```
import pandas as pd
import numpy as np
import nltk

from nltk.tokenize import word_tokenize

import re
from bs4 import BeautifulSoup
import contractions


from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split

from sklearn.linear_model import Perceptron
from sklearn.svm import LinearSVC
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import precision_recall_fscore_support, accuracy_score

import gensim
import gensim.downloader as api

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


import warnings
warnings.filterwarnings("ignore")


#! pip install bs4 # in case you don't have it installed
#! pip install contractions

#! pip install gensim
#gensim version 4.3.0
#!pip install --upgrade gensim

#nltk.download('punkt')
#nltk.download('wordnet')
#nltk.download('stopwords')
#nltk.download('omw-1.4')
# Dataset: https://s3.amazonaws.com/amazon-reviews-pds/tsv/amazon_reviews_us_Beauty_v1_00.tsv.gz
```

1. Dataset Generation

```
data = pd.read_csv("amazon_reviews_us_Beauty_v1_00.tsv", sep = '\t', on_bad_lines='skip')
data.head()
```

	marketplace	customer_id	review_id	product_id	product_parent	product_title	product_cat
0	US	1797882	R3I2DHQBR577SS	B001ANOOOE	2102612	The Naked Bee Vitmin C Moisturizing Sunscreen ...	
1	US	18381298	R1QNE9NQFJC2Y4	B0016J22EQ	106393691	Alba Botanica Sunless Tanning Lotion, 4 Ounce	
2	US	19242472	R3LIDG2Q4LJBAO	B00HU6UQAG	375449471	Elysee Infusion Skin Therapy Elixir, 2oz.	
3	US	19551372	R3KSZHPAEVPEAL	B002HWS7RM	255651889	Diane D722 Color, Perm And Conditioner Process...	
						Biore UV Aqua Rich Waterv	

```
data = data[["review_body", "star_rating"]]
data.dropna(inplace = True)
data = data.astype({'star_rating': 'int'})
```

```
# Create a new column 'class' based on the 'star_rating' column
data['class'] = data['star_rating'].apply(lambda x: 1 if x in [1, 2] else 2 if x == 3 else 3)


data_class_1 = data[data['class'] == 1].sample(n=20000, random_state=1)
data_class_2 = data[data['class'] == 2].sample(n=20000, random_state=1)
data_class_3 = data[data['class'] == 3].sample(n=20000, random_state=1)

# Concatenate the resulting dataframes to create a balanced dataset
data = pd.concat([data_class_1, data_class_2, data_class_3])
data['class'].value_counts()
```

```
1    20000
2    20000
3    20000
Name: class, dtype: int64
```

```
# print average length of reviews before cleaning
data['review_length'] = data['review_body'].str.len()
review_len_before_cleaning = data['review_length'].mean()
```

```
# Convert all reviews to lowercase
data['review_body'] = data['review_body'].str.lower()

# Remove HTML and URLs from the reviews
data['review_body'] = data['review_body'].apply(lambda x: re.sub(r'(<.*?>|https?://\S+)', '', x))

# remove non-alphabetical characters
data['review_body'] = data['review_body'].apply(lambda x: re.sub('[^a-zA-Z]', ' ', x))

# remove extra spaces
data['review_body'] = data['review_body'].str.strip()

# Perform contractions on the reviews
data['review_body'] = data['review_body'].apply(lambda x: contractions.fix(x))


# Print average length of reviews before and after cleaning
review_lengths = data['review_body'].str.len()
review_len_after_cleaning = review_lengths.mean()
print("Average review length before and after cleaning:", review_len_before_cleaning,",", review_len_afte
```

```
Average review length before and after cleaning: 268.995 , 265.76646666666664
```

```
# remove stopwords
stopwords_list = stopwords.words('english')
data['review_body'] = data['review_body'].apply(lambda x: ' '.join([word for word in x.split() if word no


lemmatizer = WordNetLemmatizer()
data['review_body'] = data['review_body'].apply(lambda x: ' '.join([lemmatizer.lemmatize(word) for word i


# Print average length of reviews before and after preprocessing
review_lengths = data['review_body'].str.len()
review_len_after_preprocessing = review_lengths.mean()
print("Average review length before and after preprocessing:", review_len_after_cleaning, ",", review_len
```

```
Average review length before and after preprocessing: 265.76646666666664 , 155.29416666666665
```

```
# split the dataset into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(data['review_body'], data['class'], stratify= data['c


training_dataset = data.sample(frac = 0.8, random_state=65)
testing_dataset = data.drop(training_dataset.index)
training_dataset = data.reset_index(drop=True)
testing_dataset = data.reset_index(drop=True)
```

