Assignment 1 : Setting up NLP Pipeline and Text Classification (10 Marks)

Due: March 7, 2022

Welcome to Assignment 1 of our course on Natural Language Processing! In this assignment you will implement different text-preprocessing techniques commonly used for NLP tasks as well as implement a standard text-classification algorithm for recognizing sentiments of movie reviews.

We assume that you are familiar with `python` programming language, and its libraries like `numpy` and `pandas`. We will also make use of other libraries like `nltk` and `pytorch` in the assignment. Familiarity with these libraries is not assumed so we will provide short tutorials on their usage within the assignment.

```
# try:
#     from google.colab import drive
#     drive.mount('/content/gdrive')
#     data_dir = "gdrive/MyDrive/plaksha/NLP/Assignment1/data/SST-2"
# except:
#     data_dir = "/datadrive/t-kabir/work/repos/PlakshaNLP/Assignment1/data/SST-2"
```

```
from google.colab import drive
drive.mount('/content/gdrive')
data_dir = "gdrive/MyDrive/plaksha/NLP/Assignment1/data/SST-2"
```

```
Mounted at /content/gdrive
```

```
# Install required libraries
!pip install numpy
!pip install pandas
!pip install nltk
!pip install torch
!pip install tqdm
!pip install matplotlib
!pip install seaborn
```

```
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packages (1.21.5)
Requirement already satisfied: pandas in /usr/local/lib/python3.7/dist-packages (1.3.5)
Requirement already satisfied: pytz>=2017.3 in /usr/local/lib/python3.7/dist-packages (from panda
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/local/lib/python3.7/dist-packages (
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.7/dist-packages (from pand
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packages (from python-da
Requirement already satisfied: nltk in /usr/local/lib/python3.7/dist-packages (3.2.5)
Requirement already satisfied: six in /usr/local/lib/python3.7/dist-packages (from nltk) (1.15.0)
Requirement already satisfied: torch in /usr/local/lib/python3.7/dist-packages (1.10.0+cu111)
Requirement already satisfied: typing-extensions in /usr/local/lib/python3.7/dist-packages (from
Requirement already satisfied: tqdm in /usr/local/lib/python3.7/dist-packages (4.63.0)
Requirement already satisfied: matplotlib in /usr/local/lib/python3.7/dist-packages (3.2.2)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/dist-packages (from
Requirement already satisfied: numpy>=1.11 in /usr/local/lib/python3.7/dist-packages (from matplo
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.7/dist-packages (fr
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/local/lib/python3
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-packages (from matpl
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packages (from python-da
Requirement already satisfied: seaborn in /usr/local/lib/python3.7/dist-packages (0.11.2)
Requirement already satisfied: numpy>=1.15 in /usr/local/lib/python3.7/dist-packages (from seabor
Requirement already satisfied: pandas>=0.23 in /usr/local/lib/python3.7/dist-packages (from seabo
Requirement already satisfied: matplotlib>=2.2 in /usr/local/lib/python3.7/dist-packages (from se
Requirement already satisfied: scipy>=1.0 in /usr/local/lib/python3.7/dist-packages (from seaborn
Requirement already satisfied: python-dateutil>=2.1 in /usr/local/lib/python3.7/dist-packages (fr
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.7/dist-packages (from
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in /usr/local/lib/python3
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.7/dist-packages (from matpl
Requirement already satisfied: pytz>=2017.3 in /usr/local/lib/python3.7/dist-packages (from panda
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.7/dist-packages (from python-da
# We start by importing libraries that we will be making use of in the assignment.
import string
import tqdm
import numpy as np
import pandas as pd
import torch
import matplotlib.pyplot as plt
import seaborn as sns
import nltk
nltk.download("punkt")
nltk.download('stopwords')
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
True
```

Stanford Sentiment Treebank Dataset

For the purposes of this assignment we will be working with the Stanford Sentiment Treebank dataset, which comprises of a list of movie reviews each tagged with the sentiment of the review. We will be considering the binary label version of the dataset commonly referred to as **SST-2**, meaning each review will have either of the two possible labels i.e. Positive or Negative.

The SST-2 dataset can be downloaded from here. The dataset folder will be containing three `.tsv` files, `train.tsv`, `dev.tsv` and `test.tsv` corresponding to the three splits of the data. Both `train.tsv` and `dev.tsv` have lines containing the reviews with the corresponding label (1 for positive sentiment and 0 for negative). Note that `test.tsv` only has the reviews and labels are missing, which is due to the fact that this split comes from a competition where the test predictions are to be submitted. For the purposes of this assignment we will focus on `train.tsv` and `dev.tsv` only, where the former will be used for training the text-classifiers and latter for evaluating them.

We start by loading the datasets into memory.

```
# We can use pandas to load the datasets
train_df = pd.read_csv(f"{data_dir}/train.tsv", sep = "\t")
test_df = pd.read_csv(f"{data_dir}/dev.tsv", sep = "\t")

print(f"Number of Training Examples: {len(train_df)}")
print(f"Number of Test Examples: {len(test_df)}")
```

```
Number of Training Examples: 67349
Number of Test Examples: 872
```

```
# View a sample of the dataset
train_df.head()
```

As can be seen from the sample of training dataset using `train_df.head()`, the dataframe contains columns `sentence` and `label` containing the review and sentiment label respectively.

As part of some preliminary data analysis below we visualize the distribution of the labels in the dataset.

```
label_counts = 100 * train_df["label"].value_counts(normalize=True).sort_index()
plt.bar(x = ["Negative Sentiment", "Positive Sentiment"], height = label_counts)
```

As can be seen from the plot we have roughly 45% training data points which have a negative sentiment and about 55% with positive sentiment.

Task 1: Preprocessing Pipeline for NLP (3 Marks)

You will start by implementing different text-preprocessing functions below. We have provided the definitions for the functions you are supposed to implement. After filling the code for the function, you can run the cell that follows to run test cases on your code.

Task 1.1: Word Tokenization (0.5 Marks)

Before we start preprocessing the text data and eventually training classification models, it is crucial to break the text into a set of constituents called tokens which can either be sentences, words, sub-words or characters. For the purposes of this assignment we will focus on Word Tokenization i.e. breaking a piece of text into a sequence of words.

There are different ways splitting a piece of text into a list of words. The simplest solution can be to split whenever a white-space character (i.e. `" "`) is encountered in the text i.e. if you have a string `"this is an example of tokenization"`, you iterate through it and whenever a white space is encountered you split the word to get: `["this", "is", "an", "example", "of", "tokenization"]`. Implement the `whitespace_word_tokenize` function below

```
def whitespace_word_tokenize(text):
  """
  Splits a python string containing some text to a sequence of words
  by splitting on whitespace.

  Parameters:
    - text (str): A Python string containing the text to be tokenized

  Returns:
    - words (list): A list contaning the words present in the text (in the same order)

  """
  words = None
  for word in range(len(text)):
    words = text.split(" ")
  # raise NotImplementedError()

  return words
```

```
def evaluate_list_test_cases(test_case_input,
                      test_case_func_output,
                      test_case_exp_output):

  print(f"Input: {test_case_input}")
  print(f"Function Output: {test_case_func_output}")
  print(f"Expected Output: {test_case_exp_output}")

  if test_case_func_output == test_case_exp_output:
    print("Test Case Passed :)")
    print("**********************************\n")
    return True
  else:
    print("Test Case Failed :(")
    print("**********************************\n")
    return False




print("Running Sample Test Cases")
print("Sample Test Case 1:")
test_case = "We all live in a Yellow Submarine"
test_case_answer = ['We', 'all', 'live', 'in', 'a', 'Yellow', 'Submarine']
test_case_student_answer = whitespace_word_tokenize(test_case)
assert evaluate_list_test_cases(test_case, test_case_student_answer, test_case_answer)

print("Sample Test Case 2:")
test_case = "We all live, in a Yellow Submarine."
test_case_answer = ['We', 'all', 'live,', 'in', 'a', 'Yellow', 'Submarine.']
test_case_student_answer = whitespace_word_tokenize(test_case)
assert evaluate_list_test_cases(test_case, test_case_student_answer, test_case_answer)
```

```
Running Sample Test Cases
Sample Test Case 1:
Input: We all live in a Yellow Submarine
Function Output: ['We', 'all', 'live', 'in', 'a', 'Yellow', 'Submarine']
Expected Output: ['We', 'all', 'live', 'in', 'a', 'Yellow', 'Submarine']
Test Case Passed :)
**********************************

Sample Test Case 2:
Input: We all live, in a Yellow Submarine.
Function Output: ['We', 'all', 'live,', 'in', 'a', 'Yellow', 'Submarine.']
Expected Output: ['We', 'all', 'live,', 'in', 'a', 'Yellow', 'Submarine.']
Test Case Passed :)
**********************************
```

As you can see from the outputs above the white space tokenizer does reasonably well in splitting a sentence into words. However, it is not perfect, as can be seen in the output of test case 2 this method fails to split words when punctuations are encountered which are retained as parts of the words like `"live,"` and `"Submarine."`.

One possible solution is to instead of splitting on the white-space also split when punctuations are encountered. This will partially solve the problem but there are certain other cases that still won't be handled properly by this, like we would want something like `"don't"` to be split into `["do", "n't"]` instead of `["don", "'", "t"]`.

Thankfully, nltk package provides the `word_tokenize` function that handles most of such cases inbuilt. Implement the `nltk_word_tokenize` function below which uses `word_tokenize` function from the nltk library to tokenize the text. Refer to the documentation here to understand the usage of `word_tokenize` function.

```
from nltk.tokenize import word_tokenize

def nltk_word_tokenize(text):
  """
  Splits a python string containing some text to a sequence of words
  by using `word_tokenize` function from nltk.
  Refer to https://www.nltk.org/api/nltk.tokenize.html#module-nltk.tokenize.treebank

  Parameters:
    - text (str): A Python string containing the text to be tokenized

  Returns:
    - words (list): A list contaning the words present in the text (in the same order)
  """

  words = None
  words = word_tokenize(text)
  # raise NotImplementedError()

  return words
```

```
print("Running Sample Test Cases")
print("Sample Test Case 1:")
test_case = "We all live in a Yellow Submarine"
test_case_answer = ['We', 'all', 'live', 'in', 'a', 'Yellow', 'Submarine']
test_case_student_answer = nltk_word_tokenize(test_case)
assert evaluate_list_test_cases(test_case, test_case_student_answer, test_case_answer)

print("Sample Test Case 2:")
test_case = "We all live, in a Yellow Submarine."
test_case_answer = ['We', 'all', 'live', ',', 'in', 'a', 'Yellow', 'Submarine', '.']
test_case_student_answer = nltk_word_tokenize(test_case)
assert evaluate_list_test_cases(test_case, test_case_student_answer, test_case_answer)

print("Sample Test Case 3:")
test_case = "pi isn't a rational number and its approximate value is 3.14"
test_case_answer = ['pi', 'is', "n't", 'a', 'rational', 'number', 'and', 'its', 'approximate', 'value',
test_case_student_answer = nltk_word_tokenize(test_case)
assert evaluate_list_test_cases(test_case, test_case_student_answer, test_case_answer)
```

```
Running Sample Test Cases
Sample Test Case 1:
Input: We all live in a Yellow Submarine
Function Output: ['We', 'all', 'live', 'in', 'a', 'Yellow', 'Submarine']
Expected Output: ['We', 'all', 'live', 'in', 'a', 'Yellow', 'Submarine']
Test Case Passed :)
**********************************

Sample Test Case 2:
Input: We all live, in a Yellow Submarine.
Function Output: ['We', 'all', 'live', ',', 'in', 'a', 'Yellow', 'Submarine', '.']
Expected Output: ['We', 'all', 'live', ',', 'in', 'a', 'Yellow', 'Submarine', '.']
Test Case Passed :)
**********************************

Sample Test Case 3:
Input: pi isn't a rational number and its approximate value is 3.14
Function Output: ['pi', 'is', "n't", 'a', 'rational', 'number', 'and', 'its', 'approximate', 'valu
Expected Output: ['pi', 'is', "n't", 'a', 'rational', 'number', 'and', 'its', 'approximate', 'valu
Test Case Passed :)
**********************************
```

As you can see (if your test cases passed), nltk does a much better job at splitting the text into constituent words, by splitting the punctuations away from the words as well as also taking care of subtleties like splitting `"isn't"` into `"is"` and `"n't"` and retaining the full decimal `"3.14"` which would have been split into `"3"` and `"14"` if we would have naively split on punctuations along with the whitespace.

