assignment-01-21307110169

March 26, 2024

0.1 Instruction

1. Rename Assignment-01-###.ipynb where ### is your student ID.
2. The deadline of Assignment-01 is 23:59pm, 03-31-2024
3. In this assignment, you will

   1) explore Wikipedia text data
   2) build language models
   3) build NB and LR classifiers

0.2 Task0 - Download datasets

Download the preprocessed data, enwiki-train.json and enwiki-test.json from the Assignment-01 folder. In the data file, each line contains a Wikipedia page with attributes, title, label, and text. There are 1000 records in the train file and 100 records in test file with ten categories.

0.3 Task1 - Data exploring and preprocessing

1) Print out how many documents are in each class (for both train and test dataset)

[4]:
```
import json
from typing import Callable

##############################################################
###          define the function we need for later use         ###
##############################################################

def load_json(file_path: str) -> list:
    """
    Fetch the data from `.json` file and concat them into a list.

    Input:
    - file_path: The relative file path of the `.json` file

    Returns:
    - join_data_list: A list containing the data, with the format of [{'title':
↪<>, 'label':<>, 'text':<>}, {}, ...]
    """
```

```python
    join_data_list = []
    with open(file_path, "r") as json_file:
        for line in json_file:
            line = line.strip()
            # guaranteen the line is not empty
            if line:
                join_data_list.append(json.loads(line))
    return join_data_list

def iterate_line_in_list(data_list: list, f: Callable) -> dict:
    """
    Iterate the `data_list` while recording the class.

    Input:
    - data_list: A list containing (train/test) data, with the format of␣
 ↪[{'title':<>, 'label':<>, 'text':<>}, {}, ...]
    - type: The type of the data, default is "train". Can take the value of␣
 ↪"train" or "test"
    - f: A function to compute the number of documents, sentences e.t.c. in␣
 ↪each `line`

    Output:
    - class_dict: A list containing dictionaries with (key, value) as (<class>,␣
 ↪<number_of_documents>)
    """
    class_dict = {}
    for line in data_list:
        line_class = line['label']
        class_dict[line_class] = class_dict.get(line_class, 0) +␣
 ↪f(line['text'])  # if the class doesn't exist, set the value as 0
    return class_dict

#################################################################
###                         end define                       ###
#################################################################

def count_docs(text):
    return 1

def print_docs_in_class(class_dict: dict, type: str = "train") -> None:
    print("The number of documents in each class for " + type + " dataset is:␣
 ↪\n")
    for _class, _times in class_dict.items():
        print("There are {:>3} documents in class {:>10}".format(_times,␣
 ↪_class))
    print('-'*60)
```

```python
# Fetch data from the json file

train_file_path, test_file_path = "enwiki-train.json", "enwiki-test.json"
train_data_list, test_data_list = map(load_json, [train_file_path,␣
 ↪test_file_path])

# print out the number of documents of each class in train and test dataset

train_docs_num = iterate_line_in_list(train_data_list, count_docs)
test_docs_num = iterate_line_in_list(test_data_list, count_docs)

print_docs_in_class(train_docs_num)
print_docs_in_class(test_docs_num, "test")
```

```
The number of documents in each class for train dataset is:

There are 100 documents in class       Film
There are 100 documents in class       Book
There are 100 documents in class Politician
There are 100 documents in class     Writer
There are 100 documents in class       Food
There are  70 documents in class      Actor
There are  80 documents in class     Animal
There are 130 documents in class   Software
There are 100 documents in class     Artist
There are 120 documents in class    Disease
------------------------------------------------------------
The number of documents in each class for test dataset is:

There are  10 documents in class       Film
There are  10 documents in class       Book
There are  10 documents in class Politician
There are  10 documents in class     Writer
There are  10 documents in class       Food
There are  10 documents in class      Actor
There are  10 documents in class     Animal
There are  10 documents in class   Software
There are  10 documents in class     Artist
There are  10 documents in class    Disease
------------------------------------------------------------
```

2) Print out the average number of sentences in each class. You may need to use sentence tokenization of NLTK. (for both train and test dataset)

[5]:
```python
from nltk.tokenize import sent_tokenize

def count_sents(text):
    return len(sent_tokenize(text))

def print_ave_sents_in_class(class_dict: dict, type: str = "train"):
    # get the dict of number of documents in each class based on the input type
    if type == "train":
        docs_num_class = train_docs_num
    elif type == "test":
        docs_num_class = test_docs_num
    else:
        raise TypeError

    # print the result
    print("The average number of sentences in each class for " + type + "␣
↪dataset is: \n")
    for _class, _times in class_dict.items():
        print("There are average {:>7.2f} sentences in class {:>10}".
↪format(_times / docs_num_class[_class], _class))
    print('-'*60)


train_ave_sents = iterate_line_in_list(train_data_list, count_sents)
test_ave_sents = iterate_line_in_list(test_data_list, count_sents)

print_ave_sents_in_class(train_ave_sents)
print_ave_sents_in_class(test_ave_sents, "test")
```

```
The average number of sentences in each class for train dataset is:

There are average  438.56 sentences in class       Film
There are average  400.36 sentences in class       Book
There are average  706.20 sentences in class Politician
There are average  420.32 sentences in class     Writer
There are average  175.24 sentences in class       Food
There are average   76.70 sentences in class      Actor
There are average   70.38 sentences in class     Animal
There are average  260.95 sentences in class   Software
There are average  306.47 sentences in class     Artist
There are average  404.90 sentences in class    Disease
------------------------------------------------------------
The average number of sentences in each class for test dataset is:

There are average  364.70 sentences in class       Film
There are average  295.90 sentences in class       Book
There are average  597.60 sentences in class Politician
```

```
There are average  294.90 sentences in class     Writer
There are average  107.60 sentences in class       Food
There are average   30.70 sentences in class      Actor
There are average   46.80 sentences in class     Animal
There are average  160.10 sentences in class   Software
There are average  234.00 sentences in class     Artist
There are average  311.70 sentences in class    Disease
------------------------------------------------------------
```

