Name: Prathvi Shetty

Python 3.11.4

In [1]:
```
import pandas as pd
import warnings
import contractions
import re
import gensim.downloader as api
from gensim.utils import simple_preprocess
import numpy as np
from gensim.models import Word2Vec
from sklearn.feature_extraction.text import TfidfVectorizer, CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.linear_model import Perceptron
from sklearn.metrics import precision_score, recall_score, f1_score, accuracy_score
from sklearn.svm import LinearSVC
import torch
from torch.utils.data import DataLoader, Dataset
import torch.nn as nn
import torch.nn.functional as F
```

In [2]:
```
warnings.filterwarnings("ignore")
```

In [3]:
```
# Read data
data = pd.read_csv('data.tsv',on_bad_lines='skip', sep="\t",usecols=['star_rating','review_body'])

# Create 2 classes
data['class'] = data['star_rating'].apply(lambda x: 1 if x in [1,2,3] else 2)

# Select 50,000 reviews randomly for both the classes
balanced_data = pd.DataFrame(columns=data.columns)
for rating in data['class'].unique():
    class_set = data[data['class'] == rating]
    random_sample = class_set.sample(n=50000)
    balanced_data = pd.concat([balanced_data,random_sample])
```

In [4]:
```
balanced_data
```

Out[4]:

	star_rating	review_body	class
595562	5	What's not to like?	2
961291	5	Excellent product at a great price.	2
2079982	5	Pros: Cheap ink. Hassle free experie...	2
200321	5	Perfect. We needed these for a trade show and...	2
1028783	5	Great for the C&C cage i built for my guinea p...	2
...	...	...	...
757222	1	Was Very surprised. Did lots of research,...	1
802600	3	Comes with all hardware to mount directly to o...	1
1813173	2	Great phone for Skype use. Phone died about 6...	1
2602978	3	It is missing a critical feature. There is no ...	1
761848	2	I was very disappointed in this calendar. My ...	1

100000 rows × 3 columns

In [5]:
```
# Converting into lower case
balanced_data['review_body'] = balanced_data['review_body'].str.lower()
# Remove HTML tags
balanced_data['review_body'] = balanced_data['review_body'].apply(lambda d: re.sub(r'<.*?>','',str(d)))
# Remove URLs
balanced_data['review_body'] = balanced_data['review_body'].apply(lambda d: re.sub(r'https?://\S+www.\S+', '',
d))
# Remove non alphabetical character
balanced_data['review_body'] = balanced_data['review_body'].apply(lambda d: re.sub(r'[^a-zA-Z\s]','',d))
# Remove extra spaces
balanced_data['review_body'] = balanced_data['review_body'].apply(lambda d: ' '.join(d.split()))
# Perform contractions
balanced_data['review_body'] = balanced_data['review_body'].apply(lambda d: contractions.fix(d))
```

2. Word Embedding (25 points). In this part the of the assignment, you will generate Word2Vec features for the dataset you generated. You can use Gensim library for this purpose.

a) Load the pretrained "word2vec-google-news-300" Word2Vec model and learn how to extract word embeddings for your dataset. Try to check semantic similarities of the generated vectors using three examples of your own, e.g., King − M an + W oman = Queen or excellent ∼ outstanding.

In [6]:
```
# Load the model
word2vect_model = api.load("word2vec-google-news-300")
```

In [7]:
```
# Using simple_preproces and replacing null values
balanced_data = balanced_data.fillna('')
balanced_data['token'] = balanced_data['review_body'].apply(lambda x: simple_preprocess(x))
```

In [8]:
```
balanced_data
```

Out[8]:

	star_rating	review_body	class	token
595562	5	what is not to like	2	[what, is, not, to, like]
961291	5	excellent product at a great price	2	[excellent, product, at, great, price]
2079982	5	proscheap inkhassle free experiencefast shippi...	2	[proscheap, inkhassle, free, experiencefast, s...
200321	5	perfect we needed these for a trade show and t...	2	[perfect, we, needed, these, for, trade, show,...
1028783	5	great for the cc cage i built for my guinea pi...	2	[great, for, the, cc, cage, built, for, my, gu...
...	...	...	...	...
757222	1	was very surpriseddid lots of research find th...	1	[was, very, surpriseddid, lots, of, research, ...
802600	3	comes with all hardware to mount directly to o...	1	[comes, with, all, hardware, to, mount, direct...
1813173	2	great phone for skype use phone died about mon...	1	[great, phone, for, skype, use, phone, died, a...
2602978	3	it is missing a critical feature there is no w...	1	[it, is, missing, critical, feature, there, is...
761848	2	i was very disappointed in this calendar my bi...	1	[was, very, disappointed, in, this, calendar, ...

