tensor = y_train_tensor - 1
y_test_tensor = y_test_tensor - 1

# Create datasets and dataloaders
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=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

In [10]:

```python
# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Function to calculate accuracy
def calculate_accuracy(y_pred, y_true):
    _, predicted = torch.max(y_pred.data, 1)
    correct = (predicted == y_true).sum().item()
    return correct / y_true.size(0)

def train_model(model, train_loader, test_loader, criterion, optimizer, epochs=10, patie
nce=3):
    best_val_acc = 0.0
    patience_counter = 0

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        running_corrects = 0

        for inputs, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item() * inputs.size(0)
            running_corrects += calculate_accuracy(outputs, labels)

        epoch_loss = running_loss / len(train_loader.dataset)
        epoch_acc = running_corrects / len(train_loader)

        # Validation phase
        model.eval()
        val_running_corrects = 0
        with torch.no_grad():
            for inputs, labels in test_loader:
                outputs = model(inputs)
                val_running_corrects += calculate_accuracy(outputs, labels)

        val_epoch_acc = val_running_corrects / len(test_loader)
        print(f'Epoch {epoch+1}/{epochs} : Training loss: {epoch_loss:.4f} | Training Ac
curacy: {epoch_acc:.4f} | Val Accuracy: {val_epoch_acc:.4f}')

        # Early Stopping Check
        if val_epoch_acc > best_val_acc:
            best_val_acc = val_epoch_acc
            patience_counter = 0  # Reset patience
        else:
            patience_counter += 1  # Increment patience

        if patience_counter >= patience:
            print("Early stopping triggered")
            break

# Train the model
train_model(model, train_loader, test_loader, criterion, optimizer, epochs=100, patience
=3)
```

```
Epoch 1/100 : Training loss: 0.4123 | Training Accuracy: 0.8165 | Val Accuracy: 0.8439
Epoch 2/100 : Training loss: 0.3417 | Training Accuracy: 0.8554 | Val Accuracy: 0.8474
Epoch 3/100 : Training loss: 0.3083 | Training Accuracy: 0.8719 | Val Accuracy: 0.8593
Epoch 4/100 : Training loss: 0.2805 | Training Accuracy: 0.8851 | Val Accuracy: 0.8601
Epoch 5/100 : Training loss: 0.2568 | Training Accuracy: 0.8959 | Val Accuracy: 0.8595
Epoch 6/100 : Training loss: 0.2348 | Training Accuracy: 0.9062 | Val Accuracy: 0.8583
Epoch 7/100 : Training loss: 0.2163 | Training Accuracy: 0.9155 | Val Accuracy: 0.8548
Early stopping triggered
```

In [11]:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import torch

def evaluate_model(model, dataloader):
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for inputs, labels in dataloader:
            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)
            all_preds.extend(preds.tolist())
            all_labels.extend(labels.tolist())
    return all_preds, all_labels

def calculate_metrics(y_true, y_pred):
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred, average='weighted')
    recall = recall_score(y_true, y_pred, average='weighted')
    f1 = f1_score(y_true, y_pred, average='weighted')
    return accuracy, precision, recall, f1


y_train_pred, y_train_true = evaluate_model(model, train_loader)
y_test_pred, y_test_true = evaluate_model(model, test_loader)

accuracy_train, precision_train, recall_train, f1_train = calculate_metrics(y_train_true
, y_train_pred)
accuracy_test, precision_test, recall_test, f1_test = calculate_metrics(y_test_true, y_t
est_pred)

# Print the results
print("Training Metrics:")
print(f"Accuracy: {accuracy_train:.4f}")
print(f"Precision: {precision_train:.4f}")
print(f"Recall: {recall_train:.4f}")
print(f"F1-Score: {f1_train:.4f}")
print("\nTesting Metrics:")
print(f"Accuracy: {accuracy_test:.4f}")
print(f"Precision: {precision_test:.4f}")
print(f"Recall: {recall_test:.4f}")
print(f"F1-Score: {f1_test:.4f}")
```

```
Training Metrics:
Accuracy: 0.9362
Precision: 0.9362
Recall: 0.9362
F1-Score: 0.9362

Testing Metrics:
Accuracy: 0.8548
Precision: 0.8548
Recall: 0.8548
F1-Score: 0.8547
```

Convolutional Neural Network Padded Value Custom Word2Vec Ternary

In [4]:

```
from gensim.models import Word2Vec
# Load model
model = Word2Vec.load("word2vec.model")
```

In [5]:

