Dependencies

```
import pandas as pd
import numpy as np

import gensim.downloader as api
import gensim.models
from gensim.test.utils import datapath
from gensim import utils
from gensim.models import KeyedVectors

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


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

1

```
class AmazonReviewDataset(Dataset):

    def __init__(self, file_path, transform=None):
        self.file_path = file_path
        self.transform = transform
        self._load_and_preprocess_data()

    def _load_and_preprocess_data(self):
        try:
            amazon_data = pd.read_csv(self.file_path, sep='\t',
on_bad_lines='skip')
        except FileNotFoundError:
            print("File not found. Ensure the correct path is
provided.")
            return

        important_fields = ['star_rating', 'review_body']
        amazon_data =
amazon_data[important_fields].dropna(subset=['star_rating'])
        amazon_data['review_body'] =
amazon_data['review_body'].fillna("")
```

```
        amazon_data['Class'] = amazon_data['star_rating'].apply(lambda
rating: 0 if rating in [1, 2, 3] else 1)

        review_size = 50000
        balanced_data = pd.concat([
            amazon_data[amazon_data['Class'] ==
0].sample(n=review_size, random_state=4),
            amazon_data[amazon_data['Class'] ==
1].sample(n=review_size, random_state=4)
        ], axis=0)

        self.dataframe = balanced_data
        self.dataframe['tokenized_review'] =
balanced_data['review_body'].apply(utils.simple_preprocess)

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

    def __getitem__(self, index):
        tokenized_review = self.dataframe.iloc[index]
['tokenized_review']
        label = self.dataframe.iloc[index]['Class']

        if self.transform:
            tokenized_review = self.transform(tokenized_review)
        return tokenized_review, label
```

2

Comparing the two approaches, it is clear that the google pretrained model does a better job of encoding semantic simlarities between words.   Pretrained Model (Google): [('queen', 0.7118193507194519)] Trained Model: [('rolodex', 0.5274915099143982)]

- MAKE SURE YOU CHANGE THE FILE PATH TO WHERE DATA.TSV IS ON YOUR LOCAL MACHINE.
- https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec. html

```
#Google
wv = api.load('word2vec-google-news-300')
res = wv.most_similar(positive=['woman', 'king'], negative = ['man'],
topn=1)
print("Pretrained Model (Google): ", res)

#Me
data_path = '/content/data.tsv'  # Replace with your actual path
amazon_data = AmazonReviewDataset(data_path)

wrd2vec_reviews = [tokens for tokens, _ in amazon_data]
```

```
wrd2vec = gensim.models.Word2Vec(sentences= wrd2vec_reviews,
vector_size=300, window=13, min_count=9)

result_own_model = wrd2vec.wv.most_similar(positive=['woman', 'king'],
negative=['man'], topn=1)
print(f"Trained Model: {result_own_model}")
```

```
---------------------------------------------------------------------------
KeyboardInterrupt                         Traceback (most recent call
last)
/Users/buckethoop/Downloads/544HW3/hw3.ipynb Cell 5 line 1
----> <a
href='vscode-notebook-cell:/Users/buckethoop/Downloads/544HW3/hw3.ipyn
b#W5sZmlsZQ%3D%3D?line=0'>1</a> wv = api.load('word2vec-google-news-
300')
      <a
href='vscode-notebook-cell:/Users/buckethoop/Downloads/544HW3/hw3.ipyn
b#W5sZmlsZQ%3D%3D?line=1'>2</a> res =
wv.most_similar(positive=['woman', 'king'], negative = ['man'],
topn=1)
      <a
href='vscode-notebook-cell:/Users/buckethoop/Downloads/544HW3/hw3.ipyn
b#W5sZmlsZQ%3D%3D?line=2'>3</a> print("google: ", res)

File
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/gensim/downloader.py:503, in load(name, return_path)
    501 sys.path.insert(0, BASE_DIR)
    502 module = __import__(name)
--> 503 return module.load_data()

File ~/gensim-data/word2vec-google-news-300/__init__.py:8, in
load_data()
      6 def load_data():
      7     path = os.path.join(base_dir, 'word2vec-google-news-300',
"word2vec-google-news-300.gz")
----> 8     model = KeyedVectors.load_word2vec_format(path,
binary=True)
      9     return model