100000 rows × 4 columns

In [9]:
```
# Generate embedings
def avg_word2vect(token_list, model):
    if len(token_list) < 1:
        return np.zeros(300)
    else:
        vector = [model[word] if word in model else np.random.rand(300) for word in token_list]
    averaged = np.mean(vector, axis=0)
    return averaged
embeddings = list(balanced_data['token'].apply(lambda x: avg_word2vect(x, word2vect_model)))
```

In [10]:
```
# Examples using Google's word2vect
print(word2vect_model.most_similar(positive=['King', 'Woman'], negative=['Man'], topn=1))
print(word2vect_model.most_similar(positive=['father', 'Woman'], negative=['Man'], topn=1))
print(word2vect_model.most_similar(positive=['English', 'India'], negative=['US'], topn=1))
print(word2vect_model.most_similar(positive=['printer', 'projector'], negative=['cartridge'], topn=1))
```

```
[('Queen', 0.4929388165473938)]
[('mother', 0.7526409029960632)]
[('Hindi', 0.530886709690094)]
[('projectors', 0.6412938237190247)]
```

In [11]:
```
# Train Word2Vec model
custom_model = Word2Vec(
    sentences=balanced_data['token'].tolist(),
    vector_size=300,
    window=13,
    min_count=9,
    workers=4)
```

In [12]:
```
# Examples using custom word2vect
print(custom_model.wv.most_similar(positive=['king', 'woman'], negative=['man'], topn=1))
print(custom_model.wv.most_similar(positive=['father', 'woman'], negative=['man'], topn=1))
print(custom_model.wv.most_similar(positive=['english', 'india'], negative=['us'], topn=1))
```

```
[('reflective', 0.5921333432197571)]
[('sister', 0.6675557494163513)]
[('language', 0.6544055342674255)]
```

The pretrained model is able to encode semantic similarities between words better than the custom model as the pretrained model gives outputs that are pretty close to the provided input words. The pretrained model seems to capture the correlations between words in a much efficient manner

3. Perceptron using Word2Vec and TF-IDF features

In [13]:
```
# TFIDF for perceptron
X = balanced_data['review_body']
Y = balanced_data['class']

tfidf_vectorizer = TfidfVectorizer()
tfidf_X = tfidf_vectorizer.fit_transform(X)
X_traintf_tfidf, X_testtf_tfidf, Y_traintf_tfidf, Y_testtf_tfidf = train_test_split(tfidf_X, Y, test_size=0.2)

perceptron_tfidf = Perceptron()
perceptron_tfidf.fit(X_traintf_tfidf, Y_traintf_tfidf.astype('int'))

y_tf = perceptron_tfidf.predict(X_testtf_tfidf)


accuracy_perceptron_tf = accuracy_score(Y_testtf_tfidf.astype('int'), y_tf.astype('int'))
print("The Accuracy for Perceptron using TFIDF features are")
print(f"{accuracy_perceptron_tf :.4f}")

# Word to vect for perceptron
X_w2v = embeddings
Y_w2v = balanced_data['class']

X_traintf_w2v, X_testtf_w2v, Y_traintf_w2v, Y_testtf_w2v = train_test_split(X_w2v, Y_w2v, test_size=0.2,
random_state=50)

perceptron_w2v = Perceptron()
perceptron_w2v.fit(X_traintf_w2v, Y_traintf_w2v.astype('int'))

y_w2v = perceptron_w2v.predict(X_testtf_w2v)

accuracy_perceptron_w2v = accuracy_score(Y_testtf_w2v.astype('int'), y_w2v.astype('int'))
print("The scores for Perceptron using Word2Vec features are")
print(f"{accuracy_perceptron_w2v :.4f}")
```

```
The Accuracy for Perceptron using TFIDF features are
0.8045
The scores for Perceptron using Word2Vec features are
0.7775
```