```
import numpy as np

def padded_word2vec(reviews, word2vec_model, vector_size, pad_size=50):
```

```
    features = []

    for review in reviews:
        valid_words = [word for word in review if word in word2vec_model.wv.key_to_index
]
        review_features = np.zeros((vector_size, pad_size), dtype=np.float32)

        for i, word in enumerate(valid_words[:pad_size]):
            review_features[:, i] = word2vec_model.wv[word]

        features.append(review_features)

    return np.array(features)

padded_features = padded_word2vec(tokenized_data, model, vector_size=300)
```

In [6]:

```
print("Filtered DataFrame shape:", df.shape)
print("Filtered padded_features shape:", padded_features.shape)
```

```
Filtered DataFrame shape: (250000, 3)
Filtered padded_features shape: (250000, 300, 50)
```

In [7]:

```
X=padded_features
y=df['class']

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42
)
```

In [8]:

```
del padded_features
del df
del tokenized_data
del model
del X
del y
```

In [9]:

```
X_train.shape,y_train.shape
```

Out[9]:

```
((200000, 300, 50), (200000,))
```

In [10]:

```
X_test.shape,y_test.shape
```

Out[10]:

```
((50000, 300, 50), (50000,))
```

In [11]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import TensorDataset, DataLoader

# Define the CNN model
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=3):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=300, out_channels=50, kernel_size=5, padding=
2)
        self.conv2 = nn.Conv1d(in_channels=50, out_channels=10, kernel_size=5, padding=2
```

```
)
        self.fc = None  # Will be initialized after the first forward pass

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool1d(x, kernel_size=2, stride=2)
        x = F.relu(self.conv2(x))
        x = F.max_pool1d(x, kernel_size=2, stride=2)


        if self.fc is None:

            n_size = x.view(x.size(0), -1).size(1)
            self.fc = nn.Linear(n_size, 3).to(x.device)

        x = x.view(x.size(0), -1)  # Flatten the tensor
        x = self.fc(x)
        return x

# Initialize the model
model = SimpleCNN(num_classes=3)
```

In [12]:

```
# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.long)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test.values, dtype=torch.long)

y_train_tensor = y_train_tensor - 1
y_test_tensor = y_test_tensor - 1
```

In [13]:

```
del X_train
del X_test
del y_train
del y_test
```

In [14]:

```
# Create datasets and dataloaders
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=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

In [15]:

```
del X_train_tensor
del X_test_tensor
del y_train_tensor
del y_test_tensor
```

In [17]:

```
del train_dataset
del test_dataset
```

In [19]:

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

# Function to calculate accuracy
def calculate_accuracy(y_pred, y_true):
    _, predicted = torch.max(y_pred.data, 1)
    correct = (predicted == y_true).sum().item()
```

```
    return correct / y_true.size(0)

def train_model(model, train_loader, test_loader, criterion, optimizer, epochs=10, patie
nce=3):
    best_val_acc = 0.0  # Track the best validation accuracy
    patience_counter = 0  # Counter for how many epochs without improvement

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        running_corrects = 0

        for inputs, labels in train_loader:
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item() * inputs.size(0)
            running_corrects += calculate_accuracy(outputs, labels)

        epoch_loss = running_loss / len(train_loader.dataset)
        epoch_acc = running_corrects / len(train_loader)

        # Validation phase
        model.eval()
        val_running_corrects = 0
        with torch.no_grad():
            for inputs, labels in test_loader:
                outputs = model(inputs)
                val_running_corrects += calculate_accuracy(outputs, labels)

        val_epoch_acc = val_running_corrects / len(test_loader)
        print(f'Epoch {epoch+1}/{epochs} : Training loss: {epoch_loss:.4f} | Training Ac
curacy: {epoch_acc:.4f} | Val Accuracy: {val_epoch_acc:.4f}')

        # Early Stopping Check
        if val_epoch_acc > best_val_acc:
            best_val_acc = val_epoch_acc
            patience_counter = 0  # Reset patience
        else:
            patience_counter += 1  # Increment patience

        if patience_counter >= patience:
            print("Early stopping triggered")
            break

# Train the model
train_model(model, train_loader, test_loader, criterion, optimizer, epochs=100, patience
=5)
```

```
Epoch 1/100 : Training loss: 0.8054 | Training Accuracy: 0.6555 | Val Accuracy: 0.6721
Epoch 2/100 : Training loss: 0.7431 | Training Accuracy: 0.6866 | Val Accuracy: 0.6807
Epoch 3/100 : Training loss: 0.7154 | Training Accuracy: 0.6985 | Val Accuracy: 0.6834
Epoch 4/100 : Training loss: 0.6923 | Training Accuracy: 0.7100 | Val Accuracy: 0.6824
Epoch 5/100 : Training loss: 0.6722 | Training Accuracy: 0.7187 | Val Accuracy: 0.6824
Epoch 6/100 : Training loss: 0.6556 | Training Accuracy: 0.7259 | Val Accuracy: 0.6762
Epoch 7/100 : Training loss: 0.6400 | Training Accuracy: 0.7335 | Val Accuracy: 0.6728
Epoch 8/100 : Training loss: 0.6260 | Training Accuracy: 0.7408 | Val Accuracy: 0.6743
Early stopping triggered
```

