```
HW2 REPORT
```

Installing libraries

```
'''
python version: 3.12.8
beautifulsoup4==4.12.2
bs4==0.0.2
gensim==4.3.3
joblib==1.4.2
nltk==3.9.1
numpy==1.26.4
pandas==2.2.3
regex==2024.11.6
scikit-learn==1.6.1
scipy==1.13.1
soupsieve==2.5
threadpoolctl==3.5.0
torch==2.6.0
torchaudio==2.6.0
torchvision==0.21.0
tqdm==4.67.1
'''

import pandas as pd
from sklearn.model_selection import train_test_split
import nltk
from nltk.tokenize import word_tokenize
from gensim.models import Word2Vec, KeyedVectors
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
import gensim
nltk.download('punkt_tab')
nltk.download('wordnet')
nltk.download('stopwords')
import re
from bs4 import BeautifulSoup
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer

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

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

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

Reading from dataset

```
pd_frame = pd.read_csv("data.tsv", sep="\t", on_bad_lines='skip', low_memory=False)
```

Dataset generation

In this, I have kept the 2 columns review_body and star_rating. Columns that have any null or bad values are dropped. 50000 reviews are picked from each star_rating values. After that, I have created a new column sentiment. The sentiment column has value 1 for positive class, 2 for negative class and 3 for neutral class. I have shuffled the dataset for better results while training.

```
pd_frame = pd_frame[['review_body', 'star_rating']]
pd_frame=pd_frame.dropna(subset=['star_rating','review_body'], how='any')
pd_frame['star_rating'] = pd.to_numeric(pd_frame['star_rating'], errors='coerce')

balanced_df = pd_frame.groupby("star_rating").apply(lambda x: x.sample(n=50000, random_state=42)).reset_index(drop=True)
balanced_df['sentiment'] = balanced_df['star_rating'].apply(
    lambda x: 1 if x > 3 else (2 if x <= 2 else 3)
)

balanced_df = balanced_df.sample(frac=1, random_state=42).reset_index(drop=True)
```

```
/var/folders/fc/t8b4fmzd1vn4x8ky425rp7l40000gn/T/ipykernel_25098/3038514753.py:5: DeprecationWarning: DataFrameGroupBy.apply
  balanced_df = pd_frame.groupby("star_rating").apply(lambda x: x.sample(n=50000, random_state=42)).reset_index(drop=True)
```

Word Embedding a

I have extracted the Word2Vec features from pretrained Google news dataset in the following cell.

```
import gensim.downloader as api

pre_model = api.load('word2vec-google-news-300')
result = pre_model.most_similar(positive=["doctor", "court"], negative=["hospital"], topn=1)
print("Doctor - Hospital + Court =", result)

similarity = pre_model.similarity("happy", "satisfied")
print("Similarity between 'happy' and 'satisfied':", similarity)
```

```
Doctor - Hospital + Court = [('judge', 0.6122931838035583)]
Similarity between 'happy' and 'satisfied': 0.6437949
```

Word Embedding b

Here, the Word2Vec features from the self trained model are extracted. I have passed the tokenized review_body along with the parameters mentioned in the question (vector_size=300, window=11, min_count=10,workers=1)

```
custom_mod = Word2Vec(sentences=balanced_df['review_body'].apply(word_tokenize), vector_size=300, window=11, min_count=10, worke

result_custom = custom_mod.wv.most_similar(positive=["doctor", "court"], negative=["hospital"], topn=1)
print("Doctor - Hospital + Court =", result_custom)

similarity_custom = custom_mod.wv.similarity("happy", "satisfied")
print("Similarity between 'happy' and 'satisfied' (Custom Model):", similarity_custom)
```