3. SVM using Word2Vec and TF-IDF features

In [14]:
```
# TFIDF for SVM
svm_model_tf = LinearSVC()
svm_model_tf.fit(X_traintf_tfidf, Y_traintf_tfidf.astype('int'))

y_tf_svm = svm_model_tf.predict(X_testtf_tfidf)

accuracy_svm_tf = accuracy_score(Y_testtf_tfidf.astype('int'), y_tf_svm.astype('int'))
print("The scores for SVM using TFIDF features are")
print(f"{accuracy_svm_tf :.4f}")

# Word to vect for perceptron
svm_model_w2v = LinearSVC()
svm_model_w2v.fit(X_traintf_w2v, Y_traintf_w2v.astype('int'))
y_w2v_svm = svm_model_w2v.predict(X_testtf_w2v)

accuracy_svm_w2v = accuracy_score(Y_testtf_w2v.astype('int'), y_w2v_svm.astype('int'))
print("The scores for SVM using Word to vec features are")
print(f"{accuracy_svm_w2v :.4f}")
```

```
The scores for SVM using TFIDF features are
0.8554
The scores for SVM using Word to vec features are
0.7970
```

From my implementation it can be concluded that the models using TF-IDF perform better resulting in better accuracies when compared to the models using Word2Vec

4. Feedforward Neural Networks

a) To generate the input features, use the average Word2Vec vectors similar to the “Simple models” section and train the neural network. Report accuracy values on the testing split for your MLP

In [15]:
```
X = torch.tensor(embeddings)
Y = torch.tensor(balanced_data['class'].values.astype(np.int64)).long() - 1

X_train, X_test, Y_train, Y_test = train_test_split(X,Y,test_size=0.2)

# An abstract class representing a Dataset.
class AmazonDataset(Dataset):
    def __init__(self, X, Y):
        self.X = X
        self.Y = Y

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

    def __getitem__(self, index):
        return self.X[index], self.Y[index]

train_data = AmazonDataset(X_train, Y_train)
test_data = AmazonDataset(X_test, Y_test)


# number of subprocesses to use for data loading
num_workers = 0
# how many samples per batch to load
batch_size = 20
# percentage of training set to use as validation
valid_size = 0.2

# prepare data loaders
train_loader = DataLoader(train_data, batch_size=batch_size, num_workers=num_workers, shuffle=True)
test_loader = DataLoader(test_data, batch_size=batch_size, num_workers=num_workers, shuffle=False)

# define the FNN architecture
class FNN(nn.Module):
    def __init__(self, dimension):
        super(FNN, self).__init__()
        self.fc1 = nn.Linear(dimension, 50)
        self.fc2 = nn.Linear(50, 5)
        self.fc3 = nn.Linear(5, 2)

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

```
        x = self.fc3(x)
        return x

model = FNN(300)

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

num_epochs = 50

for epoch in range(num_epochs):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        outputs = model(data.float())
        loss = criterion(outputs, target)
        loss.backward()
        optimizer.step()

    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {loss.item():.4f}")

#Evaluate the Model
model.eval()
correct = 0
total = 0
prediction_list = []
with torch.no_grad():
    for data, target in test_loader:
        outputs = model(data.float())
        _, predicted = torch.max(outputs.data, 1)
        prediction_list.extend(list(np.array(predicted.cpu())))

print(f"Accuracy on test set: {100 * accuracy_score(Y_test, prediction_list)}%")
```