In [20]:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import torch

def evaluate_model(model, dataloader):
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
```

```
        for inputs, labels in dataloader:
            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)
            all_preds.extend(preds.tolist())
            all_labels.extend(labels.tolist())
    return all_preds, all_labels

def calculate_metrics(y_true, y_pred):
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred, average='weighted')
    recall = recall_score(y_true, y_pred, average='weighted')
    f1 = f1_score(y_true, y_pred, average='weighted')
    return accuracy, precision, recall, f1


y_train_pred, y_train_true = evaluate_model(model, train_loader)
y_test_pred, y_test_true = evaluate_model(model, test_loader)

accuracy_train, precision_train, recall_train, f1_train = calculate_metrics(y_train_true
, y_train_pred)
accuracy_test, precision_test, recall_test, f1_test = calculate_metrics(y_test_true, y_t
est_pred)

# Print the results
print("Training Metrics:")
print(f"Accuracy: {accuracy_train:.4f}")
print(f"Precision: {precision_train:.4f}")
print(f"Recall: {recall_train:.4f}")
print(f"F1-Score: {f1_train:.4f}")
print("\nTesting Metrics:")
print(f"Accuracy: {accuracy_test:.4f}")
print(f"Precision: {precision_test:.4f}")
print(f"Recall: {recall_test:.4f}")
print(f"F1-Score: {f1_test:.4f}")
```

```
Training Metrics:
Accuracy: 0.7621
Precision: 0.7478
Recall: 0.7621
F1-Score: 0.7469

Testing Metrics:
Accuracy: 0.6744
Precision: 0.6478
Recall: 0.6744
F1-Score: 0.6557
```

Convolutional Neural Network Padded Value Google Word2Vec Ternary

In [4]:

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

In [5]:

```
import numpy as np

def padded_word2vec_google(reviews,vector_size, pad_size=50):
    features = []

    for review in reviews:
        valid_words = [word for word in review if word in wv.key_to_index]
        review_features = np.zeros((vector_size, pad_size), dtype=np.float32)

        for i, word in enumerate(valid_words[:pad_size]):
            review_features[:, i] = wv[word]
```

```
        features.append(review_features)

    return np.array(features)


padded_features_pretrained = padded_word2vec_google(tokenized_data,vector_size=300)
```

In [6]:

```
print("Filtered DataFrame shape:", df.shape)
print("Filtered padded_features shape:", padded_features_pretrained.shape)
```

```
Filtered DataFrame shape: (250000, 3)
Filtered padded_features shape: (250000, 300, 50)
```

In [7]:

```
X=padded_features_pretrained
y=df['class']

from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42
)
```

In [ ]:

```
del padded_features_pretrained
del df
del tokenized_data
del wv
del X
del y
```

In [11]:

```
X_train.shape,y_train.shape
```

Out[11]:

```
((200000, 300, 50), (200000,))
```

In [12]:

```
X_test.shape,y_test.shape
```

Out[12]:

```
((50000, 300, 50), (50000,))
```

In [13]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import TensorDataset, DataLoader

# Define the CNN model
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=3):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=300, out_channels=50, kernel_size=5, padding=
2)
        self.conv2 = nn.Conv1d(in_channels=50, out_channels=10, kernel_size=5, padding=2
)
        self.fc = None  # Will be initialized after the first forward pass

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = F.max_pool1d(x, kernel_size=2, stride=2)
        x = F.relu(self.conv2(x))
        x = F.max_pool1d(x, kernel_size=2, stride=2)
```

```python
        if self.fc is None:
            # Calculate the correct input feature size
            n_size = x.view(x.size(0), -1).size(1)
            self.fc = nn.Linear(n_size, 3).to(x.device)

        x = x.view(x.size(0), -1)  # Flatten the tensor
        x = self.fc(x)
        return x

# Initialize the model
model = SimpleCNN(num_classes=3)
```

In [14]:

```python
# Convert data to PyTorch tensors
X_train_tensor = torch.tensor(X_train, dtype=torch.float32)
y_train_tensor = torch.tensor(y_train.values, dtype=torch.long)
X_test_tensor = torch.tensor(X_test, dtype=torch.float32)
y_test_tensor = torch.tensor(y_test.values, dtype=torch.long)

y_train_tensor = y_train_tensor - 1
y_test_tensor = y_test_tensor - 1
```