Word Embedding

2. a: Load the pretrained “word2vec-google-news-300” Word2Vec model and check semantic similarities of the generated vectors using three examples of your own

```
#loading the pretrained word2vec model
#wv = api.load('word2vec-google-news-300')
```

```
#pretrained = api.load('word2vec-google-news-300')
#pretrained.save('word2vec-google-news.kv')
```

```
[==================================================] 100.0% 1662.8/1662.8MB downloaded
```

```
pretrained = gensim.models.KeyedVectors.load('word2vec-google-news.kv')
```

> checking semantic similarities between vectors

```
similarity = pretrained.most_similar(positive=['excellent','outstanding'], topn=1)
print(similarity)
```

```
[('oustanding', 0.750198483467102)]
```

```
similarity_1a = pretrained.most_similar(positive=['cat','dog'], topn=1)
print(similarity_1a)
```

```
[('puppy', 0.8089798092842102)]
```

```
similarity_2a = pretrained.most_similar(positive=['happy','sad'], topn=1)
print(similarity_2a)
```

```
[('glad', 0.7112970352172852)]
```

```
similarity_3a = pretrained.most_similar(positive=['laptop','computer'], topn=1)
print(similarity_3a)
```

```
[('laptop_computer', 0.7891943454742432)]
```

```
print(pretrained.most_similar('laptop'))
```

```
[('laptops', 0.8053741455078125), ('laptop_computer', 0.7848465442657471), ('notebook', 0.6785782575
```

```
print(pretrained.most_similar('computer'))
```

```
[('computers', 0.7979379892349243), ('laptop', 0.6640493273735046), ('laptop_computer', 0.6548868417
```

2. b: Train a Word2Vec model using your own dataset and heck the semantic similarities for the same two examples in part (a)

```
from gensim.models import Word2Vec

model = Word2Vec(sentences= data['review_body'].apply(lambda x: nltk.word_tokenize(x)), vector_size=300,
#model.save("word2vec.model")
```

```
similarity_1b = model.wv.most_similar(positive=['cat','dog'], topn=1)
print(similarity_1b)
```

```
[('gagging', 0.7637972831726074)]
```

```
similarity_2b = model.wv.most_similar(positive=['happy','sad'], topn=1)
print(similarity_2b)
```

```
[('glad', 0.6962702870368958)]
```

```
similarity_3b = model.wv.most_similar(positive=['laptop','computer'], topn=1)
print(similarity_3b)
```

```
[('booty', 0.8579108119010925)]
```

```
print(model.wv.most_similar('laptop'))
```

```
[('zippered', 0.8697407841682434), ('organized', 0.8600019216537476), ('handbag', 0.847605705261230
```

```
print(model.wv.most_similar('computer'))
```

```
[('walked', 0.8297677636146545), ('conversation', 0.7957174181938171), ('prepaid', 0.79370313882827
```

What do you conclude from comparing vectors generated by yourself and the pretrained model? Which of the Word2Vec models seems to encode semantic similarities between words better?

> Based on comparison of the semantic similarities between vectors for the 2 models, the word2vec model created using amazon reviews data gives results with a higher similarity score. But, the results from pre-trained google news word2vec model are more logical, so overall we can't say that one model is better than the other.

3. Simple models

Pre-trained word2vec

Here, we use the pre-trained word2vec model to extract features from the tokenized text data by filtering out non-existent words, retrieving their word vectors, and calculating the average vector for each sentence

```
def get_word2vec_features(X, Y, word2vec_model):
    wv_X = []
    wv_Y = []
    for sentence, label in zip(X, Y):
        tokens = word_tokenize(sentence)
        # Get only the tokens that exist in the word2vec model
        filtered_tokens = [token for token in tokens if token in word2vec_model.key_to_index]
        #filtered_tokens = [token for token in tokens if token in word2vec_model.vocab]
        # If no tokens are left, skip this sentence
        if not filtered_tokens:
            continue
        # Get the word vectors for the filtered tokens
        vectors = [word2vec_model.get_vector(token) for token in filtered_tokens]
        # Calculate the average vector for the sentence
        average_vector = sum(vectors) / len(vectors)
        wv_X.append(average_vector)
        wv_Y.append(label)
    return wv_X, wv_Y
```

```
X_train_wv, Y_train_wv = get_word2vec_features(X_train, y_train, pretrained)
```

```
X_test_wv, Y_test_wv = get_word2vec_features(X_test, y_test, pretrained)
```