```
Epoch 1/50, Loss: 0.3328
Epoch 2/50, Loss: 0.3016
Epoch 3/50, Loss: 0.2124
Epoch 4/50, Loss: 0.1901
Epoch 5/50, Loss: 0.3681
Epoch 6/50, Loss: 0.4741
Epoch 7/50, Loss: 0.4968
Epoch 8/50, Loss: 0.5736
Epoch 9/50, Loss: 0.4107
Epoch 10/50, Loss: 0.3572
Epoch 11/50, Loss: 0.3188
Epoch 12/50, Loss: 0.3158
Epoch 13/50, Loss: 0.1604
Epoch 14/50, Loss: 0.4736
Epoch 15/50, Loss: 0.4189
Epoch 16/50, Loss: 0.5407
Epoch 17/50, Loss: 0.5133
Epoch 18/50, Loss: 0.4673
Epoch 19/50, Loss: 0.3893
Epoch 20/50, Loss: 0.3141
Epoch 21/50, Loss: 0.1827
Epoch 22/50, Loss: 0.5038
Epoch 23/50, Loss: 0.3588
Epoch 24/50, Loss: 0.4236
Epoch 25/50, Loss: 0.3920
Epoch 26/50, Loss: 0.4644
Epoch 27/50, Loss: 0.4323
Epoch 28/50, Loss: 0.1715
Epoch 29/50, Loss: 0.5116
Epoch 30/50, Loss: 0.4135
Epoch 31/50, Loss: 0.3885
Epoch 32/50, Loss: 0.3552
Epoch 33/50, Loss: 0.3500
Epoch 34/50, Loss: 0.5384
Epoch 35/50, Loss: 0.4258
Epoch 36/50, Loss: 0.3157
Epoch 37/50, Loss: 0.2802
Epoch 38/50, Loss: 0.3341
Epoch 39/50, Loss: 0.4260
Epoch 40/50, Loss: 0.2596
Epoch 41/50, Loss: 0.3421
Epoch 42/50, Loss: 0.3301
Epoch 43/50, Loss: 0.3613
```

```
Epoch 44/50, Loss: 0.2410
Epoch 45/50, Loss: 0.3944
Epoch 46/50, Loss: 0.3695
Epoch 47/50, Loss: 0.2425
Epoch 48/50, Loss: 0.2520
Epoch 49/50, Loss: 0.2013
Epoch 50/50, Loss: 0.3831
Accuracy on test set: 79.42%
```

In [16]:
```
def concatenate_word2vec(token_list, model, num_words=10):
    # Initialize a list to hold vectors
    vectors = []
    # Iterate through the tokens in the list
    for token in token_list[:num_words]:
        # Check if the token exists in the Word2Vec model
        if token in model:
            vectors.append(model[token])
        else:
            vectors.append(np.random.rand(300))

    # If the review has less than num_words, pad it with zero vectors
    while len(vectors) < num_words:
        vectors.append(np.zeros(300))

    # Concatenate the vectors
    concatenated_vector = np.concatenate(vectors)

    return concatenated_vector

# Generate the concatenated embeddings for each review
concatenated_embeddings = list(balanced_data['token'].apply(lambda x: concatenate_word2vec(x,
word2vect_model)))

X = torch.tensor(concatenated_embeddings)
Y = torch.tensor(balanced_data['class'].values.astype(np.int64)).long() - 1

X_train, X_test, Y_train, Y_test = train_test_split(X,Y,test_size=0.2)

train_data = AmazonDataset(X_train, Y_train)
test_data = AmazonDataset(X_test, Y_test)


# number of subprocesses to use for data loading
num_workers = 0
# how many samples per batch to load
batch_size = 20
# percentage of training set to use as validation
valid_size = 0.2

# prepare data loaders
train_loader = DataLoader(train_data, batch_size=batch_size, num_workers=num_workers, shuffle=True)
test_loader = DataLoader(test_data, batch_size=batch_size, num_workers=num_workers, shuffle=False)
```

```
# define the FNN architecture
class FNN(nn.Module):
    def __init__(self, dimension):
        super(FNN, self).__init__()
        self.fc1 = nn.Linear(dimension, 50)
        self.fc2 = nn.Linear(50, 5)
        self.fc3 = nn.Linear(5, 2)

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

model = FNN(3000)
# specify loss function
criterion = nn.CrossEntropyLoss()
# specify optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

num_epochs = 50

for epoch in range(num_epochs):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        optimizer.zero_grad()
        outputs = model(data.float())
        loss = criterion(outputs, target)
        loss.backward()
        optimizer.step()

    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {loss.item():.4f}")

# Evaluate the Model
model.eval()
correct = 0
total = 0
prediction_list = []
with torch.no_grad():
    for data, target in test_loader:
        outputs = model(data.float())
        _, predicted = torch.max(outputs.data, 1)
```