File
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/gensim/models/keyedvectors.py:1719, in
KeyedVectors.load_word2vec_format(cls, fname, fvocab, binary,
encoding, unicode_errors, limit, datatype, no_header)
   1672 @classmethod
   1673 def load_word2vec_format(
   1674         cls, fname, fvocab=None, binary=False,
```

```
encoding='utf8', unicode_errors='strict',
   1675         limit=None, datatype=REAL, no_header=False,
   1676     ):
   1677     """Load KeyedVectors from a file produced by the original
C word2vec-tool format.
   1678
   1679     Warnings
   (...)
   1717
   1718     """
-> 1719     return _load_word2vec_format(
   1720         cls, fname, fvocab=fvocab, binary=binary,
encoding=encoding, unicode_errors=unicode_errors,
   1721         limit=limit, datatype=datatype, no_header=no_header,
   1722     )

File
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/gensim/models/keyedvectors.py:2065, in
_load_word2vec_format(cls, fname, fvocab, binary, encoding,
unicode_errors, limit, datatype, no_header, binary_chunk_size)
   2062 kv = cls(vector_size, vocab_size, dtype=datatype)
   2064 if binary:
-> 2065     _word2vec_read_binary(
   2066         fin, kv, counts, vocab_size, vector_size, datatype,
unicode_errors, binary_chunk_size, encoding
   2067     )
   2068 else:
   2069     _word2vec_read_text(fin, kv, counts, vocab_size,
vector_size, datatype, unicode_errors, encoding)

File
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/gensim/models/keyedvectors.py:1960, in
_word2vec_read_binary(fin, kv, counts, vocab_size, vector_size,
datatype, unicode_errors, binary_chunk_size, encoding)
   1958 new_chunk = fin.read(binary_chunk_size)
   1959 chunk += new_chunk
-> 1960 processed_words, chunk = _add_bytes_to_kv(
   1961     kv, counts, chunk, vocab_size, vector_size, datatype,
unicode_errors, encoding)
   1962 tot_processed_words += processed_words
   1963 if len(new_chunk) < binary_chunk_size:

File
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/gensim/models/keyedvectors.py:1943, in _add_bytes_to_kv(kv,
counts, chunk, vocab_size, vector_size, datatype, unicode_errors,
encoding)
   1941 word = word.lstrip('\n')
```

```
   1942 vector = frombuffer(chunk, offset=i_vector, count=vector_size,
dtype=REAL).astype(datatype)
-> 1943 _add_word_to_kv(kv, counts, word, vector, vocab_size)
   1944 start = i_vector + bytes_per_vector
   1945 processed_words += 1

File
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/gensim/models/keyedvectors.py:1911, in _add_word_to_kv(kv,
counts, word, weights, vocab_size)
   1909     logger.warning("duplicate word '%s' in word2vec file,
ignoring all but first", word)
   1910     return
-> 1911 word_id = kv.add_vector(word, weights)
   1913 if counts is None:
   1914     # Most common scenario: no vocab file given. Just make up
some bogus counts, in descending order.
   1915     # TODO (someday): make this faking optional, include more
realistic (Zipf-based) fake numbers.
   1916     word_count = vocab_size - word_id

File
/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-
packages/gensim/models/keyedvectors.py:548, in
KeyedVectors.add_vector(self, key, vector)
    527 """Add one new vector at the given key, into existing slot if
available.
    528
    529 Warning: using this repeatedly is inefficient, requiring a
full reallocation & copy,
   (...)
    545
    546 """
    547 target_index = self.next_index
--> 548 if target_index >= len(self) or
self.index_to_key[target_index] is not None:
    549     # must append at end by expanding existing structures
    550     target_index = len(self)
    551     warnings.warn(
    552         "Adding single vectors to a KeyedVectors which grows
by one each time can be costly. "
    553         "Consider adding in batches or preallocating to the
required size.",
    554         UserWarning)