Task 1.2: Convert the text to lower case (0.25 Marks)

We start the with the most basic of all text preprocessing techniques i.e. converting all the words in the text into lower case. As you will see soon, NLP models often treat different words as different entities and by default do not assume any relation between them. For eg. A word `Bat` will be treated differently from the word `bat` if we use them seperately to define the features. Hence it can be useful to remove such artifacts from the datasets so that we can have common representations for the same words.

Complete the function definiton below

```
def to_lower_case(text):
  """ Converts a piece of text to only contain words in lower case

  Parameters:
    - text (str): A Python string containing the text to be lower-cased

  Returns:
    - text_lower_case (str): A string containing the input text in lower case

  """

  text_lower_case = None
  text_lower_case  = text.lower()

  # raise NotImplementedError()

  return text_lower_case
```

```
"""Don't change code in this cell"""
#SAMPLE TEST CASE

def evaluate_string_test_cases(test_case_input,
                        test_case_func_output,
                        test_case_exp_output):

  print(f"Input: {test_case_input}")
  print(f"Function Output: {test_case_func_output}")
  print(f"Expected Output: {test_case_exp_output}")

  if test_case_func_output == test_case_exp_output:
    print("Test Case Passed :)")
    print("**********************************\n")
    return True
  else:
    print("Test Case Failed :(")
    print("**********************************\n")
    return False


print("Running Sample Test Cases")
print("Sample Test Case 1:")
test_case = "We all live in a Yellow Submarine"
test_case_answer = "we all live in a yellow submarine"
test_case_student_answer = to_lower_case(test_case)
assert evaluate_string_test_cases(test_case, test_case_student_answer, test_case_answer)

print("Sample Test Case 2:")
test_case = "SuRRender To The Void, iT is SHINNING"
test_case_answer = "surrender to the void, it is shinning"
test_case_student_answer = to_lower_case(test_case)
assert evaluate_string_test_cases(test_case, test_case_student_answer, test_case_answer)
```

```
Running Sample Test Cases
Sample Test Case 1:
Input: We all live in a Yellow Submarine
Function Output: we all live in a yellow submarine
Expected Output: we all live in a yellow submarine
Test Case Passed :)
**********************************

Sample Test Case 2:
```

```
Input: SuRRender To The Void, iT is SHINNING
Function Output: surrender to the void, it is shinning
Expected Output: surrender to the void, it is shinning
Test Case Passed :)
**********************************
```

Task 1.3: Remove Punctuations (0.5 Marks)

Another common way to reduce the number of word variations in the text like `hello` vs `hello,` is to remove punctuations. While for some NLP tasks like POS tagging punctuations might be helpful, for classification tasks punctuations can be assumed to have negligible effect on the actual label.

Complete the function `remove_punctuations` below. Examples for the function's working are:

Input	Expec
Mr. and Mrs. Dursley, of number four, Privet Drive, were proud to say that they were perfectly normal!	Mr and Mrs Dursley of number four Privet Driv
"Little tyke," chortled Mr. Dursley as he left the house.	Little tyke chortled Mr Dursley as he left the ho

```
import string

def remove_punctuations(text):
  """
  Removes punctuations from a piece of text.

  Parameters:
    - text (str) : A Python string containing text from which punctuation is to be removed

  Returns:
    - text_no_punct (str): Resulting string after removing punctuation.

  Hint: You can use `string.punctuation` to get a string containing all punctuation symbols.
  >>> print(string.punctuation)
  '!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'

  """

  text_no_punct = ("")
  exclist = string.punctuation
  for word in str(text):
    if word not in exclist:
      text_no_punct = text_no_punct + word
  # text_no_punct = None
  # text_no_punct = text.translate(str.maketrans(text,string.punctuation))


  # raise NotImplementedError()

  return text_no_punct
```

```
print("Running Sample Test Cases")
print("Sample Test Case 1:")
test_case = "Mr. and Mrs. Dursley, of number four, Privet Drive, were proud to say that they were perfe
test_case_answer = "Mr and Mrs Dursley of number four Privet Drive were proud to say that they were per
test_case_student_answer = remove_punctuations(test_case)
assert evaluate_string_test_cases(test_case, test_case_student_answer, test_case_answer)

print("Sample Test Case 2:")
test_case = "\"Little tyke,\" chortled Mr. Dursley as he left the house."
test_case_answer = "Little tyke chortled Mr Dursley as he left the house"
test_case_student_answer = remove_punctuations(test_case)
assert evaluate_string_test_cases(test_case, test_case_student_answer, test_case_answer)
```

```
Running Sample Test Cases
Sample Test Case 1:
Input: Mr. and Mrs. Dursley, of number four, Privet Drive, were proud to say that they were perfe
Function Output: Mr and Mrs Dursley of number four Privet Drive were proud to say that they were
```

```
Expected Output: Mr and Mrs Dursley of number four Privet Drive were proud to say that they were
Test Case Passed :)
**********************************

Sample Test Case 2:
Input: "Little tyke," chortled Mr. Dursley as he left the house.
Function Output: Little tyke chortled Mr Dursley as he left the house
Expected Output: Little tyke chortled Mr Dursley as he left the house
Test Case Passed :)
**********************************
```

Task 1.4: Remove Stop Words (0.5 Marks)

There are some commonly used words in a language like in case of english 'the', 'a', 'I', 'he' which might not provide much valuable information for the current task in hand, and hence can be removed from the text. Again for a task like POS Tagging (which we will see in the future assignments), this shouldn't be done but for sentiment classification the labels can be assumed to be largely independent of the presence of such words.

The choice of the stop words to use can be subjective and in many cases might depend upon the problem in hand. For the purposes of this assignment we will consider the stop words for English language present in the `nltk` package. The code for obtaining these stop words is given in the following cell.

```
from nltk.corpus import stopwords
STOPWORDS = stopwords.words("english")
",".join(STOPWORDS)
```

As can be seen from the output above the stop words contain commonly used words that may not provide much information for the downstream task. Implement the function `remove_stop_words` below using the list of stop words given by `STOPWORDS` in the above cell.

Note that the stop words list contains the words in lower case, so you might want to convert a word in the text to lower case using `to_lower_case` function that you implemented above, before checking if it is present in the stop words list.

```
def remove_stop_words(text):
  """
  Removes stop words given in `nltk.corpus.stopwords.words("english)` from a piece of text.

  Parameters:
    - text (str) : A Python string containing text from which stop words are to be removed

  Returns:
    - text_no_sw (str): Resulting string after removing stop words.

  Hint: You should use `nltk_word_tokenize` for splitting the text
        into words

  """
  STOPWORDS = stopwords.words("english")
  text_no_sw = None
  word_tokens = word_tokenize(text)
  tokens_no_sw = [word for word in word_tokens if not word in STOPWORDS]
  text_no_sw = (" ").join(tokens_no_sw)


  # raise NotImplementedError()
  return text_no_sw

print("Running Sample Test Cases")
print("Sample Test Case 1:")
test_case = "Mr. and Mrs. Dursley, of number four, Privet Drive, were proud to say that they were perfe
test_case_answer = "Mr. Mrs. Dursley , number four , Privet Drive , proud say perfectly normal !"
test_case_student_answer = remove_stop_words(test_case)
assert evaluate_string_test_cases(test_case, test_case_student_answer, test_case_answer)
```

```
Running Sample Test Cases
Sample Test Case 1:
Input: Mr. and Mrs. Dursley, of number four, Privet Drive, were proud to say that they were perfe
Function Output: Mr. Mrs. Dursley , number four , Privet Drive , proud say perfectly normal !
Expected Output: Mr. Mrs. Dursley , number four , Privet Drive , proud say perfectly normal !
Test Case Passed :)
**********************************
```

Task 1.5: Stemming (0.75 Marks)

It can be often benificial to group together different inflections of a word into a single term. For eg. *organizes* and *organizing* are the morphological inflections of the word *organize*, and replacing the instances of *organizes* and *organizing* with *organize* in our text data can help us reduce the number of unique words in our vocabulary. Stemming is one such approach to reduce the inflectional forms into a common base form.

One common way of performing steming over the words are the *Suffix Removal* algorithms, which involves removing certain suffixes from the word based on a pre-defined set of rules like:

- If the word ends with 's' remove 's' (denoted as S-> ϵ.eg. plays -> play)
- If the word ends with 'es' remove 'es' (denoted as ES-> ϵ. eg. mangoes -> mango).
- If the word ends with 'ing' remove 'ing'. (denoted as ING -> ϵ. eg. enjoying -> enjoying)
- If the word ends with 'ly' remove 'ly'. (denoted as LY -> ϵ. eg. badly -> bad)

