Natural Language Processing (Words)

Budditha Hettige
Department of Computer Engineering
Faculty of Computing
General Sir John Kotelawala Defence University
WHENEVER I LEARN A NEW SKILL, I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.

OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!

BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!

IT'S HOPELESS!

EVERYBODY STAND BACK.

I KNOW REGULAR EXPRESSIONS.

REGULAR EXPRESSION

Natural Language Processing
Regular expression

- Regular expressions are a powerful string manipulation tool
- All modern languages have similar library packages for regular expressions
- Use regular expressions to:
  - Search a string \((\text{search \ and \ match})\)
  - Replace parts of a string \((\text{sub})\)
  - Break strings into smaller pieces \((\text{split})\)
Python’s Regular Expression Syntax

• Most characters match themselves
  The regular expression “test” matches the string ‘test’, and only that string

• [x] matches any one of a list of characters
  “[abc]” matches ‘a’, ‘b’, or ‘c’

• [^x] matches any one character that is not included in x
  “[^abc]” matches any single character except ‘a’, ‘b’, or ‘c’
Python’s Regular Expression Syntax

- “.” matches any single character
- Parentheses can be used for grouping
  
  “(abc)+” matches ‘abc’, ‘abcabc’, ‘abcabcabc’, etc.
- x/y matches x or y
  
  “this|that” matches ‘this’ and ‘that’, but not ‘thisthat’.
Python’s  
Regular Expression Syntax

• $x^*$ matches zero or more $x$’s
  “$a^*$” matches ‘’, ’a’, ’aa’, etc.

• $x+$ matches one or more $x$’s
  “$a+$” matches ’a’, ’aa’, ’aaa’, etc.

• $x?$ matches zero or one $x$’s
  “$a?$” matches ‘’ or ’a’

• $x\{m, n\}$ matches $i$ $x$’s, where $m \leq i \leq n$
  “$a\{2,3\}$” matches ’aa’ or ’aaa’
Python’s Regular Expression Syntax

- "\d" matches any digit; "\D" any non-digit
- "\s" matches any whitespace character; "\S" any non-whitespace character
- "\w" matches any alphanumeric character; "\W" any non-alphanumeric character
- "^" matches the beginning of the string; "$" the end of the string
- "\b" matches a word boundary; "\B" matches a character that is not a word boundary
## Basic Regular Expression Patterns

<table>
<thead>
<tr>
<th>RE</th>
<th>Example Patterns Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/woodchucks/</code></td>
<td>“interesting links to woodchucks and lemurs”</td>
</tr>
<tr>
<td><code>/a/</code></td>
<td>“Mary Ann stopped by Mona’s”</td>
</tr>
<tr>
<td><code>/Claire_says,/</code></td>
<td>“Dagmar, my gift please,” Claire says,”</td>
</tr>
<tr>
<td><code>/song/</code></td>
<td>“all our pretty songs”</td>
</tr>
<tr>
<td><code>!/</code></td>
<td>“You’ve left the burglar behind again!” said Nori</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
<th>Example Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>/^[wW]oodchuck/</code></td>
<td>Woodchuck or woodchuck</td>
<td>“Woodchuck”</td>
</tr>
<tr>
<td><code>/^[abc]/</code></td>
<td>‘a’, ‘b’, or ‘c’</td>
<td>“In uomini, in soldati”</td>
</tr>
<tr>
<td><code>/^[1234567890]/</code></td>
<td>any digit</td>
<td>“plenty of 7 to 5”</td>
</tr>
</tbody>
</table>

**Figure 2.1** The use of the brackets [] to specify a disjunction of characters.
## Basic Regular Expression Patterns

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
<th>Example Patterns Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td>/[A-Z]/</td>
<td>an uppercase letter</td>
<td>“we should call it ‘Drenched Blossoms’”</td>
</tr>
<tr>
<td>/[a-z]/</td>
<td>a lowercase letter</td>
<td>“my beans were impatient to be hoed!”</td>
</tr>
<tr>
<td>/ [0-9] /</td>
<td>a single digit</td>
<td>“Chapter 1: Down the Rabbit Hole”</td>
</tr>
</tbody>
</table>

