Assignment 1

Information Retrieval and Web Search
Winter 2024
Total points: 90

Issued: 01/16/2024 Due: 01/30/2024

All the code has to be your own (exceptions to this rule are specifically noted below). The code must run on the CAEN environment without additional installation or additional files (except for the data files specified in the assignment).

You can discuss the assignment with others, but the code is to be written individually. You are to abide by the University of Michigan/Engineering honor code; violations will be reported to the Honor Council.

Please use Python 3 (%python3) for the assignment. Whenever something is not specified in the assignment, that is a design choice you can make; if you make any assumptions, please include them in your write-up.

General submission instructions:

- Include all the files for this assignment in a folder called *[your-uniqname].Assignment1/* **Do not** include the data folders, i.e., *cranfieldDocs/*, *languageIdentification.data/*.
    - For instance, mihalcea.Assignment1/ will contain preprocess.py, preprocess.output, preprocess.answers,
    - languageIdentification.py,languageIdentification.output, languageIdentification.answers.
- Archive the folder using zip and submit on Canvas by the due date.
- Include your name and uniqname in each program and in all the .answers files
- Make sure all your programs run correctly on the CAEN machines.

1. [40 points] Text preprocessing

Write a Python program that preprocesses a collection of documents. You will test this program on the Cranfield dataset, which is a standard Information Retrieval text collection, consisting of 1400 documents from the aerodynamics field. The dataset cranfield.zip is available from the Files section on Canvas under Assignments/.

Programming guidelines:

Write a program called *preprocess.py* that preprocesses the collection. Assume that the documents are available in a folder whose name you will read from the command line. For testing purposes, use the *cranfieldDocs/* folder that you will obtain after unpacking the cranfield.zip archive.

Include the following functions in *preprocess.py*:

a. Function that removes the SGML tags from the input string.
Name: *removeSGML*;
Input: string;
Output: string

b. Function that tokenizes the text.
Name: *tokenizeText*;
Input: string;
Output: list (of tokens)

The tokenizer should not consider punctuation on its own as a token.

For instance:

The current population of U.S.A. is 332,087,410 as of Friday, 01/22/2021, based on Worldometer elaboration of the latest United Nations' data.

should be tokenized as

The current population of U.S.A. is 332,087,410 as of Friday 01/22/2021 based on Worldometer elaboration of the latest United Nations data

Your tokenizer should represent your best effort to correctly address the following cases, among others (cases not listed here are up to you for how to handle them):

- tokenization of . (do not tokenize acronyms, abbreviations, numbers)
- tokenization of ' (expand when needed, e.g., I'm -> I am; tokenize the possessive, e.g., Sunday's -> Sunday 's, etc.)
- tokenization of dates (keep dates together)
- tokenization of - (keep phrases separated by - together)
- tokenization of , (do not tokenize numbers)

Note that the use of regex is acceptable to the extent that your solution is not trivialized.

c. Function that performs Byte-Pair Encoding tokenization
Name: BPE;
Input: list (of tokens), vocabSize;
Output: list (of subword tokens), list (of merge rules)

This tokenizer is going to split the tokens obtained from the tokenizeText function into subwords. Please refer to BPE in the lecture slides. Your BPE implementation should contain the following parts (either as functions or as code blocks).

Note: for the purpose of this assignment, you will not perform two separate BPE training and BPE tokenization steps; you will only apply what is referred to in the slides as "BPE training"

i. calculate character frequencies, save the initial set of characters as your vocabulary
ii. calculate character pair frequencies
iii. merge the most common pair in the character frequencies, save the resulting pair in the vocabulary and save the merge rule
iv. Main loop where you perform steps ii) and iii) until you reach the desired vocabulary size

The main program should perform the following sequence of steps:
i. open the folder containing the data collection, provided as the first argument on the command line (e.g., *cranfieldDocs/),* and read one file at a time from this folder. Hint: use encoding ISO-8859-1 when opening files.
ii. for each file, apply, in order: *removeSGML*, *tokenizeText*
iii. apply *BPE* to the entire list of all the tokens resulting from all the files in the collection
iv. write code to determine and list (this is <u>after</u> step ii above):