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

# Train Perceptron and test data
perceptron_wv = Perceptron(max_iter = 75, eta0 = 0.005, random_state=65)
perceptron_wv.fit(X_train_wv, Y_train_wv)
y_pred_perceptron_wv = perceptron_wv.predict(X_test_wv)

# Compute the precision, recall, and f1-score per class
precision, recall, f1, _ = precision_recall_fscore_support(Y_test_wv, y_pred_perceptron_wv, average=None)

# Compute the average precision, recall, and f1-score
average_precision = precision.mean()
average_recall = recall.mean()
average_f1 = f1.mean()

# Compute the accuracy
acc = accuracy_score(Y_test_wv, y_pred_perceptron_wv)

#print("Precision for perceptron model on pre-trained word2vec:" ,average_precision*100)
#print("Recall for perceptron model on pre-trained word2vec:" ,average_recall*100)
#print("F1 Score for perceptron model on pre-trained word2vec:", average_f1*100)
print("Test Accuracy for perceptron model on pre-trained word2vec:", acc*100)
```

```
Test Accuracy for perceptron model on pre-trained word2vec: 47.44842562432139
```

```
# Train SVC and test data
svc_wv = LinearSVC(max_iter = 1000, random_state=65)
svc_wv.fit(X_train_wv, Y_train_wv)
y_pred_svc_wv = svc_wv.predict(X_test_wv)

# Compute the precision, recall, and f1-score per class
precision, recall, f1, _ = precision_recall_fscore_support(Y_test_wv, y_pred_svc_wv, average=None)

# Compute the average precision, recall, and f1-score
average_precision = precision.mean()
average_recall = recall.mean()
average_f1 = f1.mean()

# Compute the accuracy
acc = accuracy_score(Y_test_wv, y_pred_svc_wv)

#print("Precision for SVC model on pre-trained word2vec:" ,average_precision*100)
#print("Recall for SVC model on pre-trained word2vec:" ,average_recall*100)
#print("F1 Score for SVC model on pre-trained word2vec:", average_f1*100)
print("Test Accuracy for SVC model on pre-trained word2vec:", acc*100)
```

```
Test Accuracy for SVC model on pre-trained word2vec: 62.45719535621815
```

Tf-idf

```
tfidf = TfidfVectorizer()
X_train_tf = tfidf.fit_transform(X_train)
X_test_tf = tfidf.transform(X_test)
```

```
# Train Perceptron and test data
perceptron_tf = Perceptron(max_iter = 75, eta0 = 0.005, random_state=65)
perceptron_tf.fit(X_train_tf, y_train)
y_pred_perceptron_tf = perceptron_tf.predict(X_test_tf)

# Compute the precision, recall, and f1-score per class
precision, recall, f1, _ = precision_recall_fscore_support(y_test, y_pred_perceptron_tf, average=None)

# Compute the average precision, recall, and f1-score
average_precision = precision.mean()
average_recall = recall.mean()
average_f1 = f1.mean()

# Compute the accuracy
acc = accuracy_score(y_test, y_pred_perceptron_tf)

#print("Precision for perceptron model with tf-idf :" ,average_precision*100)
#print("Recall for perceptron model with tf-idf:" ,average_recall*100)
#print("F1 Score for perceptron model with tf-idf:", average_f1*100)
print("Test Accuracy for perceptron model with tf-idf:", acc*100)
```

```
Test Accuracy for perceptron model with tf-idf: 58.808333333333333
```

```
# Train SVC and test data
svc_tf = LinearSVC(max_iter = 1000, random_state=65)
svc_tf.fit(X_train_tf, y_train)
y_pred_svc_tf = svc_tf.predict(X_test_tf)

# Compute the precision, recall, and f1-score per class
precision, recall, f1, _ = precision_recall_fscore_support(y_test, y_pred_svc_tf, average=None)