KeyboardInterrupt:
```

3

The Word2Vec approach yields similar results than that of TFIDF. You can see the differences below:

Perceptron Accuracy (Word2Vec): 79.155% SVM Accuracy (Word2Vec): 80.979%   TFIDF (PERC): Prec: 0.7817998994469583 ,Rec: 0776956130708504 , F1: 0.7793704891740175 TFIDF TFIDF (SVM): Prec: 0.8193973258830572 ,Rec: 0.8206255621065255 , F1: 0.8200109840730939

*Note: I used the TFIDF values from the previous HW1

```
def avg_word2vec(review, w2v, num_features=300):
    feature_vec = np.zeros((num_features,), dtype='float32')
    n_words = 0
    for word in review:
        if word in w2v:
            n_words += 1
            feature_vec = np.add(feature_vec, w2v[word])
    if n_words:
        feature_vec = np.divide(feature_vec, n_words)
    return feature_vec

data_features = np.array([avg_word2vec(review, wv) for review, _ in
amazon_data])
labels = np.array([label for _, label in amazon_data])

X_train, X_test, y_train, y_test = train_test_split(data_features,
labels, test_size=0.2, random_state=30)

perc = Perceptron(max_iter=5000)
perc.fit(X_train, y_train)
perc_pred = perc.predict(X_test)

svm = LinearSVC(max_iter=1000)
svm.fit(X_train, y_train)
svm_pred = svm.predict(X_test)


print(f"Perceptron Accuracy (Word2Vec):, {100 * accuracy_score(y_test,
perc_pred)}%")
print(f"SVM Accuracy (Word2Vec):, {100 * accuracy_score(y_test,
svm_pred)}%")
```

```
---------------------------------------------------------------------------
-----
KeyboardInterrupt                         Traceback (most recent call
last)
/Users/buckethoop/Downloads/544HW3/hw3.ipynb Cell 9 line 1
     <a
href='vscode-notebook-cell:/Users/buckethoop/Downloads/544HW3/hw3.ipyn
```

```
b#X10sZmlsZQ%3D%3D?line=12'>13</a>     return feature_vector
     <a
href='vscode-notebook-cell:/Users/buckethoop/Downloads/544HW3/hw3.ipyn
b#X10sZmlsZQ%3D%3D?line=15'>16</a> num_features = 300
---> <a
href='vscode-notebook-cell:/Users/buckethoop/Downloads/544HW3/hw3.ipyn
b#X10sZmlsZQ%3D%3D?line=16'>17</a> X =
np.array([calculate_average_word2vec(wv, review, num_features) for
review in amazon_data.preprocessed_data])
     <a
href='vscode-notebook-cell:/Users/buckethoop/Downloads/544HW3/hw3.ipyn
b#X10sZmlsZQ%3D%3D?line=17'>18</a> Y =
amazon_data._load_and_preprocess_data['Class'].values  # Ensure this
line matches your data structure
     <a
href='vscode-notebook-cell:/Users/buckethoop/Downloads/544HW3/hw3.ipyn
b#X10sZmlsZQ%3D%3D?line=20'>21</a> # Split data

/Users/buckethoop/Downloads/544HW3/hw3.ipynb Cell 9 line 1
     <a
href='vscode-notebook-cell:/Users/buckethoop/Downloads/544HW3/hw3.ipyn
b#X10sZmlsZQ%3D%3D?line=12'>13</a>     return feature_vector
     <a
href='vscode-notebook-cell:/Users/buckethoop/Downloads/544HW3/hw3.ipyn
b#X10sZmlsZQ%3D%3D?line=15'>16</a> num_features = 300
---> <a
href='vscode-notebook-cell:/Users/buckethoop/Downloads/544HW3/hw3.ipyn
b#X10sZmlsZQ%3D%3D?line=16'>17</a> X =
np.array([calculate_average_word2vec(wv, review, num_features) for
review in amazon_data.preprocessed_data])
     <a
href='vscode-notebook-cell:/Users/buckethoop/Downloads/544HW3/hw3.ipyn
b#X10sZmlsZQ%3D%3D?line=17'>18</a> Y =
amazon_data._load_and_preprocess_data['Class'].values  # Ensure this
line matches your data structure
     <a
href='vscode-notebook-cell:/Users/buckethoop/Downloads/544HW3/hw3.ipyn
b#X10sZmlsZQ%3D%3D?line=20'>21</a> # Split data

