Assignment 1: Word2Vec Representations (10 Marks)

Due: May 15, 2024 11:59PM IST

Welcome to the Assignment 1 of the course. This week we will learn about vector representations for words and how can we utilize them to solve the topic classification task that we discussed in the previous lab.

In [ ]:
```
data_dir = "./data/ag_news_csv" #commented out as using Github and Local Device
```

In [ ]:
```
# Install required libraries - commented out as already installed
# %pip install numpy
# %pip install pandas
# %pip install nltk
# %pip install torch
# %pip install tqdm
# %pip install matplotlib
# %pip install seaborn
# %pip install gensim
```

In [ ]:
```
# 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 gensim
import matplotlib.pyplot as plt
import seaborn as sns
import nltk

nltk.download("punkt")
nltk.download('stopwords')
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\HP\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\HP\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

Out[ ]: True

Similar to last time we will again be working on the AG News Dataset. Below we load the dataset into the memory

In [ ]:
```
NUM_LABELS = 4
LABELS_MAP = ["World", "Sports", "Business", "Sci/Tech"]

def load_dataset(split):
    ## Load the datasets and specify the column names
    df = pd.read_csv(f"{data_dir}/{split}.csv", names=["label", "title", "description"])

    ## Merge the title and description columns
    df["news"] = df["title"] + " " + df["description"]

    ## Remove the title and description columns
    df = df.drop(["title", "description"], axis=1)

    ## Have the labels start from 0
    df["label"] = df["label"] - 1

    ## Map the label to the corresponding class
    df["label_readable"] = df["label"].apply(lambda x: LABELS_MAP[int(x)])

    df = df[["news", "label", "label_readable"]]

    # Shuffle the dataset
    df = df.sample(frac=1, random_state=42).reset_index(drop=True)

    return df

## Load the datasets and specify the column names
train_df = load_dataset("train")
test_df = load_dataset("test")

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

```
Number of Training Examples: 120000
Number of Test Examples: 7600
```

In [ ]:
```
# View a sample of the dataset
train_df.head()
```

Out[ ]:

	news	label	label_readable
0	BBC set for major shake-up, claims newspaper L...	2	Business
1	Marsh averts cash crunch Embattled insurance b...	2	Business
2	Jeter, Yankees Look to Take Control (AP) AP - ...	1	Sports
3	Flying the Sun to Safety When the Genesis caps...	3	Sci/Tech
4	Stocks Seen Flat as Nortel and Oil Weigh NEW ...	2	Business

Task 0: Warm Up Excercise (2 Marks)

To start we ask you to re-implement some functions from the Lab 1. Mainly you will implement the preprocessing pipeline and vocabulary building functions again as well as some new but related functions. Details about the functions will be given in their Doc Strings.

Task 0.1: Preprocessing Pipeline (1 Mark)

Implement the preprocessing pipeline like we did in Lab1, however, this time we will only implement converting the text to lower case and removing punctuations.

We are not doing any stemming this time as we will be using pre-trained word representations in this assignment, and like it was discussed in the lectures stemming often results in the words that may not exist in common dictionaries.

We are also skipping stop words removal this time around, the reason being that removing stop words can often hurt the structural integrity of a sentence and the choice of stop words to use can be very subjective and depend upon the task at hand. For example: In the stop words list that we used last time contained the word `not`, removing which can change the sentiment of the sentence, eg. I did not like this movie -> I did like this movie. In this assignment we will explore more sophisticated ways to handle the stop words than just directly removing them from the text.

In [ ]:
```
def preprocess_pipeline(text):
    """
    Given a piece of text applies preprocessing techniques
    like converting to lower case, removing stop words and punctuations.

    Apply the functions in the following order:
    1. to_lower_case
    2. remove_punctuations

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

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

    Note: You may implement the functions for the two steps seperately in this cell
            or just write all the code in this function only we leave that up to you.
    """

    text_preprocessed = None

    # YOUR CODE HERE

    text_preprocessed = text.lower()

    for char in string.punctuation:
        text_preprocessed = text_preprocessed.replace(char, "")

    if not text_preprocessed:
        raise NotImplementedError()

    return text_preprocessed
```

In [ ]:
```
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 = "Mr. and Mrs. Dursley, of number four, Privet Drive, were proud to say that they were perfectly normal!"
test_case_answer = "mr and mrs dursley of number four privet drive were proud to say that they were perfectly 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 = "little tyke chortled mr dursley as he left the house"
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 perfectly normal!
Function Output: mr and mrs dursley of number four privet drive were proud to say that they were perfectly normal
Expected Output: mr and mrs dursley of number four privet drive were proud to say that they were perfectly normal
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 :)
**********************************
```

In [ ]:

```
## Preprocess the dataset

train_df["news"] = train_df["news"].apply(lambda x : preprocess_pipeline(x))
test_df["news"] = test_df["news"].apply(lambda x : preprocess_pipeline(x))
```

Task 0.2: Create Vocabulary (0.25 Marks)

Implement the `create_vocab` function below like you did during the lab. Do not forget using `nltk.tokenize.word_tokenize` to tokenize the text into words.

In [ ]:

```
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.
    (0.25 Marks)

    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.tokenize.word_tokenize` function may come in handy

    """

    vocab = []

    # YOUR CODE HERE
    for doc in documents:
        vocab.extend(nltk.word_tokenize(doc))
```

```python
    vocab = list(set(vocab))

    if len(vocab) == 0:
        raise NotImplementedError()

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

In [ ]:
```python
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 = ["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 games']
Function Output: ['also', 'football', 'games', 'john', 'likes', 'mary', 'movies', 'to', 'too', 'watch']
Expected Output: ['also', 'football', 'games', 'john', 'likes', 'mary', 'movies', 'to', 'too', 'watch']
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 :)
**********************************
```

In [ ]:
```python
# Create vocabulary from training data
train_documents = train_df["news"].values.tolist()
train_vocab = create_vocab(train_documents)
```

Task 0.3: Get Word Frequencies (0.75 Marks)

We define the normalized frequency of a word `w` in a corpus as:

p(w) = Number of occurences of `w` in all documents / Total Number of occurences of all words in all documents

Word frequencies can be helpful as it can help us recognize the most common words which in most cases will be stop words as well as rare words that occur in the documents. Later we will be making use of word frequencies to create sentence representations, but for now just implement the `get_word_frequencies` below

In [ ]:
```python
def get_word_frequencies(documents):
    """
    Gets the normalized frequency of each word w i.e.
    p(w) =  #num_of_occurences_of_w / #total_occurences_of_all_words
    present in documents