3) Print out the average number of tokens in each class (for both train and test dataset)

[6]:
```python
from nltk.tokenize import word_tokenize

def count_tokens(text):
    return len(word_tokenize(text))

def print_ave_tokens_in_class(class_dict: dict, type: str = "train"):
    # get the dict of number of documents in each class based on the input type
    if type == "train":
        docs_num_class = train_docs_num
    elif type == "test":
        docs_num_class = test_docs_num
    else:
        raise TypeError

    # print the result
    print("The average number of tokens in each class for " + type + " dataset␣
 ↪is: \n")
    for _class, _times in class_dict.items():
        print("There are average {:>8.2f} tokens in class {:>10}".format(_times␣
 ↪/ docs_num_class[_class], _class))
    print('-'*60)

train_ave_tokens = iterate_line_in_list(train_data_list, count_tokens)
test_ave_tokens = iterate_line_in_list(test_data_list, count_tokens)

print_ave_tokens_in_class(train_ave_tokens)
print_ave_tokens_in_class(test_ave_tokens, "test")
```

```
The average number of tokens in each class for train dataset is:

There are average 11895.28 tokens in class       Film
There are average 10540.51 tokens in class       Book
There are average 18644.30 tokens in class Politician
There are average 11849.91 tokens in class     Writer
There are average  3904.15 tokens in class       Food
There are average  1868.84 tokens in class      Actor
```

```
There are average  1521.92 tokens in class     Animal
There are average  6302.30 tokens in class   Software
There are average  8212.91 tokens in class     Artist
There are average  9322.96 tokens in class    Disease
------------------------------------------------------------
The average number of tokens in each class for test dataset is:

There are average  9292.90 tokens in class       Film
There are average  7711.10 tokens in class       Book
There are average 15204.30 tokens in class Politician
There are average  8499.40 tokens in class     Writer
There are average  2445.50 tokens in class       Food
There are average   677.50 tokens in class      Actor
There are average   885.60 tokens in class     Animal
There are average  3972.80 tokens in class   Software
There are average  5706.40 tokens in class     Artist
There are average  6988.80 tokens in class    Disease
------------------------------------------------------------
```

4) For each sentence in the document, remove punctuations and other special characters so that each sentence only contains English words and numbers. To make your life easier, you can make all words as lower cases. For each class, print out the first article's name and the processed first 40 words. (for both train and test dataset)