```
Doctor - Hospital + Court = [('documenting', 0.4553646147251129)]
Similarity between 'happy' and 'satisfied' (Custom Model): 0.9003303
```

Conclusion from comparing vectors generated by custom and the pretrained model

Question a

Since the pretrained google news model is trained on a diverse corpus, it encompasses multiple domains. In my example, “Doctor - Hospital + Court”, the model predicts “judge”, which aligns with the fact that courts are linked with judges.
Also, the similarity score of 0.6438 between “happy” and “satisfied” indicates a moderate semantic relationship.

On the other hand, our custom model gives different results for the same example. It predicts “documenting”, which used in the context of legal claims, refunds, policies, or disputes. This explains why “documenting” could be linked with “court” in pretrained model rather than "judge". Some reviews may discuss about legal issues.

However, the custom model assigns a higher similarity score (0.9003) between “happy” and “satisfied”, indicating that in the context of customer reviews, these words are used more interchangeably.

Overall, the custom model better captures word associations specific to the Amazon reviews dataset but lacks the general knowledge encoded in the Google News model.

Question b

The Google News model encodes semantic relationships more effectively in a general sense. Its ability to correctly map “Doctor - Hospital + Court” = “judge” implies that it has learned real-world relations between professions and locations.

However, the custom model performs better in domain-specific contexts. The higher similarity score for “happy” and “satisfied” suggests that it effectively handles sentiment-related words used in customer reviews.
In conclusion, if the goal is general-purpose word understanding, the Google News model is superior. But for sentiment analysis within Amazon reviews, the custom model may be more useful as it learns contextually relevant relationships.

Data preprocessing from HW1

```
contract_dict = {
    "he'll": "he will","he'll've": "he will have","isn't": "is not","it'd": "it would",
    "it'd've": "it would have","it'll": "it will","it'll've": "it will have","it's": "it is",
    "let's": "let us","ma'am": "madam","mayn't": "may not","mightn't": "might not","might've": "might have",
    "must've": "must have","mustn't": "must not","needn't": "need not","ain't": "is not",
    "aren't": "are not","can't": "cannot","couldn't": "could not","could've": "could have",
    "couldn't've": "could not have","didn't": "did not","doesn't": "does not","don't": "do not",
    "hadn't": "had not","hadn't've": "had not have","hasn't": "has not","haven't": "have not",
    "haven't've": "have not have","he'd": "he would","he'd've": "he would have","needn't've": "need not have",
    "o'clock": "of the clock","shalln't": "shall not","shan't": "shall not","she'd": "she would",
    "she'd've": "she would have","he's": "he is","how'd": "how did","how'd'y": "how do you",
    "how'll": "how will","how's": "how is","I'd": "I would","I'd've": "I would have",
    "I'll": "I will","I'll've": "I will have","I'm": "I am","I've": "I have","I'd": "I had",
    "I'd've": "I had have","I'm": "I am","I've": "I have","she'll": "she will","she'll've": "she will have",
    "she's": "she is","should've": "should have","shouldn't": "should not","shouldn't've": "should not have",
    "so've": "so have","so's": "so is","that'd": "that would","that'd've": "that would have","that's": "that is",
    "there'd": "there would","there'd've": "there would have","there's": "there is","they'd": "they would",
    "they'd've": "they would have","they'll": "they will","they'll've": "they will have","they're": "they are",
    "they've": "they have","to've": "to have","wasn't": "was not","we'd": "we would","we'd've": "we would have",
    "we'll": "we will","we'll've": "we will have","we're": "we are","we've": "we have","weren't": "were not",
    "what'd": "what did","what'd'y": "what do you","what'll": "what will","what'll've": "what will have",
    "what're": "what are","what's": "what is","where've": "where have","who'd": "who would",
    "who'd've": "who would have","who'll": "who will","who'll've": "who will have","who's": "who is",
    "who've": "who have","why'd": "why did","why'll": "why will","why's": "why is",
    "why've": "why have","you'd": "you would","you'd've": "you would have","you'll": "you will",
    "you'll've": "you will have","you're": "you are","you've": "you have","what've": "what have",
    "when'd": "when did","when'll": "when will","when's": "when is","when've": "when have",
    "where'd": "where did","where'll": "where will","where's": "where is",
}

def review_cleaning(review):
    review = review.lower()
    if "<" in review and ">" in review:
        review = BeautifulSoup(review, "html.parser").get_text()
    review = re.sub(r'http\S+|www\S+', '', review)
    pattern = re.compile(r'\b(' + '|'.join(re.escape(key) for key in contract_dict.keys()) + r')\b')
    review = pattern.sub(lambda x: contract_dict[x.group()], review)
    review = re.sub(r'[^a-zA-Z\s]', '', review)
    review = re.sub(r'\s+', ' ', review).strip()
    return review

balanced_df['review_body'] = balanced_df['review_body'].astype(str)
balanced_df['review_body'] = balanced_df['review_body'].apply(review_cleaning)


stop_words = set(stopwords.words('english'))

def stopwords_removal(review):
    words = review.split()
    words = [word for word in words if word not in stop_words]
    review = ' '.join(words)
    return review
```

