CIS*4500 (Fall 2019) Machine Learning for Natural Language Processing

Assignment One

Due on Sunday, Oct. 6, 2019 by 11:59 pm

General Description:

For many NLP tasks, a document collection often needs to be preprocessed and analyzed in order to know the characteristics of the dataset and apply suitable machine learning techniques. As described in the lectures, there can be many steps involved in the process, depending on the tasks to be performed as well. In this assignment, we will take one day's worth of news articles from a daily newspaper as our dataset and implement several programs for data preprocessing and analysis. The news articles were originally formatted with SGML tags but have been simplified into the following text format that contains only the required components. Each article starts with the label "`$DOC`", followed by its ID such as "`LA010190-0001`" on the same line. After the label "`$TITLE`", we can have multiple lines of text as the title, and after the label "`$TEXT`", we can have multiple lines of text as the content of the article.

```
$DOC LA010190-0001
$TITLE
NEW YEAR'S DAY
$TEXT
Today is New Year's Day. The following will be closed:
MAIL: All post offices will be closed and no mail will be
delivered.
BANKS: Most banks and savings and loan institutions will be
closed.
COURTS: All courts will be closed.
SCHOOLS: Most public schools in Los Angeles County -- including
schools in them Los Angeles Unified School District -- are
closed until Jan. 2
GOVERNMENT: All federal, state, county and municipal offices --
except those
providing emergency services, will be closed.
$DOC LA010190-0002
$TITLE
COUNTY'S PLAN IS GARBAGE;
TRASH DISPOSAL: IF SOMETHING ISN'T DONE QUICKLY, LOS ANGELES
CITY RESIDENTS
WILL END UP MAKING UNJUSTIFIED CONCESSIONS.
$TEXT
The County of Los Angeles, which has historically embraced some
of the nation's
sharpest land swindles, is out to break new ground in its legacy
of shady real-estate deals.
...
```

Given the kind of input above, you are asked to implement four separate programs for data preprocessing and analysis: (1) split the text of documents into sentences; (2) tokenize the sentences into sequences of words; (3) associate the words with their POS tags; and (4) analyze the dataset to get some statistics about the dataset so that we can know its characteristics. All the implementation should be done in Java and related tools. In our CourseLink account, you will be provided with a compressed folder that contains the dataset as "articles.txt", two sample programs for sentence splitting and POS-tagging using Apache OpenNLP toolkit, and a sample scanner for a very simple programming language. *You are allowed to use and adapt the given sample code for the implementation of this assignment. Just acknowledge it in the README file of your submission.*

Sentence Splitting:

Sentence splitting (also called sentence segmentation or sentence boundary detection) involves the disambiguation of three common end-of-sentence marks, including period ("`.`"), question ("`?`"), and exclamation ("`!`") so that a body of text can be split into a sequence of sentences. It's an NLP task on its own, but for this assignment, we will rely on Apache OpenNLP tools to simplify our implementation. The OpenNLP toolkit is implemented in Java and supports many common NLP tasks such as language identification, sentence splitting, part-of-speech tagging, named entity recognition, parsing, and coreference resolution. Among the sample programs provided in our CourseLink account, "SentenceDetectionME.java" will store a body of text as a string with all newlines replaced by spaces, and output an array of strings, one for each sentence. To use this program, you will need a jar file for OpenNLP tools and a pre-trained model for English in the folder "OpenNLP_models".

For this assignment, we will do sentence splitting for both title and body texts. The output file will be in the same format as the input file except that the corresponding texts will be replaced by sequences of sentences, one per line in the output file.

Tokenization:

Since different datasets often require different patterns for their token categories, it's generally more flexible and useful to use a scanner-generation tool such as JFlex to do tokenization for them. For this assignment, you need to write regular expressions for the following token categories:

- LABEL: the three common labels (`$DOC`, `$TITLE`, and `$TEXT`) used to indicate the structure of a document.
- WORD: strings of letters and digits separated by spaces, tabs, newlines, and most of the punctuation marks. For example, "`John`", "`computer`", "`mp3`", "`123abc`" are all treated as WORD tokens.
- NUMBER: integers and real numbers, with possible positive and negative signs.

- APOSTROPHIZED: words like "`John's`", "`O'Reily`", and "`O'Reily's`" should be treated as single tokens. However, "`world'cup`" and "`this'is'just'a'test`" are likely to be typos and should be split further. Due to the longest possible match in JFlex, however, it's hard to separate these two cases in a scanner. As a result, you can treat strings like "`world'cup`" and "`this'is'just'a'test`" as APOSTROPHIZED for now.
- HYPHENATED: words such as "`data-base`" and "`father-in-law`" should be treated as single tokens. However, "`---`" should be treated as a sequence of punctuation marks. Note that when a hyphenated token is ended with an apostrophized suffix, it will be classified as APOSTROPHIZED such as "`father-in-law's`". Again, due to the longest possible match, you can write a general pattern so that strings like "`this-is-just-a-test`" as HYPHENATED for now.
- NEWLINE: capture all the newlines explicitly so that we can preserve the line structures in the input file for output.
- PUNCTUATION: Any symbol that does not contribute to the tokens above should be treated as a punctuation mark.

Once again for this assignment, we will do tokenization for both title and body texts. The output file will be in the same format as the file generated from the Sentence Splitting step above except that the sentences on each line will be replaced by sequences of tokenized words while keeping the original line structure for each document. Such an output file can be generated from "Scanner.java" in the sample scanner provided in our CourseLink account since it allows us to access the type and value of each token from the scanner. In addition, Scanner.java provides us an opportunity to fix some of the issues with APOSTROPHIZED and HYPHENATED tokens described above. Once we get a token of these two types, we can examine its string value, and split it if necessary, based on the following two rules:

(1) For a HYPHENATED token, it should only contain two or three parts when separated by the hyphens, and if there are three parts, the middle part can only have one or two characters. Otherwise, the token should be split into a sequence of tokens. Such a rule will keep strings like "`data-base`" and "`father-in-law`" as single tokens, but split strings like "`this-is-just-a-test`" into sequences of tokens such as "`this - is - just - a - test`".

(2) For an APOSTROPHIZED token, it should only contain two or three parts when separated by the apostrophes. If there are two parts, either the first part contains a single character and the second part contains more than two characters, or the last part contains the character '`s`'. If there are three parts, the first part should contain only one character and the last part contains the character '`s`'. Otherwise, the token should be split into a sequence of tokens. For example, "`John's`", "`O'Reily`", and "`O'Reily's`" will be treated as single tokens. When splitting an APOSTROPHIZED token with two parts, we need to keep the apostrophe with the second part if it has one or two characters; otherwise, add spaces on both sides of the apostrophes. For example, we will split "`You're`" and "`I've`" to "`You 're`" and "`I 've`", but "`world'cup`" will

be split into a sequence of three tokens "`world ' cup`". Note that the above two rules may be tested together so that "`father-in-law's`" is treated as one token, but "`this-is-just-a-test's`" should be split into a sequence of tokens "`this - is - just - a - test ' s`".

POS-Tagging:

In our CourseLink account, a sample program "TokenizerMEDemo.java" shows how to use the POS-tagger in OpenNLP to tag a sequence of words with their POS tags such as "`two/CD cancer-stricken/JJ smokers/NNS`". For this assignment, your program will read one sentence at a time and split it into an array of tokens before calling the POS-tagger. The output is the same as the file generated from the Tokenization step above except all the tokens in a sentence are paired up with their POS tags.

Data Analysis:

Based on the file generated from the POS-Tagging step above, you are required to perform the following data analysis: (1) How many documents are there in the data collection? (2) What are the min, avg, and max document lengths by the number of sentences? (3) What are the min, avg, and max document lengths by the number of tokens? and (4) What are the avg sentence lengths by the number of tokens for the whole data collection as well as all individual documents. Note that each document should consist of both title and body texts, and you can use any format to show these values as long as all the related information can be easily identified in the output file.

Guidelines for the Implementation:

- Your regular expressions for the Tokenization step should be as general and descriptive as possible, defining the largest set of possible matches. For example, a number should include any reasonable representation of an integer or a float, either positive or negative.

- Your programs should consume input from stdin until EOF is read and send output to stdout, and do not prompt for input.

- You are responsible for creating a Makefile to compile your programs. Typing "make clean" will remove all the compiled and generated files, and typing "make" or "make all" should create a new build for all the compiled programs. You should ensure that all required dependencies are specified correctly.

Guidelines for the Submission:

The following should be handed in for your assignment submissions:

- Source Code: Turn in all source code you have written or used along with any other files necessary to compile and run your program (such as Makefile and datafiles). Source code should be appropriately commented and follow reasonable style guidelines (see below).

- Documentation: a file called README should always be present where you can describe the general problem you are trying to solve; what are the assumptions and limitations of your solution; how can a user build and test your program; how is the program tested for correctness (i.e., the test plan should be part of the README file); and what possible improvements could be done if you were to do it again or have extra time available.
- Testing Environment: For the purpose of marking, we will run your programs on a Linux server at linux.socs.uoguelph.ca where both Java and JFlex are installed, and you are strongly encouraged to test your code on this server before uploading your submissions.
- Late policy: please refer to the updated course outline for details.

Steps for submitting your assignment:

1. Organize all of your files into a directory named by the pattern "<userid>_a1" (e.g., my email userid is "fsong" and I will name the directory "fsong_a1").

2. Run tar and gzip to the above directory to create a single compressed file using the following commands on Linux:
   "tar -cvf fsong_a1.tar fsong_a1" and "gzip fsong_a1.tar" to create "fsong_a1.tar.gz",
   or simply "tar –czvf fsong_a1.tgz fsong_a1" to create "fsong_a1.tgz".

3. Upload the compressed file into the corresponding dropbox on CourseLink by the specified due time.

4. Verify that your submission is indeed uploaded successfully by downloading your compressed file from CourseLink to your local machine and examine the details. You can expand your compressed file using the following Linux commands:
   "gzip –d fsong_a1.tar.gz" and "tar –xvf fsong_a1.tar" to recreate "fsong_a1" directory,
   or "tar –xzvf fsong_a1.tgz" to do the same.

Coding Style Guidelines:

- Each source file should have a comment header, describing the purpose of the related file; and similarly, each method should have a comment header, describing the purpose of the related function.
- All identifiers for files, variables, and functions should have descriptive names (with the possible exception of index variables for loops).
- Literal constants (or so-called "magic numbers") should be avoided; instead use named constants for counting limits, array sizes, and so forth.
- The indentation and use of whitespaces should be consistent throughout the code.
- A method should ideally be a small block of code that performs a single task. If a method is getting too long or is doing too many things, you should probably divide it into several methods.
- Other commenting should be appropriate (e.g., you would not comment an increment statement as it is too obvious) and helpful for someone to understand your code (usually highlighting the major steps).