CS159 - Assignment 1
Data Analysis and Regular Expressions

Due: Thursday, September 11, 2:45pm

`http://www.smbc-comics.com/index.php?db=comics&id=2884`

Regular expressions are one of the more useful tools for many NLP tasks. The goals of this assignment are to familiarize yourself with regular expressions, to play with them in different languages/environments, to get some experience with some useful command-line tools and to get a feeling for text data analysis.

You will only be handing in a write-up for this assignment, however, it will involve coding and experimentation. You will not hand-in any code, but you are still expected to write the code yourself (we will do group projects soon) and should not be copying material from online, etc. If you have any question about what is appropriate, feel free to come talk to me.

Before starting, you may find it useful to read the notes section at the end of this handout which discusses `grep` and `sed` as well as review some of the links posted with the notes.

1. String matching **[10 points]**

   For all of the following problems you MUST use regular expressions exclusively.

   (a) Write two Java functions that take as input a `String` and return a `boolean` indicating whether that string is a valid social security number. They should both use regular expressions and should just be one or two lines of code.

      i. To start with, we'll define a social security number (SSN) as a number consisting of 9 digits. Optionally, the SSN may be separated into 3 digits, 2 digits and 4 digits by two dashes '-' (note that it must either have 2 dashes or none).

      ii. The first version isn't exactly correct for social security numbers. Currently, the first three digits of the SSN must be less than or equal to 772[1], though the other 6 numbers can be any number. Write a second Java function that enforces this additional restriction. For simplicity, you may assume that for this version the user must enter dashes.

   Include these two functions in your writeup.

   (b) For this problem, we're going to be analyzing twitter posts. I've put a file with 100K tweets on the Pomona CS network at:

   `/common/cs/cs159/assignments/assignment1/twitter.posts.gz`

   Write a statement using `grep` that selects all the lines in the file that contain retweets. A retweet is signified by the characters "rt" (all capitalization variants) or the word "Retweet", optionally follow by a colon, optionally followed by some number of spaces followed by an '@', directly followed by a username, which consists of only alphanumeric characters and concluded with either whitespace, a colon or the end of the string. A retweet must be preceded by a space or be at the beginning of the string, so be careful not to match something like "blurt @bobby".

   The following are examples of valid retweets:

   RT @profkauchak ...
   ... rt @steve471
   blah blah Retweet:@theman blah blah

   Include your `grep` statement in your writeup as well as your answers to the following questions:

      i. Of the 100K tweets, how many included a retweet? (*Hint:* The `wc` command might be useful for counting.)

      ii. How many used "RT" to denote the retweet? "Rt"? "rt"? "Retweet"?

   A quick overview of `grep` can be found at the end of this handout.

2. String replacement **[10 points]**

   (a) HTML processing

[1] along with a few other restrictions we won't worry about

Using `sed`, write one command-line expression, i.e. that you type it all and then press return only once (though you can use multiple `sed` statements chained together with pipes), to do all of the following to an input file:

- remove all the html tags from the document (an html tag is any text surrounded by '<>' and may contain spaces). Be careful about greedy matching. Both '*' and '+' try and match as much as they can, so if you just use them with say '<.*>', it can match the whole line if the whole line starts and ends with brackets (even though it may be multiple tags). Think about what characters can and, more importantly, can't be inside an html tag.
- change any sequences of ` ` to spaces
- the title of an html article is enclosed with "<title> ... </title>" tags. Before removing the html tags, identify the title and prepend it with "Title:". For example, if in the file the title was "<title>A Brave New World</title>", then you would replace this sequence with "Title: A Brave New World".

The input file should be called *filename*.

A quick overview of `sed` can be found at the end of this handout.

(b) Write a function in Perl, Python or Java that takes as input a string and returns a pig latin version of that word. Any string processing should be done using ONLY the regular expression functionality of the language. To turn a string into pig latin, move any consonant (or consonant cluster) at the beginning of each word to the end, then, regardless of what letter the word stars with, add 'ay' to the end of it. For example "crouton → outoncray" or "snake → akesnay" or "another" → "anotheray".

3. Data analysis **[30 points]**

For this last problem, you will be analyzing a real corpus consisting of two files:

/common/cs/cs159/assignments/assignment1/normal.txt.gz
/common/cs/cs159/assignments/assignment1/simple.txt.gz

Each of these files contains text from Wikipedia articles, `normal.txt` from English Wikipedia and `simple.txt` from Simple English Wikipedia. The files each contain articles on the same 500 topics. The articles are delimited by the TITLE descriptor and paragraphs are delimited by blank lines.