/Users/buckethoop/Downloads/544HW3/hw3.ipynb Cell 9 line 5
      <a
href='vscode-notebook-cell:/Users/buckethoop/Downloads/544HW3/hw3.ipyn
b#X10sZmlsZQ%3D%3D?line=1'>2</a> feature_vector =
np.zeros((num_features,), dtype="float32")
      <a
href='vscode-notebook-cell:/Users/buckethoop/Downloads/544HW3/hw3.ipyn
b#X10sZmlsZQ%3D%3D?line=2'>3</a> n_words = 0
----> <a
href='vscode-notebook-cell:/Users/buckethoop/Downloads/544HW3/hw3.ipyn
```

```
b#X10sZmlsZQ%3D%3D?line=4'>5</a> index2word_set =
set(word2vec_model.index_to_key)
      <a
href='vscode-notebook-cell:/Users/buckethoop/Downloads/544HW3/hw3.ipyn
b#X10sZmlsZQ%3D%3D?line=6'>7</a> for word in text:
      <a
href='vscode-notebook-cell:/Users/buckethoop/Downloads/544HW3/hw3.ipyn
b#X10sZmlsZQ%3D%3D?line=7'>8</a>     if word in index2word_set:

KeyboardInterrupt:
```

4a

My accuracy for the section was: 82.72%

- https://www.kaggle.com/mishra1993/pytorch-multi-layer-perceptron-mnist

```
data_features = [avg_word2vec(review, wv) for review, _ in
amazon_data]
labels = [label for _, label in amazon_data]

features_tensor = torch.tensor(data_features, dtype=torch.float32)
labels_tensor = torch.tensor(labels, dtype=torch.long)

dataset = TensorDataset(features_tensor, labels_tensor)
train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_dataset, test_dataset = torch.utils.data.random_split(dataset,
[train_size, test_size])

batch_size = 64
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=True)

class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(300, 50)
        self.fc2 = nn.Linear(50, 5)
        self.fc3 = nn.Linear(5, 2)
        self.relu = nn.ReLU()

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

```
        return x

model = MLP()
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
epochs = 100

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

correct = 0
total = 0
with torch.no_grad():
    for inputs, labels in test_loader:
        outputs = model(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"Accuracy: {100 * correct / total}%")
```

4b

The accuracy value for this section was: 72.43%

Comparing the accuracies from 4a and 4b you can see that the concatenation approach doesn't perform as well as the average Word2Vec approach. I think that this is because when we only consider the first 10 Word2Vec vectors we run the risk of losing information. With the average vectors approach we get a more holistic look at that information.

```
def review_to_vector(review, w2v_model, max_len=10):
    vectors = []
    for word in review:
        if word in w2v_model.wv:
            vectors.append(w2v_model.wv[word])

    if len(vectors) < max_len:
        vectors.extend([np.zeros(w2v_model.vector_size) for _ in
range(max_len - len(vectors))])
    else:
        vectors = vectors[:max_len]

    return np.concatenate(vectors, axis=0)
```

```
X = [review_to_vector(review, wrd2vec) for review, _ in amazon_data]
# Note: Using wrd2vec model
y = [label for _, label in amazon_data]

X_tensor = torch.tensor(X, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.long)

dataset = TensorDataset(X_tensor, y_tensor)
train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_dataset, test_dataset = random_split(dataset, [train_size,
test_size])

batch_size = 64
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)


class MLP(nn.Module):
    def __init__(self, input_size, hidden_size1=50, hidden_size2=5,
output_size=2):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size1)
        self.fc2 = nn.Linear(hidden_size1, hidden_size2)
        self.fc3 = nn.Linear(hidden_size2, output_size)
        self.relu = nn.ReLU()

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

input_size = wrd2vec.vector_size * 10  # 10 concatenated Word2Vec
vectors
mlp_model = MLP(input_size)

criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(mlp_model.parameters(), lr=0.001)
epochs = 100

for epoch in range(epochs):
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = mlp_model(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
```

```
        optimizer.step()

correct = 0
total = 0
with torch.no_grad():
    for inputs, labels in test_loader:
        outputs = mlp_model(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"Accuracy: {100 * correct / total}%")
```