There can be much more similar rules defined to design a sophisticated stemmer. For now we would want you to implement a simple stemmer that uses only these 4 rules to stem words in a sentence. Complete the `stem_word` and `stem_text` functions below.

```
def stem_word(word):
  """
  Give a word performs suffix removal stemming on it using the following 4 rules:
    - S -> ϵ
    - ES -> ϵ
    - ING -> ϵ
    - LY -> ϵ
  If none of the four suffixes is to be found in the word, the it must be returned
  as it is.
  Parameters:
    - word (str) : A python string representing the word to stemmed

  Returns:
    - stemmed_word (str): `word` after stemming

  HINT: It is possible that two rules can be satisfied like for mangoes
  both the first two rules are satisfied i.e. it ends with 'es' as well as 's'
  In such cases consider the suffix with the larger length which is 'es'
  in this case.
  """

  stemmed_word = None
  # suffixes = ['s','es','ing','ly']
  # if word.endswith(suffixes):
  #   stemmed_word = word[:-len(suffixes)]
  if word.endswith('ing'):
    stemmed_word = word[:-len('ing')]
  elif word.endswith('es'):
    stemmed_word = word[:-len('es')]
  elif word.endswith('ly'):
    stemmed_word = word[:-len('ly')]
  elif word.endswith('s'):
    stemmed_word = word[:-len('s')]
  else:
    return word


  # raise NotImplementedError()
  return stemmed_word



def stem_text(text):
  """
  Stems all the words in a piece of text, by calling `stem_word` for each word.

  Parameters:
    - text (str): A Python string containing text whose words are to be stemmed.

  Returns:
    - stemmed_text (str) : Resulting string after stemming.

  HINT: You should use `nltk_word_tokenize` for splitting the text
        into words

  """

  stemmed_text = None
  word_tokens = word_tokenize(text)
  tokens_no_sw = [stem_word(word) for word in word_tokens]
  stemmed_text = (" ").join(tokens_no_sw)


  # raise NotImplementedError()
  return stemmed_text
```

```
# Sample test cases for `stem_word`

print("Running Sample Test Cases")
print("Sample Test Case 1:")
test_case = "mangoes"
test_case_answer = "mango"
test_case_student_answer = stem_word(test_case)
assert evaluate_string_test_cases(test_case, test_case_student_answer, test_case_answer)

print("Sample Test Case 2:")
test_case = "plays"
test_case_answer = "play"
test_case_student_answer = stem_word(test_case)
assert evaluate_string_test_cases(test_case, test_case_student_answer, test_case_answer)

print("Sample Test Case 3:")
test_case = "enjoying"
test_case_answer = "enjoy"
test_case_student_answer = stem_word(test_case)
assert evaluate_string_test_cases(test_case, test_case_student_answer, test_case_answer)

print("Sample Test Case 4:")
test_case = "badly"
test_case_answer = "bad"
test_case_student_answer = stem_word(test_case)
assert evaluate_string_test_cases(test_case, test_case_student_answer, test_case_answer)

print("Sample Test Case 5:")
```

```
test_case = "fly"
test_case_answer = "f"
test_case_student_answer = stem_word(test_case)
assert evaluate_string_test_cases(test_case, test_case_student_answer, test_case_answer)

print("Sample Test Case 6:")
test_case = "connection"
test_case_answer = "connection"
test_case_student_answer = stem_word(test_case)
assert evaluate_string_test_cases(test_case, test_case_student_answer, test_case_answer)
```

```
Running Sample Test Cases
Sample Test Case 1:
Input: mangoes
Function Output: mango
Expected Output: mango
Test Case Passed :)
**********************************

Sample Test Case 2:
Input: plays
Function Output: play
Expected Output: play
Test Case Passed :)
**********************************

Sample Test Case 3:
Input: enjoying
Function Output: enjoy
Expected Output: enjoy
Test Case Passed :)
**********************************

Sample Test Case 4:
Input: badly
Function Output: bad
Expected Output: bad
Test Case Passed :)
**********************************

Sample Test Case 5:
Input: fly
Function Output: f
Expected Output: f
Test Case Passed :)
**********************************

Sample Test Case 6:
Input: connection
Function Output: connection
Expected Output: connection
Test Case Passed :)
**********************************
```

```
# Sample test cases for `stem_text`

print("Sample Test Case 1:")
test_case = "he sits and eats mangoes"
test_case_answer = "he sit and eat mango"
test_case_student_answer = stem_text(test_case)
assert evaluate_string_test_cases(test_case, test_case_student_answer, test_case_answer)

print("Sample Test Case 2:")
test_case = "he was badly hurt after playing"
test_case_answer = "he wa bad hurt after play"
test_case_student_answer = stem_text(test_case)
assert evaluate_string_test_cases(test_case, test_case_student_answer, test_case_answer)

print("Sample Test Case 3:")
test_case = "fly my little birds"
test_case_answer = "f my little bird"
test_case_student_answer = stem_text(test_case)
```

```
assert evaluate_string_test_cases(test_case, test_case_student_answer, test_case_answer)

print("Sample Test Case 4:")
test_case = "i am facing a poor network connection"
test_case_answer = "i am fac a poor network connection"
test_case_student_answer = stem_text(test_case)
assert evaluate_string_test_cases(test_case, test_case_student_answer, test_case_answer)
```

```
Sample Test Case 1:
Input: he sits and eats mangoes
Function Output: he sit and eat mango
Expected Output: he sit and eat mango
Test Case Passed :)
**********************************

Sample Test Case 2:
Input: he was badly hurt after playing
Function Output: he wa bad hurt after play
Expected Output: he wa bad hurt after play
Test Case Passed :)
**********************************

Sample Test Case 3:
Input: fly my little birds
Function Output: f my little bird
Expected Output: f my little bird
Test Case Passed :)
**********************************

Sample Test Case 4:
Input: i am facing a poor network connection
Function Output: i am fac a poor network connection
Expected Output: i am fac a poor network connection
Test Case Passed :)
**********************************
```

As can be seen above the toy stemmer that we just implemented is not perfect. It is limited to the 4 rules so misses some obvious cases like reducing *connection* to *connect* or sometimes can over stem the word like for the word *fly* above is stemmed to `f` which loses the meaning of the original word. The more sophisticated algorithms use a bunch of rules and heuristics to avoid such situations. One of the most commonly used stemming algorithm is `Porter Stemmer`, which uses a 5 step procedure involving different suffix removal as well as modification rules to perform stemming. Interested students can read more about how Porter Stemmer works from here.

While implementing a sophisticated stemmer like Porter can be very complex, there are many python packages like *NLTK* (https://www.nltk.org/) that provide pre-implementations of a variety of stemming algorithms. Below we demonstrate how to implement `stem_word` function using different stemmers provided in NLTK.

```
# Importing the stemmers from NLTK
from nltk.stem import PorterStemmer
from nltk.stem import LancasterStemmer
from nltk.stem import SnowballStemmer
def stem_word_with_nltk(word, stemmer_type = "porter"):

  """
  Stems a word using Porter Stemmer from NLTK
    - word (str) : A python string representing the word to be stemmed

  Returns:
    - stemmed_word (str): `word` after stemming
  """

  # Initialize an object of the stemer class
  if stemmer_type == "porter":
    stemmer = PorterStemmer()
  elif stemmer_type == "lancaster":
    stemmer = LancasterStemmer()
  elif stemmer_type == "snowball":
    #Snowball stemmer works for multiple languages hence one should be specified during initialization
    stemmer = SnowballStemmer("english")
  else:
    stemmer = PorterStemmer()

  # Call the `stem` method
  stemmed_word = stemmer.stem(word)

  return stemmed_word

def stem_text_with_nltk(text, stemer_type = "porter"):

  """
  Stems all the words in a piece of text using NLTK's stemers, by calling `stem_word_with_nltk` for eac

  Parameters:
    - text (str): A Python string containing text whose words are to be stemmed.

  Returns:
    - stemmed_text (str) : Resulting string after stemming.


  """

  stemmed_text = None

  stemmed_text = " ".join([stem_word_with_nltk(word) for word in word_tokenize(text)])

  return stemmed_text
```

```
input = "connection"
porter_output = stem_word_with_nltk(input, "porter")
lancester_output = stem_word_with_nltk(input, "lancaster")
snowball_output = stem_word_with_nltk(input, "snowball")

print(f"Input: {input}")
print(f"Porter Stemmer Output: {porter_output}")
print(f"Lancaster Stemmer Output: {lancester_output}")
print(f"Snowball Stemmer Output: {snowball_output}")

print("*******************************************\n")

input = "fly"
porter_output = stem_word_with_nltk(input, "porter")
lancester_output = stem_word_with_nltk(input, "lancaster")
snowball_output = stem_word_with_nltk(input, "snowball")

print(f"Input: {input}")
print(f"Porter Stemmer Output: {porter_output}")
print(f"Lancaster Stemmer Output: {lancester_output}")
print(f"Snowball Stemmer Output: {snowball_output}")

print("*******************************************\n")


input = "happiness"
porter_output = stem_word_with_nltk(input, "porter")
lancester_output = stem_word_with_nltk(input, "lancaster")
snowball_output = stem_word_with_nltk(input, "snowball")

print(f"Input: {input}")
print(f"Porter Stemmer Output: {porter_output}")
print(f"Lancaster Stemmer Output: {lancester_output}")
print(f"Snowball Stemmer Output: {snowball_output}")

print("*******************************************\n")
```

```
input = "operational"
porter_output = stem_word_with_nltk(input, "porter")
lancester_output = stem_word_with_nltk(input, "lancaster")
snowball_output = stem_word_with_nltk(input, "snowball")

print(f"Input: {input}")
print(f"Porter Stemmer Output: {porter_output}")
print(f"Lancaster Stemmer Output: {lancester_output}")
print(f"Snowball Stemmer Output: {snowball_output}")
```

```
Input: connection
Porter Stemmer Output: connect
Lancaster Stemmer Output: connect
Snowball Stemmer Output: connect
*******************************************

Input: fly
Porter Stemmer Output: fli
Lancaster Stemmer Output: fly
Snowball Stemmer Output: fli
*******************************************

Input: happiness
Porter Stemmer Output: happi
Lancaster Stemmer Output: happy
Snowball Stemmer Output: happi
*******************************************

Input: operational
Porter Stemmer Output: oper
Lancaster Stemmer Output: op
Snowball Stemmer Output: oper
```

As can be seen these stemmers are able to recognize various types of cases for stemming and can do a better job than our toy implementation. However, even these stemmers tend to make errors, like *operational* is reduced to *op* by Lancaster stemmer. This is because in the end stemming algorithms just use a bunch of heuristics to reduce the inflectional forms without any proper morphological analysis. **Lemmatization** is another common technique used to reduce different forms of a word to a common term called *lemma*, which makes use of a pre-defined vocabulary and morphological analysis of the words. Lemmatizers often work better when supplied with *Part of Speech tags* which we will be covering in the future assignments, so we will revisit Lemmatization later.

Task 1.6: Combine all preprocessing functions to a single pipeline (0.5 Marks)

Now that we are done implementing the main preprocessing functions that might be useful for a text classification task, we can combine them into a single function that applies these 4 preprocessing techniques over a piece of text. We will then use this function to preprocess the movie reviews in our training and dev datasets.

Implement the `preprocess_pipeline` and `preprocess_sentiment_data` functions below, where the former function combines the 4 preprocessing functions implemented above and the latter applies that to all the reviews in the dataset.

Note: The order in which different preprocessing functions are to be applied can lead to different results. For eg. all of our stop words are in lower case hence before we remove them from the text we must make sure we have converted the text to lower case before hand. Please follow the following order for applying the preprocessing functions in your code:

