Regular Expressions and Finite State Automata

Mausam

(Based on slides by Jurafsky & Martin, Julia Hirschberg)
Regular Expressions and Text Searching

• Everybody does it
  ◦ Emacs, vi, perl, grep, etc..

• Regular expressions are a compact textual representation of a set of strings representing a language.
<table>
<thead>
<tr>
<th>RE</th>
<th>Example Patterns Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td>/woodchucks/</td>
<td>“interesting links to woodchucks and lemurs”</td>
</tr>
<tr>
<td>/a/</td>
<td>“Mary Ann stopped by Mona’s”</td>
</tr>
<tr>
<td>/Claire_says,/</td>
<td>“Dagmar, my gift please,” Claire says,”</td>
</tr>
<tr>
<td>/DOROTHY/</td>
<td>“SURRENDER DOROTHY”</td>
</tr>
<tr>
<td>/!/</td>
<td>“You’ve left the burglar behind again!” said Nori</td>
</tr>
<tr>
<td>RE</td>
<td>Match</td>
</tr>
<tr>
<td>----------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td><code>/\wW\oodchuck/</code></td>
<td>Woodchuck or woodchuck</td>
</tr>
<tr>
<td><code>/\abc/</code></td>
<td>‘a’, ‘b’, <em>or</em> ‘c’</td>
</tr>
<tr>
<td><code>/\[1234567890]/</code></td>
<td>any digit</td>
</tr>
<tr>
<td>RE</td>
<td>Match</td>
</tr>
<tr>
<td>-------</td>
<td>------------------------</td>
</tr>
<tr>
<td>/A-Z/</td>
<td>an upper case letter</td>
</tr>
<tr>
<td>/a-z/</td>
<td>a lower case letter</td>
</tr>
<tr>
<td>/0-9/</td>
<td>a single digit</td>
</tr>
<tr>
<td>RE</td>
<td>Match (single characters)</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------------------------------</td>
</tr>
<tr>
<td>[^A-Z]</td>
<td>not an upper case letter</td>
</tr>
<tr>
<td>[^Ss]</td>
<td>neither ‘S’ nor ‘s’</td>
</tr>
<tr>
<td>[^.]</td>
<td>not a period</td>
</tr>
<tr>
<td>[e^]</td>
<td>either ‘e’ or ‘^’</td>
</tr>
<tr>
<td>a^b</td>
<td>the pattern ‘a^b’</td>
</tr>
</tbody>
</table>
### Figure 2.4

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
<th>Example Patterns Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td>woodchucks?</td>
<td>woodchuck or woodchucks</td>
<td>“woodchuck”</td>
</tr>
<tr>
<td>colour</td>
<td>color or colour</td>
<td>“colour”</td>
</tr>
<tr>
<td>RE</td>
<td>Match</td>
<td>Example Patterns</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------</td>
<td>------------------</td>
</tr>
<tr>
<td>/beg.n/</td>
<td>any character between <code>beg</code> and <code>n</code></td>
<td><code>begin, beg’n, begun</code></td>
</tr>
</tbody>
</table>
Parenthesis  ( )
Counters     * + ? { }
Sequences and anchors the ^my end$
Disjunction  |
<table>
<thead>
<tr>
<th>RE</th>
<th>Expansion</th>
<th>Match</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>\d</td>
<td>[0–9]</td>
<td>any digit</td>
<td>Party_of_5</td>
</tr>
<tr>
<td>\D</td>
<td>[^0–9]</td>
<td>any non-digit</td>
<td>Blue_moon</td>
</tr>
<tr>
<td>\w</td>
<td>[a-zA-Z0-9_]</td>
<td>any alphanumeric/underscore</td>
<td>Daiyu</td>
</tr>
<tr>
<td>\W</td>
<td>[^\w]</td>
<td>a non-alphanumeric</td>
<td>!!!!</td>
</tr>
<tr>
<td>\s</td>
<td>[\r\t\n\f]</td>
<td>whitespace (space, tab)</td>
<td></td>
</tr>
<tr>
<td>\S</td>
<td>[^\s]</td>
<td>Non-whitespace</td>
<td>in_Concord</td>
</tr>
</tbody>
</table>
## Figure 2.8

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
<th>Example Patterns Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>an asterisk &quot;*&quot;</td>
<td>&quot;K<em>A</em>P<em>L</em>A*N&quot;</td>
</tr>
<tr>
<td>.</td>
<td>a period &quot;.&quot;</td>
<td>&quot;Dr. Livingston, I presume&quot;</td>
</tr>
<tr>
<td>?</td>
<td>a question mark</td>
<td>&quot;Why don’t they come and lend a hand?&quot;</td>
</tr>
<tr>
<td>\n</td>
<td>a newline</td>
<td></td>
</tr>
<tr>
<td>\t</td>
<td>a tab</td>
<td></td>
</tr>
</tbody>
</table>
Example