5a

My accuracy in this section is: 81.8%

- https://pytorch.org/tutorials/intermediate/char_rnn_classification_ tutorial.html

```
def review_to_vector(review, w2v_model, max_len=10):
    vectors = [w2v_model.wv[word] if word in w2v_model.wv else
np.zeros(w2v_model.vector_size) for word in review[:max_len]]
    while len(vectors) < max_len:
        vectors.append(np.zeros(w2v_model.vector_size))
    return np.array(vectors)

X = [review_to_vector(review, wrd2vec) for review, _ in amazon_data]
y = [label for _, label in amazon_data]

X_tensor = torch.tensor(X, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.long)

dataset = TensorDataset(X_tensor, y_tensor)
train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_dataset, test_dataset = random_split(dataset, [train_size,
test_size])

batch_size = 64
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)

class RNNModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(RNNModel, self).__init__()
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)
```

```
    def forward(self, x):
        h0 = torch.zeros(1, x.size(0), hidden_size).to(x.device)  #
Initial hidden state
        out, _ = self.rnn(x, h0)
        out = self.fc(out[:, -1, :])  # Use last sequence output as
input to FC layer
        return out

input_size = wrd2vec.vector_size
hidden_size = 10
output_size = 2

model = RNNModel(input_size, hidden_size, output_size)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
epochs = 100

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

correct = 0
total = 0
with torch.no_grad():
    for inputs, labels in test_loader:
        outputs = model(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"Accuracy: {100 * correct / total}%")
```

5b

My accuracy in this section is: 83.24%

```
def review_to_vector(review, w2v_model, max_len=10):
    vectors = [w2v_model.wv[word] if word in w2v_model.wv else
np.zeros(w2v_model.vector_size) for word in review[:max_len]]
    while len(vectors) < max_len:
        vectors.append(np.zeros(w2v_model.vector_size))
    return np.array(vectors)
```

```
X = [review_to_vector(review, wrd2vec) for review, _ in amazon_data]
y = [label for _, label in amazon_data]

X_tensor = torch.tensor(X, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.long)

dataset = TensorDataset(X_tensor, y_tensor)
train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_dataset, test_dataset = random_split(dataset, [train_size,
test_size])

batch_size = 64
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)

class GRUModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(GRUModel, self).__init__()
        self.gru = nn.GRU(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(1, x.size(0), hidden_size).to(x.device)  #
Initial hidden state
        out, _ = self.gru(x, h0)
        out = self.fc(out[:, -1, :])  # Use last sequence output as
input to FC layer
        return out

input_size = wrd2vec.vector_size
hidden_size = 10
output_size = 2

model = GRUModel(input_size, hidden_size, output_size)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
epochs = 100

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

correct = 0
```

```
total = 0
with torch.no_grad():
    for inputs, labels in test_loader:
        outputs = model(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"Accuracy using GRU: {100 * correct / total}%")
```

5c

My accuracy in this section is: 82.725%

```
def review_to_vector(review, w2v_model, max_len=10):
    vectors = [w2v_model.wv[word] if word in w2v_model.wv else
np.zeros(w2v_model.vector_size) for word in review[:max_len]]
    while len(vectors) < max_len:
        vectors.append(np.zeros(w2v_model.vector_size))
    return np.array(vectors)

X = [review_to_vector(review, wrd2vec) for review, _ in amazon_data]
y = [label for _, label in amazon_data]

X_tensor = torch.tensor(X, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.long)

dataset = TensorDataset(X_tensor, y_tensor)
train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_dataset, test_dataset = random_split(dataset, [train_size,
test_size])

batch_size = 64
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)

class LSTMModel(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(LSTMModel, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

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

```
        c0 = torch.zeros(1, x.size(0), hidden_size).to(x.device)  # Initial cell state
        out, _ = self.lstm(x, (h0, c0))
        out = self.fc(out[:, -1, :])  # Use last sequence output as input to FC layer
        return out

input_size = wrd2vec.vector_size
hidden_size = 10
output_size = 2

model = LSTMModel(input_size, hidden_size, output_size)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
epochs = 100

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

correct = 0
total = 0
with torch.no_grad():
    for inputs, labels in test_loader:
        outputs = model(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"Accuracy using LSTM: {100 * correct / total}%")
```