[7]:
```python
import re
from copy import deepcopy
from nltk.tokenize import sent_tokenize

def clean_doc(document: str) -> list:
    document = document.lower()
    cleaned_document = []
    sentences = sent_tokenize(document)
    for sentence in sentences:
        # remove punctuations and special characters
        sentence = re.sub(r'[^a-zA-Z0-9\s]', '', sentence)
        # remove extra whitespaces
        sentence = re.sub(r'\s+', ' ', sentence).strip()
        cleaned_document.append(sentence)
    return cleaned_document

def process_data_list(data_list: list, type: str = "train") -> list:
    explored = []
    print("The result of the " + type + " data list:")
    # process the data_list
    for line in data_list:
        class_label = line["label"]
        former_line_text = line["text"]                          # former text
```

```
        line["sentences"] = clean_doc(line["text"])             # cleaned␣
↪sentences list
        line["text"] = ". ".join(line["sentences"]) + "."       # join the␣
↪sentence list with ". " to generate the processed text
        if class_label not in explored:
            explored.append(class_label)
            # print the result
            print()
            print("The first article's name of class {:>10} is {:>20}".
↪format(class_label, line["title"]))
            print("The cleaned text is: [{}] ==> [{}]".format(former_line_text[:
↪40], line["text"][:40]))
    print("-"*120)
    return data_list

# make a deepcopy of the origin data list to avoid over-write
cleaned_train_data_list = deepcopy(train_data_list)
cleaned_test_data_list = deepcopy(test_data_list)

# process the copyed data list in place by `process_data_list`
cleaned_train_data_list = process_data_list(cleaned_train_data_list)
cleaned_test_data_list = process_data_list(cleaned_test_data_list, "test")
```

```
The result of the train data list:

The first article's name of class       Film is          Citizen_Kane
The cleaned text is: [Citizen Kane is a 1941 American drama fi] ==> [citizen
kane is a 1941 american drama fi]

The first article's name of class       Book is The_Spirit_of_the_Age
The cleaned text is: [The Spirit of the Age (full title "The S] ==> [the spirit
of the age full title the spi]

The first article's name of class Politician is    Charles_de_Gaulle
The cleaned text is: [Charles André Joseph Marie de Gaulle (; ] ==> [charles
andr joseph marie de gaulle 22 n]

The first article's name of class     Writer is        Mircea_Eliade
The cleaned text is: [Mircea Eliade (; - April 22, 1986) was a] ==> [mircea
eliade april 22 1986 was a romani]

The first article's name of class       Food is       Korean_cuisine
The cleaned text is: [
Korean cuisine has evolved through cen] ==> [korean cuisine has evolved through
centu]

The first article's name of class      Actor is       Roman_Polanski
```

```
The cleaned text is: [Roman Polanski ( ; ; born Raymond Thierr] ==> [roman
polanski born raymond thierry lieb]

The first article's name of class     Animal is      Oesophagostomum
The cleaned text is: [Oesophagostomum is a genus of parasitic ] ==>
[oesophagostomum is a genus of parasitic ]

The first article's name of class   Software is Android_(operating_system)
The cleaned text is: [Android is a mobile operating system bas] ==> [android is
a mobile operating system bas]

The first article's name of class     Artist is           Mihai_Olos
The cleaned text is: [Mihai Olos (born 26 February 1940 in Ari] ==> [mihai olos
born 26 february 1940 in arin]

The first article's name of class    Disease is    Domestic_violence
The cleaned text is: [Domestic violence (also called domestic ] ==> [domestic
violence also called domestic a]
--------------------------------------------------------------------------------
----------------------------------------
The result of the test data list:

The first article's name of class       Film is Monty_Python's_Life_of_Brian
The cleaned text is: [Monty Python's Life of Brian, also known] ==> [monty
pythons life of brian also known a]

The first article's name of class       Book is         Cousin_Bette
The cleaned text is: [La Cousine Bette (, "Cousin Bette") is a] ==> [la cousine
bette cousin bette is an 1846]

The first article's name of class Politician is    Olusegun_Obasanjo
The cleaned text is: [Chief Olusegun Matthew Okikiola Aremu Ob] ==> [chief
olusegun matthew okikiola aremu ob]

The first article's name of class     Writer is         Horia_Gârbea
The cleaned text is: [Horia-Răzvan Gârbea or Gîrbea (; born Au] ==> [horiarzvan
grbea or grbea born august 10]

The first article's name of class       Food is          Sponge_cake
The cleaned text is: [Sponge cake is a light cake made with eg] ==> [sponge cake
is a light cake made with eg]

The first article's name of class      Actor is       Kom_Chuanchuen
The cleaned text is: [Akom Preedakul (, , ; 5 January 1958 - 3] ==> [akom
preedakul 5 january 1958 30 april 2]

The first article's name of class     Animal is Articulata_hypothesis
The cleaned text is: [The Articulata hypothesis is the groupin] ==> [the
```

```
articulata hypothesis is the groupin]

The first article's name of class   Software is                 Unix
The cleaned text is: [Unix (; trademarked as UNIX) is a family] ==> [unix
trademarked as unix is a family of ]

The first article's name of class     Artist is     Camille_Pissarro
The cleaned text is: [Camille Pissarro ( , ; 10 July 1830 – 13] ==> [camille
pissarro 10 july 1830 13 novembe]

The first article's name of class    Disease is Staphylococcus_aureus
The cleaned text is: [Staphylococcus aureus is a Gram-positive] ==>
[staphylococcus aureus is a grampositive ]
--------------------------------------------------------------------------------
----------------------------------------
```

0.4 Task2 - Build language models

1) Based on the training dataset, build unigram, bigram, and trigram language models using Add-one smoothing technique. It is encouraged to implement models by yourself. If you use public code, please cite it.