```
        prediction_list.extend(list(np.array(predicted.cpu())))

print(f"Accuracy on test set: {100 * accuracy_score(Y_test, prediction_list)}%")
```

```
Epoch 1/50, Loss: 0.5847
Epoch 2/50, Loss: 0.3547
Epoch 3/50, Loss: 0.5817
Epoch 4/50, Loss: 0.4619
Epoch 5/50, Loss: 0.2506
Epoch 6/50, Loss: 0.3506
Epoch 7/50, Loss: 0.3388
Epoch 8/50, Loss: 0.3736
Epoch 9/50, Loss: 0.2418
Epoch 10/50, Loss: 0.3006
Epoch 11/50, Loss: 0.3013
Epoch 12/50, Loss: 0.1590
Epoch 13/50, Loss: 0.0686
Epoch 14/50, Loss: 0.2021
Epoch 15/50, Loss: 0.0719
Epoch 16/50, Loss: 0.2718
Epoch 17/50, Loss: 0.0773
Epoch 18/50, Loss: 0.3071
Epoch 19/50, Loss: 0.1482
Epoch 20/50, Loss: 0.3042
Epoch 21/50, Loss: 0.0282
Epoch 22/50, Loss: 0.1709
Epoch 23/50, Loss: 0.2143
Epoch 24/50, Loss: 0.1521
Epoch 25/50, Loss: 0.2340
Epoch 26/50, Loss: 0.1867
Epoch 27/50, Loss: 0.0677
Epoch 28/50, Loss: 0.0172
Epoch 29/50, Loss: 0.0328
Epoch 30/50, Loss: 0.0393
Epoch 31/50, Loss: 0.1344
Epoch 32/50, Loss: 0.0623
Epoch 33/50, Loss: 0.1307
Epoch 34/50, Loss: 0.1306
Epoch 35/50, Loss: 0.0309
Epoch 36/50, Loss: 0.0057
Epoch 37/50, Loss: 0.0147
Epoch 38/50, Loss: 0.0087
Epoch 39/50, Loss: 0.1367
Epoch 40/50, Loss: 0.0292
Epoch 41/50, Loss: 0.1408
Epoch 42/50, Loss: 0.1814
Epoch 43/50, Loss: 0.0118
```

```
Epoch 44/50, Loss: 0.0174
Epoch 45/50, Loss: 0.0217
Epoch 46/50, Loss: 0.0765
Epoch 47/50, Loss: 0.0026
Epoch 48/50, Loss: 0.1557
Epoch 49/50, Loss: 0.0351
Epoch 50/50, Loss: 0.0029
Accuracy on test set: 70.855%
```

From the accuracies obtained for the feedforward neural networks it can be concluded that Simple models performed better with the Word2Vec embeddings that I generated. This might be due to the data that is being used

5. Recurrent Neural Networks

a) Train a simple RNN for sentiment analysis. You can consider an RNN cell with the hidden state size of 10. To feed your data into our RNN, limit the maximum review length to 10 by truncating longer reviews and padding shorter reviews with a null value (0). Report accuracy values on the testing split for your RNN model. What do you conclude by comparing accuracy values you obtain with those obtained with feedforward neural network models.

In [17]:

```
def prepare_sequence(token_list, model):
    rnn_vectors = []
    for token in token_list[:10]:
        if token in model:
            rnn_vectors.append(model[token])
        else:
            rnn_vectors.append(np.zeros(300))

    while len(rnn_vectors) < 10:
        rnn_vectors.append(np.zeros(300))

    return rnn_vectors


rnn_sequences = balanced_data['token'].apply(lambda x: prepare_sequence(x, word2vect_model))

X_rnn = torch.tensor(np.array(rnn_sequences.tolist()))
Y_rnn = torch.tensor(balanced_data['class'].values).long() - 1

X_train_rnn, X_test_rnn, Y_train_rnn, Y_test_rnn = train_test_split(X_rnn, Y_rnn, test_size=0.2,
random_state=42)
```

In [18]:
```
X = torch.tensor(embeddings)
Y = torch.tensor(balanced_data['class'].values.astype(np.int64)).long() - 1