**Figure 2.2** The use of the brackets [] plus the dash - to specify a range.

<table>
<thead>
<tr>
<th>RE</th>
<th>Match (single characters)</th>
<th>Example Patterns Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td>[^A-Z]</td>
<td>not an uppercase letter</td>
<td>“Oyfn pripetchik”</td>
</tr>
<tr>
<td>[^Ss]</td>
<td>neither ‘S’ nor ‘s’</td>
<td>“I have no exquisite reason for’t”</td>
</tr>
<tr>
<td>[^.]</td>
<td>not a period</td>
<td>“our resident Djinn”</td>
</tr>
<tr>
<td>[e^]</td>
<td>either ‘e’ or ‘^’</td>
<td>“look up _ now”</td>
</tr>
<tr>
<td>a^b</td>
<td>the pattern ‘a^b’</td>
<td>“look up a^b now”</td>
</tr>
</tbody>
</table>

**Figure 2.3** Uses of the caret ^ for negation or just to mean ^
## Basic Regular Expression Patterns

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
<th>Example Patterns Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>woodchucks?</code></td>
<td><code>woodchuck</code> or <code>woodchucks</code></td>
<td>“woodchuck”</td>
</tr>
<tr>
<td><code>colou?r</code></td>
<td><code>color</code> or <code>colour</code></td>
<td>“colour”</td>
</tr>
</tbody>
</table>

**Figure 2.4** The question-mark `?` marks optionality of the previous expression.

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
<th>Example Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>beg.n/</code></td>
<td>any character between ‘beg’ and ‘n’</td>
<td><code>begin</code>, <code>beg’n</code>, <code>begun</code></td>
</tr>
</tbody>
</table>

**Figure 2.5** The use of the period `.` to specify any character.
## Advanced Operators

<table>
<thead>
<tr>
<th>RE</th>
<th>Expansion</th>
<th>Match</th>
<th>Example Patterns</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 or space</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>in_Concord</td>
</tr>
<tr>
<td>\S</td>
<td>[^\s]</td>
<td>Non-whitespace</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2.6** Aliases for common sets of characters.
### More

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
<th>Example Patterns Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>*</code></td>
<td>zero or more occurrences of the previous char or expression</td>
<td>“K<em>A</em>P<em>L</em>A*N”</td>
</tr>
<tr>
<td><code>+</code></td>
<td>one or more occurrences of the previous char or expression</td>
<td>“Dr. Livingston, I presume”</td>
</tr>
<tr>
<td><code>?</code></td>
<td>exactly zero or one occurrence of the previous char or expression</td>
<td>“Would you light my candle?”</td>
</tr>
<tr>
<td><code>{n}</code></td>
<td><code>n</code> occurrences of the previous char or expression</td>
<td></td>
</tr>
<tr>
<td><code>{n, m}</code></td>
<td>from <code>n</code> to <code>m</code> occurrences of the previous char or expression</td>
<td></td>
</tr>
<tr>
<td><code>{n,}</code></td>
<td>at least <code>n</code> occurrences of the previous char or expression</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2.7** Regular expression operators for counting.

### Figure 2.8

<table>
<thead>
<tr>
<th>RE</th>
<th>Match</th>
<th>Example Patterns Matched</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\*</code></td>
<td>an asterisk “*”</td>
<td>“K<em>A</em>P<em>L</em>A*N”</td>
</tr>
<tr>
<td><code>\.</code></td>
<td>a period “.”</td>
<td>“Dr. Livingston, I presume”</td>
</tr>
<tr>
<td><code>\?</code></td>
<td>a question mark</td>
<td>“Would you light my candle?”</td>
</tr>
<tr>
<td><code>\n</code></td>
<td>a newline</td>
<td></td>
</tr>
<tr>
<td><code>\t</code></td>
<td>a tab</td>
<td></td>
</tr>
</tbody>
</table>

Some characters that need to be backslashed.
The two basic functions are `re.search` and `re.match`
- Search looks for a pattern anywhere in a string
- Match looks for a match staring at the beginning

Both return `None` (logical false) if the pattern isn’t found and a “match object” instance if it is

```python
>>> import re
>>> pat = "a*b"
>>> re.search(pat,"fooaaabbcde")
<_sre.SRE_Match object at 0x809c0> <<<re.SRE_Match object at 0x809c0>>>
>>> re.match(pat,"fooaaabbcde")
>>> ```
Q: What’s a match object?

A: an instance of the match class with the details of the match result