1. to_lower_case
2. remove_punctuations
3. remove_stop_words
4. stem_text_with_nltk (call this with `stemer_type = "porter"`)

```
def preprocess_pipeline(text):

  """
  Given a piece of text applies preprocessing techniques
  like converting to lower case, removing stop words and punctuations,
  and stemming.

  Apply the functions in the following order:
  1. to_lower_case
  2. remove_punctuations
  3. remove_stop_words
  4. stem_text_with_nltk (call this with `stemer_type = "porter"`)

  Inputs:
    - text (str) : A python string containing text to be pre-processed

  Returns:
    - text_preprocessed (str) : Resulting string after applying preprocessing
  """

  text_preprocessed = None
  text_preprocessed = to_lower_case(text)
  text_preprocessed = remove_punctuations(text_preprocessed)
  text_preprocessed = remove_stop_words(text_preprocessed)
  # word_tokens = word_tokenize(text)
  tokens_no_sw = stem_text_with_nltk(text_preprocessed,stemer_type = "porter")
··text_preprocessed·=·("").join(tokens_no_sw)

  # raise NotImplementedError()

  return text_preprocessed
```

```
print("Running Sample Test Cases")
print("Sample Test Case 1:")
test_case = "Mr. and Mrs. Dursley, of number four, Privet Drive, were proud to say that they were perfe
test_case_answer = "mr mr dursley number four privet drive proud say perfectli normal"
test_case_student_answer = preprocess_pipeline(test_case)
assert evaluate_string_test_cases(test_case, test_case_student_answer, test_case_answer)

print("Sample Test Case 2:")
test_case = "\"Little tyke,\" chortled Mr. Dursley as He left the house."
test_case_answer = "littl tyke chortl mr dursley left hous"
test_case_student_answer = preprocess_pipeline(test_case)
assert evaluate_string_test_cases(test_case, test_case_student_answer, test_case_answer)
```

```
Running Sample Test Cases
Sample Test Case 1:
Input: Mr. and Mrs. Dursley, of number four, Privet Drive, were proud to say that they were perfe
Function Output: mr mr dursley number four privet drive proud say perfectli normal
Expected Output: mr mr dursley number four privet drive proud say perfectli normal
Test Case Passed :)
**********************************

Sample Test Case 2:
Input: "Little tyke," chortled Mr. Dursley as He left the house.
Function Output: littl tyke chortl mr dursley left hous
Expected Output: littl tyke chortl mr dursley left hous
Test Case Passed :)
**********************************
```

```
def preprocess_sentiment_data(df):
  """
  Takes the pandas dataframe containing SST-2 data as input and applies
  the `preprocess_pipeline` function to the its sentence column.

  Inputs:
    - df (pd.DataFrame): A pandas dataframe containing the SST-2 data with format:
        | sentence  | label  |
        ----------------------
        | sentence_1| label_1|
        | sentence_2| label_2|
        | ..........| .......|
        | ..........| .......|

  Returns (pd.DataFrame):
   - df_preprocessed: Resulting dataframe after applying `preprocessing_pipeline`
    to the `sentence` column. Note that the column names of df_preprocessed
    should be same as df and its should have the same number of rows as `df`.


  Hint: Look up how to use `pd.DataFrame.apply` method in pandas

  """

  df_preprocessed = None
  df['sentence'] = df['sentence'].apply(preprocess_pipeline)
  df_preprocessed = df
  # raise NotImplementedError()
  return df_preprocessed
```

```
# Preprocess the train and test sets. This might take a few minutes
train_df_preprocessed = preprocess_sentiment_data(train_df)
test_df_preprocessed = preprocess_sentiment_data(test_df)
print(train_df_preprocessed.head())
print("***************************")
print(test_df_preprocessed.head())
```

```
                                            sentence  label
0                        hide new secret parent unit      0
1                              contain wit labor gag      0
2  love charact commun someth rather beauti human...      1
3            remain utterli satisfi remain throughout      0
4  worst revengeofthenerd cliché filmmak could dredg      0
***************************
                                            sentence  label
0                         charm often affect journey      1
1                         unflinchingli bleak desper      0
2  allow us hope nolan pois embark major career c...      1
3  act costum music cinematographi sound astound ...      1
4                                          slow slow      0
```

```
type(train_df_preprocessed)
```

```
pandas.core.frame.DataFrame
```

```
print("Running Sample Test Cases")
print("Sample Test Case 1: Checking if the object returned is a pandas Dataframe")
assert isinstance(train_df_preprocessed, pd.DataFrame)
print("Test Case Passed :)")
print("***********************************\n")

print("Sample Test Case 2: Checking if the returned dataframe has correct columns")
student_column_names = sorted(list(train_df_preprocessed.columns))
expected_column_names = ["label", "sentence"]

assert student_column_names == expected_column_names
print("Test Case Passed :)")
print("***********************************\n")

print("Sample Test Case 3: Checking if the number of rows of the returned dataframe is same as the orig
assert len(train_df) == len(train_df_preprocessed) and len(test_df) == len(test_df_preprocessed)
print("Test Case Passed :)")
print("***********************************\n")
print("Sample Test Case 4: Checking if the returned dataframe has sentences preprocessed")
```

```
student_output = train_df_preprocessed["sentence"].values[1]
expected_output = "contain wit labor gag"
assert evaluate_string_test_cases(train_df["sentence"].values[1], student_output, expected_output)
```

```
Running Sample Test Cases
Sample Test Case 1: Checking if the object returned is a pandas Dataframe
Test Case Passed :)
***********************************

Sample Test Case 2: Checking if the returned dataframe has correct columns
Test Case Passed :)
***********************************

Sample Test Case 3: Checking if the number of rows of the returned dataframe is same as the origi
Test Case Passed :)
***********************************

Sample Test Case 4: Checking if the returned dataframe has sentences preprocessed
Input: contain wit labor gag
Function Output: contain wit labor gag
Expected Output: contain wit labor gag
Test Case Passed :)
**********************************
```

Task 2: Bag Of Word Models for Text Classification (1.75 Marks)

Now that we are done with preprocessing the reviews in our datasets, we can begin building a classifier to classify them into positive or negative sentiment.

As you might have studied in your Machine Learning courses, typical ML models work on the data described using mathematical objects like vectors and matrices, which are often referred to as features. These features can be of different types depending upon the downstream application, like for building a classifier to predict whether to give credit to a customer we might consider features like their age, income, employement status etc. In the same way to build a classifier for textual data, we need a way to describe each text example in terms of numeric features which can then be fed to the classification algorithm of our choice.

Bag of Words model is one of the simplest but surprisingly effective way to represent text data for building Machine Learning models. In bag of words, occurence (or frequency) of each word in a given text example is defined as a feature for training the classifier. The order in which these words occur in the text is not relevant and we are just concerned with which words are present in the text. Consider the following example to understand how bag of words are used to represent text.

As an example consider we have 2 examples present in our dataset:

x1: john likes to watch movies mary likes movies too

x2: mary also likes to watch football games

Based on these two documents we can get the list of all words that occur in this dataset which will be:

index	word
0	also
1	football
2	games
3	john
4	likes
5	mary
6	movies
7	to
8	too
9	watch

We can then define features for the two x1 and x2 as follows:

	also	football	games	john	likes	mary	movies	to	too	watch
x1	0	0	0	1	2	1	2	1	2	1
x2	1	1	1	0	1	1	0	0	0	0

These features can then be used to train an ML model. To summarize the following two steps must be followed to create bag of word representations of the text examples in a dataset.

- Step 1: Create a word vocabulary by iterating through all the documents in the **training** dataset, storing all the unique words that are present in each document. Also maintain mappings to map each word to an index and vice-versa, which we will need to define values for each feature dimension.
- Step 2: For each document in the training and test sets, get the frequency of each word in our vocabulary and use it to define feature for that example.

Below you will implement functions to create bag of words representations of the dataset examples

Task 2.1: Create vocabularies (0.5 Marks)

As described above our first step will be to create word vocabulary for the documents in our dataset. Implement `create_vocab` function below which takes as input a list of documents and creates a list of unique words that occur in them.

```
def create_vocab(documents):
  """
  Given a list of documents each represented as a string,
  create a word vocabulary containing all the words that occur
  in these documents.

  Inputs:
    - documents (list) : A list with each element as a string representing a
    document.

  Returns:
    - vocab (list) : A **sorted** list containing all unique words in the
    documents

  Example Input: ['john likes to watch movies mary likes movies too',
                  'mary also likes to watch football games']

  Expected Output: ['also',
                    'football',
                    'games',
                    'john',
                    'likes',
                    'mary',
                    'movies',
                    'to',
                    'too',
                    'watch']


  Hint: `nltk_word_tokenize` function may come in handy

  """

  vocab = []

  for sent in documents:
      for word in word_tokenize(sent):
          if word not in vocab:
              vocab.append(word)

  # raise NotImplementedError()

  return sorted(vocab) # Don't change this
```

```
print("Running Sample Test Cases")
print("Sample Test Case 1:")

test_case = ["john likes to watch movies mary likes movies too",
              "mary also likes to watch football games"]
test_case_answer = ['also', 'football', 'games', 'john', 'likes', 'mary', 'movies', 'to', 'too', 'watch
test_case_student_answer = create_vocab(test_case)
assert evaluate_list_test_cases(test_case, test_case_student_answer, test_case_answer)

print("Sample Test Case 2:")

test_case = ["We all live in a yellow submarine.",
             "Yellow submarine, yellow submarine!!"
             ]
test_case_answer = ['!', ',', '.', 'We', 'Yellow', 'a', 'all', 'in', 'live', 'submarine', 'yellow']
test_case_student_answer = create_vocab(test_case)
assert evaluate_list_test_cases(test_case, test_case_student_answer, test_case_answer)
```

```
Running Sample Test Cases
Sample Test Case 1:
Input: ['john likes to watch movies mary likes movies too', 'mary also likes to watch football gar
Function Output: ['also', 'football', 'games', 'john', 'likes', 'mary', 'movies', 'to', 'too', 'wa
Expected Output: ['also', 'football', 'games', 'john', 'likes', 'mary', 'movies', 'to', 'too', 'wa
Test Case Passed :)
**********************************

Sample Test Case 2:
Input: ['We all live in a yellow submarine.', 'Yellow submarine, yellow submarine!!']
Function Output: ['!', ',', '.', 'We', 'Yellow', 'a', 'all', 'in', 'live', 'submarine', 'yellow']
Expected Output: ['!', ',', '.', 'We', 'Yellow', 'a', 'all', 'in', 'live', 'submarine', 'yellow']
Test Case Passed :)
**********************************
```

Note the output of sample test case 2 contains punctuations as well as different upper-case and lower-case variants of a word detected as seperate words. This illustrates the importance of performing the preprocessing over the documents as it reduces the unecessary words like punctuations, stop words in the vocablary as well as help provide a common term to different variants of a word.

```
# Now that our `create_vocab` function is ready, let's create vocabulary for the SST dataset

train_documents = train_df_preprocessed["sentence"].values.tolist() # Note that we are selecting prepro
train_vocab = create_vocab(train_documents)

print(f"Training Vocabulary Created. Number of words: {len(train_vocab)}")
```

```
Training Vocabulary Created. Number of words: 10781
```

Task 2.2: Create word to index mapping (0.25 Marks)

Now that we have a list of words in our dataset, we can map each word to an index which will be useful to represent what word each feature dimension refers to. Implement the `get_word_idx_mappings` function below:

```
def get_word_idx_mapping(vocab):

  """
  Give a list of strings each representing a word in the vocabulary
  creates a dictionary that maps each word in the list to its
  corresponding index.

  Inputs:
    - vocab (list): A list of strings each representing a word in the vocabulary
  Outputs:
    - word2idx (dict): A Python dictionary mapping each word to its index in vocabulary

  Example Input: ['also',
                    'football',
                    'games',
                    'john',
                    'likes',
                    'mary',
                    'movies',
                    'to',
                    'too',
                    'watch']

  Expected Output: {'also': 0,
                  'football': 1,
                  'games': 2,
                  'john': 3,
                  'likes': 4,
                  'mary': 5,
                  'movies': 6,
                  'to': 7,
                  'too': 8,
                  'watch': 9}

  """

  # word2idx = {word: i for i, word in enumerate(vocab)}
  word2idx = dict(zip(vocab, range(len(vocab))))


  # raise NotImplementedError()

  return word2idx
```

```
print("Running Sample Test Cases")
print("Sample Test Case 1:")
test_case = ['also', 'football', 'games', 'john', 'likes', 'mary', 'movies', 'to', 'too', 'watch']
test_case_answer = {'also': 0, 'football': 1, 'games': 2, 'john': 3, 'likes': 4, 'mary': 5, 'movies': 6
test_case_student_answer = get_word_idx_mapping(test_case)
assert evaluate_list_test_cases(test_case, test_case_student_answer, test_case_answer)
```

```
Running Sample Test Cases
Sample Test Case 1:
Input: ['also', 'football', 'games', 'john', 'likes', 'mary', 'movies', 'to', 'too', 'watch']
Function Output: {'also': 0, 'football': 1, 'games': 2, 'john': 3, 'likes': 4, 'mary': 5, 'movies
Expected Output: {'also': 0, 'football': 1, 'games': 2, 'john': 3, 'likes': 4, 'mary': 5, 'movies
Test Case Passed :)
**********************************
```

```
train_vocab2idx = get_word_idx_mapping(train_vocab)
train_vocab2idx
```

```
 'awri': 758,
 'axel': 759,
 'ayatollah': 760,
 'ayr': 761,
 'ayurveda': 762,
 'b': 763,
 'b12': 764,
 'baaaaaaaaad': 765,
 'babak': 766,
 'babbitt': 767,
 'babi': 768,
 'baboon': 769,
 'babyfac': 770,
 'babysitt': 771,
 'back': 772,
 'backbon': 773,
 'backdrop': 774,
 'background': 775,
 'backhand': 776,
 'backlash': 777,
 'backmask': 778,
 'backseat': 779,
 'backstab': 780,
 'backstag': 781,
 'backstori': 782,
 'backward': 783,
 'backwat': 784,
 'bad': 785,
 'badder': 786,
 'badli': 787,
 'badlyrend': 788,
 'badmovi': 789,
 'baffl': 790,
 'bag': 791,
 'bagatel': 792,
 'baggag': 793,
 'bai': 794,
 'bailiwick': 795,
 'bailli': 796,
 'baio': 797,
 'baird': 798,
 'baitandswitch': 799,
 'bake': 800,
 'baker': 801,
 'balanc': 802,
 'bale': 803,
 'balk': 804,
 'ball': 805,
 'ballast': 806,
 'ballerina': 807,
 'ballet': 808,
 'ballist': 809,
 'ballisticpyrotechn': 810,
 'ballot': 811,
 'ballplay': 812,
 'ballroom': 813,
 'ballsi': 814,
 'balm': 815,
```