In [15]:

```python
del X_train
del X_test
del y_train
del y_test
```

In [16]:

```python
# Create datasets and dataloaders
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=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

In [17]:

```python
del X_train_tensor
del X_test_tensor
del y_train_tensor
del y_test_tensor
del train_dataset
del test_dataset
```

In [18]:

```python
# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

# Function to calculate accuracy
def calculate_accuracy(y_pred, y_true):
    _, predicted = torch.max(y_pred.data, 1)
    correct = (predicted == y_true).sum().item()
    return correct / y_true.size(0)

def train_model(model, train_loader, test_loader, criterion, optimizer, epochs=10, patie
nce=3):
    best_val_acc = 0.0  # Track the best validation accuracy
    patience_counter = 0  # Counter for how many epochs without improvement

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0
        running_corrects = 0

        for inputs, labels in train_loader:
```

```
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            running_loss += loss.item() * inputs.size(0)
            running_corrects += calculate_accuracy(outputs, labels)

        epoch_loss = running_loss / len(train_loader.dataset)
        epoch_acc = running_corrects / len(train_loader)

        # Validation phase
        model.eval()
        val_running_corrects = 0
        with torch.no_grad():
            for inputs, labels in test_loader:
                outputs = model(inputs)
                val_running_corrects += calculate_accuracy(outputs, labels)

        val_epoch_acc = val_running_corrects / len(test_loader)
        print(f'Epoch {epoch+1}/{epochs} : Training loss: {epoch_loss:.4f} | Training Ac
curacy: {epoch_acc:.4f} | Val Accuracy: {val_epoch_acc:.4f}')

        # Early Stopping Check
        if val_epoch_acc > best_val_acc:
            best_val_acc = val_epoch_acc
            patience_counter = 0  # Reset patience
        else:
            patience_counter += 1  # Increment patience

        if patience_counter >= patience:
            print("Early stopping triggered")
            break

# Train the model
train_model(model, train_loader, test_loader, criterion, optimizer, epochs=100, patience
=3)
```

```
Epoch 1/100 : Training loss: 0.8213 | Training Accuracy: 0.6449 | Val Accuracy: 0.6752
Epoch 2/100 : Training loss: 0.7363 | Training Accuracy: 0.6887 | Val Accuracy: 0.6865
Epoch 3/100 : Training loss: 0.6987 | Training Accuracy: 0.7053 | Val Accuracy: 0.6843
Epoch 4/100 : Training loss: 0.6710 | Training Accuracy: 0.7185 | Val Accuracy: 0.6891
Epoch 5/100 : Training loss: 0.6478 | Training Accuracy: 0.7309 | Val Accuracy: 0.6878
Epoch 6/100 : Training loss: 0.6270 | Training Accuracy: 0.7395 | Val Accuracy: 0.6888
Epoch 7/100 : Training loss: 0.6089 | Training Accuracy: 0.7481 | Val Accuracy: 0.6804
Early stopping triggered
```

In [19]:

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
import torch

def evaluate_model(model, dataloader):
    model.eval()
    all_preds = []
    all_labels = []
    with torch.no_grad():
        for inputs, labels in dataloader:
            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)
            all_preds.extend(preds.tolist())
            all_labels.extend(labels.tolist())
    return all_preds, all_labels

def calculate_metrics(y_true, y_pred):
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred, average='weighted')
    recall = recall_score(y_true, y_pred, average='weighted')
    f1 = f1_score(y_true, y_pred, average='weighted')
    return accuracy, precision, recall, f1
```

```python
y_train_pred, y_train_true = evaluate_model(model, train_loader)
y_test_pred, y_test_true = evaluate_model(model, test_loader)

accuracy_train, precision_train, recall_train, f1_train = calculate_metrics(y_train_true
, y_train_pred)
accuracy_test, precision_test, recall_test, f1_test = calculate_metrics(y_test_true, y_t
est_pred)

# Print the results
print("Training Metrics:")
print(f"Accuracy: {accuracy_train:.4f}")
print(f"Precision: {precision_train:.4f}")
print(f"Recall: {recall_train:.4f}")
print(f"F1-Score: {f1_train:.4f}")
print("\nTesting Metrics:")
print(f"Accuracy: {accuracy_test:.4f}")
print(f"Precision: {precision_test:.4f}")
print(f"Recall: {recall_test:.4f}")
print(f"F1-Score: {f1_test:.4f}")
```

```
Training Metrics:
Accuracy: 0.7704
Precision: 0.7598
Recall: 0.7704
F1-Score: 0.7622

Testing Metrics:
Accuracy: 0.6803
Precision: 0.6635
Recall: 0.6803
F1-Score: 0.6697
```