(a) How many paragraphs does each data set contain? What is the average number of paragraphs per article?

(b) How many sentences does each data set contain? What is the average number of sentences per article?

To do this, write a simple sentence segmenter. We will use a fairly simple definition of a sentence. First, split the paragraphs into sentences based on any text that ends with a '?' or '!' or '.' optionally proceeded by quotes. Now, go back through and merge sentences with the following criterion: 1) sentences should not start with a lowercase letter, if one does, merge it with the previous sentence 2) if a sentence ends with a period and the last word is a capitalized word with three or less characters, then merge it with the sentence that follows it.

(c) List one example (you can make it up) where this sentence splitting approach does not do the correct thing.

(d) How many words does each data set contain? What is the average number of words per article? per sentence?

To do this, write a simple word tokenizer. First split the sentence into tokens based on whitespace. Then, for each token, as long as it starts or ends with: `" , : ; ' ( ) . ? !` then split that character off as a new token. For example, the token `banana."` would end up as three different tokens since you would continue to split off trailing characters.

Only count those words that contain some alphanumeric portion as a "word", for example, you wouldn't count a period by itself or "()".

(e) List one example (you can make it up) where this tokenization technique does not do the correct thing.

(f) How many *different* words are there in each of the data sets (i.e what is the vocabulary size)? How does this change if you lowercase all of the words? Again, only count alphanumeric strings as words.

(g) List the top ten most frequent words in each data set with their frequencies. Are there any surprises? Do they differ much?

(h) Find two more interesting pieces of data to compare/contrast between these data sets. You will be graded on how creative your analysis is, how difficult the analysis was and how profound your results are. If you need ideas, come talk to me (earlier than later) and we can bounce some ideas around. Extra credit will be given for particularly insightful analysis.

4. Extra credit [**up to 4 points**]

In problem 3, we implemented fairly basic word tokenizers and sentence segmenters. For extra credit, you may improve one or both of these. If you do, include in your write a) what you did and b) how this changed your results compared to the "basic" version.

When you're done

Submit your writeup in .doc, .pdf or .txt using the link on the course web page.

Useful notes

- `grep`

  `grep` is a command line tool that allows you to find lines in a file that match particular regular expressions. Most frequently, `grep` takes two parameters, a double quoted regular expression and a filename. `grep` will then print to the console all lines in the file that match the regular expression (the `-v` flag causes `grep` to return all non-matching lines). For example

  ```
  grep "banana" file
  ```

  would output all lines in the file that contained the string banana, including lines that had other variants like bananas. `grep` does not have the character class shortcuts (like \d, \w, etc.) however, you can make your own using brackets (like [0-9]). `grep` has many other useful features that you can experiment with. I would suggest always using `grep -E`, which has uses the more traditional notation.

  You can find documentation for `grep` online, for example:

  ```
  http://www.panix.com/~elflord/unix/grep.html
  ```

  is pretty good. You can also type `man grep` to get some information. Using `grep -E` does have a few extra useful features, for example it will allow you to use the `\b` character for denoting a boundary (whitespace or the beginning/end).

- `sed`

  `sed` is another command-line program. `sed` is particularly useful for doing string substitutions in a file. Like `grep`, the most common way to use `sed` is to pass it a command in quotes as the first argument and then a filename as the second. The most common way to use `sed` is with the substitution command:

  ```
  sed -E "s/aa/bb/" filename
  ```

  which will substitute *the first* occurrent of "aa" on each line with "bb" and print it out to the console (or you could redirect it to an file using '`>`'). If you want it to substitute all occurrences, use the global flag:

  ```
  sed -E "s/aa/bb/g" filename
  ```

  You can use other standard regular expression techniques with `sed` including things like '.' for matching everything, character classes using '[]', and the use of groups identified by parenthesis in the matching expression and then denoted by '\\1', '\\2', etc. For example:

  ```
  sed -E "s/([dD]ave)/\\1 is great/g" filename
  ```

  would add " is great" after every occurrence of "dave" or "Dave" in a file.

  `sed` will also read input from the console if you don't provide a filename. This features is frequently used to chain sed statements together. For example:

  ```
  sed -E "s/[0-9]+//g" filename | sed -E "s/ +/ /g"
  ```

  which first removes all numbers then pipes/sends the output from that to another `sed` call using '|', which then removes multiple spaces.

For additional information type `man sed` or see resources online, for example http://www.grymoire.com/Unix/Sed.html.