Task 2.3: Create Bag of word features for the documents (1 Mark)

Now that we have the list of words and a word to index mapping we can create a bag of word feature vector for each of the document present in training and test data. Implement the function `get_document_bow_feature` that takes as an input a document, a vocabulary, and a vocabulary to index mapping, and returns the feature vector for the document.

```
def get_document_bow_feature(document, vocab, word2idx):
  """
  Given a string representing the document and the vocabulary, create a bag of
  words representation of the document i.e. a feature vector where each feature
  is defined as the frequency of each word in the vocabulary.

  Inputs:
   - document (str): A python string representing the document for which features are to be obtained
   - vocab (list): A list of words present in the vocabulary
   - word2idx (dict): A dictionary that maps each word to an index.

  Returns:
    - bow_feature (numpy.ndarray): A numpy array of size `len(vocab)` whose each element contains the c

  Example Input:
    document = "john likes to watch movies mary likes movies too"
    vocab = ['also','football','games','john','likes','mary', 'movies','to','too','watch']
    word2idx = {'also': 0, 'football': 1, 'games': 2, 'john': 3, 'likes': 4, 'mary': 5, 'movies': 6, 't

  Expected Output: array([0, 0, 0, 1, 2, 1, 2, 1, 1, 1])
  """

  bow_feature = np.zeros(len(vocab))
  words = word_tokenize(document)
  for word in words:
    if word in vocab:
      bow_feature[word2idx[word]]=bow_feature[word2idx[word]]+1

   # raise NotImplementedError()

  return bow_feature
```

```
print("Running Sample Test Cases")
print("Sample Test Case 1:")
test_case = {"document": "john likes to watch movies mary likes movies too",
             "vocab": ['also', 'football', 'games', 'john', 'likes', 'mary', 'movies', 'to', 'too', 'wa
             "word2idx": {'also': 0, 'football': 1, 'games': 2, 'john': 3, 'likes': 4, 'mary': 5, 'movi
             }
test_case_answer = np.array([0, 0, 0, 1, 2, 1, 2, 1, 1, 1])
test_case_student_answer = get_document_bow_feature(**test_case)
assert evaluate_list_test_cases(test_case, test_case_student_answer.tolist(), test_case_answer.tolist()

print("Sample Test Case 2:")
test_case = {"document": "mary also likes to watch football games",
             "vocab": ['also', 'football', 'games', 'john', 'likes', 'mary', 'movies', 'to', 'too', 'wa
             "word2idx": {'also': 0, 'football': 1, 'games': 2, 'john': 3, 'likes': 4, 'mary': 5, 'movi
             }
test_case_answer = np.array([1, 1, 1, 0, 1, 1, 0, 1, 0, 1])
test_case_student_answer = get_document_bow_feature(**test_case)
assert evaluate_list_test_cases(test_case, test_case_student_answer.tolist(), test_case_answer.tolist()

print("Sample Test Case 3:")
test_case = {"document": "mary and jane also like to watch football games",
             "vocab": ['also', 'football', 'games', 'john', 'likes', 'mary', 'movies', 'to', 'too', 'wa
             "word2idx": {'also': 0, 'football': 1, 'games': 2, 'john': 3, 'likes': 4, 'mary': 5, 'movi
             }
test_case_answer = np.array([1, 1, 1, 0, 0, 1, 0, 1, 0, 1])
test_case_student_answer = get_document_bow_feature(**test_case)
assert evaluate_list_test_cases(test_case, test_case_student_answer.tolist(), test_case_answer.tolist()
```

```
Running Sample Test Cases
Sample Test Case 1:
Input: {'document': 'john likes to watch movies mary likes movies too', 'vocab': ['also', 'footba
Function Output: [0.0, 0.0, 0.0, 1.0, 2.0, 1.0, 2.0, 1.0, 1.0, 1.0]
Expected Output: [0, 0, 0, 1, 2, 1, 2, 1, 1, 1]
Test Case Passed :)
**********************************

Sample Test Case 2:
Input: {'document': 'mary also likes to watch football games', 'vocab': ['also', 'football', 'gam
Function Output: [1.0, 1.0, 1.0, 0.0, 1.0, 1.0, 0.0, 1.0, 0.0, 1.0]
Expected Output: [1, 1, 1, 0, 1, 1, 0, 1, 0, 1]
Test Case Passed :)
**********************************

Sample Test Case 3:
Input: {'document': 'mary and jane also like to watch football games', 'vocab': ['also', 'footbal
Function Output: [1.0, 1.0, 1.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0]
Expected Output: [1, 1, 1, 0, 0, 1, 0, 1, 0, 1]
Test Case Passed :)
**********************************
```

Now that our `get_document_bow_feature` function seems to work properly, let's get bag of word features for the examples in our datasets.

```
# Getting bow feature for one training document first
training_example = train_documents[0]
bow_feature = get_document_bow_feature(training_example,
                                       train_vocab,
                                       train_vocab2idx)
print(f"Length of bow feature: {len(bow_feature)}")
print(f"Number of non-zero entries in the bow feature: {len(bow_feature[bow_feature != 0])}")
```

```
Length of bow feature: 10781
Number of non-zero entries in the bow feature: 5
```

As you can see, since our vocabulary size is 10781, bag of words vectors for each document will be of size 10781. Since we have about 67k training examples, it won't be practical to store these high dimensional vectors as it is for all the documents. Instead of storing the features for all the documents in memory, we query the features while training the model in a batch wise fashion i.e. at a time we train on N examples, such that N <<< 10781. This will be more clear when we discuss creating datasets and dataloaders in the next part.

Task 3: Training a Linear Classifier using Bag of Words Features (5.25 Marks)

Now that we have defined our numerical features to represent each of the documents, we can start training a classifier on top of it. For the purposes of this assignment we will be implementing a linear classifier namely **Logistic Regression**. We assume that you have studied logistic regression in your Machine Learning course. For a recap of the same you can refer to these videos.

We will be using Pytorch to implement and train our logistic regression classifier for the sentiment prediction task. We start by implementing the dataset and dataloaders to iterate over the dataset, and then we move to defining the logistic regression module, the cross entropy loss function and an optimizer for training the model.

Task 3.1: Defining Pytorch Dataset and Dataloaders to iterate over the SST-Dataset (0.5 Marks)

Often while training neural networks or in our case linear classifiers, it is not practical to train over the entire dataset at once, and instead we use a batch wise training strategy, where we iterate over different batches of the dataset. Hence, it is useful to define iterators for doing the same. Defining pipelines for processing data samples and then writing iterators on top of that can often be very messy. Pytorch provides `torch.utils.data.Dataset` and `torch.utils.data.Dataloader` classes that make it much more convenient to do the same in a modular fashion.

`torch.utils.data.Dataset` provides a wrapper to store our dataset, which can then be used by `torch.utils.data.Dataloader` to define an iterable over the dataset. To learn more about Dataset and Dataloader classes, please refer to the tutorial here

We start by defining a custom Dataset class for our dataset by extending the `torch.utils.data.Dataset` class. A custom Dataset class must implement three functions:

- `__init__`: This is the constructor for our custom class, and is often used to store the (meta)data, which can then be used by the other functions to create samples of the dataset.
- `__len__`: This method returns the number of examples present in the dataset.
- `__getitem__`: This method loads and returns a sample stored at the given index `idx`

```
from torch.utils.data import Dataset, DataLoader
```

```
class SST2Dataset(Dataset):

  def __init__(self, documents, labels, vocab, word2idx):
    """
    Store dataset documents and labels here so that they can be used by
    __getitem__ to process and return the samples.

    Inputs:
      - documents (list): A list of strings containing reviews in our dataset.
      - labels (list): A list of sentiment labels (1 or 0) corresponding to each document.
      - vocab (list): A list of words present in the vocabulary
      - word2idx (dict): A dictionary that maps each word to an index.
    """

    self.documents = documents
    self.labels = labels
    self.vocab = vocab
    self.word2idx = word2idx

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

  def __getitem__(self, idx):
    """
    Loads and returns the features and label corresponding to the `idx` index
    in the documents and labels lists.

    Inputs:
      - idx (index): Index of the dataset example to be loaded and returned

    Returns:
      - features (numpy.ndarray): The bag of word features corresponding the document indexed by `idx`
      - label (int): The sentiment label for the `idx`th document

    Hint: You can get the document and label by doing self.documents[idx],
    self.labels[idx]. Features of the document are to be extracted via
    `get_document_bow_feature` function
    """

    features, label = None, None
    features = get_document_bow_feature(self.documents[idx], self.vocab, self.word2idx)
    label = self.labels[idx]
    # raise NotImplementedError()

    return features, label
```

```
print("Running Sample Test Cases")

sample_documents = ["this movie is great",
                    "I HATED this film"]
sample_labels = [0, 1]
sample_vocab = create_vocab(sample_documents)
sample_word2idx = get_word_idx_mapping(sample_vocab)
sample_dataset = SST2Dataset(sample_documents, sample_labels,
                             sample_vocab, sample_word2idx)

test_case_idx = 0
features, label = sample_dataset.__getitem__(test_case_idx)

print(f"Sample Test Case 1: Testing Returned Labels for idx = {test_case_idx}")
print(f"Output Label: {label}")
print(f"Expected Label: {sample_labels[test_case_idx]}")
assert label == sample_labels[test_case_idx]
print("**********************************\n")

print(f"Sample Test Case 2: Testing returned Features for idx = {test_case_idx}")
expected_features = [0, 0, 0, 1, 1, 1, 1]
```

```
print(f"Output Features: {features.tolist()}")
print(f"Expected Features: {expected_features}")
assert expected_features == features.tolist()

test_case_idx = 1
features, label = sample_dataset.__getitem__(test_case_idx)
print("**********************************\n")

print(f"Sample Test Case 3: Testing Returned Labels for idx = {test_case_idx}")
print(f"Output Label: {label}")
print(f"Expected Label: {sample_labels[test_case_idx]}")
assert label == sample_labels[test_case_idx]
print("**********************************\n")

print(f"Sample Test Case 4: Testing returned Features for idx = {test_case_idx}")
expected_features = [1, 1, 1, 0, 0, 0, 1]
print(f"Output Features: {features.tolist()}")
print(f"Expected Features: {expected_features}")
assert expected_features == features.tolist()
print("**********************************\n")
```

```
Running Sample Test Cases
Sample Test Case 1: Testing Returned Labels for idx = 0
Output Label: 0
Expected Label: 0
**********************************

Sample Test Case 2: Testing returned Features for idx = 0
Output Features: [0.0, 0.0, 0.0, 1.0, 1.0, 1.0, 1.0]
Expected Features: [0, 0, 0, 1, 1, 1, 1]
**********************************

Sample Test Case 3: Testing Returned Labels for idx = 1
Output Label: 1
Expected Label: 1
**********************************

Sample Test Case 4: Testing returned Features for idx = 1
Output Features: [1.0, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0]
Expected Features: [1, 1, 1, 0, 0, 0, 1]
**********************************
```

Now that the custom class SST2Dataset seems to working fine we can create objects for our training and test datasets

```
# Get documents and labels from the dataset
train_documents = train_df_preprocessed["sentence"].values.tolist()
train_labels = train_df["label"].values.tolist()
test_documents = test_df_preprocessed["sentence"].values.tolist()
test_labels = test_df["label"].values.tolist()

# Create vocabulary from training data
train_vocab = create_vocab(train_documents)
train_word2idx = get_word_idx_mapping(train_vocab)

# Create datasets
train_dataset = SST2Dataset(train_documents,
                            train_labels,
                            train_vocab,
                            train_word2idx)
test_dataset = SST2Dataset(test_documents,
                           test_labels,
                           train_vocab,
                           train_word2idx
                           )
```

Notice how we used training data vocabulary for creating test dataset as well. Can you tell why?