# Compute the average precision, recall, and f1-score
average_precision = precision.mean()
average_recall = recall.mean()
average_f1 = f1.mean()

# Compute the accuracy
acc = accuracy_score(y_test, y_pred_svc_tf)

#print("Precision for SVC model with tf-idf :" ,average_precision*100)
#print("Recall for SVC model with tf-idf:" ,average_recall*100)
#print("F1 Score for SVC model with tf-idf:", average_f1*100)
print("Test Accuracy for SVC model with tf-idf:", acc*100)
```

```
Test Accuracy for SVC model with tf-idf: 66.108333333333333
```

Task 3: Accuracy Summary

1. Using pre-trained word2vec:
   a. Perceptron accuracy = 47.448%
   b. SVM accuracy = 62.457%
2. Using tf-idf:
   a. Perceptron accuracy = 58.808%
   b. SVM accuracy = 66.108%

What do you conclude from comparing performances for the models trained using the two different feature types

> Based on comparison of accuracy between the models using pre-trained features and the tf-idf features, tf-idf performs better. So we can say that tf-idf is a more robust input feature

4. FNN

Using the Word2Vec features, train a feedforward multilayer perceptron network for classification

Here we are defining a multilayer perceptron (MLP) using PyTorch's nn.Sequential module. The MLP has three layers: an input layer with 300 nodes, a hidden layer with 100 nodes, and an output layer with 3 nodes.

The activation function used in the hidden layers is ReLU (rectified linear unit)

```
# Feedforward MLP model network with two hidden layers, each with 100 and 10 nodes, respectively
mlp = nn.Sequential(
    # Input layer to hidden layer
    nn.Linear(300, 100),
    # ReLU activation function
    nn. ReLU(),
    # Hidden layer to output layer
    nn.Linear(100, 10),
    nn. ReLU(),
    # Output layer
    nn.Linear(10, 3))
print(mlp)
```

```
Sequential(
  (0): Linear(in_features=300, out_features=100, bias=True)
  (1): ReLU()
  (2): Linear(in_features=100, out_features=10, bias=True)
  (3): ReLU()
  (4): Linear(in_features=10, out_features=3, bias=True)
)
```

```
# Loss function -> CrossEntropyLoss
loss_fn = nn.CrossEntropyLoss()
# Select Optimizer = Adam
optimizer = optim.Adam(mlp.parameters(), lr=0.001)
```

(a) Use the average Word2Vec vectors and train the neural network

```
#The train_model function is responsible for training a neural network model on a given dataset using the
def train_model(num_epochs):
    # Set the model to training mode
    mlp.train()

    # Iterate over the training data in mini-batches
    for inputs, labels in train_loader:
        # Reset the gradients to zero
        optimizer.zero_grad()
        # Forward pass: compute the predicted outputs of the model
        outputs = mlp(inputs)
        # Compute the loss between the predicted outputs and the true labels
        loss = loss_fn(outputs, labels)
        # Backward pass: compute the gradients of the loss with respect to the model parameters
```

```
        loss.backward()
        # Update the model parameters using the computed gradients
        optimizer.step()
    # Print the current epoch number and the training loss every 10 epochs
    #if epoch % 10 == 0:
    print(f"Epoch {epoch:4d} Loss: {loss.item():.6f}")

def test_model():
    #set the MLP model to evaluation mode
    mlp.eval()
    #variable to count the number of correct predictions made by the model on the test dataset
    correct = 0
    with torch.no_grad():
        for inputs, labels in test_loader:
            #Feed the input data into the MLP model to get the predicted outputs
            outputs = mlp(inputs)
            #Find the predicted labels for each input sample
            predicted = torch.argmax(outputs, dim=1)
            #Compare the predicted labels with the ground truth labels to count the number of correct pre
            correct += (predicted == labels).sum().item()
    #Calculate the overall accuracy on the test dataset
    accuracy = 100. * correct / len(test_loader.dataset)
    print('Accuracy on test set: {:.2f}%'.format(accuracy))