[8]:
```
import nltk
from nltk.tokenize import sent_tokenize
from itertools import product
import math

'''
The framework of the class `NGramModels` follows from the repo "https://github.
↪com/joshualoehr/ngram-language-model" with some \
modification to fit into the task 1.
'''

class NGramModels(object):
    def __init__(self, n, laplace=1) -> None:
        self.n = n
        self.laplace = laplace
        self._model = None
        self._tokens = None
        self._vocab = None
        self._masks = list(reversed(list(product((0,1), repeat=n))))

    def _preprocess(self, sentences: list) -> list:
        """
        Preprocess the raw text by adding (n-1)*"<s>" (or one single <s>) on␣
↪the front of the sentence
        and replacing the tokens which occur only once with "<UNK>".
```

```
        Input:
        - sentences: A list with each element as a `sent_tokenized` sentence.

        Return:
        - tokens: A list containing the processed tokens
        """
        sos = "<s> " * (self.n - 1) if self.n > 1 else "<s> "
        tokenized_sentences = ['{}{} {}'.format(sos, sent, "</s>").split() for␣
↪sent in sentences]
        tokenized_sentences = [token for sublist in tokenized_sentences for␣
↪token in sublist]  # flatten
        # Replace tokens which appear only once in the corpus with <UNK>
        vocab = nltk.FreqDist(tokenized_sentences)
        tokens = [token if vocab[token] > 1 else "<UNK>" for token in␣
↪tokenized_sentences]
        return tokens

    def _smooth(self) -> dict:
        """
        Smooth the frequency distribution based on Laplace smoothing.

        Return:
        - A dictionary {<ngram>: <count>, ...} containing the information of␣
↪the frequency distribution
        """
        vocab_size = len(self._vocab)
        if self.n == 1:          # if n equals 1, we don't need to smooth it
            num_tokens = len(self._tokens)
            return {(unigram,): count / num_tokens for unigram, count in self.
↪_vocab.items()}
        else:
            n_grams = nltk.ngrams(self._tokens, self.n)
            n_vocab = nltk.FreqDist(n_grams)
            n_minus_one_grams = nltk.ngrams(self._tokens, self.n-1)
            n_minus_one_vocab = nltk.FreqDist(n_minus_one_grams)
            return {ngram: (n_freq + self.laplace) / (n_minus_one_vocab[ngram[:
↪-1]] + self.laplace * vocab_size) for ngram, n_freq in n_vocab.items()}

    def train(self, sentences: list) -> None:
        """
        Train the model based on the given raw text.

        Input:
        - sentences: A list with each element as a `sent_tokenized` sentence.
        """
        tokens = self._preprocess(sentences)
```

```
        self._tokens = tokens
        self._vocab = nltk.FreqDist(self._tokens)
        self._model = self._smooth()

    def _find_match(self, ngram: tuple) -> str:
        """
        Find the best match of the given ngram token in the trained model by␣
↪masking the ngram in iteration

        Input:
        - ngram: A tuple representing a test ngram token

        Return:
        - tokens: The best match of the ngram in the trained model
        """
        mask = lambda ngram, bitmask: tuple((token if flag == 1 else "<UNK>"␣
↪for token, flag in zip(ngram, bitmask)))
        possible_tokens = [mask(ngram, bitmask) for bitmask in self._masks]
        for tokens in possible_tokens:
            if tokens in self._model:
                return tokens

    def perplexity(self, test_sentences: list) -> float:
        """
        Compute the perplexity of the given `test_sentences` based on the train␣
↪tokens.

        Input:
        - test_sentences: A list containing the test sentences

        Return:
        - perplexity: The perplexity of the test material, computed by the␣
↪geomteric mean of the
                      log probabilities.
        """
        test_tokens = self._preprocess(test_sentences)
        test_ngrams = nltk.ngrams(test_tokens, self.n)
        known_ngrams  = (self._find_match((ngram,)) if isinstance(ngram, str)␣
↪else self._find_match(ngram) for ngram in test_ngrams)
        probabilities = [self._model[ngram] for ngram in known_ngrams]

        return math.exp((-1 / len(test_tokens)) * sum(map(math.log,␣
↪probabilities)))

    def _best_candidate(self, prev: tuple, i: int, blacklist: list=[]) -> tuple:
        """
```

```
        Find the best candidate from the trained model based on the previous text and blacklist.

        Input:
        - prev: A tuple containing the information of the previous text
        - i: current index
        - blacklist: A list of values that can't be taken

        Return:
        - candidate: A tuple with the format (<candidate_token>, <prob>)
        """
        blacklist += ["<UNK>"]
        candidates = [(ngram[-1], prob) for ngram, prob in self._model.items() if ngram[:-1] == prev] # find the candidates based on the trained moel
        candidates = [candidate for candidate in candidates if candidate[0] not in blacklist]          # filter out the candidate in blacklist
        candidates = sorted(candidates, key=lambda candidate: candidate[1], reverse=True)              # sort the candidates based on the prob
        if len(candidates) == 0:
            return ("</s>", 1)
        return candidates[0 if prev != () and prev[-1] != "<s>" else i]

    def generate(self, num: int, min_len: int=12, max_len: int=24):
        """
        Generate sentences based on the trained model for given number of sentences, minimum length and maximun length

        Input:
        - num: The number of sentences we need to generate
        - min_len: The minmum length of the generated sentence
        - max_len: The maximum length of the generated sentence

        Return (Yield):
        - The generated sentence one by one
        """
        for i in range(num):
            sent, prob = ["<s>"] * max(1, self.n - 1), 1
            while sent[-1] != "</s>":
                prev = () if self.n == 1 else tuple(sent[-(self.n-1):])
                blacklist = sent + (["</s>"] if len(sent) < min_len else [])
                next_token, next_prob = self._best_candidate(prev, i, blacklist)
                sent.append(next_token)
                prob *= next_prob

                if len(sent) >= max_len:
                    sent.append("</s>")
```

```python
            yield ' '.join(sent), -1/math.log(prob)

# generate the train sentences from `cleaned_train_data_list`
train_sentences = []
for each in cleaned_train_data_list:
    train_sentences.extend(each["sentences"])

# unigram language model
unigramModel = NGramModels(1)
unigramModel.train(train_sentences)
print("The unigram language model has been successfully built!")
# bigram language model
bigramModel = NGramModels(2)
bigramModel.train(train_sentences)
print("The  bigram language model has been successfully built!")
# trigram language model
trigramModel = NGramModels(3)
trigramModel.train(train_sentences)
print("The trigram language model has been successfully built!")
```

```
The unigram language model has been successfully built!
The  bigram language model has been successfully built!
The trigram language model has been successfully built!
```

2) Report the perplexity of these 3 trained models on the testing dataset and explain your findings.