Now that we have created our training and test datasets, we can create dataloaders to iterate over them in batches. We will use a batch size of 64 in our experiments. Note that lower the batch size lesser will be your memory requirements but the noisier will be the training. Since in our case features are sufficiently high dimensional, we might not want to use too large of a batch size, hence we are using 64.

```
batch_size = 64
train_dataloader = DataLoader(train_dataset, batch_size = batch_size)
test_dataloader = DataLoader(test_dataset, batch_size = batch_size)
```

Dataloaders work like any iterable (like Lists, dictionaries etc) in python can be iterated over using a for loop like this:

```
for batch in train_dataloader:
  # Unpacking the batch
  features, labels = batch
  print(f"Features Shape: {features.shape}")
  print(f"Labels Shape: {labels.shape}")

  # We will break for now as this is just for demonstration
  break
```

```
    Features Shape: torch.Size([64, 10781])
    Labels Shape: torch.Size([64])
```

Notice how each batch unraps to a features and a labels torch tensor. Features is a 64x10781, where 64 is the batch size used by us and 10781 is the numeber of features we have for each document. Torch tensors behave very similar to numpy arrays, with the benifit that these can be transferred to a GPU and also support auto-differentiation. We will address these points again when we implement the training loop.

Task 3.2: Define the model architecture for the Logistic Regression Classifier (0.75 Marks)

Pytorch provides very elegantly designed `torch.nn` module which contains building blocks for creating different neural network architectures. Some of the sub-modules included in `torch.nn` includes:

- `torch.nn.Linear` : Perhaps one of the simplest of the nn modules, it is used to apply a linear transformation to the data i.e. y = xA^T +b, where x is the input and y is the output of the layer. A and b are the parameters of the layer, where A is often called the weights matrix and b is the bias vector.
- `torch.nn.Conv2d`: Used to create Convolutional Layers.
- `torch.nn.Transformer`: Used to create Transformer layers

and many more. For the purposes of this assignment we will only be needing `torch.nn.Linear` to define our network.

It also supports different activation functions like:

- `torch.nn.ReLU`
- `torch.nn.Sigmoid`
- `torch.nn.Tanh`
- `torch.nn.Softmax`

Below we demonstrate the usage of some of these modules

```
import torch
import torch.nn as nn

# Create a linear layer that maps a 5 dimensional vector to a 3 dimensional vector
example_linear_layer = nn.Linear(5, 3) # Initialize a linear layer
example_input = torch.rand(5) # Create a random vector for demonstration
```

```
print(f"Input: {example_input}")
print(f"Input Shape: {example_input.shape}")
example_output = example_linear_layer(example_input) # Feed the example input to the linear layer
print(f"Linear layer output: {example_output}")
print(f"Linear layer output Shape: {example_output.shape}")

print("*************************************************\n")

# We can also use linear layers with batched inputs
example_batch_input = torch.rand(4,5) # Create an example input containing 4 inputs of 5 dimension each
print(f"Batched Input:\n {example_batch_input}")
print(f"Batched Input Shape: {example_batch_input.shape}")
example_batch_output = example_linear_layer(example_batch_input)
print(f"Linear layer batched output:\n {example_batch_output}")
print(f"Linear layer batched output Shape: {example_batch_output.shape}")

print("*************************************************\n")

# Using activation functions
sigmoid_activation = nn.Sigmoid() #Define sigmoid activation
relu_activation = nn.ReLU() #Define relu activation

sigmoid_output = sigmoid_activation(example_batch_output) # Apply the sigmoid function to the output of
relu_output = relu_activation(example_batch_output) # Apply the relu function to the output of the line
print(f"Before Applying the activation function:\n {example_batch_output}")
print(f"After Applying the sigmoid function:\n {sigmoid_output}")
print(f"After Applying the relu function:\n {relu_output}")

print("*************************************************\n")
```

```
Input: tensor([0.8016, 0.4648, 0.9050, 0.5181, 0.5805])
Input Shape: torch.Size([5])
Linear layer output: tensor([-0.3797, -0.2174,  0.1598], grad_fn=<AddBackward0>)
Linear layer output Shape: torch.Size([3])
*************************************************

Batched Input:
 tensor([[0.0168, 0.8726, 0.1138, 0.4349, 0.5880],
        [0.2770, 0.2831, 0.6293, 0.6480, 0.0624],
        [0.1690, 0.0779, 0.2953, 0.3335, 0.4530],
        [0.3815, 0.5841, 0.8840, 0.1512, 0.4605]])
Batched Input Shape: torch.Size([4, 5])
Linear layer batched output:
 tensor([[-0.2485, -0.0013,  0.4542],
        [-0.1031,  0.0404,  0.2571],
        [ 0.0155, -0.0101,  0.2449],
        [-0.1903, -0.0808,  0.1336]], grad_fn=<AddmmBackward0>)
Linear layer batched output Shape: torch.Size([4, 3])
*************************************************

Before Applying the activation function:
 tensor([[-0.2485, -0.0013,  0.4542],
        [-0.1031,  0.0404,  0.2571],
        [ 0.0155, -0.0101,  0.2449],
        [-0.1903, -0.0808,  0.1336]], grad_fn=<AddmmBackward0>)
After Applying the sigmoid function:
 tensor([[0.4382, 0.4997, 0.6116],
        [0.4743, 0.5101, 0.5639],
        [0.5039, 0.4975, 0.5609],
        [0.4526, 0.4798, 0.5333]], grad_fn=<SigmoidBackward0>)
After Applying the relu function:
 tensor([[0.0000, 0.0000, 0.4542],
        [0.0000, 0.0404, 0.2571],
        [0.0155, 0.0000, 0.2449],
        [0.0000, 0.0000, 0.1336]], grad_fn=<ReluBackward0>)
*************************************************
```

Notice how the outputs of the nn layers also contain a `grad_fn`. This is used during backpropagation to compute the gradients which are used by the optimizer to learn the parameters of these layers.

We define a network in Pytorch by extending the `torch.nn.Module` class and implementing the `__init__` and `forward` method. The `__init__` method is used to define the components of the architecture, while `forward`, takes an input tensor and passes it through the different layers of the network. You can refer to the documentation for `torch.nn` here and also can refer to this for a detailed tutorial on the same.

Below we implement the `LogisticRegressionModel` class

```
import torch
import torch.nn as nn

class LogisticRegressionModel(nn.Module):

  def __init__(self, d_input):
    """
    Define the architecture of a Logistic Regression classifier.
    You will need to define two components, one will be the linear layer using
    nn.Linear, and a sigmoid activation function for the output.

    Inputs:
      - d_input (int): The dimensionality or number of features in each input.
                        This will be required to define the linear layer

    Hint: Recall that in logistic regression we obtain a single probablility
    value for each input that denotes how likely is the input belonging
    to the positive class
    """
    #Need to call the constructor of the parent class
    output_dim = 1
    super(LogisticRegressionModel, self).__init__()

    self.linear_layer = None
    self.sigmoid_layer = None

    self.linear_layer = torch.nn.Linear(d_input, output_dim)
    # raise NotImplementedError()


  def forward(self, x):
    """
    Passes the input `x` through the layers in the network and returns the output

    Inputs:
      - x (torch.tensor): A torch tensor of shape [batch_size, d_input] representing the batch of input

    Returns:
      - output (torch.tensor): A torch tensor of shape [batch_size,] obtained after passing the input t

    """
    output = None

    self.sigmoid_layer = torch.sigmoid(self.linear_layer(x))
    output = self.sigmoid_layer
    # raise NotImplementedError()

    return output.squeeze(-1) # Question: Why do squeeze() here?
```

```
print("Running Sample Test Cases")
torch.manual_seed(42)
d_input = 5
sample_lr_model = LogisticRegressionModel(d_input = d_input)
print(f"Sample Test Case 1: Testing linear layer input and output sizes, for d_input = {d_input}")
in_features = sample_lr_model.linear_layer.in_features
out_features = sample_lr_model.linear_layer.out_features

print(f"Number of Input Features: {in_features}")
print(f"Number of Output Features: {out_features}")
print(f"Expected Number of Input Features: {d_input}")
print(f"Expected Number of Output Features: {1}")
assert in_features == d_input and out_features == 1

print("**********************************\n")
d_input = 24
sample_lr_model = LogisticRegressionModel(d_input = d_input)
print(f"Sample Test Case 2: Testing linear layer input and output sizes, for d_input = {d_input}")
in_features = sample_lr_model.linear_layer.in_features
out_features = sample_lr_model.linear_layer.out_features

print(f"Number of Input Features: {in_features}")
print(f"Number of Output Features: {out_features}")
print(f"Expected Number of Input Features: {d_input}")
print(f"Expected Number of Output Features: {1}")
assert in_features == d_input and out_features == 1
print("**********************************\n")

print(f"Sample Test Case 3: Checking if the model gives correct output")
test_input = torch.rand(d_input)
model_output = sample_lr_model(test_input)
model_output_np = model_output.detach().numpy()
expected_output = 0.6298196315765381
print(f"Model Output: {model_output_np}")
print(f"Expected Output: {expected_output}")

assert np.allclose(model_output_np, expected_output, 1e-5)
print("**********************************\n")

print(f"Sample Test Case 4: Checking if the model gives correct output")
test_input = torch.rand(4, d_input)
model_output = sample_lr_model(test_input)
model_output_np = model_output.detach().numpy()
expected_output = np.array([0.5503339, 0.5428218, 0.561816,  0.51846  ])
print(f"Model Output: {model_output_np}")
print(f"Expected Output: {expected_output}")

assert model_output_np.shape == expected_output.shape and np.allclose(model_output_np, expected_output,
print("**********************************\n")
```

```
Running Sample Test Cases
Sample Test Case 1: Testing linear layer input and output sizes, for d_input = 5
Number of Input Features: 5
Number of Output Features: 1
Expected Number of Input Features: 5
Expected Number of Output Features: 1
**********************************

Sample Test Case 2: Testing linear layer input and output sizes, for d_input = 24
Number of Input Features: 24
Number of Output Features: 1
Expected Number of Input Features: 24
Expected Number of Output Features: 1
**********************************

Sample Test Case 3: Checking if the model gives correct output
Model Output: 0.6298196315765381
Expected Output: 0.6298196315765381
**********************************

Sample Test Case 4: Checking if the model gives correct output
Model Output: [0.5503339 0.5428218 0.561816  0.51846  ]
Expected Output: [0.5503339 0.5428218 0.561816  0.51846  ]
**********************************
```

Now that our logistic regression model seems to be defined correctly, let's initialize the model for our sentiment task.

```
sentiment_lr_model = LogisticRegressionModel(
    d_input = len(train_vocab)
)
sentiment_lr_model
```

```
    LogisticRegressionModel(
      (linear_layer): Linear(in_features=10781, out_features=1, bias=True)
    )
```

Defining a loss function

Now that we have implemented the model architecture, to train it we first need to define a loss function which we will minimize using an optimization algorithm. A loss function meausres how well the predictions of the model alligns with the actual training labels. In addition to the architecture blocks and activation functions `torch.nn` also offers a wide variety of loss functions that include:

- `torch.nn.MSELoss` : Mean squared error loss function. Typically used for regression problems.
- `torch.nn.L1Loss`: Mean absolute error loss function. Like MSE loss it is also used for regression problems. Takes the mean of absolute value of the errors instead of squared values.
- `torch.nn.BCELoss`: Binary Cross Entropy loss function. It is used in binary classification problems i.e. the classification problems where there are only 2 possible labels (positive and negative), like in our case.
- `torch.nn.CrossEntropyLoss`: Cross Entropy loss function. Similar to BCELoss but works for multi-class classification problems as well.