    Inputs:
        - documents(list): A list of documents

    Returns:
        - word2freq(dict): A dictionary containing words as keys
```

```
                            and values as their corresponding frequencies

    """

    word2freq = {}

    # YOUR CODE HERE

    # counting the number of occurences of each word
    for doc in documents:
        for word in nltk.word_tokenize(doc):
            if word in word2freq:
                word2freq[word] += 1
            else:
                word2freq[word] = 1

    # calculating the total number of words
    total_words = sum(word2freq.values())

    # normalizing the frequency of each word
    for k, v in word2freq.items():
        word2freq[k] = v / total_words


    if len(word2freq) == 0:
        raise NotImplementedError()

    return word2freq
```

In [ ]:
```
def check_dicts_same(dict1, dict2):
    if not isinstance(dict1, dict):
        print("Your function output is not a dictionary!")
        return False
    if len(dict1) != len(dict2):
        return False

    for key in dict1:
        val1 = dict1[key]
        val2 = dict2[key]
        if isinstance(val1, float) and isinstance(val1, float):
            if not np.allclose(val1, val2, 1e-4):
                return False
        if val1 != val2:
            return False

    return True

print("Running Sample Test Case 1")
sample_documents = [
    'john likes to watch movies mary likes movies too',
    'mary also likes to watch football games'
]
actual_word2freq = {'john': 0.0625,
                     'likes': 0.1875,
                     'to': 0.125,
                     'watch': 0.125,
                     'movies': 0.125,
                     'mary': 0.125,
                     'too': 0.0625,
                     'also': 0.0625,
                     'football': 0.0625,
                     'games': 0.0625}

output_word2freq = get_word_frequencies(sample_documents)
print(f"Input Documents: {sample_documents}")
print(f"Output Word Frequencies: {output_word2freq}")
print(f"Expected Word Frequencies: {actual_word2freq}")

assert check_dicts_same(output_word2freq, actual_word2freq)
print("*****************************************\n")

print("Running Sample Test Case 2")
sample_documents = [
    'We all live in a yellow submarine.',
    'Yellow submarine, yellow submarine!!'
]
actual_word2freq = {'We': 0.06666666666666667,
                    'all': 0.06666666666666667,
                    'live': 0.06666666666666667,
                    'in': 0.06666666666666667,
                    'a': 0.06666666666666667,
                    'yellow': 0.13333333333333333,
                    'submarine': 0.2,
                    '.': 0.06666666666666667,
                    'Yellow': 0.06666666666666667,
                    ',': 0.06666666666666667,
```

```
                    '!': 0.13333333333333333}

output_word2freq = get_word_frequencies(sample_documents)
print(f"Input Documents: {sample_documents}")
print(f"Output Word Frequencies: {output_word2freq}")
print(f"Expected Word Frequencies: {actual_word2freq}")

assert check_dicts_same(output_word2freq, actual_word2freq)
print("****************************************\n")
```

```
Running Sample Test Case 1
Input Documents: ['john likes to watch movies mary likes movies too', 'mary also likes to watch football games']
Output Word Frequencies: {'john': 0.0625, 'likes': 0.1875, 'to': 0.125, 'watch': 0.125, 'movies': 0.125, 'mary': 0.125, 'to
o': 0.0625, 'also': 0.0625, 'football': 0.0625, 'games': 0.0625}
Expected Word Frequencies: {'john': 0.0625, 'likes': 0.1875, 'to': 0.125, 'watch': 0.125, 'movies': 0.125, 'mary': 0.125,
'too': 0.0625, 'also': 0.0625, 'football': 0.0625, 'games': 0.0625}
****************************************

Running Sample Test Case 2
Input Documents: ['We all live in a yellow submarine.', 'Yellow submarine, yellow submarine!!']
Output Word Frequencies: {'We': 0.06666666666666667, 'all': 0.06666666666666667, 'live': 0.06666666666666667, 'in': 0.06666
666666666667, 'a': 0.06666666666666667, 'yellow': 0.13333333333333333, 'submarine': 0.2, '.': 0.06666666666666667, 'Yello
w': 0.06666666666666667, ',': 0.06666666666666667, '!': 0.13333333333333333}
Expected Word Frequencies: {'We': 0.06666666666666667, 'all': 0.06666666666666667, 'live': 0.06666666666666667, 'in': 0.066
66666666666667, 'a': 0.06666666666666667, 'yellow': 0.13333333333333333, 'submarine': 0.2, '.': 0.06666666666666667, 'Yello
w': 0.06666666666666667, ',': 0.06666666666666667, '!': 0.13333333333333333}
****************************************
```

Task 1: Word2Vec Representations

In this task you will learn how to use word2vec for obtaining vector representations for words and then how to use them further to create sentence/document level vector representations. We will be using the popular gensim package that has great support for vector space models and supports various popular word embedding methods like word2vec, fasttext, LSA etc. For the purposes of this assignment we will be working with the pretrained word2vec vectors on the google news corpus containing about 100 billion tokens. Below we provide a tutorial on how to use gensim for obtaining these word vectors.

We start by downloading pretrained word2vec vectors and create a `gensim.models.keyedvectors` obect. The download has a size of about 2GB, so might take a few minutes to download and load.

In [ ]:
```
import gensim.downloader as api
wv = api.load('word2vec-google-news-300')
```

The `wv` object has a bunch of methods that we can use to obtain vector representations of words, finding similar words etc. We start with how to obtain vectors for words using it, which can be done using the `get_vector` method as demonstrated below.

In [ ]:
```
word = "bad"
vector = wv.get_vector(word)
print(f"Word : {word}")
print(f"Length of the vector: {len(vector)}")
print(f"Vector:")
print(vector)
```

```
Word : bad
Length of the vector: 300
Vector:
[ 0.06298828  0.12451172  0.11328125  0.07324219  0.03881836  0.07910156
  0.05078125  0.171875    0.09619141  0.22070312 -0.04150391 -0.09277344
 -0.02209473  0.14746094 -0.21582031  0.15234375  0.19238281 -0.05078125
 -0.11181641 -0.3203125   0.00506592  0.15332031 -0.02563477 -0.0234375
  0.36328125  0.20605469  0.04760742 -0.02624512  0.09033203  0.00457764
 -0.15332031  0.06591797  0.3515625  -0.12451172  0.03015137  0.16210938
  0.00242615 -0.02282715  0.02978516  0.00531006  0.25976562 -0.22460938
  0.29492188 -0.18066406  0.07910156  0.02282715  0.12109375 -0.17382812
 -0.03735352 -0.06933594 -0.21972656  0.1875     -0.03320312 -0.06225586
 -0.04492188  0.11621094 -0.23339844 -0.11669922  0.09814453 -0.11962891
  0.13964844  0.28710938 -0.26953125 -0.05493164  0.03112793 -0.05029297
  0.1328125  -0.01831055 -0.37695312 -0.06298828  0.12597656 -0.07910156
 -0.04467773  0.10400391 -0.41210938  0.22851562 -0.07080078  0.24511719
  0.06494141  0.12890625 -0.05102539 -0.00308228 -0.17871094  0.25976562
 -0.13476562 -0.21289062 -0.234375    0.21777344 -0.07910156  0.01977539
  0.19726562  0.17285156  0.03613281 -0.17578125 -0.02966309 -0.00939941
  0.25976562  0.12353516  0.19140625 -0.03930664  0.15917969  0.05664062
 -0.01977539 -0.14941406  0.12597656 -0.00350952 -0.05957031 -0.14648438
  0.01660156 -0.35742188 -0.0300293   0.03149414 -0.0324707  -0.3203125
  0.35351562 -0.19433594  0.13964844  0.07470703 -0.10888672  0.10107422
 -0.296875   -0.01348877 -0.14160156  0.06982422 -0.20703125 -0.25195312
  0.03955078  0.04345703  0.05957031 -0.15429688 -0.43359375 -0.13671875
  0.00436401  0.13867188 -0.13867188 -0.125       0.00118256  0.08203125
 -0.01989746 -0.10449219  0.04638672  0.03735352  0.078125   -0.00656128
 -0.12402344 -0.3125     -0.23046875  0.0065918   0.22949219 -0.21875
  0.2421875  -0.01062012 -0.26367188  0.3359375  -0.19140625  0.02636719
 -0.0112915  -0.20898438  0.06298828 -0.07763672 -0.11572266  0.14648438
  0.10400391 -0.02819824  0.12109375 -0.11083984 -0.02893066 -0.171875
  0.1953125  -0.12451172 -0.19140625 -0.03857422 -0.01507568  0.05151367
 -0.06884766  0.07177734  0.25195312 -0.09570312  0.08251953 -0.0135498
  0.07177734 -0.27734375  0.00350952 -0.11035156 -0.15039062  0.08642578
 -0.27148438  0.10009766 -0.02746582  0.07470703  0.11865234  0.08740234
 -0.03955078  0.05004883 -0.03735352  0.03369141 -0.01977539 -0.16210938
  0.00460815 -0.0390625   0.10302734  0.18066406 -0.01495361 -0.08105469
  0.02905273 -0.02490234 -0.21875     0.04492188 -0.09472656 -0.07519531
 -0.1640625  -0.13476562  0.02111816  0.10888672 -0.08251953  0.10644531
  0.04345703 -0.1484375  -0.02038574  0.02734375 -0.11767578 -0.03735352
  0.10400391 -0.11572266  0.0546875  -0.05664062 -0.11669922  0.00180817
 -0.04736328  0.13085938 -0.00089645  0.01831055  0.13378906 -0.12060547
  0.13671875  0.05053711 -0.19238281 -0.24414062  0.02062988  0.11035156
  0.42773438  0.11572266  0.0480957  -0.11572266  0.00787354 -0.08251953
  0.03808594  0.06542969 -0.14453125 -0.13769531  0.02001953 -0.05395508
  0.17675781  0.06298828 -0.05981445 -0.25195312  0.24414062  0.17382812
  0.09619141 -0.30664062 -0.21875     0.28710938 -0.00897217  0.01818848
  0.06445312  0.01660156 -0.07177734 -0.15625     0.06738281 -0.05371094
  0.08154297  0.29101562  0.11523438 -0.02258301  0.01306152 -0.10595703
  0.19824219 -0.03393555 -0.05419922  0.07763672  0.05859375 -0.07910156
  0.09863281 -0.06054688 -0.09765625 -0.01269531 -0.12695312 -0.06982422
 -0.13574219 -0.10058594  0.01135254  0.34179688 -0.09033203  0.07666016
 -0.0324707   0.13378906 -0.15429688 -0.06347656  0.11474609  0.03100586]
```

You can also obtain the brackets by using angular brackets notation i.e. `wv["bad"]`

In [ ]:
```python
word = "bad"
vector = wv[word]
print(f"Word : {word}")
print(f"Length of the vector: {len(vector)}")
print(f"Vector:")
print(vector)
```