# Set data
X_train = torch.Tensor(X_train_wv)
X_test = torch.Tensor(X_test_wv)

# changing classes to 0,1,2 by reducing each class number by 1
# Subtract 1 from each element in Y_train_wv
Y_train_wv = [y - 1 for y in Y_train_wv]
# Subtract 1 from each element in Y_test_wv
Y_test_wv = [y - 1 for y in Y_test_wv]

y_train = torch.LongTensor(Y_train_wv)
y_test = torch.LongTensor(Y_test_wv)

train_data = TensorDataset(X_train, y_train)
test_data = TensorDataset(X_test, y_test)

#create data loaders to load batches of input features and output labels during training and testing
train_loader = DataLoader(train_data, batch_size=32, shuffle=True)
test_loader = DataLoader(test_data, batch_size=32, shuffle=False)


# Train with epoch = 100, and test data
for epoch in range(100):
    train_model(epoch)
test_model()
```

```
Epoch   56 Loss: 0.073553
Epoch   57 Loss: 0.561730
Epoch   58 Loss: 0.029989
Epoch   59 Loss: 0.177583
Epoch   60 Loss: 0.054957
Epoch   61 Loss: 0.125054
Epoch   62 Loss: 0.231713
Epoch   63 Loss: 0.087247
Epoch   64 Loss: 0.609809
Epoch   65 Loss: 0.152007
Epoch   66 Loss: 0.096239
Epoch   67 Loss: 0.123272
Epoch   68 Loss: 0.087379
Epoch   69 Loss: 0.048163
Epoch   70 Loss: 0.025534
Epoch   71 Loss: 0.358235
Epoch   72 Loss: 0.399552
Epoch   73 Loss: 0.430765
Epoch   74 Loss: 0.065172
Epoch   75 Loss: 0.268065
Epoch   76 Loss: 0.030238
Epoch   77 Loss: 0.056783
Epoch   78 Loss: 0.434328
Epoch   79 Loss: 0.654119
Epoch   80 Loss: 0.057290
Epoch   81 Loss: 0.106597
Epoch   82 Loss: 0.046246
Epoch   83 Loss: 0.435028
Epoch   84 Loss: 0.638065
Epoch   85 Loss: 0.123029
Epoch   86 Loss: 0.010198
Epoch   87 Loss: 0.107779
Epoch   88 Loss: 0.049329
Epoch   89 Loss: 0.269081
Epoch   90 Loss: 0.035462
Epoch   91 Loss: 0.241868
Epoch   92 Loss: 0.088481
Epoch   93 Loss: 0.001846
Epoch   94 Loss: 0.044079
Epoch   95 Loss: 0.019699
Epoch   96 Loss: 0.169109
Epoch   97 Loss: 0.041571
Epoch   98 Loss: 0.100528
Epoch   99 Loss: 0.224995
Accuracy on test set: 56.94%
```

(b) concatenate the first 10 Word2Vec vectors for each review as the input feature

Here, we are concatenating the first 10 word2vec vectors using the concatenate_word2vec function. This function tokenizes the sentence using nltk.word_tokenize and filters out tokens that are not present in the word2vec_model vocabulary. It then retrieves the word embeddings for the remaining tokens using the word2vec_model and concatenates the first 10 word embeddings into a single vector and pads the concatenated vector with zeros to ensure that it has a fixed length of 3000.

```
def concatenate_word2vec(X, Y, word2vec_model):
    wv_X_c = []
    wv_Y_c = []
    for sentence, label in zip(X, Y):
        tokens = nltk.word_tokenize(sentence)
        #filtered_tokens = [token for token in tokens if token in word2vec_model.key_to_index]
        filtered_tokens = [token for token in tokens if token in word2vec_model.vocab]
        if len(filtered_tokens) > 0:
            embeddings = [word2vec_model[token] for token in filtered_tokens[:10]]
```

```
            concatenated = np.concatenate(embeddings)
            padded = np.pad(concatenated, (0, 3000 - len(concatenated)), 'constant', constant_values=0)
            wv_X_c.append(padded)
            wv_Y_c.append(label)
    return wv_X_c, wv_Y_c

temp_X, temp_Y = concatenate_word2vec(data['review_body'], data['class'], pretrained)
X_train_wv_c, X_test_wv_c, Y_train_wv_c, Y_test_wv_c = train_test_split(temp_X, temp_Y, test_size=0.2)


#The train_model function is responsible for training a neural network model on a given dataset using the
def train_model(epochs):
    # Set the model to training mode
    mlp.train()