[9]:
```python
# generate the test sentences from `cleaned_test_data_list`
test_sentences = []
for each in cleaned_test_data_list:
    test_sentences.extend(each["sentences"])

# compute the perplexity of the test dataset
u_perp = unigramModel.perplexity(test_sentences)
b_perp = bigramModel.perplexity(test_sentences)
t_perp = trigramModel.perplexity(test_sentences)

print("The perplexity of the testing dataset in unigram language model is {:>7.
 ↪2f}".format(round(u_perp, 2)))
print("The perplexity of the testing dataset in  bigram language model is {:>7.
 ↪2f}".format(round(b_perp, 2)))
print("The perplexity of the testing dataset in trigram language model is {:>7.
 ↪2f}".format(round(t_perp, 2)))
```

```
The perplexity of the testing dataset in unigram language model is 1155.88
The perplexity of the testing dataset in  bigram language model is 1587.09
The perplexity of the testing dataset in trigram language model is 3859.54
```

- `Unigram Model` (Perplexity: 1155.88):
  - The unigram model, despite its simplicity, achieves a relatively low perplexity. This suggests that even with minimal context (each word considered independently), the model is able to capture some of the underlying patterns within the Wikipedia text data across the 10 classes. **This could be due to the wide variety of topics covered by Wikipedia, allowing for some generalization even with a unigram model.**
- `Bigram Model` (Perplexity: 1587.09):
  - The bigram model, which considers the previous word as context, exhibits a higher perplexity compared to the unigram model. However, it still performs reasonably well, indicating that incorporating some contextual information improves prediction accuracy. The performance of the bigram model suggests that **there are significant dependencies between adjacent words within the Wikipedia text data**, contributing to the lower perplexity compared to the trigram model.
- `Trigram Model` (Perplexity: 3859.54):
  - The trigram model, with the highest perplexity among the three models, struggles more with accurately predicting the next word in the testing dataset. Despite considering two preceding words as context, the model's performance is not as effective as expected. **This could be due to data sparsity issues, especially considering the relatively small training sample size (1000) and the wide range of topics covered by Wikipedia across the 10 classes**. The trigram model might encounter challenges in capturing sufficient instances of trigrams within each class, leading to higher perplexity.
- In summary, while the unigram and bigram models demonstrate reasonable performance in capturing language patterns within the Wikipedia text data across the 10 classes, the trigram model's performance is comparatively weaker, possibly due to data sparsity and the complexity of capturing trigram dependencies within each class.

3) Use each built model to generate five sentences and explain these generated patterns.

[10]:
```python
num_sentence = 5

model_dict = {"unigram": unigramModel, "bigram": bigramModel, "trigram": trigramModel}
for _name, _model in model_dict.items():
    print("-" * 60)
    print("For the {} language model:".format(_name))
    for sentence, prob in _model.generate(num_sentence):
        print("{} ({:.5f})".format(sentence, prob))
```

```
------------------------------------------------------------
For the unigram language model:
<s> the of and in to a was that as for with </s> (0.02075)
<s> of and in to a was that as for with his </s> (0.01990)
<s> and in to a was that as for with his is of he on by it from an at be which
had or </s> (0.00895)
<s> in to a was that as for with his is he and on by it from an at be which had
or are </s> (0.00877)
<s> to a was that as for with his is he on in by it from an at be which had or
```

```
are not </s> (0.00860)
------------------------------------------------------------
For the bigram language model:
<s> the film was a new york city of his own and in which he had been found that
it is not be used </s> (0.00987)
<s> in the film was a new york city of his own </s> (0.01976)
<s> he was a new york city of the film and his own </s> (0.01801)
<s> it was a new york city of the film and his own </s> (0.01782)
<s> this is a new york city of the film was not be used to his own </s>
(0.01341)
------------------------------------------------------------
For the trigram language model:
<s> <s> the film was released on october 31 2014 a new version of windows 8 and
9 to 10 times more likely than </s> (0.00579)
<s> <s> in the united states and canada on november 22 1963 he was a member of
parliament </s> (0.00868)
<s> <s> he was a member of the film and television arts bafta awards for best
actor in his own </s> (0.00734)
<s> <s> it is not a single day of the film was released on october 31 2014 and
in his own </s> (0.00714)
<s> <s> this is the most common cause of death in a letter to his own </s>
(0.01020)
```

- `Unigram`:
  - Analysis of the generated sentences reveals a dominance of high-frequency words such as "the," "of," and "as," among others.
  - These sentences lack coherence and can be considered mere concatenations of single words.
  - Minimal semantic relationships exist between words at different positions within each sentence.
- `Bigram`:
  - Notably, phrases like "new york city," "the film," and "his own" recur in all five sentences, indicating the model's ability to learn two-word combinations from the training dataset.
  - While there is an improvement from the unigram model, the sentences generated by the bigram model still exhibit some oddness.
- `Trigram`:
  - Sentences generated by the trigram model demonstrate improved coherence and semblance of meaning.
  - However, the average probabilities of these sentences are comparatively lower, reflecting a trade-off between mirroring the training data and generalizing the language model.