- Find all the instances of the word “the” in a text.
  - /the/
  - /[tT]he/
  - /\b[tT]he\b/
Errors

• The process we just went through was based on two fixing kinds of errors
  ▶ Matching strings that we should not have matched (*there, then, other*)
    ▪ False positives (Type I)
  ▶ Not matching things that we should have matched (*The*)
    ▪ False negatives (Type II)
Errors

• We’ll be telling the same story for many tasks, all semester. Reducing the error rate for an application often involves two antagonistic efforts:
  - Increasing accuracy, or precision, (minimizing false positives)
  - Increasing coverage, or recall, (minimizing false negatives).
Finite State Automata

- Regular expressions can be viewed as a textual way of specifying the structure of finite-state automata.
- FSAs and their probabilistic relatives are at the core of much of what we’ll be doing all semester.
- They also capture significant aspects of what linguists say we need for morphology and parts of syntax.
FSAs as Graphs

• Let’s start with the sheep language from Chapter 2
  ♦ /baa+!/
Sheep FSA

We can say the following things about this machine:

- It has 5 states
- b, a, and ! are in its alphabet
- q₀ is the start state
- q₄ is an accept state
- It has 5 transitions
But Note

- There are other machines that correspond to this same language

- More on this one later
More Formally

- You can specify an FSA by enumerating the following things.
  - The set of states: $Q$
  - A finite alphabet: $\Sigma$
  - A start state
  - A set of accept/final states
  - A transition function that maps $Q\times\Sigma$ to $Q$
About Alphabets

• Don’t take term *alphabet* word too narrowly; it just means we need a finite set of symbols in the input.

• These symbols can and will stand for bigger objects that can have internal structure.
Dollars and Cents

The diagram illustrates the states and transitions for processing spoken numbers. The states include:

- $q_0$: Begins with one through five.
- $q_1$: Moves to six through ten.
- $q_2$: Proceeds to eleven through nineteen.

Transitions are indicated by arrows connecting these states, with the numbers corresponding to the transitions labeled:

- From $q_0$ to $q_1$: Twenty, thirty, forty, fifty.
- From $q_1$ to $q_2$: Sixty, seventy, eighty, ninety.
- From $q_2$: One, two, three, four, five.

This representation helps in understanding the process of recognizing spoken numbers in a speech processing context.
Dollars and Cents
Yet Another View

- The guts of FSAs can ultimately be represented as tables

If you’re in state 1 and you’re looking at an a, go to state 2
Recognition

- Recognition is the process of determining if a string should be accepted by a machine
- Or... it’s the process of determining if a string is in the language we’re defining with the machine
- Or... it’s the process of determining if a regular expression matches a string
- Those all amount the same thing in the end
Recognition

- Traditionally, (Turing’s notion) this process is depicted with a tape.
Recognition

- Simply a process of starting in the start state
- Examining the current input
- Consulting the table
- Going to a new state and updating the tape pointer.
- Until you run out of tape.
function D-RECOGNIZE(tape, machine) returns accept or reject

index ← Beginning of tape
current-state ← Initial state of machine
loop
  if End of input has been reached then
    if current-state is an accept state then
      return accept
    else
      return reject
  elsif transition-table[current-state, tape[index]] is empty then
    return reject
  else
    current-state ← transition-table[current-state, tape[index]]
    index ← index + 1
  end
Key Points

• Deterministic means that at each point in processing there is always one unique thing to do (no choices).

• D-recognize is a simple table-driven interpreter

• The algorithm is universal for all unambiguous regular languages.
  ♦ To change the machine, you simply change the table.
Key Points

- Crudely therefore... matching strings with regular expressions (ala Perl, grep, etc.) is a matter of
  - translating the regular expression into a machine (a table) and
  - passing the table and the string to an interpreter
Recognition as Search

• You can view this algorithm as a trivial kind of state-space search.
• States are pairings of tape positions and state numbers.
• Operators are compiled into the table
• Goal state is a pairing with the end of tape position and a final accept state
• It is trivial because?
Generative Formalisms

- **Formal Languages** are sets of strings composed of symbols from a finite set of symbols.
- Finite-state automata define formal languages (without having to enumerate all the strings in the language)
- The term *Generative* is based on the view that you can run the machine as a generator to get strings from the language.
Generative Formalisms

- FSAs can be viewed from two perspectives:
  - Acceptors that can tell you if a string is in the language
  - Generators to produce *all and only* the strings in the language
Non-Determinism

\[ q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2 \xrightarrow{a} q_3 \xrightarrow{a} q_4 \]

\[ q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2 \xrightarrow{a} q_3 \xrightarrow{!} q_4 \]
Equivalence

- Non-deterministic machines can be converted to deterministic ones with a fairly simple construction.
- That means that they have the same power; non-deterministic machines are not more powerful than deterministic ones in terms of the languages they can accept.
ND Recognition

• Two basic approaches (used in all major implementations of regular expressions, see Friedl 2006)
  1. Either take a ND machine and convert it to a D machine and then do recognition with that.
  2. Or explicitly manage the process of recognition as a state-space search (leaving the machine as is).
Non-Deterministic Recognition: Search

• In a ND FSA there exists at least one path through the machine for a string that is in the language defined by the machine.