    # Iterate over the training data in mini-batches
    for inputs, labels in train_loader:
        # Reset the gradients to zero
        optimizer.zero_grad()
        # Forward pass: compute the predicted outputs of the model
        outputs = mlp(inputs)
        # Compute the loss between the predicted outputs and the true labels
        loss = loss_fn(outputs, labels)
        # Backward pass: compute the gradients of the loss with respect to the model parameters
        loss.backward()
        # Update the model parameters using the computed gradients
        optimizer.step()
    # Print the current epoch number and the training loss every 10 epochs
    if epoch % 10 == 0:
      print(f"Loss: {loss.item():.6f}")


def test_model():
    #set the MLP model to evaluation mode
    mlp.eval()
    #variable to count the number of correct predictions made by the model on the test dataset
    correct = 0
    with torch.no_grad():
        for inputs, labels in test_loader:
            #Feed the input data into the MLP model to get the predicted outputs
            outputs = mlp(inputs)
            #Find the predicted labels for each input sample
            predicted = torch.argmax(outputs, dim=1)
            #Compare the predicted labels with the ground truth labels to count the number of correct pre
            correct += (predicted == labels).sum().item()
    #Calculate the overall accuracy on the test dataset
    accuracy = 100. * correct / len(test_loader.dataset)
    print('Accuracy on test set: {:.0f}%'.format(accuracy))


# Feedforward MLP model
mlp = nn.Sequential(
    # Input layer to hidden layer
    nn.Linear(300, 100),
    # ReLU activation function
    nn. ReLU(),
    # Hidden layer to output layer
    nn.Linear(100, 10),
    nn. ReLU(),
    # Output layer
    nn.Linear(10, 3))
print(mlp)
```

```
Sequential(
  (0): Linear(in_features=300, out_features=100, bias=True)
  (1): ReLU()
  (2): Linear(in_features=100, out_features=10, bias=True)
  (3): ReLU()
  (4): Linear(in_features=10, out_features=3, bias=True)
)
```

```
# Loss function -> CrossEntropyLoss
loss_fn = nn.CrossEntropyLoss()
# Select Optimizer = Adam
optimizer = optim.Adam(mlp.parameters(), lr=0.001)


# Set data
X_train = torch.Tensor(X_train_wv_c)
X_test = torch.Tensor(X_test_wv_c)

# changing classes to 0,1,2 by reducing each class number by 1
# Subtract 1 from each element in Y_train_wv
Y_train_wv_c = [y - 1 for y in Y_train_wv_c]
# Subtract 1 from each element in Y_test_wv
Y_test_wv_c = [y - 1 for y in Y_test_wv_c]

y_train = torch.LongTensor(Y_train_wv_c)
y_test = torch.LongTensor(Y_test_wv_c)

train_data = TensorDataset(X_train, y_train)
test_data = TensorDataset(X_test, y_test)

#create data loaders to load batches of input features and output labels during training and testing
train_loader = DataLoader(train_data, batch_size=32, shuffle=True)
test_loader = DataLoader(test_data, batch_size=32, shuffle=False)


# Train with epoch = 100, and test data
for epoch in range(50):
    train_model(epoch)
test_model()
```

```
Loss: 0.577560
Loss: 0.346125
Loss: 0.493343
Loss: 0.513139
Loss: 0.253340
Accuracy on test set: 60%
```

Task 4: Accuracy Summary

4.a Accuracy = 56.94%

4.b Accuracy = 60%

What do you conclude by comparing accuracy values you obtain with those obtained in the "'Simple Models" section?

> According to the accuracy scores for the simple models and the Feedforward Neural Networks, SVM using tf-idf features has the best performance

5. RNN

Using the Word2Vec features, train a recurrent neural network (RNN) for classification

The pad_reviews function pads or truncates a list of reviews to a specified maximum length. If a review is longer than the max_length, the function truncates it to max_length by slicing the first max_length elements of the review. If a review is shorter than max_length, the function pads it with zeros by concatenating the review with a list of max_length - len(review) zeros.