```
Word : bad
Length of the vector: 300
Vector:
[ 0.06298828  0.12451172  0.11328125  0.07324219  0.03881836  0.07910156
  0.05078125  0.171875    0.09619141  0.22070312 -0.04150391 -0.09277344
 -0.02209473  0.14746094 -0.21582031  0.15234375  0.19238281 -0.05078125
 -0.11181641 -0.3203125   0.00506592  0.15332031 -0.02563477 -0.0234375
  0.36328125  0.20605469  0.04760742 -0.02624512  0.09033203  0.00457764
 -0.15332031  0.06591797  0.3515625  -0.12451172  0.03015137  0.16210938
  0.00242615 -0.02282715  0.02978516  0.00531006  0.25976562 -0.22460938
  0.29492188 -0.18066406  0.07910156  0.02282715  0.12109375 -0.17382812
 -0.03735352 -0.06933594 -0.21972656  0.1875     -0.03320312 -0.06225586
 -0.04492188  0.11621094 -0.23339844 -0.11669922  0.09814453 -0.11962891
  0.13964844  0.28710938 -0.26953125 -0.05493164  0.03112793 -0.05029297
  0.1328125  -0.01831055 -0.37695312 -0.06298828  0.12597656 -0.07910156
 -0.04467773  0.10400391 -0.41210938  0.22851562 -0.07080078  0.24511719
  0.06494141  0.12890625 -0.05102539 -0.00308228 -0.17871094  0.25976562
 -0.13476562 -0.21289062 -0.234375    0.21777344 -0.07910156  0.01977539
  0.19726562  0.17285156  0.03613281 -0.17578125 -0.02966309 -0.00939941
  0.25976562  0.12353516  0.19140625 -0.03930664  0.15917969  0.05664062
 -0.01977539 -0.14941406  0.12597656 -0.00350952 -0.05957031 -0.14648438
  0.01660156 -0.35742188 -0.0300293   0.03149414 -0.0324707  -0.3203125
  0.35351562 -0.19433594  0.13964844  0.07470703 -0.10888672  0.10107422
 -0.296875   -0.01348877 -0.14160156  0.06982422 -0.20703125 -0.25195312
  0.03955078  0.04345703  0.05957031 -0.15429688 -0.43359375 -0.13671875
  0.00436401  0.13867188 -0.13867188 -0.125       0.00118256  0.08203125
 -0.01989746 -0.10449219  0.04638672  0.03735352  0.078125   -0.00656128
 -0.12402344 -0.3125     -0.23046875  0.0065918   0.22949219 -0.21875
  0.2421875  -0.01062012 -0.26367188  0.3359375  -0.19140625  0.02636719
 -0.0112915  -0.20898438  0.06298828 -0.07763672 -0.11572266  0.14648438
  0.10400391 -0.02819824  0.12109375 -0.11083984 -0.02893066 -0.171875
  0.1953125  -0.12451172 -0.19140625 -0.03857422 -0.01507568  0.05151367
 -0.06884766  0.07177734  0.25195312 -0.09570312  0.08251953 -0.0135498
  0.07177734 -0.27734375  0.00350952 -0.11035156 -0.15039062  0.08642578
 -0.27148438  0.10009766 -0.02746582  0.07470703  0.11865234  0.08740234
 -0.03955078  0.05004883 -0.03735352  0.03369141 -0.01977539 -0.16210938
  0.00460815 -0.0390625   0.10302734  0.18066406 -0.01495361 -0.08105469
  0.02905273 -0.02490234 -0.21875     0.04492188 -0.09472656 -0.07519531
 -0.1640625  -0.13476562  0.02111816  0.10888672 -0.08251953  0.10644531
  0.04345703 -0.1484375  -0.02038574  0.02734375 -0.11767578 -0.03735352
  0.10400391 -0.11572266  0.0546875  -0.05664062 -0.11669922  0.00180817
 -0.04736328  0.13085938 -0.00089645  0.01831055  0.13378906 -0.12060547
  0.13671875  0.05053711 -0.19238281 -0.24414062  0.02062988  0.11035156
  0.42773438  0.11572266  0.0480957  -0.11572266  0.00787354 -0.08251953
  0.03808594  0.06542969 -0.14453125 -0.13769531  0.02001953 -0.05395508
  0.17675781  0.06298828 -0.05981445 -0.25195312  0.24414062  0.17382812
  0.09619141 -0.30664062 -0.21875     0.28710938 -0.00897217  0.01818848
  0.06445312  0.01660156 -0.07177734 -0.15625     0.06738281 -0.05371094
  0.08154297  0.29101562  0.11523438 -0.02258301  0.01306152 -0.10595703
  0.19824219 -0.03393555 -0.05419922  0.07763672  0.05859375 -0.07910156
  0.09863281 -0.06054688 -0.09765625 -0.01269531 -0.12695312 -0.06982422
 -0.13574219 -0.10058594  0.01135254  0.34179688 -0.09033203  0.07666016
 -0.0324707   0.13378906 -0.15429688 -0.06347656  0.11474609  0.03100586]
```

Also note that the word2vec model might not have vectors for all words, you can check for Out of Vocabulary (OOV) words using the `in` operator as shown in the code block below.

In [ ]:
```python
print("book" in wv)
print("blastoise" in wv)
```

```
True
False
```

Just looking at the vectors we cannot really gain any insights about them, but it is the relation between the vectors of different words that is much more easier to interpet. `wv` object has a `most_similar` method that for a given word obtains the words that are most similar to it by computing cosine similarity between them.

In [ ]:
```python
wv.most_similar("bad",topn=5)
```

Out[ ]:
```
[('good', 0.7190051674842834),
 ('terrible', 0.6828612089157104),
 ('horrible', 0.6702597737312317),
 ('Bad', 0.669891893863678),
 ('lousy', 0.6647640466690063)]
```

In [ ]:
```python
wv.most_similar("king",topn=5)
```

Out[ ]:
```
[('kings', 0.7138045430183411),
 ('queen', 0.6510956883430481),
 ('monarch', 0.6413194537162781),
 ('crown_prince', 0.6204220056533813),
 ('prince', 0.6159993410110474)]
```

You can see that the we obtain very reasonable similar words in both examples. We can also use `most_similar` to do the analogy comparison that was discussed in the class. For eg: man : king :: woman : ?

```
In [ ]: wv.most_similar(positive=['woman', 'king'], negative=['man'], topn = 1)
```

```
Out[ ]: [('queen', 0.7118193507194519)]
```

```
In [ ]: wv.most_similar(positive=['woman', 'father'], negative=['man'], topn = 1)
```

```
Out[ ]: [('mother', 0.8462507128715515)]
```

Task 1.1 Sentence representations using Word2Vec : Bag of Words Methods (2 Marks)

Now that we know how to obtain the vectors of each word, how can we obtain a vector representation for a sentence or a document? One of the simplest way is to add the vectors of all the words in the sentence to obtain sentence vector. This is also called the Bag of Words approach. Can you think of why? Last time when we discussed bag of words features for a sentence, it contained counts of each word occuring in the sentence. This can be just thought of as just adding one hot vectors for all the words in a sentence. Hence, adding word2vec vectors for each word in the sentence can also be viewed as a bag of words representation.

Implement the `get_bow_sent_vec` function below that takes in a sentence and adds the word2vec vectors for each word occuring in the sentence to obtain the sentence vector. Also, in practice it is helpful to divide the sum of word vectors by the number of words to normalize the representation obtained.

```
In [ ]: def get_bow_sent_vec(sentence, wv):
            """
            Obtains the vector representation of a sentence by adding the word vectors
            for each word occuring in the sentence (and dividing by the number of words) i.e

            v(s) = sum_{w \in s}(v(w)) / N(s)
            where N(s) is the number of words in the sentence,
            v(w) is the word2vec representation for word w
            and v(s) is the obtained vector representation of sentence s

            Inputs:
                - sentence (str): A string containing the sentence to be encoded
                - wv (gensim.models.keyedvectors.KeyedVectors) : A gensim word vector model object.

            Returns:
                - sentence_vec (np.ndarray): A numpy array containing the vector representation
                of the sentence

            Note : Not all the words might be present in `wv` so you will need to check for that,
                   and only add vectors for the words that are present. Also while normalization
                   divide by the number of words for which a word vector was actually present in `wv`

            Important Note: In case no word in the sentence is present in `wv`, return an all zero vector!

            """

            sentence_vec = None

            # YOUR CODE HERE
            sentence_vec = np.zeros(wv.vector_size) # initializing the sentence vector
            count = 0 # initializing counter for words in wv
            for word in nltk.word_tokenize(sentence):
                if word in wv:
                    sentence_vec += wv[word]
                    count += 1
            if count > 0:
                sentence_vec /= count

            return sentence_vec
```

```
In [ ]: print("Running Sample Test Case 1")
        sample_sentence ='john likes watching movies mary likes movies too'
        sentence_vec = get_bow_sent_vec(sample_sentence, wv)
        expected_sent_vec = np.array([ 0.03330994,  0.11713409,  0.00738525,  0.24951172, -0.0202179 ])
        print(f"Input Sentence: {sample_sentence}")
        print(f"First five elements of output vector: {sentence_vec[:5]}")
        print(f"Expected first five elements of output vector: {expected_sent_vec}")
        assert np.allclose(sentence_vec[:5], expected_sent_vec, 1e-4)
        print("Sample Test Case Passed")
        print("*******************************\n")

        print("Running Sample Test Case 2")
        sample_sentence ='We all live in a yellow submarine.'
        sentence_vec = get_bow_sent_vec(sample_sentence, wv)
        expected_sent_vec = np.array([-0.08424886,  0.14601644,  0.0727946 ,  0.09978231, -0.02655029])
        print(f"Input Sentence: {sample_sentence}")
        print(f"First five elements of output vector: {sentence_vec[:5]}")
        print(f"Expected first five elements of output vector: {expected_sent_vec}")
        assert np.allclose(sentence_vec[:5], expected_sent_vec, 1e-4)
        print("Sample Test Case Passed")
```

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

print("Running Sample Test Case 3")
sample_sentence ='blastoise pikachu charizard'
sentence_vec = get_bow_sent_vec(sample_sentence, wv)
expected_sent_vec = np.array([0.,  0.,  0. ,  0., 0.])
print(f"Input Sentence: {sample_sentence}")
print(f"First five elements of output vector: {sentence_vec[:5]}")
print(f"Expected first five elements of output vector: {expected_sent_vec}")
assert np.allclose(sentence_vec[:5], expected_sent_vec, 1e-4)
print("Sample Test Case Passed")
print("******************************\n")
```

```
Running Sample Test Case 1
Input Sentence: john likes watching movies mary likes movies too
First five elements of output vector: [ 0.03330994  0.11713409  0.00738525  0.24951172 -0.0202179 ]
Expected first five elements of output vector: [ 0.03330994  0.11713409  0.00738525  0.24951172 -0.0202179 ]
Sample Test Case Passed
******************************

Running Sample Test Case 2
Input Sentence: We all live in a yellow submarine.
First five elements of output vector: [-0.08424886  0.14601644  0.0727946   0.09978231 -0.02655029]
Expected first five elements of output vector: [-0.08424886  0.14601644  0.0727946   0.09978231 -0.02655029]
Sample Test Case Passed
******************************

Running Sample Test Case 3
Input Sentence: blastoise pikachu charizard
First five elements of output vector: [0. 0. 0. 0. 0.]
Expected first five elements of output vector: [0. 0. 0. 0. 0.]
Sample Test Case Passed
******************************
```

Task 1.2 Sentence representations using Word2Vec : Inverse Frequency Weighted Sum Method (2 Marks)

Instead of directly adding the vectors for all the words in the sentence, we can do something slightly better which tends to work very well in practice. Arora et al. 2017 proposes the following method for computing sentence embedding from word vectors

$$v_s \leftarrow \frac{1}{|s|} \sum_{w \in s} \frac{a}{a + p(w)} v_w$$

Here v_w is the vector representation of the word w, p(w) is the frequency of the word w, |s| is the number of words in the sentence, and `a` is just a constant with a typical value between 1e-3 to 1e-4.

Intuitively, we take a weighted sum of all the word vectors where the weights are inversely propotional to the frequency of the word (p(w)). This ensures that very frequent words which are often stop words like "the", "I" etc. are given lower weightage when constructing the sentence vector. `a` is used as smoothing constant, such that when p(w) = 0 we still have finite weights.

In [ ]:

```
def get_weighted_bow_sent_vec(sentence, wv, word2freq, a = 1e-4):
    """
    Obtains the vector representation of a sentence by adding the word vectors
    for each word occuring in the sentence (and dividing by the number of words) i.e

    v(s) = (sum_{w \in s} a / (a + p(w)) * (v(w))) / N(s)

    Inputs:
        - sentence (str): A string containing the sentence to be encoded
        - wv (gensim.models.keyedvectors.KeyedVectors) : A gensim word vector model object.
        - word2freq (dict): A dictionary with words as keys and their frequency in the
                            entire training dataset as values
        - a (float): Smoothing constant

    Returns:
        - sentence_vec (np.ndarray): A numpy array containing the vector representation
        of the sentence

    Important Note: In case no word in the sentence is present in `wv`, return an all zero vector!

    Hint: If a word is not present in the `word2freq` dictionary, you can consider frequency
          of that word to be zero

    """

    sentence_vec = None

    # YOUR CODE HERE
    count = 0 # intializing the counter for words in wv
    sentence_vec = np.zeros(wv.vector_size) # initializing the sentence vector
```

```python
    for word in nltk.word_tokenize(sentence):
        if word not in word2freq:
            word2freq[word] = 0 # setting the frequency of the word to 0 if not present in word2freq
        if word in wv:
            sentence_vec += (a / (a + word2freq[word])) * wv[word]
            count += 1
    if count > 0:
        sentence_vec /= count

    return sentence_vec
```