• But not all paths directed through the machine for an accept string lead to an accept state.

• No paths through the machine lead to an accept state for a string not in the language.
Non-Deterministic Recognition

• So success in non-deterministic recognition occurs when a path is found through the machine that ends in an accept.

• Failure occurs when all of the possible paths for a given string lead to failure.
Example

\[ q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2 \xrightarrow{a} q_3 \xrightarrow{!} q_4 \]
Example

1. b a a a !

2. q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow q_3 \rightarrow q_4

b \rightarrow q_1
a \rightarrow q_2
a \rightarrow q_3
!

4
Key Points

- States in the search space are pairings of tape positions and states in the machine.
- By keeping track of as yet unexplored states, a recognizer can systematically explore all the paths through the machine given an input.
Why Bother?

- Non-determinism doesn’t get us more formal power and it causes headaches so why bother?
  - More natural (understandable) solutions
FSTs (Contd)
FST-based Tokenization

```perl
#!/usr/bin/perl

$letternumber = "[A-Za-z0-9]";
$notletter = "([\-\d\s]\"\)|\'\[\])";
$alwayssep = "([\-\:\.\-\d\s]\"\)|\'\[\])";
$clitic = "'(\.|\-)|\'D|\'H|\'LL|\'RE|\'VE|\'N\'T|\'s|\'d|\'m|\'ll|\'re|\'ve|\'n\'t)";

$abbr("Co." ) = 1; $abbr("Dr." ) = 1; $abbr("Jan." ) = 1; $abbr("Feb." ) = 1;

while ($line = <>) {
  # read the next line from standard input
  # put whitespace around unambiguous separators
  $line =~ s/$alwayssep/ /g;
  # put whitespace around commas that aren't inside numbers
  $line =~ s/(/\1)/, /g;
  # distinguish single quotes from apostrophes by
  # segmenting off single quotes not preceded by letter
  $line =~ s/\'$/\1 /g;
  $line =~ s/\1'(single quote)\1'/g;

  # segment off unambiguous word-final clitics and punctuation
  $line =~ s/\s+$/\1 /g;
  $line =~ s/\s+($notletter)/\1 \1/g;

  # now deal with periods.  For each possible word
  @possiblewords = split(/\s*\$, $line/);
  foreach $word (@possiblewords) {
    # if it ends in a period,
    if ($word =~ /$letternumber\$/)
      if (!($word eq $abbr($word))
        && !($word eq $letternumber))
        &&#$'\$ is't a sequence of letters and periods (U.S.)
        &&#$'\$ doesn't resemble an abbreviation (no vowels: Inc.)
      && !($word =~
        '/'([A-Za-z]\.|([A-Za-z]+)[bcdfghj-nptvwx]|[a-z][bcdfghj-nptvwx]\")/)
        {
          # then segment off the period
          $word =~ s/\s+/\1 /g;
        }
    # expand clitics
    $word =~ s/\'ve\have/\;
    $word =~ s/\'m\am/\;
    print $word, " ",
  }
  print "\n";
}```
Porter Stemmer (1980)

- Common algorithm for stemming English

- Conventions + 5 phases of reductions
  - phases applied sequentially
  - each phase consists of a set of commands
  - sample convention: *Of the rules in a compound command, select the one that applies to the longest suffix.*
Porter Stemmer (1980)

- Standard, very popular and usable stemmer (IR, IE) – identify a word’s stem
- Sequence of cascaded rewrite rules, e.g.
  - IZE → ε (e.g. unionize → union)
  - CY → T (e.g. frequency → frequent)
  - ING → ε , if stem contains vowel (motoring → motor)
- Can be implemented as a lexicon-free FST (many implementations available on the web)
- [http://text-processing.com/demo/stem/](http://text-processing.com/demo/stem/)
Summing Up

• Regular expressions and FSAs can represent subsets of natural language as well as regular languages
  - Both representations may be difficult for humans to use for any real subset of a language
  - Can be hard to scale up: e.g., when many choices at any point
  - But quick, powerful and easy to use for small problems
  - AT&T Finite State Toolkit *does* scale

• Finite state transducers and rules are common ways to incorporate linguistic ideas in NLP for small applications