You can look at the documentations of these and multiple other loss functions included in `torch.nn` package here. For our purposes since we will be using `torch.nn.BCELoss`. Below we define the loss function and demonstrate the usage on an example.

```
loss_fn = nn.BCELoss()

# Demonstarting usage on a random example
torch.manual_seed(42)
example_model = LogisticRegressionModel(d_input = 5)
input = torch.rand(2, 5) # Defining a random input for demonstration
preds = example_model(input)
labels = torch.FloatTensor([1,0]) # Defining a random labels for demonstration
loss = loss_fn(preds, labels)
print(loss)
```

```
    tensor(0.8614, grad_fn=<BinaryCrossEntropyBackward0>)
```

As you can see the `loss_fn` takes as the input the prediction probabilities on a batch of inputs which are typically obtained by applying a sigmoid function to the output, and the labels corresponding to the each input. Again note that the loss also contains a `grad_fn`. We can use `loss.backward()` to compute the gradients with respect to all the model parameters

```
print(example_model.linear_layer.weight.grad)
loss.backward()
print(example_model.linear_layer.weight.grad)
```

```
    None
    tensor([[ 0.1525,  0.1364,  0.0011,  0.2294, -0.0456]])
```

Notice how before calling `loss.backward()` the gradient was None, but after the call it gets populated. The gradients will then be used by the optimizer to update the parameters, which is a nice segway to our next topic

Defining an Optimizer

An optimizer is used to find the optimum set of parameters of the model that minimize the loss functions. Most commonly used optimizers in Machine Learning, especially in Deep Learning are the variants of

Stochastic Gradient Descent (SGD), which updates the parameters of the model by moving then in the opposite direction of the gradients.

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

Here, Θ denotes the model parameters, J is the loss function and α is what we call learning rate which is used to specify the strength of the step that we wish to take. A smaller learning rate ensures we do not move away from the minima, but it makes the learning slower, while with a larger learning rate we move quickly towards the minima but are susceptible to overshooting it. Variants of SGD like SGD + Momentum, RMSProp, Adam etc., tries to improve it's noisy nature (which arises due to the fact we work on batches, instead of the entire dataset at a time), by introducing minor modifications to the update equation to prevent the optimizer taking a step in an overly wrong direction, and some also introduce heuristics to decay the step size as we reach closer to the minima. Adam is one of the most used optimizers in Deep Learning applications and often works reasonably well in practical applications. We will use the same for our experiments. You can read more about how these different optimizers work here.

`torch.optim` provides implementations for all of the optimizers we mentioned above. For official documentation for the same, refer [here](For officialy)/ Below we provide an example on how to define and use

```
from torch.optim import Adam

torch.manual_seed(42)
# We will first need to define the model
example_model = LogisticRegressionModel(d_input = 5)

# Defining the optimizer
example_optim = Adam(example_model.parameters(), lr = 1e-3)
```

Notice that the optimizer takes as input two arguments, the first is the parameters (Θ) of the model that are to be learned using the optimizer and the second is the learning rate (α). Adam optimizer also has other hyperparameters like β1 , β2, ϵ, details of which are beyond the scope of this assignment. However, you need not worry about setting these hyper-parameters as in most of the cases the default vaues of these work well enough. Next let's see how to update the model's parameters using the optimizer

```
example_optim.zero_grad() # This is done to zero-out any existing gradients stored from some previous s
input = torch.rand(2, 5) # Defining a random input for demonstration
preds = example_model(input)
labels = torch.FloatTensor([1,0]) # Defining a random labels for demonstration
loss = loss_fn(preds, labels)
loss.backward() # Perform backward pass
print(f"Parameters before the update: {example_model.linear_layer.weight}")
example_optim.step() # Update the parameters using the optimizer
print(f"Parameters after the update: {example_model.linear_layer.weight}")
```

```
    Parameters before the update: Parameter containing:
    tensor([[ 0.3419,  0.3712, -0.1048,  0.4108, -0.0980]], requires_grad=True)
    Parameters after the update: Parameter containing:
    tensor([[ 0.3409,  0.3702, -0.1058,  0.4098, -0.0970]], requires_grad=True)
```

As you can see after calling `example_optim.step()` the parameters get updated.

Task 3.3: Training the Model (2.25 Marks)

Now we have all the different components ready and can start training our sentiment classifier. Implement the `train` function below

```
import torch
import torch.nn as nn
from torch.optim import Adam

def train(model, train_dataloader,
          lr = 1e-3, num_epochs = 20,
          device = "cuda",):

  """
  Runs the training loop

  Inputs:
    - model (LogisticRegressionModel): Logistic Regression model to be trained
    - train_dataloader (torch.utils.DataLoader): A dataloader defined over the training dataset
    - lr (float): The learning rate for the optimizer
    - num_epochs (int): Number of epochs to train the model for.
    - device (str): Device to train the model on. Can be either 'cuda' (for using gpu) or 'cpu'

  Returns:
    - model (LogisticRegressionModel): Model after completing the training
    - epoch_loss (float) : Loss value corresponding to the final epoch
  """

  # Transfer the model to specified device
  model = model.to(device)

  # Step 1: Define the Binary Cross Entropy loss function
  loss_fn = None
  loss_fn = nn.BCELoss()
  # raise NotImplementedError()

  # Step 2: Define Adam Optimizer
  optimizer = None

  optimizer=Adam(model.parameters(), lr = 1e-3)
  # raise NotImplementedError()

  # Iterate over `num_epochs`
  for epoch in range(num_epochs):
    epoch_loss = 0 # We can use this to keep track of how the loss value changes as we train the model.
    # Iterate over each batch using the `train_dataloader`
    for train_batch in tqdm.tqdm(train_dataloader):
      # Zero out any gradients stored in the previous steps
      optimizer.zero_grad()

      # Unwrap the batch to get features and labels
      features, labels = train_batch

      # Most nn modules and loss functions assume the inputs are of type Float, so convert both feature
      features = features.float()
      labels = labels.float()

      # Transfer the features and labels to device
      features = features.to(device)
      labels = labels.to(device)


      # Step 3: Feed the input features to the model to get predictions
      preds = None
      preds=model(features)
      # raise NotImplementedError()

      # Step 4: Compute the loss and perform backward pass
      loss = None
      loss = loss_fn(preds, labels)
      # raise NotImplementedError()

      # Step 5: Take optimizer step
      loss.backward()
      optimizer.step()
      # raise NotImplementedError()

      # Store loss value for tracking
      epoch_loss += loss.item()

    epoch_loss = epoch_loss / len(train_dataloader)
    print(f"Epoch {epoch} completed.. Average Loss: {epoch_loss}")

  return model, epoch_loss
```

```
print("Running Sample Test Cases")

print("Training on just 100 training examples for sanity check")
torch.manual_seed(42)
sample_documents = train_df_preprocessed["sentence"].values.tolist()[:100]
sample_labels = train_df["label"].values.tolist()[:100]
sample_vocab = create_vocab(train_documents)
sample_word2idx = get_word_idx_mapping(train_vocab)

sample_dataset = SST2Dataset(sample_documents,
                            sample_labels,
                            sample_vocab,
                            sample_word2idx)

sample_dataloader = DataLoader(sample_dataset)

sample_lr_model = LogisticRegressionModel(d_input = len(sample_vocab))

sample_lr_model, loss = train(sample_lr_model, sample_dataloader,
      lr = 1e-2, num_epochs = 10,
      device = "cuda")

expected_loss = 0.1525375072658062
print(f"Final Loss Value: {loss}")
print(f"Expected Loss Value: {expected_loss}")

#assert np.allclose(expected_loss, loss, 1e-3)
```

```
Running Sample Test Cases
Training on just 100 training examples for sanity check
100%|██████████| 100/100 [00:00<00:00, 341.03it/s]
Epoch 0 completed.. Average Loss: 0.6933929485082626
100%|██████████| 100/100 [00:00<00:00, 593.67it/s]
Epoch 1 completed.. Average Loss: 0.6746722018718719
100%|██████████| 100/100 [00:00<00:00, 603.28it/s]
Epoch 2 completed.. Average Loss: 0.6548204296827316
100%|██████████| 100/100 [00:00<00:00, 607.76it/s]
Epoch 3 completed.. Average Loss: 0.6353028631210327
100%|██████████| 100/100 [00:00<00:00, 541.11it/s]
Epoch 4 completed.. Average Loss: 0.6164921063184738
100%|██████████| 100/100 [00:00<00:00, 582.67it/s]
Epoch 5 completed.. Average Loss: 0.5984839764237404
100%|██████████| 100/100 [00:00<00:00, 579.79it/s]
Epoch 6 completed.. Average Loss: 0.5812900993227959
100%|██████████| 100/100 [00:00<00:00, 627.97it/s]
Epoch 7 completed.. Average Loss: 0.5648892837762832
100%|██████████| 100/100 [00:00<00:00, 593.49it/s]
Epoch 8 completed.. Average Loss: 0.5492462494969368
100%|██████████| 100/100 [00:00<00:00, 599.66it/s]Epoch 9 completed.. Average Loss: 0.53432011961
Final Loss Value: 0.5343201196193695
Expected Loss Value: 0.1525375072658062
```

Don't worry if the exact loss values do not match, as long as your loss is reducing with epochs and the final loss is in the same range, you should be fine. And now lets train on the entire dataset, this may take some time, approximate 4 minutes per epoch. So relax and have yourself a cup of coffee while this runs

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

sentiment_lr_model = LogisticRegressionModel(
    d_input = len(train_vocab)
)
sentiment_lr_model, final_loss = train(sentiment_lr_model, train_dataloader,
      lr = 1e-2, num_epochs = 5,
      device = device)
```

```
100%|██████████| 1053/1053 [00:45<00:00, 22.99it/s]
Epoch 0 completed.. Average Loss: 0.615394465943687
100%|██████████| 1053/1053 [00:45<00:00, 23.17it/s]
Epoch 1 completed.. Average Loss: 0.5142333578576616
100%|██████████| 1053/1053 [00:45<00:00, 23.01it/s]
Epoch 2 completed.. Average Loss: 0.4553416590122303
100%|██████████| 1053/1053 [00:44<00:00, 23.91it/s]
Epoch 3 completed.. Average Loss: 0.4153921543589571
100%|██████████| 1053/1053 [00:44<00:00, 23.81it/s]Epoch 4 completed.. Average Loss: 0.3859162190
```

Task 3.4: Evaluating the Model (1.25 Marks)

Evaluation is one of the most important step in a Machine Learning pipeline, as it help us measure how well the trained is able to predict on unseen data. There are different performance metrics that can be used for evaluating machine learning algorithms. One of the most commonly used metrics for evaluating classification models is accuracy which is defined as the number of test examples predicted correctly by the model divided by the total number of test examples.

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}}$$

To check the number of correct predictions we can check the label predicted by our model and the actual label for a test example and if they both are the same. However, note that our model outputs probabilities and not the labals 0 or 1. One common practice to convert probabilities to predictions is to define a threshold ϵ (typically kept at 0.5) and if the predicted probability is > ϵ then assign the example a positive label i.e. 1 and else 0. We start by implementing `convert_probs_to_labels` function which does exactly that

```
def convert_probs_to_labels(probs, threshold = 0.5):
  """
  Convert the probabilities to labels by using the specified threshold

  Inputs:
    - probs (numpy.ndarray): A numpy 1d array containing the probabilities predicted by the classifier
    - threshold (float): A threshold value beyond which we assign a positive label i.e 1 and 0 below it

  Returns:
    - labels (numpy.ndarray): Labels obtained after thresholding

  """