- the total number of merge rules learned by your BPE tokenizer
- the first 20 merge rules learned by your BPE tokenizer
- most frequent 50 BPE tokens in the collection, along with their frequencies (list in descending order of their frequency, i.e., from most to least frequent)

The *preprocess.py* program should be run using a command like this:
% *python3 preprocess.py cranfieldDocs/ vocabSize*

It should produce a file called *preprocess.output* with the following content (tokens are to be listed in descending order of their frequency; ties can be listed in any order):

Tokens [total-number-of-BPE-tokens]
Merge rules [total-number-of-merge-rules]
The first 20 merge rules
(tok1, tok2) -> tok1tok2
….
Top 50 tokens
Tok1 [frequency-of-Tok1]
…
Tok50 [frequency-of-Tok50]

For the assignment submission, include the file *preprocess.output* produced for a vocabSize of 10000

<u>Write-up guidelines:</u>
Create a text file called *preprocess.answers*, and include the following information, for a target vocabulary of 10000:

- Total number of BPE tokens in the Cranfield collection
- Total number of merge rules
- The minimum number of unique BPE tokens in the Cranfield collection accounting for 25% of the total number of BPE tokens in the collection?
  Example: if the total number of tokens in the collection is 100,
  and we have the following token-frequency pairs: airplane - 30 space - 10
  clear - 8 cut - 7 etc. the answer to this question will be 1 (1 token accounts for 25% of the total 100 tokens)

2. [50 points] Language identification.

Implement a language identifier, using the letter-based bigram probability model discussed in class, with add-one smoothing. The dataset to be used for this assignment is included in languageIdentification.data.zip, available from the Files section on Canvas, under Assignments/.

Programming guidelines:

Once you unpack the data archive, you will obtain a folder called *languageIdentification.data/*. Store this data folder in the same folder as your program. There are five files included in the *languageIdentification.data/* folder: three files with data to use for training (*English*, *French*, *Italian,* stored under a subfolder called *training/*), one file with data to use for test (*test*), and one solution file to use for evaluation (*solution*).

Write a Python program called *languageIdentification.py*. The program should include the following functions:

a. Function to train a bigram language model
Name: *trainBigramLanguageModel*
Input: string (the training text in a given language);
Output: two dictionaries: one dictionary with character-unigram frequencies collected from the string; one dictionary with character-bigram frequencies collected from the string

Given an input (training) string, this function will calculate the frequencies for all the single characters and for all the bigram characters in the string.

b. Function to determine the language of a string.
Name: *identifyLanguage*
Input: string (the test text for which the language is to be identified); list of strings (each string corresponding to a language name); list of dictionaries with unigram character frequencies (each dictionary corresponding to the single character frequencies in a language); list of dictionaries with bigram character frequencies (each dictionary corresponding to the bigram character frequencies in a language);
Output: string (the name of the most likely language).
Note: in the input lists, elements at a given position K in the lists correspond to the same language L.

Given an input (test) string and the unigram and bigram frequencies for the languages being considered as candidates, this function will determine the most likely language by (i) calculating the bigram probability of the input string, and (ii) determining the language that leads to the highest probability.

The *main* program should perform the following sequence of steps:
i. Use the *trainBigramLanguageModel* function to build unigram and bigram dictionaries for each of the three language files provided as training.
ii. Open the test file, provided as the first argument on the command line, and for each line in the test file, apply the *identifyLanguage* function.

The *languageIdentification.py* program should be run using a command like this:
% python languageIdentification.py [path to training data folder] [test file]

E.g.,
%python languageIdentification.py
languageIdentification.data/training/
languageIdentification.data/test

It should produce a file called *languageIdentification.output*, with the following content:

Line1 Language1
Line2 Language2
…
Line300 Language300

Where LineN represents the line number in the test file, and LanguageN is the language determined by the *identifyLanguage* function as the most likely one for that particular line in the test file. For example:
1 English
2 French
…
300 Italian

Write-up guidelines:
Create a text file called *languageIdentification.answers*, and include the following information:
- Accuracy of your language identifier when comparing the output of your system with the solution provided? (in other words, what is the percentage of predictions that are correct)