```
balanced_df['review_body'] = balanced_df['review_body'].apply(stopwords_removal)


lemma = WordNetLemmatizer()

def review_lemmatization(review):
    words = review.split()
    words = [lemma.lemmatize(word) for word in words]
    review = ' '.join(words)
    return review



balanced_df['review_body'] = balanced_df['review_body'].apply(review_lemmatization)
```

Data preparation for binary classification

I have created another dataframe binary_df to use this for binary classification by removing the neutral class(sentiment=3) rows from the balanced_df(which contains all the classes).
Also, the tokenized version of the training and testing data are stored separately for future use in the following sections of the assignment.

```
binary_df = balanced_df[balanced_df['sentiment'] != 3]

X_train_bin, X_test_bin, y_train_bin, y_test_bin = train_test_split(binary_df['review_body'], binary_df['sentiment'], test_size=

X_train_token = [word_tokenize(text) for text in X_train_bin]
X_test_token = [word_tokenize(text) for text in X_test_bin]
```

Simple models

Using each feature extraction method(TF-IDF, pretrained model, custom model), I trained and evaluated two classification models(Perceptron and SVM).
TF-IDF vectors were generated using TfidfVectorizer().
Word2Vec feature vectors were computed by averaging the word embeddings of all tokens in each review in the avg_vector function.

```

tf_idf = TfidfVectorizer()
tf_idf.fit(binary_df['review_body'])
X_train_tfidf = tf_idf.transform(X_train_bin)
X_test_tfidf = tf_idf.transform(X_test_bin)

def avg_vector(review, model, vector_size=300):
    vectors = [model[word] for word in review if word in model]
    return np.mean(vectors, axis=0) if vectors else np.zeros(vector_size)


X_train_bin_avg_pre = np.array([avg_vector(review, pre_model) for review in X_train_token])
X_test_bin_avg_pre = np.array([avg_vector(review, pre_model) for review in X_test_token])

X_train_bin_avg_cus = np.array([avg_vector(review, custom_mod.wv) for review in X_train_token])
X_test_bin_avg_cus = np.array([avg_vector(review, custom_mod.wv) for review in X_test_token])

def train_eval(X_train, X_test, y_train, y_test, f_type):

    perceptron = Perceptron(max_iter=1000,eta0=0.01, random_state=42)
    perceptron.fit(X_train, y_train)
    y_pred_perceptron = perceptron.predict(X_test)
    perceptron_acc = accuracy_score(y_test, y_pred_perceptron)

    svm = LinearSVC()
    svm.fit(X_train, y_train)
    y_pred_svm = svm.predict(X_test)
    svm_acc = accuracy_score(y_test, y_pred_svm)

    print(f"Feature: {f_type}")
    print(f"Perceptron Accuracy: {perceptron_acc:.4f}")
    print(f"SVM Accuracy: {svm_acc:.4f}")
    print()
```

```
train_eval(X_train_tfidf, X_test_tfidf, y_train_bin, y_test_bin, "TF-IDF")
train_eval(X_train_bin_avg_pre, X_test_bin_avg_pre, y_train_bin, y_test_bin, "Word2Vec-Google-News-300")
train_eval(X_train_bin_avg_cus, X_test_bin_avg_cus, y_train_bin, y_test_bin, "Custom Word2Vec")
```

```
Feature: TF-IDF
Perceptron Accuracy: 0.8150
SVM Accuracy: 0.8637

Feature: Word2Vec-Google-News-300
Perceptron Accuracy: 0.7544
SVM Accuracy: 0.8161