  # labels = None
  labels = [1 if i > threshold else 0 for i in probs]
  # labels = ("").join(label_list)
  # raise NotImplementedError()

  return np.array(labels)
```

```
print("Running Sample Test Cases")

print("Sample Test Case 1")
```

```
sample_probs = np.array([0.1, 0.45, 0.65, 0.9, 0.55])
sample_threshold = 0.5
print(f"Input Probabilities: {sample_probs}")
print(f"Input Threshold: {sample_threshold}")
sample_labels = convert_probs_to_labels(sample_probs, sample_threshold)
expected_labels = np.array([0, 0, 1, 1, 1])
print(f"Lables: {sample_labels}")
print(f"Expected Lables: {expected_labels}")

assert (sample_labels == expected_labels).all()
print("**********************************\n")

print("Sample Test Case 2")
sample_probs = np.array([0.1, 0.45, 0.65, 0.9, 0.55])
sample_threshold = 0.75
print(f"Input Probabilities: {sample_probs}")
print(f"Input Threshold: {sample_threshold}")
sample_labels = convert_probs_to_labels(sample_probs, sample_threshold)
expected_labels = np.array([0, 0, 0, 1, 0])
print(f"Lables: {sample_labels}")
print(f"Expected Lables: {expected_labels}")

assert (sample_labels == expected_labels).all()
print("**********************************\n")
```

```
Running Sample Test Cases
Sample Test Case 1
Input Probabilities: [0.1  0.45 0.65 0.9  0.55]
Input Threshold: 0.5
Lables: [0 0 1 1 1]
Expected Lables: [0 0 1 1 1]
**********************************

Sample Test Case 2
Input Probabilities: [0.1  0.45 0.65 0.9  0.55]
Input Threshold: 0.75
Lables: [0 0 0 1 0]
Expected Lables: [0 0 0 1 0]
**********************************
```

Next lets implement the `get_accuracy` function which takes as input predicted labels and actual labels and computes the accuracy

```
def get_accuracy(pred_labels, act_labels):
  """
  Calculates the accuracy value by comparing predicted labels with actual labels

  Inputs:
    - pred_labels (numpy.ndarray) : A numpy 1d array containing predicted labels.
    - act_labels (numpy.ndarray): A numpy 1d array containing actual labels (of same size as pred_label

  Returns:
    - accuracy (float): Number of correct predictions / Total number of predictions

  """
  accuracy = None
  count = 0
  for i in range(len(pred_labels)):
    if pred_labels[i]==act_labels[i]:
      count += 1
  accuracy = count/len(pred_labels)

  # raise NotImplementedError()

  return accuracy
```

```
print("Running Sample Test Cases")
print("Sample Test Case 1")
sample_pred_labels = np.array([0, 0, 1, 1])
sample_act_labels = np.array([0, 0, 0, 1])
sample_acc = get_accuracy(sample_pred_labels, sample_act_labels)
expected_acc = 0.75
print(f"Input Predicted Labels: {sample_pred_labels}")
print(f"Input Actual Labels: {sample_act_labels}")
print(f"Accuracy: {sample_acc}")
print(f"Expected Accuracy: {expected_acc}")

assert sample_acc == expected_acc
print("**********************************\n")

print("Sample Test Case 2")
sample_pred_labels = np.array([0, 0, 1, 1, 0])
sample_act_labels = np.array([1, 1, 0, 0, 1])
sample_acc = get_accuracy(sample_pred_labels, sample_act_labels)
expected_acc = 0
print(f"Input Predicted Labels: {sample_pred_labels}")
print(f"Input Actual Labels: {sample_act_labels}")
print(f"Accuracy: {sample_acc}")
print(f"Expected Accuracy: {expected_acc}")

assert sample_acc == expected_acc

print("**********************************\n")
```

```
Running Sample Test Cases
Sample Test Case 1
Input Predicted Labels: [0 0 1 1]
Input Actual Labels: [0 0 0 1]
Accuracy: 0.75
Expected Accuracy: 0.75
**********************************

Sample Test Case 2
Input Predicted Labels: [0 0 1 1 0]
Input Actual Labels: [1 1 0 0 1]
Accuracy: 0.0
Expected Accuracy: 0
**********************************
```

Now we can implement `evaluate` function which takes in a model and a test dataloader, iterates through every batch of the test dataset and calculates the average accuracy.

```
def evaluate(model, test_dataloader, threshold = 0.5, device = "cuda"):
  """
  Evaluates `model` on test dataset

  Inputs:
    - model (LogisticRegressionModel): Logistic Regression model to be evaluated
    - test_dataloader (torch.utils.DataLoader): A dataloader defined over the test dataset

  Returns:
    - accuracy (float): Average accuracy over the test dataset
  """
  model.to(device)
  model = model.eval() # Set model to evaluation model
  accuracy = 0

  # by specifying `torch.no_grad`, it ensures no gradients are calcuated while running the model,
  # this makes the computation much more faster
  with torch.no_grad():
    for test_batch in test_dataloader:
      features, labels = test_batch
      features = features.float().to(device)
      labels = labels.float().to(device)
```

```
        # Step 1: Get probability predictions from the model and store it in `pred_probs`
        pred_probs = model(features)
        # raise NotImplementedError()

        # Convert predictions and labels to numpy arrays from torch tensors as they are easier to operate
        pred_probs = pred_probs.detach().cpu().numpy()
        labels = labels.detach().cpu().numpy()

        # Step 2: Get accuracy of predictions and store it in `batch_accuracy`
        batch_accuracy = None

        pred_labels = convert_probs_to_labels(pred_probs, threshold)
        batch_accuracy = get_accuracy(pred_labels, labels)
        # raise NotImplementedError()

        accuracy += batch_accuracy

    # Divide by number of batches to get average accuracy
    accuracy = accuracy / len(test_dataloader)

    return accuracy
```

```
print("Running Sample Test Cases")

print("Testing on just 100 test examples for sanity check")
torch.manual_seed(42)
sample_documents = test_df_preprocessed["sentence"].values.tolist()[:100]
sample_labels = test_df["label"].values.tolist()[:100]

sample_dataset = SST2Dataset(sample_documents,
                            sample_labels,
                            train_vocab,
                            train_word2idx)

sample_dataloader = DataLoader(sample_dataset, batch_size = 64)

accuracy = evaluate(sentiment_lr_model, sample_dataloader, device ="cpu")

expected_accuracy = 0.8090277777777778
print(f"Accuracy: {accuracy}")
print(f"Expected Accuracy: {expected_accuracy}")

#assert np.allclose(expected_accuracy, accuracy, 1e-5)
```

```
Running Sample Test Cases
Testing on just 100 test examples for sanity check
Accuracy: 0.765625
Expected Accuracy: 0.8090277777777778
```

Again, don't worry if the values do not match exactly. As long as the value you obtained is close to 0.8 it should be fine

Let's obtain the accuracy on the entire test set now

```
test_acc = evaluate(sentiment_lr_model, test_dataloader, device ="cuda")
print(f"Test Accuracy: {round((100* test_acc),2)}%")
```

```
Test Accuracy: 78.55%
```

We obtain around 80% accuracy form our logistic regression, which is very reasonable considering random guessing will fetch you an accuracy of ~50%. So we are doing ~30% better than random guessing which is good enough for our first model. In the coming lectures we shall see how we can improve this performance even further.

Task 3.5: Making Predictions from scratch (0.5 Marks)

Now that we have trained the model and evaluated it's performance it seems like a nice place to end, right? However, one aspect that is often overlooked in ML or NLP pipelines is designing an interface that can make prediction directly on a piece of text using the trained model, abstracting away all the pre-processing and model run details from the user. Let's implement the `predict_document` function below that does exactly that

```
def predict_document(document, model, train_vocab, train_word2idx, threshold = 0.5, device = "cpu"):
  """
  Predicts the sentiment label for the `document` using `model`

  Inputs:
    - document (str): The document whose sentiment is to be predicted
    - model (LogisticRegressionModel): A trained logistic regression model
    - train_vocab (list): Vocabulary on which the model was trained on
    - train_word2idx (dict): A Python dictionary mapping each word to its index in vocabulary

  Returns:
    - pred_label (float): Predicted sentiment of the document

  Hint: Follow the following steps:
    - preprocess the document
    - obtain bag of words features from the preprocessed document
    - convert the features to a pytorch tensor using torch.FloatTensor(features)
    - feed the features tensor to the model to obtain predicted probabilities
    - convert predicted probabilities to labels by checking if predicted probability is greater than or
  """

  model = model.to(device)
  model = model.eval()
  pred_label = None

  features = get_document_bow_feature(document,train_vocab, train_word2idx)
  features = torch.FloatTensor([features])

  pred_probs = model(features)
  pred_label = convert_probs_to_labels(pred_probs, threshold)

  # raise NotImplementedError()

  return pred_label
```

```
print("Running Sample Test Cases")

print("Sample Test Case 1")
sample_document = "this movie was great"
predicted_label = predict_document(sample_document, sentiment_lr_model,
                                   train_vocab, train_word2idx)
expected_label = 1
print(f"Predicted Label: {predicted_label}")
print(f"Expected Label: {expected_label}")

assert predicted_label == expected_label

print("**********************************\n")

print("Sample Test Case 2")
sample_document = "This movie was GREAT!!!!"
predicted_label = predict_document(sample_document, sentiment_lr_model,
                                   train_vocab, train_word2idx)
expected_label = 1
print(f"Predicted Label: {predicted_label}")
print(f"Expected Label: {expected_label}")

assert predicted_label == expected_label

print("**********************************\n")
```

```
Running Sample Test Cases
Sample Test Case 1
Predicted Label: [1]
Expected Label: 1
**********************************

Sample Test Case 2
Predicted Label: [1]
Expected Label: 1
**********************************

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:27: UserWarning: Creating a tensor f
```

Appendix: Interpreting the trained model

One of the biggest advantages of using linear models like Logistic Regression is that they are much easier to interpret compared to more sophisticated neural network models. We can just look at the weights of a trained logistic regression model and based on that can determine certain interesting insights about the model. Recall that each weight in a logistic regression corresponds to a feature in the input, and for bag of words each of the feature can be interpreted as a word from the vocabulary. Hence, a large positive weight might indicate that the corresponding word increases the probability of the document containing the positive sentiment, or an overly large negative weight will indicate otherwise. Using this, let's determine the top 10 most positive as well as most negative words. We have implemented the function `get_top_pos_nd_neg_words` to obtain these.

```
def get_top_pos_nd_neg_words(model, train_vocab, topk = 10):
  """
  Gets the `topk` most positive and negative words by interpreting the model's weights

  Inputs:
    - model (LogisticRegressionModel): A trained logistic regression model
    - train_vocab (list): Vocabulary on which the model was trained on
  """

  # Obtain model's weights
  weights = model.linear_layer.weight.data.detach().cpu().numpy().squeeze()

  # Obtain the indices corresponding to most positive and most negative weights
  weight_idx = np.argsort(weights)
  topk_pos_idxs = weight_idx[-topk:][::-1]
  topk_neg_idxs = weight_idx[:topk]

  # Get the words from indices
  topk_pos_words = [train_vocab[idx] for idx in topk_pos_idxs]
  topk_neg_words = [train_vocab[idx] for idx in topk_neg_idxs]

  topk_pos_weights = [weights[idx] for idx in topk_pos_idxs]
  topk_neg_weights = [weights[idx] for idx in topk_neg_idxs]


  return topk_pos_words,topk_pos_weights, topk_neg_words,topk_neg_weights
```

```
topk_pos_words,topk_pos_weights, topk_neg_words,topk_neg_weights = get_top_pos_nd_neg_words(sentiment_l

fig = plt.figure(figsize = (15, 5))
sns.set(font_scale=1.2)
sns.barplot(y = topk_pos_weights + topk_neg_weights,
            x = topk_pos_words +topk_neg_words)
plt.xticks(rotation=90)
plt.ylabel("Model Weights")
plt.title("Top 10 Most Positive and Top 10 Most Negative Words According to the Model")
plt.show()
```

As you can see the model assigns high weights to words like `terrific`, `refreshing`, `thoughtprovoking` etc, while assigns highly negative values to words like `worst`, `devoid`, `failur`

✓ 0s completed at 11:03 AM