```python
>>> r1 = re.search("a*b","fooaaabcde")
>>> r1.group()  # group returns string matched
'aaab'
>>> r1.start()  # index of the match start
3
>>> r1.end()    # index of the match end
7
>>> r1.span()   # tuple of (start, end)
(3, 7)
```
What got matched?

• Here’s a pattern to match simple email addresses
  \w+@\(\w+\.)+(com|org|net|edu)

```python
>>> pat1 = '\w+@(\w+\.)+(com|org|net|edu)'
>>> r1 = re.match(pat,'finin@cs.umbc.edu')
>>> r1.group()
'finin@cs.umbc.edu'
```

• We might want to extract the pattern parts, like the email name and host
What got matched?

- We can put parentheses around groups we want to be able to reference

```python
>>> pat2 = "\((\w+)@((\w+\.)+(com|org|net|edu))\"
>>> r2 = re.match(pat2,"finin@cs.umbc.edu")
>>> r2.group(1)
'finin'
>>> r2.group(2)
'cs.umbc.edu'
>>> r2.groups()
('finin', 'cs.umbc.edu', 'umbc.', 'edu')
```

- Note that the ‘groups’ are numbered in a preorder traversal of the forest
• We can ‘label’ the groups as well...

```python
>>> pat3 = "(?P<name>\w+)@(?P<host>(\w+\.)+(com|org|net|edu))"

>>> r3 = re.match(pat3,"finin@cs.umbc.edu")

>>> r3.group('name')
'finin'

>>> r3.group('host')
'cs.umbc.edu'
```

• And reference the matching parts by the labels
More re functions

• `re.split()` is like `split()` but can use patterns
  >>> re.split("\W+", "This... is a test, short and sweet, of split().")
  ['This', 'is', 'a', 'test', 'short', 'and', 'sweet', 'of', 'split', '']

• `re.sub()` substitutes one string for a pattern
  >>> re.sub('(blue|white|red)', 'black', 'blue socks and red shoes')
  'black socks and black shoes'

• `re.findall()` finds all matches
  >>> re.findall("\d+", "12 dogs, 11 cats, 1 egg")
  ['12', '11', '1']
• If you plan to use a re pattern more than once, compile it to a re object
• Python produces a special data structure that speeds up matching

```python
>>> capt3 = re.compile(pat3)
>>> cpat3
<_sre.SRE_Pattern object at 0x2d9c0>
>>> r3 = cpat3.search("finin@cs.umbc.edu")
>>> r3
<_sre.SRE_Match object at 0x895a0>
>>> r3.group()
'finin@cs.umbc.edu'
```
Pattern objects have methods that parallel the re functions (e.g., match, search, split, findall, sub), e.g.:

```python
>>> p1 = re.compile(r"\w+@\w+\.+com|org|net|edu")
>>> p1.match("steve@apple.com").group(0)
'steve@apple.com'
>>> p1.search("Email steve@apple.com today.").group(0)
'steve@apple.com'
>>> p1.findall("Email steve@apple.com and bill@msft.com now.")
['steve@apple.com', 'bill@msft.com']
>>> p2 = re.compile(r"[.?!]+\s+")
>>> p2.split("Tired? Go to bed! Now!! ")
['Tired', 'Go to bed', 'Now', ' ']
```
Exercise

• Write regular expressions for the following languages
  – the set of all alphabetic strings.
  – the set of all lowercase alphabetic strings ending in a $b$.
  – the set of all strings with two consecutive repeated words

Example
  ‘Humbert Humbert’ and ‘the the’ but not ‘the bug’ or ‘the big bug’).
FINITE-STATE AUTOMATA
FSA

- Any regular expression can be implemented as a finite-state automaton
- Both regular expressions and finite-state automata can be used to described regular languages
After a while, with the parrot’s help, the Doctor got to learn the language of the animals so well that he could talk to them himself and understand everything they said.

Hugh Lofting, The Story of Doctor Dolittle

baa!
baaa!
baaaaa!
baaaaaaa!
...

Using an FSA to Recognize

Diagram of a finite state automaton (FSA): q₀ → q₁ → q₂ → q₃ → q₄.
Automaton

- automaton (finite automaton, finite-state automaton, or FSA) recognizes a set of strings

- Example
  - Automaton has number of states
  - Start State
  - End State
  - Accepting state,
Deterministic Finite Automaton (DFA)

Input Tape

String

Finite Automation

Output

“Accept” or “Reject”
Transition Graph

- Initial state: $q_0$
- Accepting state: $q_5$
- Transitions:
  - $q_0 \rightarrow a \rightarrow q_1 \rightarrow b \rightarrow q_2 \rightarrow a \rightarrow q_3 \rightarrow b \rightarrow q_4$
  - $q_5$ is connected by $a, b$

Natural Language Processing
For every state, there is a transition for every symbol in the alphabet
**Initial Configuration**

**Input Tape**

| \(a\) | \(b\) | \(b\) | \(a\) |

**Input String**

\(a, b, ba, 0\)

**Initial state**

\(q_0\)

**Diagram**

- **States**: \(q_0, q_1, q_2, q_3, q_4, q_5\)
- **Transition Rules**:
  - \(q_0\) on \(a\) moves to \(q_1\)
  - \(q_1\) on \(a\) moves to \(q_2\)
  - \(q_2\) on \(b\) moves to \(q_3\)
  - \(q_3\) on \(a\) moves to \(q_4\)
  - \(q_4\) on \(a, b\) moves to \(q_5\)
  - \(q_5\) on \(a, b\) moves to \(q_5\) (loop)

**Head Position**

Pointed at \(a\)
Scanning the Input

\[
\begin{array}{cccc}
  a & b & b & a \\
\end{array}
\]
Scanning the Input

\begin{center}
\begin{tabular}{c|c|c|c}
  a & b & b & a \\
\end{tabular}
\end{center}

\begin{center}
\begin{tikzpicture}
  \node (q0) at (0,0) [circle,draw] {$q_0$};
  \node (q1) at (2,0) [circle,draw] {$q_1$};
  \node (q2) at (4,0) [circle,draw,fill=red] {$q_2$};
  \node (q3) at (6,0) [circle,draw] {$q_3$};
  \node (q4) at (8,0) [circle,draw] {$q_4$};
  \node (q5) at (6,2) [circle,draw] {$q_5$};

  \draw[->,thick] (q0) -- node[above] {$a$} (q1);
  \draw[->,thick] (q1) -- node[above] {$b$} (q2);
  \draw[->,thick] (q2) -- node[above] {$a$} (q3);
  \draw[->,thick] (q3) -- node[above] {$b$} (q4);
  \draw[->,thick] (q4) -- node[above] {$a$} (q5);

  \draw[->,thick] (q5) -- node[above] {$a,b$} (q2);

  \draw[->,thick] (q0) -- node[above] {$b$} (q1);
  \draw[->,thick] (q1) -- node[above] {$a$} (q2);
  \draw[->,thick] (q2) -- node[above] {$a$} (q3);
  \draw[->,thick] (q3) -- node[above] {$b$} (q4);
  \draw[->,thick] (q4) -- node[above] {$a$} (q5);

  \draw[->,thick] (q5) -- node[above] {$a,b$} (q2);

\end{tikzpicture}
\end{center}
Scanning the Input

\[
\begin{array}{cccc}
 a & b & b & a \\
\end{array}
\]
Scanning the Input

$q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{b} q_3 \xrightarrow{a} q_4$

Input finished

$q_5$

$q_4$ accept
A Rejection Case

Input String

\[
\begin{array}{ccc}
q_0 & a & b \\
q_1 & a & b \\
q_2 & a & b \\
q_3 & a & b \\
q_4 & a & b \\
q_5 & a, b \\
\end{array}
\]
A Rejection Case

\[
\begin{array}{ccc}
  a & b & a \\
\end{array}
\]
A Rejection Case

\[
\begin{array}{ccc}
  a & b & a \\
\end{array}
\]
A Rejection Case

Input finished

Reject
Another Rejection Case

(\lambda) Tape is empty

Input Finished

reject
Language Accepted:

\[ L = \{abba\} \]
Another Example

$L = \{\lambda, ab, abba\}$

\[
\begin{align*}
q_0 & \xrightarrow{a} q_1 \xrightarrow{b} q_2 \xrightarrow{b} q_3 \xrightarrow{a} q_4 \xrightarrow{a,b} q_5 \\
q_1 & \xrightarrow{a} q_2 \xrightarrow{b} q_3 \xrightarrow{a,b} q_5 \\
q_2 & \xrightarrow{a} q_3 \xrightarrow{b} q_5 \\
q_3 & \xrightarrow{a,b} q_5 \\
q_4 & \xrightarrow{a,b} q_5
\end{align*}
\]
Empty Tape

(\lambda)

Input Finished

accept

q_0 \rightarrow a \rightarrow q_1 \rightarrow b \rightarrow q_2 \rightarrow a \rightarrow q_3 \rightarrow b \rightarrow q_4 \rightarrow a,b

q_5 \rightarrow a,b

q_0 (\text{red})
Another Example

\[
\begin{align*}
q_0 & \xrightarrow{a} q_0 \\
q_0 & \xrightarrow{b} q_1 \\
q_1 & \xrightarrow{a,b} q_2 \\
q_1 & \xrightarrow{a,b} q_2 \\
q_2 & \xrightarrow{a,b} q_2 \\
q_2 & \xrightarrow{a,b} q_2
\end{align*}
\]

- \( q_0 \): Start state
- \( q_1 \): Accept state
- \( q_2 \): Trap state

Input symbols: \( a, b \)
Input String

q₀ → a → q₁ → a, b → q₂

a a b
The diagram illustrates a finite automaton with the following states and transitions:

- **States:**
  - $q_0$ (initial state)
  - $q_1$
  - $q_2$

- **Transitions:**
  - From $q_0$ to $q_0$ on input $a$
  - From $q_0$ to $q_1$ on input $b$
  - From $q_1$ to $q_2$ on input $a, b$

The input string $a a b$ is processed as follows:

1. Start at $q_0$.
2. Read $a$ from left to right.
3. Transition to $q_0$.
4. Read $a$.
5. Transition to $q_0$.
6. Read $b$.
7. Transition to $q_1$.
8. The automaton accepts the input string, ending at state $q_1$. The transitions are represented by arrows labeled with the input symbols.
Input finished

\[
\begin{array}{c|c|c}
q_0 & q_1 & q_2 \\
\end{array}
\]

- q_0 \rightarrow a \rightarrow q_0
- q_0 \rightarrow b \rightarrow q_1
- q_1 \rightarrow \text{accept}
- q_1 \rightarrow a,b \rightarrow q_2
- q_2 \rightarrow a,b \rightarrow q_2
A rejection case

Input String

\[ \begin{array}{c}
  b \\
  a \\
  b \\
\end{array} \]
Input finished

$$b\ a\ b$$

Graph:

- **$q_0$**
  - Transition on $a$ to $q_0$
  - Transition on $b$ to $q_1$

- **$q_1$**
  - Transition on $b$ to $q_1$
  - Transition on $a, b$ to $q_2$

- **$q_2$**
  - Final state
  - Transition on $a, b$ (Reject)

The input finished with the string $b\ a\ b$. The automaton moves from $q_0$ to $q_1$ on the input $b$, and then to $q_2$ on the input $a, b$, resulting in a reject state.
Language Accepted:

\[ L = \{a^n b : n \geq 0\} \]
Another Example

Alphabet: \[ \Sigma = \{1\} \]

Language Accepted:

\[ \text{EVEN} = \{x : x \in \Sigma^* \text{ and } x \text{ is even} \} = \{\lambda, 11, 1111, 111111, \ldots\} \]
State-transition table

• Represent an automaton
• Example

![Diagram of an automaton with states and transitions]

<table>
<thead>
<tr>
<th>State</th>
<th>Input</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>b</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>a</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>!</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Finite automaton

- $Q$: a finite set of $N$ states $q_0, q_1, \ldots q_N$
- $\Sigma$: a finite input alphabet of symbols
- $q_0$: the start state
- $F$: the set of final states, $F \subseteq Q$
- $\delta(q, i)$: the transition function or transition matrix between states. Given a state $q \in Q$ and an input symbol $i \in \Sigma$, $\delta(q, i)$ returns a new state $q' \in Q$. $\delta$ is thus a relation from $Q \times \Sigma$ to $Q$;
A deterministic algorithm

```plaintext
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
        elseif transition-table[current-state, tape[index]] is empty then
            return reject
        else
            current-state ← transition-table[current-state, tape[index]]
            index ← index + 1
        end
    end
```
Fail State
Language Example

- A finite-state automaton for English nominal inflection
Language Example

• A finite-state automaton for English verbal inflection