0.5 Task3 - Build NB/LR classifiers

1) Build a Naive Bayes classifier (with Laplace smoothing) and test your model on test dataset

[11]:
```
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import classification_report, f1_score
```

```python
import numpy as np

class LanguageNaiveBayes(object):
    def __init__(self, laplace: int=1) -> None:
        self._data_set = None
        self._vocab = None
        self._features = None
        self._labels = None
        self._model = None
        self.laplace = laplace
        self.stopwords = [
            'a', 'about', 'above', 'across', 'after', 'afterwards', 'again',␣
 ↪'against', 'all', 'almost', 'alone',
              'along', 'already', 'also', 'although', 'always', 'am', 'among',␣
 ↪'amongst', 'amoungst', 'amount',
              'an', 'and', 'another', 'any', 'anyhow', 'anyone', 'anything',␣
 ↪'anyway', 'anywhere', 'are', 'around',
              'as', 'at', 'back', 'be', 'became', 'because', 'become',␣
 ↪'becomes', 'becoming', 'been', 'before',
              'beforehand', 'behind', 'being', 'below', 'beside', 'besides',␣
 ↪'between', 'beyond', 'bill', 'both',
              'bottom', 'but', 'by', 'call', 'can', 'cannot', 'cant', 'co',␣
 ↪'con', 'could', 'couldnt', 'cry', 'de',
              'describe', 'detail', 'did', 'do', 'does', 'doing', 'don', 'done',␣
 ↪'down', 'due', 'during', 'each', 'eg',
              'eight', 'either', 'eleven', 'else', 'elsewhere', 'empty',␣
 ↪'enough', 'etc', 'even', 'ever', 'every', 'everyone',
              'everything', 'everywhere', 'except', 'few', 'fifteen', 'fify',␣
 ↪'fill', 'find', 'fire', 'first', 'five', 'for',
              'former', 'formerly', 'forty', 'found', 'four', 'from', 'front',␣
 ↪'full', 'further', 'get', 'give', 'go', 'had',
              'has', 'hasnt', 'have', 'having', 'he', 'hence', 'her', 'here',␣
 ↪'hereafter', 'hereby', 'herein', 'hereupon',
              'hers', 'herself', 'him', 'himself', 'his', 'how', 'however',␣
 ↪'hundred', 'i', 'ie', 'if', 'in', 'inc', 'indeed',
              'interest', 'into', 'is', 'it', 'its', 'itself', 'just', 'keep',␣
 ↪'last', 'latter', 'latterly', 'least', 'less',
              'ltd', 'made', 'many', 'may', 'me', 'meanwhile', 'might', 'mill',␣
 ↪'mine', 'more', 'moreover', 'most', 'mostly',
              'move', 'much', 'must', 'my', 'myself', 'name', 'namely',␣
 ↪'neither', 'never', 'nevertheless', 'next', 'nine',
              'no', 'nobody', 'none', 'noone', 'nor', 'not', 'nothing', 'now',␣
 ↪'nowhere', 'of', 'off', 'often', 'on', 'once',
              'one', 'only', 'onto', 'or', 'other', 'others', 'otherwise',␣
 ↪'our', 'ours', 'ourselves', 'out', 'over', 'own',
```

```
             'part', 'per', 'perhaps', 'please', 'put', 'rather', 're', 's', 'same', 'see', 'seem', 'seemed', 'seeming',
             'seems', 'serious', 'several', 'she', 'should', 'show', 'side', 'since', 'sincere', 'six', 'sixty', 'so',
             'some', 'somehow', 'someone', 'something', 'sometime', 'sometimes', 'somewhere', 'still', 'such', 'system',
             't', 'take', 'ten', 'than', 'that', 'the', 'their', 'theirs', 'them', 'themselves', 'then', 'thence', 'there',
             'thereafter', 'thereby', 'therefore', 'therein', 'thereupon', 'these', 'they', 'thickv', 'thin', 'third', 'this',
             'those', 'though', 'three', 'through', 'throughout', 'thru', 'thus', 'to', 'together', 'too', 'top', 'toward',
             'towards', 'twelve', 'twenty', 'two', 'un', 'under', 'until', 'up', 'upon', 'us', 'very', 'via', 'was', 'we',
             'well', 'were', 'what', 'whatever', 'when', 'whence', 'whenever', 'where', 'whereafter', 'whereas', 'whereby',
             'wherein', 'whereupon', 'wherever', 'whether', 'which', 'while', 'whither', 'who', 'whoever', 'whole', 'whom',
             'whose', 'why', 'will', 'with', 'within', 'without', 'would', 'yet', 'you', 'your', 'yours', 'yourself',
             'yourselves'
        ]

    def train(self, data_list: list, cut_freq: int=5):
        self._vocab = self._bulid_vocab(data_list)
        self._features = self._extract_features(cut_freq)
        self._data_set, self._labels = self._convert_to_dataset(data_list)
        self._model = MultinomialNB(alpha=self.laplace)
        self._model.fit(self._data_set, self._labels)
        return self._model.score(self._data_set, self._labels)

    def test(self, data_list: list) -> tuple:
        test_dataset, test_labels = self._convert_to_dataset(data_list)
        test_pred = self._model.predict(test_dataset)
        test_score = self._model.score(test_dataset, test_labels)
        report = classification_report(test_labels, test_pred)
        return test_score, report

    def set_stopwords(self, stopwords: list[str]) -> None:
        if isinstance(stopwords, list[str]):
            self.stopwords = stopwords
        else:
            raise TypeError("The type of the stopwords should be List[str]")

    def _bulid_vocab(self, data_list: list) -> dict:
        """
```

```
        Build a vocabulary for words which have length greater than 2 and are␣
↪not in stopwords.

        Input:
        - data_list: A list with the format of [{"title": <title>, "label":␣
↪<label>, "text": <text>}, ...]

        Return:
        - vocab: A dictionary with the format of {"word": <count>, ...}, where␣
↪each word has length greater than 2 and is not in stopwords
        """
        vocab = {}
        for i in range(len(data_list)):
            sentence = data_list[i]["text"]
            for word in sentence.split():
                if len(word) > 2 and word not in self.stopwords:
                    if word not in vocab:
                        vocab[word] = 1
                    else:
                        vocab[word] += 1
        return vocab

    def _extract_features(self, cut_freq: int) -> dict:
        """
        Extract features from the vocabulary, with the threshold `cut_freq` for␣
↪frequency of a word.

        Input:
        - cut_freq: The cutting frequency of the occurrence times of a word

        Return:
        - features: A dict with (key, value) as the (<extracted_feature>,␣
↪<index of the feature>)
        """
        features = {}
        count = 0
        for key, value in self._vocab.items():
            if value >= cut_freq:
                features[key] = count
                count += 1
        return features

    def _convert_to_dataset(self, data_list: list) -> tuple:
        """
        Convert the `data_list` to `train_dataset` and `labels` which can be␣
↪accepted by the `MultinomialNB()`
```

```
        Input:
        - data_list: A list with the format of [{"title": <title>, "label": <label>, "text": <text>}, ...]

        Return:
        - dataset: A 2-d numpy array with rows and columns as the index numbers and feature indexes respectively
        - labels: The corresponding y label of the dataset
        """
        data_length = len(data_list)
        dataset = np.zeros((data_length, len(self._features)))
        labels = [0] * data_length
        for i in range(data_length):
            word_list = [word for word in data_list[i]["text"].split()]
            for word in word_list:
                if word in self._features:
                    dataset[i][self._features[word]] += 1
            labels[i] = data_list[i]["label"]
        return dataset, labels

    def _f1_score(self, data_list: list) -> tuple:
        """
        Return the micro-f1 and macro-f1 scores of the model in test dataset
        """
        test_dataset, test_labels = self._convert_to_dataset(data_list)
        micro_f1 = f1_score(test_labels, self._model.predict(test_dataset), average="micro")
        macro_f1 = f1_score(test_labels, self._model.predict(test_dataset), average="macro")
        return micro_f1, macro_f1

LMNaiveBayes = LanguageNaiveBayes(1)
train_score = LMNaiveBayes.train(cleaned_train_data_list)
test_score, report = LMNaiveBayes.test(cleaned_test_data_list)

print("The train score of the Naive Bayes model with laplace smoothing is: {:7.6f}".format(train_score))
print("The  test score of the Naive Bayes model with laplace smoothing is: {:7.6f}\n".format(test_score))
print(report)
```

```
The train score of the Naive Bayes model with laplace smoothing is: 0.998000
The  test score of the Naive Bayes model with laplace smoothing is: 0.920000

              precision    recall  f1-score   support
```

```
       Actor       1.00      0.80      0.89        10
      Animal       1.00      1.00      1.00        10
      Artist       1.00      1.00      1.00        10
        Book       1.00      0.70      0.82        10
     Disease       1.00      1.00      1.00        10
        Film       0.90      0.90      0.90        10
        Food       1.00      1.00      1.00        10
  Politician       0.71      1.00      0.83        10
    Software       1.00      1.00      1.00        10
      Writer       0.73      0.80      0.76        10

    accuracy                           0.92       100
   macro avg       0.93      0.92      0.92       100
weighted avg       0.93      0.92      0.92       100
```