```
def pad_reviews(reviews, max_length):
    padded_reviews = []
    for review in reviews:
        if len(review) > max_length:
            # Truncate longer reviews
            padded_review = review[:max_length]
        else:
            # Pad shorter reviews with zeros
            padded_review = review + [0] * (max_length - len(review))
        padded_reviews.append(padded_review)
    return padded_reviews
```

Here, I have writeen a function that converts a list of tokenized reviews into a list of integer reviews. For each review, the function creates a new list called int_review. It then iterates over each word in the input review and checks whether that word is in the pre-trained model. If the word is in the model, the function retrieves the index number of that word in the model using the key_to_index attribute of the pretrained model. If the word is not in the model, the function assigns the index number 0 to that word.

```
# Change the tokenized reviews to int type
def convert_reviews_to_int(reviews):
  int_reviews = []
  for review in reviews:
  # if specific word is in my word2vec model -> use index number. If not, put 0 instead of the words' ind
    int_reviews.append([pretrained.key_to_index[word] if word in pretrained.key_to_index else 0 for word
  return int_reviews
```

(a) Train a simple RNN for sentiment analysis

> Here, we are training a simple RNN for sentiment analysis using PyTorch. The RNN implementation has the following layers:

1. An embedding layer, which maps each input index to a dense vector of embedding_dim dimensions.
2. A layer with hidden_size hidden units. The batch_first=True argument specifies that the input tensor has dimensions (batch_size, sequence_length, embedding_dim).
3. A linear layer (fully connected layer) that maps the output of the previous layer to the output classes.
   We use the CrossEntropyLoss loss function and the Adam optimizer with a learning rate of 0.001.

```
class RNN(nn.Module):
  def __init__(self, input_dim, hidden_size, num_classes):
```

```
      super(RNN, self).__init__()
      self.hidden_size = hidden_size
      self.embedding = nn.Embedding(input_dim, hidden_size)
      self.rnn = nn.RNN(hidden_size, hidden_size, batch_first=True, nonlinearity='relu')
      self.fc = nn.Linear(hidden_size, num_classes)
    def forward(self, x):
      embedded = self.embedding(x)
      out, _ = self.rnn(embedded)
      out = self.fc(out)
      return out

  def train(epoch, batch_size):
    model.train()
    epoch_loss = 0
    # Train model with mini batch
    for inputs, labels in train_loader:
          # Reset the gradients to zero
          optimizer.zero_grad()
          # Forward pass: compute the predicted outputs of the model
          outputs = model(inputs)
          # Compute the loss between the predicted outputs and the true labels
          loss = loss_fn(outputs, labels.reshape(1,batch_size).t())
          # Backward pass: compute the gradients of the loss with respect to the model parameters
          loss.backward()
          # Update the model parameters using the computed gradients
          optimizer.step()
          epoch_loss += loss.item()
    print('Loss: {:.6f}'.format(loss.item()))


  def test(model, data_loader):
    #set the model to evaluation mode
    model.eval()
    #variable to count the number of correct predictions made by the model on the test dataset
    correct = 0
      #Create minibatch
    with torch.no_grad():
      for data, labels in data_loader:
        #Feed the input data into the model to get the predicted outputs
        outputs = model(data)
              #Find the predicted labels for each input sample
        _, predicted = torch.max(outputs.data, 1)
        #Compare the predicted labels with the ground truth labels to count the number of correct predictio
        correct += predicted.eq(labels.data.view_as(predicted)).sum()
      #Print accuracy
      data_num = len(data_loader.dataset)
      #print('\nAccuracy with test data: {}/{} ({:.0f}%)\n'.format(correct,data_num, 100. * correct / data_
      print('\nAccuracy with test data: {:.2f}%\n'.format(100. * correct / data_num))


  #Change words in train data to number values using google-word2vec-news model
  x_train = convert_reviews_to_int(word_tokenize(sentence) for sentence in X_train)
  #padding shorter reviews with a null value (0)
  x_train = np.array(pad_reviews(x_train, 20))

  #Change words in test data to number values using google-word2vec-news model
  x_test = convert_reviews_to_int(word_tokenize(sentence) for sentence in X_test)
  #padding shorter reviews with a null value (0)
  x_test = np.array(pad_reviews(x_test, 20))


  from torch.utils.data import TensorDataset, DataLoader
  X_train = torch.LongTensor(x_train)
  X_test = torch.LongTensor(x_test)
```

```
# changing classes to 0,1,2 by reducing each class number by 1
y_train = [y - 1 for y in y_train]
y_test = [y - 1 for y in y_test]
Y_train = torch.LongTensor(y_train)
Y_test = torch.LongTensor(y_test)

