Even More Perl

Info 427

Announcements

• Assignment 1 released!
  – Due Monday Sept 30 (about 2 weeks)
  – You can work individually or in groups of 2
  – Please start early!

Part 1: Analyze text documents

Part 2: Spam filtering

Spam

• Spam = junk e-mail and web pages
  
  – A big problem! (~96% of all email traffic on the Internet
  – ~150 billion junk emails per day
  – Spreads malware, worms, phishing schemes, etc.

Part 1:

```bash
[djcran@raichu i417]$ perl spellcheck.pl myessay.txt
Running spell check on myessay.txt:
- Potential problem with word: "freind"
  "ei" only allowed after the letter "c"!
- Potential problem with word: "An"
  The next word ("good") does not start with a vowel!
- Potential problem with word: "qickly"
  The word has a "q" not followed by "u"!
- Potential problem with word: "brght"
  The word does not have a vowel!
```

Part 2:

```bash
[djcran@raichu i427]$ perl spam.pl known_spam.txt known_notspam.txt new_document.txt
Evaluating the spaminess of new_document.txt...
Spam score: 346.017439426476
Document is probably spam.
```

Using a “spaminess score:”

\[
\text{Score} = \sum_{\text{word} \in \text{Document}} \log S(\text{word}) - \log N(\text{word}).
\]

– where \(S(\text{word})\) is # of times word occurs in a spam corpus, and \(N(\text{word})\) is # of times in non-spam corpus

Spam

– Spam = junk e-mail and web pages

A big problem! (~96% of all email traffic on the Internet

– ~150 billion junk emails per day

– Spreads malware, worms, phishing schemes, etc.
Modeling a document

• Use natural language processing techniques?
  – Parse the web page, understand the meaning, decide if email is spam
  – Too difficult for now
• Simpler alternative
  – Represent a document as an unordered collection of words (a bag of words model)

Statistical motivation

• Spam and (my) non-spam are statistically very different

An example

• We can take some documents known to be spam and known to be legitimate, to estimate relative importance of words

<table>
<thead>
<tr>
<th>Word</th>
<th># times in spam</th>
<th># times in non-spam</th>
<th>Ratio (# spam / # nonspam)</th>
<th>log(Ratio)</th>
</tr>
</thead>
<tbody>
<tr>
<td>debt</td>
<td>308</td>
<td>4</td>
<td>77.25</td>
<td>1.88</td>
</tr>
<tr>
<td>news</td>
<td>215</td>
<td>39</td>
<td>5.51</td>
<td>0.74</td>
</tr>
<tr>
<td>investment</td>
<td>288</td>
<td>13</td>
<td>22.15</td>
<td>1.34</td>
</tr>
<tr>
<td>david</td>
<td>12</td>
<td>575</td>
<td>0.021</td>
<td>-1.68</td>
</tr>
<tr>
<td>wants</td>
<td>101</td>
<td>268</td>
<td>0.38</td>
<td>-0.42</td>
</tr>
<tr>
<td>thanks</td>
<td>49</td>
<td>196</td>
<td>0.25</td>
<td>-1.39</td>
</tr>
</tbody>
</table>

Classifying spam

• Once we’ve constructed this table, we can use a Bayesian classifier to decide if a new document is spam
  – You can learn more about classifiers in a Machine Learning class
  – Multiply together the ratios for each word in the document; if greater than 1, it’s spam, and otherwise not spam
  – Or, equivalently, add up the log(ratios) for each word; if greater than 0, it’s spam, and otherwise not spam

Learning

• An advantage of a Bayesian classifier is that it “learns” what spam looks like automatically
  – Just by counting #’s of words in spam and non-spam.
  – No need for hand-crafted rules.
  – But a good set of training data is critical.
• The classifier can continue to learn with time
  – User corrects the classifier’s errors, classifier adjusts its word counts accordingly
Bayesian poisoning

- Spammers try to confuse the Bayesian filters
- Passive attacks
  - Add many non-spam words to web pages
  - Disguise spam words by misspelling (e.g. viagra -> vi@gra)
- Active attacks
  - Assume that it’s possible for spammer to see if an email (or webpage) is filtered out by the classifier
  - Send many email variants, observing the filter’s decision
  - Tune the emails to stay “one step ahead” of the filter

Image-based spam

Regular expressions

- Last time we covered regular expressions
  - For matching strings
  - Important characters to know: + * . ? [] ^ $
- These will be very useful in A1
- Regular expressions have many more features
  - Check them out in the Perl book, or by googling: “perldoc perlre”

Even More Perl

Quantifiers

- So far we’ve seen several ways of specifying repeated patterns: *, ?, +
- Quantifiers let you specify a specific number of repetitions, or a specific range of repetitions, e.g.:
  - [a-z]{5} matches exactly 5 lowercase letters
  - (hello){3,5} matches 3, 4, or 5 copies of “hello”
  - (\b.*\b){10} matches 10 words that end in “s”

Alternations

- Specify alternatives with the | symbol
  - the pattern is matched if at least one alternative matches
  - a|b|c|d matches either a, b, c, or d
  - (((hi)|(bye))|song|blue)bird matches songbird and bluebird
Regular expression references