Feature: Custom Word2Vec
Perceptron Accuracy: 0.7825
SVM Accuracy: 0.8435
```

From the results, we observe the following performance differences across the three feature extraction methods:

Feature Type	Perceptron Accuracy	SVM Accuracy
TF-IDF	0.8150	0.8637
Google Word2Vec	0.7544	0.8161
Custom Word2Vec	0.7825	0.8435

Conclusion from comparing performances for the models trained using the three different feature types (TF-IDF, pretrainedWord2Vec, custom Word2Vec)

TF-IDF outperforms both Word2Vec models
•Highest accuracy (Perceptron: 0.8150, SVM: 0.8637)
•Since TF-IDF is purely frequency-based, it effectively captures important terms in the dataset without relying on word meanings. This suggests that sentiment classification in this dataset benefits more from word importance rather than contextual meaning.

Custom Word2Vec performs better than Google Word2Vec in Perceptron
•Perceptron with Custom Word2Vec (0.7825) is higher than Perceptron with Google Word2Vec (0.7544).
•This might produce different results as we change random_state. Sometimes, the pretrained model may produce better results on perceptron.

Custom Word2Vec performs better than Google Word2Vec with SVM
•SVM with Custom Word2Vec (0.8435) is higher than SVM with Google Word2Vec (0.8161).
•This indicates that domain-specific embeddings (Custom Word2Vec) are more effective than general-purpose embeddings (Google Word2Vec) for sentiment classification in Amazon reviews.

Feedforward Neural Networks

Setting device

Using the GPU of mac, MPS for accelaration.

```
device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")

print(f"Using device: {device}")
```

```
Using device: mps
```

MLP

This code implements and trains a Multi-Layer Perceptron (MLP) to classify sentiment using different feature representations. The neural network consists of three fully connected layers with ReLU activation. The model is trained using Cross-Entropy Loss and optimized with Adam.

The network consists of:
• Input layer → Fully connected (linear) layer with 50 neurons
• Hidden layer → Fully connected layer with 10 neurons
• Output layer → Fully connected layer matching the number of output classes

ReLU activation is used in the first two layers to introduce non-linearity.

In the train_mlp function, an MLP instance is created based on the input and output sizes. The model is trained for 10 epochs using mini-batches (batch size = 32) from the DataLoader.

The prep_tensors function converts input features into PyTorch tensors. Also, I am shifting labels in this function to have zero-based indexing which is needed for classification. Since, our labels are {1, 2, 3}, it is changed to {0, 1, 2}.

```
class MLP(nn.Module):
    def __init__(self, input_size, output_size):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, 50)
        self.fc2 = nn.Linear(50, 10)
        self.fc3 = nn.Linear(10, output_size)
        self.relu = nn.ReLU()

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

def train_mlp(X_train, y_train, X_test, y_test, input_size, output_size, model_name):
    mod = MLP(input_size, output_size).to(device)
    loss_fn = nn.CrossEntropyLoss()
    optimizer = optim.Adam(mod.parameters(), lr=0.001)
    train_loader = DataLoader(TensorDataset(X_train, y_train), batch_size=32, shuffle=True)

    for epoch in range(10):
        for batch_X, batch_y in train_loader:
            optimizer.zero_grad()
            outputs = mod(batch_X)
            loss = loss_fn(outputs, batch_y)
            loss.backward()
            optimizer.step()

    with torch.no_grad():
        test_outputs = mod(X_test)
        predictions = torch.argmax(test_outputs, dim=1)
        accuracy = (predictions == y_test).float().mean().item()

    print(f"{model_name}: {accuracy:.4f}")

def prep_tensors(X, y, shift_labels=True):
    y_tensor = torch.tensor(y.values, dtype=torch.long)
    if shift_labels:
        y_tensor -= 1
    return torch.tensor(X, dtype=torch.float32).to(device), y_tensor.to(device)
```

Data prep for ternary classification

Similar to data preparation of the binary classification, here I am splitting data for ternary classification which is stored in balanced_df. Again, I have stored the tokenized version of the review_body separately. I have applied the avg_vector function to get the average Word2Vec features for ternary classification.