# Make a dataset and dataloader
train_data = TensorDataset(X_train, Y_train)
test_data = TensorDataset(X_test, Y_test)

#create data loaders to load batches of input features and output labels during training and testing
train_loader = DataLoader(train_data, batch_size=64, shuffle=True)
test_loader = DataLoader(test_data, batch_size=64, shuffle=False)


input_dim = len(pretrained)+1
hidden_dim = 20
output_dim = 1
model = RNN(input_dim, hidden_dim, output_dim)

# Loss function -> CrossEntropyLoss
loss_fn = nn.CrossEntropyLoss()
# Select Optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)


for epoch in range(5):
  train(epoch, 64)
test(model, test_loader)
```

```
Loss: 0.989170
Loss: 0.963754
Loss: 0.982743
Loss: 0.914864
Loss: 0.858872

Accuracy with test data: 52.08%
```

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

> Here, we are implementing a GRU (Gated Recurrent Unit) neural network using PyTorch. It uses an embedding layer followed by a single GRU layer with 20 hidden units, and a linear layer to map the output to the classes. We use the CrossEntropyLoss loss function and the Adam optimizer with a learning rate of 0.001.

```
# Define the GRU model
class GRU(nn.Module):
    def __init__(self, input_dim, hidden_size, num_classes):
        super(GRU, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(input_dim, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)
    def forward(self, x):
        embedded = self.embedding(x)
        out, _ = self.gru(embedded)
```

```
        out = self.fc(out)
```

```
input_dim = len(pretrained)+1
hidden_dim = 20
output_dim = 1
model = GRU(input_dim, hidden_dim, output_dim)

# Loss function used: CrossEntropyLoss
loss_fn = nn.CrossEntropyLoss()
# Adam Optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)


for epoch in range(5):
    train(epoch, 64)
test(model,test_loader)
```

```
Loss: 1.122906
Loss: 1.068928
Loss: 0.985442
Loss: 1.010378
Loss: 0.797255

Accuracy with test data: 51.67%
```

(c) Repeat part (a) by considering an LSTM unit cell

> The LSTM model uses an embedding layer followed by a single LSTM layer with 20 hidden units, and a linear layer to map the output to the classes. We use the CrossEntropyLoss loss function and the Adam optimizer with a learning rate of 0.001.

```
# Define the LSTM model
class LSTM(nn.Module):
    def __init__(self, input_dim, hidden_size, num_classes):
        super(LSTM, self).__init__()
        self.hidden_size = hidden_size
        self.embedding = nn.Embedding(input_dim, hidden_size)
        self.lstm = nn.LSTM(hidden_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        embedded = self.embedding(x)
        out, _ = self.lstm(embedded)
        #out = self.fc(out[:, -1, :])
        out = self.fc(out)
        return out


input_dim = len(pretrained)+1
hidden_dim = 20
output_dim = 1
model = LSTM(input_dim, hidden_dim, output_dim)

# Loss function used: CrossEntropyLoss
loss_fn = nn.CrossEntropyLoss()
# Adam Optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
for epoch in range(5):
    train(epoch, 64)
test(model, test_loader)
```

```
Loss: 1.151444
Loss: 1.025713
Loss: 1.108427
Loss: 0.950206
Loss: 0.910374

Accuracy with test data: 52.12%
```

Task 5: Accuracy Summary

5a: RNN accuracy: 52.12%

5b. GRU accuracy: 51.67%

5c: LSTM accuracy: 52.08%

What do you conclude by comparing accuracy values you obtain by GRU, LSTM, and simple RNN

> Based on the accuracy values, LSTM performs the best but the simple RNN also comes close with very little difference in performance

```
#References

#https://radimrehurek.com/gensim/models/word2vec.html
#https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html
#https://arxiv.org/abs/1301.3781
#https://www.kaggle.com/c/word2vec-nlp-tutorial/discussion/27022
#https://towardsdatascience.com/word2vec-for-phrases-learning-embeddings-for-more-than-one-word-727b6cf72
#https://www.kaggle.com/mishra1993/pytorch-multi-layer-perceptron-mnist
#https://pytorch.org/docs/stable/generated/torch.nn.RNN.html
#https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html
```