- Perl lets you refer to portions of a matched regular expression later in your program.
- Use parens to mark portions of your pattern. Then use the variables $1, $2, $3, etc. to refer to the strings that matched that portion.

```perl
print "Enter a name in the format lastname, firstname: ";
my $name = <STDIN>;
chomp($name);
if ($name =~ /^([A-Za-z]+), ([A-Za-z]+)$/) {
    print "The name is $2 $1
} else {
    die "Invalid name!"
}
```

References: another example

- Suppose we want to read in a URL, and then separate it into four parts: protocol://siteaddress//directory/file

```perl
chomp(my $url = <STDIN>);
if ($url =~ /^((https?|ftp)://([a-z.]+)/([a-z.]+)$/) {
    print "The protocol is $1
    print "The server address is $2
    print "The filename is $3
} else {
    die "Invalid url!"
}
```

Parsing

- Split breaks up a string into tokens, and returns the result as a list
  - Using a specified pattern as the separator
    - e.g.

```perl
my $address="http://www.google.com/test/index.html";
my @list = split(/,\/\/,$address);
foreach my $i (@list) {
    print "$i
```

Substring parsing

- Index function: Search a string for a substring
  - $location = index("Hello world", "w");
  - # result is 6

- Substr function: Return portion of a string
  - $var = "Hello world";
  - $part = substr($var, index($var, "w")+1, 2);
  - # $part contains "or"

Search and Replace

- Perl’s s operator lets you perform search and replace operations on strings, using regular expressions:
  - s/search-pattern/replace-pattern/
  - By default, only replaces the first occurrence; use the g suffix to replace all occurrences

```perl
my $string = "Purdue is #1! Go Purdue!";
$string =~ s/Purdue/IU/g;
print "$string\n```

```
IU is #1! Go IU!
```

The protocol is http
The server address is en.wikipedia.org
The filename is wiki/facebook.html

http://www.google.com
http://www.google.com/test/index.html
http://www.google.com/test/index.html
Another example

- Suppose we want to remove all HTML tags from a file—i.e. remove everything between < and >
  my $string = "Some &lt;b&gt;bolded&lt;/b&gt; text";
  $string =~ s/&lt;.*?&gt;/g;
  print "$string\n";

- The * and + operators are greedy, consuming as much of the input string as possible
  — Use special *? and +? operators to consume as little as possible
  my $string = "Some &lt;b&gt;bolded&lt;/b&gt; text";
  $string =~ s/&lt;.*?&gt;/g;
  print "$string\n";

Subroutines

- Create your own subroutines (functions) to modularize your code
  — Define a subroutine like this:
  sub subroutinename {
    # code
  }
  Then call the subroutine like this:
  subroutinename()

Example

sub complain {
    print "Input invalid!\n";
}
my $input = <STDIN>;
chomp($input);
complain() unless($input =~ /^[yn]$/);

Parameters

• Input parameters to your function appear in a special array, @_

• Return a value from your function using:
  return expression;

Example

sub add {
    return $_[0] + $_[1];
}
print "Sum of 1 and 2 is " . add(1,2);

More readable version: Copy these into more meaningful variable names first
sub add {
    my ($num1, $num2) = @_; 
    return $num1 + $num2;
}

Example

sub add {
    my ($num1, $num2) = @_; 
    return $num1 + $num2;
}
my $sum = add(add(3,4), add(1,2));
print "1+2+3+4 = $sum\n";
File Processing

• To read from or write to a file, you first need to open it:

    unless (open (FILE, "filename")) {
        # error occurred
    }
    # can read from filename using FILE here...
    # when done, close it:
    close FILE;

Reading data

• You can read an entire file into an array in a single statement:

    my @file = <FILE>;

    – Each element in the array is a single line of the file

• Or, you can read single lines into a scalar:

    while (my $line = <FILE>) {
        # $line has the data
        chomp $line; # to remove newline character:
    }

Writing Output

open (OUTPUT, ">test.out");
while (my $i = <STDIN>) {
    print OUTPUT $i;
} close OUTPUT;

• print is a shortcut for print STDOUT
• Note that there is no comma between the output file handle and the other print args

Reading Directories

• You can read in a list of files using opendir and readdir:

    opendir(DIR, ".") or die("can't open directory");
    my @list = readdir(DIR);
    print "Files in current directory:
    foreach my $i (@list) {
        print $i . "\n";
    }

File Tests

• Checking certain conditions
• Syntax:

    if (–test filename) {
        print "filename has property –test";
    }

Tests

• -r file or directory is readable
• -w file or directory is writable
• -x file or directory is executable
• -o file or directory is owned
• -e file or directory exists
• -z file exists and has zero size
• -s file or directory exists and has non-zero size
• -f/-d/-l if file/directory/symlink
More tests

• -T file looks like a text file
• -B file looks like a binary file
• -M modification age in days
• -A access age in days

Example

opendir(DIR, "./" or die("can't open directory");
my @list = readdir(DIR);

print "Directories in current directory:\n";
foreach my $i (@list) {
  if(-d $i) {
    print $i . "\n";
  }
}