In [ ]:
```python
print("Running Sample Test Case 1")
sample_sentence ='john likes watching movies mary likes movies too'
sample_word2freq = {
    "john" : 0.001,
    "likes": 0.01,
    "watching" : 0.01,
    "movies": 0.05,
    "mary" : 0.001,
    "too": 0.1
}
sentence_vec = get_weighted_bow_sent_vec(sample_sentence, wv, sample_word2freq)
expected_sent_vec = np.array([-0.00384654,  0.00208942,  0.00010824,  0.00648482, -0.00236967])
print(f"Input Sentence: {sample_sentence}")
print(f"First five elements of output vector: {sentence_vec[:5]}")
print(f"Expected first five elements of output vector: {expected_sent_vec}")
assert np.allclose(sentence_vec[:5], expected_sent_vec, 1e-4)
print("Sample Test Case Passed")
print("*******************************\n")

print("Running Sample Test Case 2")
sample_sentence ='We all live in a yellow submarine.'
sentence_vec = get_weighted_bow_sent_vec(sample_sentence, wv, word2freq = {}, a = 1e-3)
expected_sent_vec = np.array([-0.08424886,  0.14601644,  0.0727946 ,  0.09978231, -0.02655029])
print(f"Input Sentence: {sample_sentence}")
print(f"First five elements of output vector: {sentence_vec[:5]}")
print(f"Expected first five elements of output vector: {expected_sent_vec}")
assert np.allclose(sentence_vec[:5], expected_sent_vec, 1e-4)
print("Sample Test Case Passed")
print("*******************************\n")

print("Running Sample Test Case 3")
sample_sentence ='blastoise pikachu charizard'
sentence_vec = get_weighted_bow_sent_vec(sample_sentence, wv, word2freq = {}, a = 1e-3)
expected_sent_vec = np.array([0.,  0.,  0. ,  0., 0.])
print(f"Input Sentence: {sample_sentence}")
print(f"First five elements of output vector: {sentence_vec[:5]}")
print(f"Expected first five elements of output vector: {expected_sent_vec}")
assert np.allclose(sentence_vec[:5], expected_sent_vec, 1e-4)
print("Sample Test Case Passed")
print("*******************************\n")
```

```
Running Sample Test Case 1
Input Sentence: john likes watching movies mary likes movies too
First five elements of output vector: [-0.00384654  0.00208942  0.00010824  0.00648482 -0.00236967]
Expected first five elements of output vector: [-0.00384654  0.00208942  0.00010824  0.00648482 -0.00236967]
Sample Test Case Passed
*******************************

Running Sample Test Case 2
Input Sentence: We all live in a yellow submarine.
First five elements of output vector: [-0.08424886  0.14601644  0.0727946   0.09978231 -0.02655029]
Expected first five elements of output vector: [-0.08424886  0.14601644  0.0727946   0.09978231 -0.02655029]
Sample Test Case Passed
*******************************

Running Sample Test Case 3
Input Sentence: blastoise pikachu charizard
First five elements of output vector: [0. 0. 0. 0. 0.]
Expected first five elements of output vector: [0. 0. 0. 0. 0.]
Sample Test Case Passed
*******************************
```

Now that you have implemented the sentence vector functions, let's obtain sentence vectors for all the sentences in our training and test sets. This will take a few minutes

In [ ]:
```python
train_documents = train_df["news"].values.tolist()
test_documents = test_df["news"].values.tolist()
train_vocab = create_vocab(train_documents)
train_word2freq = get_word_frequencies(train_documents)

train_bow_vectors = np.array([
    get_bow_sent_vec(document, wv)
```

```
    for document in train_documents
])
test_bow_vectors = np.array([
    get_bow_sent_vec(document, wv)
    for document in test_documents
])

train_w_bow_vectors = np.array([
    get_weighted_bow_sent_vec(document, wv, train_word2freq, a = 1e-3)
    for document in train_documents
])
test_w_bow_vectors = np.array([
    get_weighted_bow_sent_vec(document, wv, train_word2freq, a = 1e-3)
    for document in test_documents
])
```

Task 2: Train a Topic Classifier using Sentence Vectors

This part will be just like Lab 1, but instead of the Bag of Word features we defined last time to train the classifier, we will use the sentence vectors obtained from word2vec.

Define a Custom Dataset class

In [ ]:
```
from torch.utils.data import Dataset, DataLoader

class AGNewsDataset(Dataset):

    def __init__(self, features, labels):
        self.features = features
        self.labels = labels

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

    def __getitem__(self, idx):
        return self.features[idx], self.labels[idx]
```

Task 2.1: Define the Multinomial Logistic Regression Model (1 Mark)

Like last time define a Multinomial Logistic Regression model that takes as input the sentence vector and predicts the label.

In [ ]:
```
import torch
import torch.nn as nn

class MultinomialLogisticRegressionModel(nn.Module):

    def __init__(self, d_input, num_labels):
        """
        Define the architecture of a Multinomial Logistic Regression classifier.
        You will need to define two components, one will be the linear layer using
        nn.Linear, and a log-softmax activation function for the output
        (log-softmax is numerically more stable and as we will see later we just need to log of the probabilities to calcu

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

        Hint: Recall that in multinomial logistic regression we obtain a `num_labels` probablilities (or log-probabilities
        value for each input that denotes how likely is the input belonging
        to each class.
        """
        #Need to call the constructor of the parent class
        super(MultinomialLogisticRegressionModel, self).__init__()

        self.linear_layer = None
        self.log_softmax_layer = None

        # YOUR CODE HERE
        self.linear_layer = nn.Linear(d_input, num_labels)
        self.log_softmax_layer = nn.LogSoftmax(dim=-1)

        if not self.linear_layer or not self.log_softmax_layer:
            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 inputs

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

        """
        output = None

        # YOUR CODE HERE
        linear = self.linear_layer(x)
        output = self.log_softmax_layer(linear)

        return output
```

In [ ]:

```
print("Running Sample Test Cases")
torch.manual_seed(42)
d_input = 5
num_labels = 4
sample_lr_model = MultinomialLogisticRegressionModel(d_input = d_input,num_labels = num_labels)
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: {4}")
assert in_features == d_input and out_features == 4

print("**********************************\n")
d_input = 24
num_labels=6
sample_lr_model = MultinomialLogisticRegressionModel(d_input = d_input,num_labels = num_labels)
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: {6}")
assert in_features == d_input and out_features == 6
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 = np.array([-1.2607676, -1.8947134, -2.0088696, -2.7715783, -2.0052252, -1.4487281])
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([-1.4812257, -1.9529424, -1.8019284, -2.575539,  -2.2114434, -1.272432 ])
print(f"Model Output: {model_output_np}")
print(f"Expected Output: {expected_output}")

assert model_output_np[0].shape == expected_output.shape and np.allclose(model_output_np[0], expected_output, 1e-5)
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: 4
Expected Number of Input Features: 5
Expected Number of Output Features: 4
**********************************

Sample Test Case 2: Testing linear layer input and output sizes, for d_input = 24
Number of Input Features: 24
Number of Output Features: 6
Expected Number of Input Features: 24
Expected Number of Output Features: 6
**********************************

Sample Test Case 3: Checking if the model gives correct output
Model Output: [-1.2607676 -1.8947134 -2.0088696 -2.7715783 -2.0052252 -1.4487281]
Expected Output: [-1.2607676 -1.8947134 -2.0088696 -2.7715783 -2.0052252 -1.4487281]
**********************************

Sample Test Case 4: Checking if the model gives correct output
Model Output: [[-1.4812257 -1.9529424 -1.8019285 -2.575539  -2.2114434 -1.272432 ]
 [-1.4630653 -1.8433273 -1.9780327 -2.5459044 -1.7756544 -1.495802 ]
 [-1.4245441 -1.9857559 -1.982151  -2.5390692 -2.0440183 -1.2877699]
 [-1.7060428 -1.8973265 -1.7597649 -2.417839  -1.9728215 -1.316079 ]]
Expected Output: [-1.4812257 -1.9529424 -1.8019284 -2.575539  -2.2114434 -1.272432 ]
**********************************
```

Task 2.2: Training and Evaluating the Model (3 Marks)

Write the training and evaluation script like the last time to train and evaluate topic classification model. You will need to write the entire functions on your own this time. You can refer to the code in Lab 1.

In [ ]:
```python
import torch
import torch.nn as nn
from torch.optim import Adam

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

    """
    Runs the training loop

    Inputs:
    - model (MultinomialLogisticRegressionModel): Multinomial 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 (MultinomialLogisticRegressionModel): Model after completing the training
    - epoch_loss (float) : Loss value corresponding to the final epoch
    """

    # YOUR CODE HERE

    # transfer the model to the device
    model = model.to(device)

    # Please note that most of the things have been used from Lab-1, and some comments pre-provided have been retained for
    # (for me personally when I review the code later)

    # Define the Binary Cross Entropy loss function
    loss_fn = nn.NLLLoss()

    # Define Adam Optimizer
    optimizer = Adam(model.parameters(), lr = lr)

    # Iterate over `num_epochs`
    for epoch in range(num_epochs):
        epoch_loss = 0 # keep track of the loss value as training proceeds

        # 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
```

```python
            # Most nn modules and loss functions assume the inputs are of type Float while the labels are expected to be o
            features = features.float()
            labels = labels.long()

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

            # Feed the input features to the model to get predictions
            preds = model(features)

            # Compute the loss and perform backward pass
            loss = loss_fn(preds, labels)
            loss.backward()

            # Take optimizer step
            optimizer.step()

            # 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
```

In [ ]:

```python
torch.manual_seed(42)
print("Training on 100 data points for sanity check")
sample_documents = train_df["news"].values.tolist()[:100]
sample_labels = train_df["label"].values.tolist()[:100]
sample_features = np.array([get_bow_sent_vec(document, wv) for document in sample_documents])
sample_dataset = AGNewsDataset(sample_features, sample_labels)
sample_dataloader = DataLoader(sample_dataset, batch_size=64)
sample_lr_model = MultinomialLogisticRegressionModel(d_input = len(sample_features[0]),num_labels=4)

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

expected_loss = 0.9724720418453217
print(f"Final Loss Value: {loss}")
print(f"Expected Loss Value: {expected_loss}")
```

```
Training on 100 data points for sanity check
100%|██████████| 2/2 [00:00<00:00, 16.26it/s]
Epoch 0 completed.. Average Loss: 1.3817952275276184
100%|██████████| 2/2 [00:00<00:00, 576.62it/s]
Epoch 1 completed.. Average Loss: 1.3134156465530396
100%|██████████| 2/2 [00:00<00:00, 434.17it/s]
Epoch 2 completed.. Average Loss: 1.2590091824531555
100%|██████████| 2/2 [00:00<00:00, 860.63it/s]
Epoch 3 completed.. Average Loss: 1.2111555337905884
100%|██████████| 2/2 [00:00<00:00, 940.95it/s]
Epoch 4 completed.. Average Loss: 1.1668499112129211
100%|██████████| 2/2 [00:00<00:00, 1938.67it/s]
Epoch 5 completed.. Average Loss: 1.1246291399002075
100%|██████████| 2/2 [00:00<00:00, 224.15it/s]
Epoch 6 completed.. Average Loss: 1.0840501189231873
100%|██████████| 2/2 [00:00<00:00, 949.04it/s]
Epoch 7 completed.. Average Loss: 1.0451085567474365
100%|██████████| 2/2 [00:00<00:00, 370.95it/s]
Epoch 8 completed.. Average Loss: 1.0078991651535034
100%|██████████| 2/2 [00:00<00:00, 290.49it/s]
Epoch 9 completed.. Average Loss: 0.9724720120429993
Final Loss Value: 0.9724720120429993
Expected Loss Value: 0.9724720418453217
```

Don't worry if the loss values do not match exactly but you should see a decreasing trend and the final value should be of the same order of magnitude

In [ ]:

```python
def evaluate(model, test_dataloader, device = "cpu"):
    """
    Evaluates `model` on test dataset

    Inputs:
    - model (MultinomialLogisticRegressionModel): 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
    - preds (np.ndarray): Predictions of the model on test dataset
    """
```

```python
    model.to(device)
    model = model.eval() # Set model to evaluation model
    accuracy = 0
    preds = []

    # YOUR CODE HERE
    with torch.no_grad():
        for test_batch in tqdm.tqdm(test_dataloader):
            features, labels = test_batch # Unwrapping each batch

            # [LAB - 1]
            features = features.float().to(device)
            labels = labels.long().to(device)

            # obtaining the predictions from the model
            output = model(features)

            # [LAB - 1] Convert predictions and labels to numpy arrays from torch tensors as they are easier to operate fo
            output = output.detach().cpu().numpy()
            labels = labels.detach().cpu().numpy()

            # Obtaining the index of the maximum value in the output - corresponds to the label
            pred = np.argmax(output, axis = 1)

            # Adding the predictions to the list
            preds.extend(pred)

            # Calculating the accuracy
            accuracy += (pred == labels).sum().item()
            # In this case, (pred == labels).sum() returns a tensor, which is a multi-dimensional array containing element
            # .sum() adds up all the elements in the tensor, resulting in a tensor with a single value.
            # .item() is then used to convert this tensor into a regular Python integer.

    # Calculating the final accuracy
    accuracy = accuracy / len(test_dataloader.dataset)


    return accuracy
```

In [ ]:
```python
print(f"Testing the sample model on 100 examples for sanity check")
torch.manual_seed(42)
sample_documents = test_df["news"].values.tolist()[:100]
sample_labels = test_df["label"].values.tolist()[:100]
sample_features = np.array([get_bow_sent_vec(document, wv) for document in sample_documents])

sample_dataset = AGNewsDataset(sample_features,
                            sample_labels)

sample_dataloader = DataLoader(sample_dataset, batch_size = 64)
accuracy = evaluate(sample_lr_model, sample_dataloader, device ="cpu")
expected_accuracy = 0.7161458333333333
print(f"Accuracy: {accuracy}")
print(f"Expected Accuracy: {expected_accuracy}")
```

```
Testing the sample model on 100 examples for sanity check
100%|██████████| 2/2 [00:00<00:00, 953.25it/s]
Accuracy: 0.73
Expected Accuracy: 0.7161458333333333
```

Now that you have implemented the training and evaluation functions, we will train (and evaluate) 2 different models and compare their performance. The 2 models are:

```
- Multinomial Logistic Regression with Bag of Word2vec features
- Multinomial Logistic Regression with Weighted Bag of Word2vec features
```

In [ ]:
```python
print(f"Training and Evaluating Multinomial Logistic Regression with Bag of Word2vec features")
device = "cuda" if torch.cuda.is_available() else "cpu"

train_labels = train_df["label"].values.tolist()
test_labels = test_df["label"].values.tolist()

train_dataset = AGNewsDataset(train_bow_vectors, train_labels)
train_loader = DataLoader(train_dataset, batch_size = 64)

test_dataset = AGNewsDataset(test_bow_vectors, test_labels)
test_loader = DataLoader(test_dataset, batch_size = 64)

lr_bow_model = MultinomialLogisticRegressionModel(
    d_input = wv.vector_size,num_labels= 4
)

lr_bow_model, loss = train(lr_bow_model, train_loader,
      lr = 1e-2, num_epochs = 10,
      device = device)
```

```
test_accuracy = evaluate(
    lr_bow_model, test_loader,
    device = device
)

print(f"Test Accuracy: {test_accuracy}")
```

```
Training and Evaluating Multinomial Logistic Regression with Bag of Word2vec features
100%|██████████| 1875/1875 [00:02<00:00, 708.13it/s]
Epoch 0 completed.. Average Loss: 0.39655469262599946
100%|██████████| 1875/1875 [00:02<00:00, 688.34it/s]
Epoch 1 completed.. Average Loss: 0.33597400794029236
100%|██████████| 1875/1875 [00:02<00:00, 655.30it/s]
Epoch 2 completed.. Average Loss: 0.3294712652762731
100%|██████████| 1875/1875 [00:02<00:00, 695.53it/s]
Epoch 3 completed.. Average Loss: 0.3269284041523933
100%|██████████| 1875/1875 [00:02<00:00, 675.79it/s]
Epoch 4 completed.. Average Loss: 0.32565709519386293
100%|██████████| 1875/1875 [00:02<00:00, 711.16it/s]
Epoch 5 completed.. Average Loss: 0.3249286182999611
100%|██████████| 1875/1875 [00:02<00:00, 699.48it/s]
Epoch 6 completed.. Average Loss: 0.32447296189069746
100%|██████████| 1875/1875 [00:02<00:00, 702.32it/s]
Epoch 7 completed.. Average Loss: 0.3241696937878927
100%|██████████| 1875/1875 [00:02<00:00, 656.49it/s]
Epoch 8 completed.. Average Loss: 0.3239580528140068
100%|██████████| 1875/1875 [00:03<00:00, 597.98it/s]
Epoch 9 completed.. Average Loss: 0.3238045896132787
100%|██████████| 119/119 [00:00<00:00, 1055.75it/s]
Test Accuracy: 0.8893421052631579
```

In [ ]:

```
print(f"Training and Evaluating Multinomial Logistic Regression with Weighted Bag of Word2vec features")
device = "cuda" if torch.cuda.is_available() else "cpu"

train_labels = train_df["label"].values.tolist()
test_labels = test_df["label"].values.tolist()

train_dataset = AGNewsDataset(train_w_bow_vectors, train_labels)
train_loader = DataLoader(train_dataset, batch_size = 64)

test_dataset = AGNewsDataset(test_w_bow_vectors, test_labels)
test_loader = DataLoader(test_dataset, batch_size = 64)

lr_bow_model = MultinomialLogisticRegressionModel(
    d_input = wv.vector_size, num_labels=4
)

lr_bow_model, loss = train(lr_bow_model, train_loader,
      lr = 1e-2, num_epochs = 10,
      device = device)

test_accuracy = evaluate(
    lr_bow_model, test_loader,
    device = device
)

print(f"Test Accuracy: {test_accuracy}")
```

```
Training and Evaluating Multinomial Logistic Regression with Weighted Bag of Word2vec features
100%|██████████| 1875/1875 [00:04<00:00, 423.00it/s]
Epoch 0 completed.. Average Loss: 0.41314818875789644
100%|██████████| 1875/1875 [00:03<00:00, 498.48it/s]
Epoch 1 completed.. Average Loss: 0.34825183312098185
100%|██████████| 1875/1875 [00:02<00:00, 634.90it/s]
Epoch 2 completed.. Average Loss: 0.3417980758865674
100%|██████████| 1875/1875 [00:03<00:00, 609.98it/s]
Epoch 3 completed.. Average Loss: 0.33926741584539416
100%|██████████| 1875/1875 [00:03<00:00, 619.20it/s]
Epoch 4 completed.. Average Loss: 0.33800648591121035
100%|██████████| 1875/1875 [00:02<00:00, 644.75it/s]
Epoch 5 completed.. Average Loss: 0.33729090749820073
100%|██████████| 1875/1875 [00:02<00:00, 655.97it/s]
Epoch 6 completed.. Average Loss: 0.3368498655796051
100%|██████████| 1875/1875 [00:02<00:00, 627.86it/s]
Epoch 7 completed.. Average Loss: 0.3365619061946869
100%|██████████| 1875/1875 [00:03<00:00, 573.48it/s]
Epoch 8 completed.. Average Loss: 0.3363656563997269
100%|██████████| 1875/1875 [00:02<00:00, 659.74it/s]
Epoch 9 completed.. Average Loss: 0.33622738288243614
```

```
100%|██████████| 119/119 [00:00<00:00, 1401.30it/s]
Test Accuracy: 0.8860526315789473
```

First thing that you can notice is that these models train substantially faster than the models in Lab 1, as now we have much more lower sized sentence representations i.e. 300, compared to last time when it was equal to the size of vocabulary i.e. around 10k!

Both models get around ~88% test accuracy, which is close to what we got with Bag of Words features in Lab 1 only. The reason we do not see much improvement in performance is because both models still take a (weighted) sum of the individual word vectors to obtain sentence vectors, and fails to encode any structural information as well as semantics properly. For eg. for sentiment analysis task, both of the following sentences:

- it was a good movie adapted from a bad book
- it was a bad movie adapted from a good book

both of these sentences will get exact similar vector representations according to both the methods and hence the model will never be able to distinguish between the sentiment of these two sentences giving same prediction for both.

In the next labs and assignments we shall see how we can learn more contextual representation of the sentences that can help us solve the task much more efficiently.