# Define RNN architechture
class RNNModel(nn.Module):
    def __init__(self, dim, hidden_dim, output_dim):
        super(RNNModel, self).__init__()
        self.rnn = nn.RNN(dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        _, h_n = self.rnn(x)
        output = self.fc(h_n.squeeze(0))
        return output

rnn_model = RNNModel(300, 10, 2)

# specify loss function
criterion = nn.CrossEntropyLoss()
# specify optimizer
optimizer = torch.optim.Adam(rnn_model.parameters(), lr=0.001)

num_epochs = 50
# number of subprocesses to use for data loading
num_workers = 0
# how many samples per batch to load
batch_size = 20
# percentage of training set to use as validation
valid_size = 0.2

train_dataset_rnn = AmazonDataset(X_train_rnn, Y_train_rnn)
train_loader_rnn = DataLoader(train_dataset_rnn, batch_size=batch_size, num_workers=num_workers, shuffle=True)
test_dataset_rnn = AmazonDataset(X_test_rnn, Y_test_rnn)
test_loader_rnn = DataLoader(test_dataset_rnn, batch_size=batch_size, num_workers=num_workers, shuffle=False)

for epoch in range(num_epochs):
    rnn_model.train()
    for batch_idx, (data, target) in enumerate(train_loader_rnn):
        optimizer.zero_grad()
        outputs = rnn_model(data.float())
        loss = criterion(outputs, target)
        loss.backward()
        optimizer.step()
```

```
    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {loss.item():.4f}")

# Evaluate the Model
rnn_model.eval()
correct = 0
total = 0
prediction_list = []
with torch.no_grad():
    for data, target in test_loader_rnn:
        outputs = rnn_model(data.float())
        _, predicted = torch.max(outputs.data, 1)
        prediction_list.extend(list(np.array(predicted.cpu())))

print(f"Accuracy on test set: {100 * accuracy_score(Y_test_rnn, prediction_list)}%")
```

```
Epoch 1/50, Loss: 0.5162
Epoch 2/50, Loss: 0.4079
Epoch 3/50, Loss: 0.4128
Epoch 4/50, Loss: 0.3414
Epoch 5/50, Loss: 0.5033
Epoch 6/50, Loss: 0.4499
Epoch 7/50, Loss: 0.3953
Epoch 8/50, Loss: 0.5981
Epoch 9/50, Loss: 0.4769
Epoch 10/50, Loss: 0.4125
Epoch 11/50, Loss: 0.3873
Epoch 12/50, Loss: 0.5468
Epoch 13/50, Loss: 0.3387
Epoch 14/50, Loss: 0.6357
Epoch 15/50, Loss: 0.4404
Epoch 16/50, Loss: 0.5140
Epoch 17/50, Loss: 0.3417
Epoch 18/50, Loss: 0.2846
Epoch 19/50, Loss: 0.6856
Epoch 20/50, Loss: 0.3207
Epoch 21/50, Loss: 0.5669
Epoch 22/50, Loss: 0.5033
Epoch 23/50, Loss: 0.5582
Epoch 24/50, Loss: 0.2831
Epoch 25/50, Loss: 0.5615
Epoch 26/50, Loss: 0.3843
Epoch 27/50, Loss: 0.4704
Epoch 28/50, Loss: 0.4151
Epoch 29/50, Loss: 0.3823
Epoch 30/50, Loss: 0.4371
Epoch 31/50, Loss: 0.5365
Epoch 32/50, Loss: 0.3994
Epoch 33/50, Loss: 0.5854
Epoch 34/50, Loss: 0.3706
Epoch 35/50, Loss: 0.4447
Epoch 36/50, Loss: 0.6450
Epoch 37/50, Loss: 0.3674
Epoch 38/50, Loss: 0.5307
Epoch 39/50, Loss: 0.4645
Epoch 40/50, Loss: 0.3483
Epoch 41/50, Loss: 0.4117
Epoch 42/50, Loss: 0.3125
Epoch 43/50, Loss: 0.7319
```