```
X_train_ter, X_test_ter, y_train_ter, y_test_ter = train_test_split(balanced_df['review_body'], balanced_df['sentiment'], test_s

X_train_ter_token = [word_tokenize(text) for text in X_train_ter]
X_test_ter_token = [word_tokenize(text) for text in X_test_ter]

X_train_ter_avg_pre = np.array([avg_vector(review, pre_model) for review in X_train_ter_token])
X_test_ter_avg_pre = np.array([avg_vector(review, pre_model) for review in X_test_ter_token])

X_train_ter_avg_cus = np.array([avg_vector(review, custom_mod.wv) for review in X_train_ter_token])
X_test_ter_avg_cus = np.array([avg_vector(review, custom_mod.wv) for review in X_test_ter_token])
```

Average w2v (part a)

Here, I am converting the input average w2v features into Pytorch tensors using the prep_tensor function. So the tensors for both custom and pretrained model for binary and ternary classification are obtained here.

```
#AVG custom-binary
X_train_avg_bin_tensor, y_train_bin_tensor = prep_tensors(X_train_bin_avg_cus, y_train_bin)
X_test_avg_bin_tensor, y_test_bin_tensor = prep_tensors(X_test_bin_avg_cus, y_test_bin)

#AVG pretrained-binary
X_train_avg_google_tensor, y_train_google_tensor = prep_tensors(X_train_bin_avg_pre, y_train_bin)
X_test_avg_google_tensor, y_test_google_tensor = prep_tensors(X_test_bin_avg_pre, y_test_bin)

#AVG custom-ternary
X_train_avg_ter_tensor, y_train_ter_tensor = prep_tensors(X_train_ter_avg_cus, y_train_ter)
X_test_avg_ter_tensor, y_test_ter_tensor = prep_tensors(X_test_ter_avg_cus, y_test_ter)

#AVG pretrained-ternary
X_train_avg_google_ter_tensor, y_train_google_ter_tensor = prep_tensors(X_train_ter_avg_pre, y_train_ter)
X_test_avg_google_ter_tensor, y_test_google_ter_tensor = prep_tensors(X_test_ter_avg_pre, y_test_ter)
```

```
train_mlp(X_train_avg_google_tensor, y_train_google_tensor, X_test_avg_google_tensor, y_test_google_tensor, 300, 2, "Binary - Av
```

```
Binary - Avg W2V (Google): 0.8434
```

```
train_mlp(X_train_avg_bin_tensor, y_train_bin_tensor, X_test_avg_bin_tensor, y_test_bin_tensor, 300, 2, "Binary - Avg Custom W2V
```

```
Binary - Avg Custom W2V: 0.8625
```

```
train_mlp(X_train_avg_google_ter_tensor, y_train_google_ter_tensor, X_test_avg_google_ter_tensor, y_test_google_ter_tensor, 300,
```

```
Ternary - Avg W2V (Google): 0.6860
```

```
train_mlp(X_train_avg_ter_tensor, y_train_ter_tensor, X_test_avg_ter_tensor, y_test_ter_tensor, 300, 3, "Ternary - Avg Custom W2
```

```
Ternary - Avg Custom W2V: 0.7031
```

These are the accuracy values using the average Word2Vec

Classification Type	Feature Representation	Accuracy
Binary	Avg W2V (Google)	0.8434
Binary	Avg Custom W2V	0.8625
Ternary	Avg W2V (Google)	0.6860
Ternary	Avg Custom W2V	0.7031

Concatenated w2v (part b)

Here, I am concatenating the first 10 Word2Vec vectors for each review as the input feature using the concat_vector function.
This function is applied to train and test tokens of both binary and ternary data. Later, the tensors are obtained similar to the previous section.

```
def concat_vector(review, model, vector_size=300, max_words=10):
    vectors = [model[word] for word in review if word in model][:max_words]
    while len(vectors) < max_words:
        vectors.append(np.zeros(vector_size))
    return np.concatenate(vectors)

X_train_bin_concat_pre = np.array([concat_vector(review, pre_model) for review in X_train_token])
X_test_bin_concat_pre = np.array([concat_vector(review, pre_model) for review in X_test_token])

X_train_bin_concat_cus = np.array([concat_vector(review, custom_mod.wv) for review in X_train_token])
X_test_bin_concat_cus = np.array([concat_vector(review, custom_mod.wv) for review in X_test_token])

X_train_ter_concat_pre = np.array([concat_vector(review, pre_model) for review in X_train_ter_token])
X_test_ter_concat_pre = np.array([concat_vector(review, pre_model) for review in X_test_ter_token])

X_train_ter_concat_cus = np.array([concat_vector(review, custom_mod.wv) for review in X_train_ter_token])
X_test_ter_concat_cus = np.array([concat_vector(review, custom_mod.wv) for review in X_test_ter_token])

#CONCAT custom-binary
X_train_concat_bin_tensor, y_train_bin_tensor = prep_tensors(X_train_bin_concat_cus, y_train_bin)
X_test_concat_bin_tensor, y_test_bin_tensor = prep_tensors(X_test_bin_concat_cus, y_test_bin)

#CONCAT pretrained-binary
X_train_concat_google_tensor, y_train_google_tensor = prep_tensors(X_train_bin_concat_pre, y_train_bin)
X_test_concat_google_tensor, y_test_google_tensor = prep_tensors(X_test_bin_concat_pre, y_test_bin)

#CONCAT custom-ternary
X_train_concat_ter_tensor, y_train_ter_tensor = prep_tensors(X_train_ter_concat_cus, y_train_ter)
X_test_concat_ter_tensor, y_test_ter_tensor = prep_tensors(X_test_ter_concat_cus, y_test_ter)

#CONCAT pretrained-ternary
X_train_concat_google_ter_tensor, y_train_google_ter_tensor = prep_tensors(X_train_ter_concat_pre, y_train_ter)
X_test_concat_google_ter_tensor, y_test_google_ter_tensor = prep_tensors(X_test_ter_concat_pre, y_test_ter)
```

```
train_mlp(X_train_concat_google_tensor, y_train_google_tensor, X_test_concat_google_tensor, y_test_google_tensor, 3000, 2, "Bina
```

```
Binary - concat W2V (Google): 0.7531
```

```
train_mlp(X_train_concat_bin_tensor, y_train_bin_tensor, X_test_concat_bin_tensor, y_test_bin_tensor, 3000, 2, "Binary - concat
```

```
Binary - concat Custom W2V: 0.7735
```

```
train_mlp(X_train_concat_google_ter_tensor, y_train_google_ter_tensor, X_test_concat_google_ter_tensor, y_test_google_ter_tensor
```

```
Ternary - concat W2V (Google): 0.5983
```

```
train_mlp(X_train_concat_ter_tensor, y_train_ter_tensor, X_test_concat_ter_tensor, y_test_ter_tensor, 3000, 3, "Ternary - concat
```

```
Ternary - concat Custom W2V: 0.6183
```

These are the accuracy values using the concatenated Word2Vec

Classification Type	Feature Representation	Accuracy
Binary	Concat W2V (Google)	0.7560
Binary	Concat Custom W2V	0.7735
Ternary	Concat W2V (Google)	0.5903
Ternary	Concat Custom W2V	0.6176

Conclusion

Feedforward Neural Network (FNN) with Average Word2Vec performs better than Simple Models:
•Binary - Avg Custom W2V (0.86225) achieves nearly the same accuracy as SVM with TF-IDF (0.8637), making it a strong alternative.
•Binary - Avg W2V (Google) (0.8434) outperforms both SVM with Google Word2Vec (0.8161) and SVM with Custom Word2Vec (0.8435).

FNN with Concatenated Word2Vec performs worse than Simple Models:
•Binary - concat Custom W2V (0.7735) and Binary - concat W2V (Google) (0.7560) have lower accuracy than all simple models.

•This suggests that averaging word embeddings is a better feature representation than concatenation for sentiment classification.

Ternary - Avg Custom W2V (0.7031) and Ternary - Avg W2V (Google) (0.6860) show a noticeable drop in accuracy compared to binary classification.
Ternary - concat Custom W2V (0.6176) and Ternary - concat W2V (Google) (0.5903) perform the worst, reinforcing that concatenation is less effective.

Overall, FNN with average embeddings is competitive with SVM models, particularly for binary classification. TF-IDF remains the strongest feature extraction method overall. Ternary classification struggles across all methods, showing that distinguishing three sentiment classes is much harder than binary classification.

Convolutional Neural Networks

w2c_embedding function generates word embeddings for a given tokenized review using a Word2Vec model. It returns a (50 × 300) matrix, where each row represents a word vector. To maintain consistency in input size, all reviews are padded or truncated to a fixed length of 50 words.

The SADataset class is designed to store and provide access to tokenized reviews and their corresponding labels in a format compatible with PyTorch's DataLoader. It accepts tokenized review text, sentiment labels(zero-based index), and a Word2Vec model.
I used __ getitem __ method retrieves the precomputed (50 × 300) embedding matrix for each review and its corresponding label.

After that, the dataset is prepared for both binary and ternary classification.

```
# Extracting Word2Vec embeddings
def w2c_embedding(tokens, model, max_len=50, vector_size=300):
    f_vectors = [model[word] if word in model else np.zeros(vector_size, dtype=np.float32) for word in tokens]
    if len(f_vectors) < max_len:
        f_vectors.extend([np.zeros(vector_size, dtype=np.float32)] * (max_len - len(f_vectors)))
    return np.array(f_vectors[:max_len], dtype=np.float32)

# PyTorch Dataset Class
class SADataset(Dataset):
    def __init__(self, review_tokens, labels, model):
        self.review_tokens = review_tokens
        self.labels = [label - 1 for label in labels]  # zero based indexing
        self.model = model.wv if hasattr(model, "wv") else model  # Word2Vec Model

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

    def __getitem__(self, idx):
        tokens = self.review_tokens[idx]
        embedding = w2c_embedding(tokens, self.model)
        label = self.labels[idx]
        return torch.tensor(embedding, dtype=torch.float32), torch.tensor(label, dtype=torch.long)

train_bin_pre = SADataset(X_train_token, y_train_bin, pre_model)
test_bin_pre = SADataset(X_test_token, y_test_bin, pre_model)
train_bin_cus = SADataset(X_train_token, y_train_bin, custom_mod.wv)
test_bin_cus = SADataset(X_test_token, y_test_bin, custom_mod.wv)
train_ter_pre = SADataset(X_train_ter_token, y_train_ter, pre_model)
test_ter_pre = SADataset(X_test_ter_token, y_test_ter, pre_model)
train_ter_cus = SADataset(X_train_ter_token, y_train_ter, custom_mod.wv)
test_ter_cus = SADataset(X_test_ter_token, y_test_ter, custom_mod.wv)
```

The CNN model is designed to process word embedding matrices of size (50 × 300), where each review is represented as a sequence of 50 words, with each word embedded into a 300-dimensional vector.
Kernel size = 3
I used ReLU function for activation.
The input and output size for the first layer are 300 and 50 channels respectively.
The input and output size for the second layer are 50 and 10 channels respectively.
The output of the second convolutional layer is flattened and fed into a fully connected layer with output neurons equal to the number of

classes (binary: 2, ternary: 3).
I used _get_conv_output function to compute the output size of the convolutional layers dynamically, ensuring compatibility with varying input lengths.

After this, train_cnn function trains the CNN model using CrossEntropyLoss and the Adam optimizer. The model is trained for a default of 10 epochs with a learning rate of 0.001. The model iterates through mini-batches from the training set.
After training, the model is set to evaluation mode (model.eval()). The accuracy is calculated on the test set by comparing predicted and actual labels.

The datasets are wrapped in PyTorch's DataLoader to facilitate efficient batch processing. I have set the batch size to 64 and also I have enabled shuffling to enhance model generalization.

```
class CNN(nn.Module):
    def __init__(self, input_channels=300, op_size=3):  # 3 classes for ternary
        super(CNN, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=input_channels, out_channels=50, kernel_size=3, padding=1)
        self.conv2 = nn.Conv1d(in_channels=50, out_channels=10, kernel_size=3, padding=1)
        self.pool = nn.MaxPool1d(kernel_size=2)

        sam_ip = torch.zeros(1, input_channels, 50)
        sam_op = self._get_conv_output(sam_ip)
        self.fc1 = nn.Linear(sam_op, op_size)

    def _get_conv_output(self, x):
        x = torch.relu(self.conv1(x))
        x = self.pool(x)
        x = torch.relu(self.conv2(x))
        x = self.pool(x)
        return x.view(x.shape[0], -1).shape[1]  # Flatten size

    def forward(self, x):
        x = x.permute(0, 2, 1)  #(batch, channel, sequence)
        x = torch.relu(self.conv1(x))
        x = self.pool(x)
        x = torch.relu(self.conv2(x))
        x = self.pool(x)
        x = x.view(x.shape[0], -1)  # Flatten before fc1
        x = self.fc1(x)
        return x

#Train CNN Model with DataLoader
def train_cnn(train_loader, test_loader, num_classes, epochs=10, lr=0.001):
    model = CNN(op_size=num_classes).to(device)
    loss_fn = nn.CrossEntropyLoss()
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)

    for epoch in range(epochs):
        model.train()
        epoch_loss = 0
        for batch_X, batch_y in train_loader:
            batch_X, batch_y = batch_X.to(device), batch_y.to(device)

            optimizer.zero_grad()
            outputs = model(batch_X)
            loss = loss_fn(outputs, batch_y)
            loss.backward()
            optimizer.step()
            epoch_loss += loss.item()

    model.eval()
    correct, total = 0, 0
    with torch.no_grad():
        for batch_X, batch_y in test_loader:
            batch_X, batch_y = batch_X.to(device), batch_y.to(device)
            outputs = model(batch_X)
            _, predicted = torch.max(outputs, 1)
            total += batch_y.size(0)
            correct += (predicted == batch_y).sum().item()

    accuracy = correct / total
    print(f"{accuracy:.4f}")
    return accuracy
```

```
# Dataloaders for binary and ternary classification
train_bin_cus_loader = DataLoader(train_bin_cus, batch_size=64, shuffle=True)
test_bin_cus_loader = DataLoader(test_bin_cus, batch_size=64, shuffle=False)
train_bin_pre_loader = DataLoader(train_bin_pre, batch_size=64, shuffle=True)
test_bin_pre_loader = DataLoader(test_bin_pre, batch_size=64, shuffle=False)
train_ter_cus_loader = DataLoader(train_ter_cus, batch_size=64, shuffle=True)
test_ter_cus_loader = DataLoader(test_ter_cus, batch_size=64, shuffle=False)
train_ter_pre_loader = DataLoader(train_ter_pre, batch_size=64, shuffle=True)
test_ter_pre_loader = DataLoader(test_ter_pre, batch_size=64, shuffle=False)
```

```
print("\nTraining CNN for Binary Classification")
print("\nAccuracy(pretrained model with CNN): ")
accuracy_cnn_bin_pre = train_cnn(train_bin_pre_loader, test_bin_pre_loader, num_classes=2)
print("\nAccuracy(custom model with CNN):")
accuracy_cnn_bin_cus = train_cnn(train_bin_cus_loader, test_bin_cus_loader, num_classes=2)
```

```
Training CNN for Binary Classification

Accuracy(pretrained model with CNN):
0.8610

Accuracy(custom model with CNN):
0.8642
```

```
print("\nTraining CNN for Ternary Classification")
print("\nAccuracy(pretrained model with CNN):")
accuracy_cnn_tern_pre = train_cnn(train_ter_pre_loader, test_ter_pre_loader, num_classes=3)
```

```
Training CNN for Ternary Classification

Accuracy(pretrained model with CNN):
0.7005
```

```
print("\nAccuracy(custom model with CNN):")
accuracy_cnn_tern_cus = train_cnn(train_ter_cus_loader, test_ter_cus_loader, num_classes=3)
```

```
Accuracy(custom model with CNN):
0.7069
```

These are the accuracy values on the testing split of binary and ternary data using CNN:

Model	Pretrained Word2Vec	Custom Word2Vec
Binary Classification	0.8610	0.8642
Ternary Classification	0.7005	0.7069