2) Build a LR classifier. This question seems to be challenging. We did not directly provide features for samples. But just use your own method to build useful features. You may need to split the training dataset into train and validation so that some involved parameters can be tuned.

[12]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics import classification_report
from sklearn.pipeline import Pipeline

class LanguageLogisticRegression(object):
    def __init__(self) -> None:
        self._data_set = None
        self._labels = None
        self._model = None

    def train(self, data_list: list) -> None:
        self._data_set, self._labels = self._convert_to_dataset(data_list)
        # create the tfidf-lr pipeline, use cv to choose the best parameter
        pipeline = Pipeline([
            ('tfidf', TfidfVectorizer()),
            ('clf', LogisticRegression(max_iter=1000))
        ])
        parameters = {
            'tfidf__max_df': (0.25, 0.5, 0.75),
        }
        self._model = GridSearchCV(pipeline, parameters, cv=5, verbose=3)

        self._model.fit(self._data_set, self._labels)
        return self._model.score(self._data_set, self._labels)
```

```python
    def test(self, data_list: list) -> None:
        test_dataset, test_labels = self._convert_to_dataset(data_list)
        test_pred = self._model.predict(test_dataset)
        test_score = self._model.score(test_dataset, test_labels)
        report = classification_report(test_labels, test_pred)
        return test_score, report

    def _convert_to_dataset(self, data_list: list) -> tuple:
        """
        Convert the `data_list` to `train_dataset` and `labels` which can be␣
 ↪accepted by the `LogisticRegression()`

        Input:
        - data_list: A list with the format of [{"title": <title>, "label":␣
 ↪<label>, "text": <text>}, ...]

        Return:
        - texts: A list with each element as <text>
        - labels: The corresponding y label of the dataset
        """
        texts, labels = [], []
        for i in range(len(data_list)):
            texts.append(data_list[i]["text"])
            labels.append(data_list[i]["label"])
        return texts, labels

    def _f1_score(self, data_list: list) -> tuple:
        """
        Return the micro-f1 and macro-f1 scores of the model in test dataset
        """
        test_dataset, test_labels = self._convert_to_dataset(data_list)
        micro_f1 = f1_score(test_labels, self._model.predict(test_dataset),␣
 ↪average="micro")
        macro_f1 = f1_score(test_labels, self._model.predict(test_dataset),␣
 ↪average="macro")
        return micro_f1, macro_f1


LMLogisticRegression = LanguageLogisticRegression()
train_score = LMLogisticRegression.train(cleaned_train_data_list)
test_score, report = LMLogisticRegression.test(cleaned_test_data_list)

print("\nThe train score of the Logistic Regression model is: {:7.6f}".
 ↪format(train_score))
print("The  test score of the Logistic Regression model is: {:7.6f}\n".
 ↪format(test_score))
```

```
print(report)
```

```
Fitting 5 folds for each of 3 candidates, totalling 15 fits
[CV 1/5] END …tfidf__max_df=0.25;, score=0.900 total time=  13.0s
[CV 2/5] END …tfidf__max_df=0.25;, score=0.925 total time=  12.5s
[CV 3/5] END …tfidf__max_df=0.25;, score=0.885 total time=  13.0s
[CV 4/5] END …tfidf__max_df=0.25;, score=0.890 total time=  12.2s
[CV 5/5] END …tfidf__max_df=0.25;, score=0.895 total time=  12.0s
[CV 1/5] END …tfidf__max_df=0.5;, score=0.895 total time=  13.8s
[CV 2/5] END …tfidf__max_df=0.5;, score=0.955 total time=  12.6s
[CV 3/5] END …tfidf__max_df=0.5;, score=0.910 total time=  13.4s
[CV 4/5] END …tfidf__max_df=0.5;, score=0.900 total time=  12.8s
[CV 5/5] END …tfidf__max_df=0.5;, score=0.945 total time=  14.2s
[CV 1/5] END …tfidf__max_df=0.75;, score=0.890 total time=  12.4s
[CV 2/5] END …tfidf__max_df=0.75;, score=0.955 total time=  13.8s
[CV 3/5] END …tfidf__max_df=0.75;, score=0.920 total time=  13.6s
[CV 4/5] END …tfidf__max_df=0.75;, score=0.910 total time=  12.8s
[CV 5/5] END …tfidf__max_df=0.75;, score=0.930 total time=  14.0s

The train score of the Logistic Regression model is: 0.996000
The  test score of the Logistic Regression model is: 0.940000

              precision    recall  f1-score   support

       Actor       1.00      0.90      0.95        10
      Animal       1.00      1.00      1.00        10
      Artist       1.00      1.00      1.00        10
        Book       1.00      0.80      0.89        10
     Disease       1.00      1.00      1.00        10
        Film       0.90      0.90      0.90        10
        Food       1.00      1.00      1.00        10
  Politician       0.71      1.00      0.83        10
    Software       1.00      1.00      1.00        10
      Writer       0.89      0.80      0.84        10

    accuracy                           0.94       100
   macro avg       0.95      0.94      0.94       100
weighted avg       0.95      0.94      0.94       100
```

3) Report Micro-F1 score and Macro-F1 score for these classifiers on testing dataset explain our results.

[14]:
```
nb_micro_f1, nb_macro_f1 = LMNaiveBayes._f1_score(cleaned_test_data_list)
print("For the    Naive      Bayes Classifier: The Micro-F1 score is {:>7.6f}, ␣
↪the Macro-F1 score is {:>7.6f}".format(nb_micro_f1, nb_macro_f1))
lr_micro_f1, lr_macro_f1 = LMLogisticRegression.
↪_f1_score(cleaned_test_data_list)
```

```python
print("For the Logistic Regression Classifier: The Micro-F1 score is {:>7.6f}, 
↪the Macro-F1 score is {:>7.6f}".format(lr_micro_f1, lr_macro_f1))
```

```
For the    Naive      Bayes Classifier: The Micro-F1 score is 0.920000, the
Macro-F1 score is 0.920766
For the Logistic Regression Classifier: The Micro-F1 score is 0.940000, the
Macro-F1 score is 0.941170
```

- Micro-F1 vs. Macro-F1:
  - Both the `Naive Bayes` and `Logistic Regression` classifiers exhibit **Micro-F1** scores slightly lower than their **Macro-F1** counterparts. This discrepancy suggests that the classifiers tend to perform better on average across individual classes (as indicated by **Macro-F1**) than when considering the overall performance across all classes equally (**Micro-F1**).
- Classifier Comparison:
  - The scores of the `Logistic Regression` classifier, obtained through **cross-validation**, surpass those of the `Naive Bayes` classifier. This suggests a superior performance of the `Logistic Regression` classifier on the given dataset."