```
Epoch 44/50, Loss: 0.4846
Epoch 45/50, Loss: 0.7604
Epoch 46/50, Loss: 0.2036
Epoch 47/50, Loss: 0.3158
Epoch 48/50, Loss: 0.3314
Epoch 49/50, Loss: 0.3841
Epoch 50/50, Loss: 0.3395
Accuracy on test set: 77.105%
```

In [19]:
```
print("Output shape:", outputs.shape)
print("Target shape:", target.shape)
```

```
Output shape: torch.Size([20, 2])
Target shape: torch.Size([20])
```

b) Repeat part (a) by considering a gated recurrent unit cell.

In [20]:
```
# specify GRU architecture
class GRU(nn.Module):
    def __init__(self):
        super(GRU, self).__init__()
        self.gru = nn.GRU(300, 10, batch_first=True)
        self.fc = nn.Linear(10,2)

    def forward(self, x):
        outputs, hidden = self.gru(x)
        out = self.fc(hidden.squeeze(0))
        return out

gru_model = GRU()

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

for epoch in range(num_epochs):
    gru_model.train()
    for batch_idx, (data, target) in enumerate(train_loader_rnn):
        optimizer.zero_grad()
        outputs = gru_model(data.float())
        loss = criterion(outputs, target)
        loss.backward()
        optimizer.step()

    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {loss.item():.4f}")

# Evaluate the Model
rnn_model.eval()
correct = 0
total = 0
prediction_list = []
with torch.no_grad():
    for data, target in test_loader_rnn:
        outputs = gru_model(data.float())
        _, predicted = torch.max(outputs.data, 1)
        prediction_list.extend(list(np.array(predicted.cpu())))
```

```
print(f"Accuracy on test set: {100 * accuracy_score(Y_test_rnn, prediction_list)}%")
```

```
Epoch 1/50, Loss: 0.7548
Epoch 2/50, Loss: 0.3663
Epoch 3/50, Loss: 0.4254
Epoch 4/50, Loss: 0.3061
Epoch 5/50, Loss: 0.6919
Epoch 6/50, Loss: 0.4189
Epoch 7/50, Loss: 0.1438
Epoch 8/50, Loss: 0.5460
Epoch 9/50, Loss: 0.5215
Epoch 10/50, Loss: 0.4524
Epoch 11/50, Loss: 0.3290
Epoch 12/50, Loss: 0.4110
Epoch 13/50, Loss: 0.5167
Epoch 14/50, Loss: 0.4712
Epoch 15/50, Loss: 0.3478
Epoch 16/50, Loss: 0.4748
Epoch 17/50, Loss: 0.2202
Epoch 18/50, Loss: 0.5135
Epoch 19/50, Loss: 0.3056
Epoch 20/50, Loss: 0.6678
Epoch 21/50, Loss: 0.4572
Epoch 22/50, Loss: 0.4655
Epoch 23/50, Loss: 0.4044
Epoch 24/50, Loss: 0.3621
Epoch 25/50, Loss: 0.4841
Epoch 26/50, Loss: 0.3916
Epoch 27/50, Loss: 0.3763
Epoch 28/50, Loss: 0.3124
Epoch 29/50, Loss: 0.4728
Epoch 30/50, Loss: 0.3608
Epoch 31/50, Loss: 0.4319
Epoch 32/50, Loss: 0.3058
Epoch 33/50, Loss: 0.4074
Epoch 34/50, Loss: 0.3467
Epoch 35/50, Loss: 0.6184
Epoch 36/50, Loss: 0.3947
Epoch 37/50, Loss: 0.2309
Epoch 38/50, Loss: 0.2335
Epoch 39/50, Loss: 0.2872
Epoch 40/50, Loss: 0.2371
Epoch 41/50, Loss: 0.4297
Epoch 42/50, Loss: 0.1880
Epoch 43/50, Loss: 0.2323
```

```
Epoch 44/50, Loss: 0.3533
Epoch 45/50, Loss: 0.1678
Epoch 46/50, Loss: 0.3348
Epoch 47/50, Loss: 0.3884
Epoch 48/50, Loss: 0.3831
Epoch 49/50, Loss: 0.1695
Epoch 50/50, Loss: 0.5356
Accuracy on test set: 78.14999999999999%
```

Repeat part (a) by considering an LSTM unit cell

In [21]:
```
# Define LSTM architecture
class LSTM(nn.Module):
    def __init__(self):
        super(LSTM, self).__init__()

        # Define the LSTM layer
        self.lstm = nn.LSTM(300,10, batch_first=True)

        # Define the output layer
        self.linear = nn.Linear(10, 2)

    def forward(self, text):
        packed_output, _ = self.lstm(text)
        out = self.linear(packed_output[:, -1, :])
        return out

lstm_model = LSTM()

# Define loss function
criterion = nn.CrossEntropyLoss()
# Define optimizers
optimizer = torch.optim.Adam(lstm_model.parameters())

for epoch in range(num_epochs):
    lstm_model.train()
    for batch_idx, (data, target) in enumerate(train_loader_rnn):
        optimizer.zero_grad()
        outputs = lstm_model(data.float())
        loss = criterion(outputs, target)
        loss.backward()
        optimizer.step()

    print(f"Epoch {epoch+1}/{num_epochs}, Loss: {loss.item():.4f}")

# Evaluate the Model
lstm_model.eval()
correct = 0
total = 0
prediction_list = []
with torch.no_grad():
    for data, target in test_loader_rnn:
        outputs = lstm_model(data.float())
        _, predicted = torch.max(outputs.data, 1)
```

```
        prediction_list.extend(list(np.array(predicted.cpu())))

print(f"Accuracy on test set: {100 * accuracy_score(Y_test_rnn, prediction_list)}%")
```

```
Epoch 1/50, Loss: 0.5085
Epoch 2/50, Loss: 0.4011
Epoch 3/50, Loss: 0.5107
Epoch 4/50, Loss: 0.2903
Epoch 5/50, Loss: 0.5231
Epoch 6/50, Loss: 0.5736
Epoch 7/50, Loss: 0.3478
Epoch 8/50, Loss: 0.3532
Epoch 9/50, Loss: 0.3659
Epoch 10/50, Loss: 0.2765
Epoch 11/50, Loss: 0.6873
Epoch 12/50, Loss: 0.6251
Epoch 13/50, Loss: 0.4043
Epoch 14/50, Loss: 0.2225
Epoch 15/50, Loss: 0.5057
Epoch 16/50, Loss: 0.3865
Epoch 17/50, Loss: 0.7284
Epoch 18/50, Loss: 0.3600
Epoch 19/50, Loss: 0.2412
Epoch 20/50, Loss: 0.5965
Epoch 21/50, Loss: 0.5539
Epoch 22/50, Loss: 0.4326
Epoch 23/50, Loss: 0.3650
Epoch 24/50, Loss: 0.3370
Epoch 25/50, Loss: 0.5511
Epoch 26/50, Loss: 0.3249
Epoch 27/50, Loss: 0.1773
Epoch 28/50, Loss: 0.3519
Epoch 29/50, Loss: 0.4769
Epoch 30/50, Loss: 0.3361
Epoch 31/50, Loss: 0.2924
Epoch 32/50, Loss: 0.1658
Epoch 33/50, Loss: 0.1995
Epoch 34/50, Loss: 0.2678
Epoch 35/50, Loss: 0.3836
Epoch 36/50, Loss: 0.2170
Epoch 37/50, Loss: 0.2131
Epoch 38/50, Loss: 0.4101
Epoch 39/50, Loss: 0.5496
Epoch 40/50, Loss: 0.1825
Epoch 41/50, Loss: 0.5015
Epoch 42/50, Loss: 0.4948
Epoch 43/50, Loss: 0.2533
```

```
Epoch 44/50, Loss: 0.4823
Epoch 45/50, Loss: 0.4412
Epoch 46/50, Loss: 0.2388
Epoch 47/50, Loss: 0.1991
Epoch 48/50, Loss: 0.3819
Epoch 49/50, Loss: 0.4203
Epoch 50/50, Loss: 0.2983
Accuracy on test set: 77.535%
```

By comparing accuracy values obtained by GRU, LSTM, and simple RNN it can be concluded that for the given embeddings GRU & LSTM perform slightly better than RNN

References:

https://www.kaggle.com/code/mishra1993/pytorch-multi-layer-perceptron-mnist/notebook (https://www.kaggle.com/code/mishra1993/pytorch-multi-layer-perceptron-mnist/notebook)

In [ ]: