Basic Text Processing

Regular Expressions

*Slides from Jurafsky & Martin*

*edited by RB with Sed and Python*
Regular expressions

A formal language for specifying text strings

How can we search for any of these?

- woodchuck
- woodchucks
- Woodchuck
- Woodchucks
Regular Expressions: Disjunctions

Letters inside square brackets []

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>[wW]oodchuck</td>
<td>Woodchuck, woodchuck</td>
</tr>
<tr>
<td>[1234567890]</td>
<td>Any digit</td>
</tr>
</tbody>
</table>

Ranges [A–Z]

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<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></td>
<td><strong>Drenched Blossoms</strong></td>
</tr>
<tr>
<td></td>
<td><strong>my beans were impatient</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Chapter 1: Down the Rabbit Hole</strong></td>
</tr>
</tbody>
</table>
Special character classes in Python

https://docs.python.org/3/howto/regex.html

\d

Matches any decimal digit; this is equivalent to the class [0-9].

\D

Matches any non-digit character; this is equivalent to the class [^0-9].

\s

Matches any whitespace character; this is equivalent to the class [ \t\n\r\f\v].

\S

Matches any non-whitespace character; this is equivalent to the class [^ \t\n\r\f\v].

\w

Matches any alphanumeric character; this is equivalent to the class [a-zA-Z0-9_].

\W

Matches any non-alphanumeric character; this is equivalent to the class [^a-zA-Z0-9_].
Regular Expressions: Negation in Disjunction

Negations  \([ ^Ss ]\)
- Carat means negation only when first in []

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>[^A−Z]</td>
<td>Not an upper case letter</td>
<td><em>Oyfn pripetchik</em></td>
</tr>
<tr>
<td>[^Ss]</td>
<td>Neither ‘S’ nor ‘s’</td>
<td><em>I have no exquisite reason”</em></td>
</tr>
<tr>
<td>[^e]</td>
<td>Neither e nor ^</td>
<td><em>Look here</em></td>
</tr>
<tr>
<td>a^b</td>
<td>The pattern a carat b</td>
<td><em>Look up a^b now</em></td>
</tr>
</tbody>
</table>
Regular Expressions: More Disjunction

Woodchuck is another name for groundhog!
Use the pipe | for disjunction

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>groundhog</td>
<td>woodchuck</td>
</tr>
<tr>
<td>yours</td>
<td>mine</td>
</tr>
<tr>
<td>a</td>
<td>b</td>
</tr>
<tr>
<td>[gG]roundhog</td>
<td>[Ww]oodchuck</td>
</tr>
</tbody>
</table>

```python
p = re.compile('[Ww]oodchucks?|[Gg]roundhogs?')
p.findall('Woodchucks, by any other name, such as groundhog, ' +
    'wouldchuck the same.')
```
## Regular Expressions: \(? *+ .\)

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>colou?r</code></td>
<td>Optional previous char</td>
</tr>
<tr>
<td></td>
<td><code>color</code></td>
</tr>
<tr>
<td></td>
<td><code>colour</code></td>
</tr>
<tr>
<td><code>oo*h!</code></td>
<td>0 or more of previous char</td>
</tr>
<tr>
<td></td>
<td><code>oh!</code></td>
</tr>
<tr>
<td></td>
<td><code>ooh!</code></td>
</tr>
<tr>
<td></td>
<td><code>oooh!</code></td>
</tr>
<tr>
<td></td>
<td><code>ooooh!</code></td>
</tr>
<tr>
<td><code>o+h!</code></td>
<td>1 or more of previous char</td>
</tr>
<tr>
<td></td>
<td><code>oh!</code></td>
</tr>
<tr>
<td></td>
<td><code>ooh!</code></td>
</tr>
<tr>
<td></td>
<td><code>oooh!</code></td>
</tr>
<tr>
<td></td>
<td><code>ooooh!</code></td>
</tr>
<tr>
<td><code>baa+</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>baa</code></td>
</tr>
<tr>
<td></td>
<td><code>baaa</code></td>
</tr>
<tr>
<td></td>
<td><code>baaaa</code></td>
</tr>
<tr>
<td></td>
<td><code>baaaaa</code></td>
</tr>
<tr>
<td><code>beg.n</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>begin</code></td>
</tr>
<tr>
<td></td>
<td><code>begun</code></td>
</tr>
<tr>
<td></td>
<td><code>begun</code></td>
</tr>
<tr>
<td></td>
<td><code>beg3n</code></td>
</tr>
</tbody>
</table>

*Stephen C Kleene*

Kleene *, Kleene +
<table>
<thead>
<tr>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>^[A-Z]</td>
<td><strong>Palo Alto</strong></td>
</tr>
<tr>
<td>^[^A-Za-z]</td>
<td>1 “Hello”</td>
</tr>
<tr>
<td>.$</td>
<td>The end.</td>
</tr>
<tr>
<td>.$</td>
<td>The end? The end!</td>
</tr>
</tbody>
</table>
Example

Find me all instances of the word “the” in a text.

the

Misses capitalized examples

[tT]he

Incorrectly returns other or theology

[^a-zA-Z][tT]he[^a-zA-Z]
Example in Python

- **Without grouping:**
  - >>> p = re.compile('[^a-zA-Z] [Tt]he[^a-zA-Z]', re.VERBOSE)
  - >>> m = p.findall('Yes. The cat chases the dogs that bathe. ')
  - >>> print(m) => [' The ', ' the ']

- **With grouping:**
  - >>> p = re.compile('[^a-zA-Z] ([Tt]he)[^a-zA-Z]', re.VERBOSE)
  - >>> m = p.findall('Yes. The cat chases the dogs that bathe. ')
  - >>> print(m) => ['The', 'the']
Errors

The process we just went through was based on fixing two kinds of errors:

1. Matching strings that we should not have matched (there, then, other)
   False positives (Type I errors)

2. Not matching things that we should have matched (The)
   False negatives (Type II errors)
Errors cont.

In NLP we are always dealing with these kinds of errors.

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).
Substitutions

Substitution in UNIX commands and Pythons:

```
s/regexp1/pattern/g
```

Unix:
```
  sed 's/colour/color/g' <file.txt>
```

Python:
```
p = re.compile('colour')
p.sub('color', <string>)
```
Capture Groups

- Say we want to put angles around all numbers:

  \( \text{the 35 boxes} \rightarrow \text{the <35> extra boxes} \)

- Use parens () to "capture" a pattern into a numbered register (1, 2, 3...)
- Use \1 to refer to the contents of the register

Unix:

\[
\text{sed } -E 's/([0-9]+)/<\1> extra/g'
\]

Python:

\[
\text{p = re.compile(''( [0-9]+ )', re.VERBOSE)}
\]
\[
\text{p.sub(r'<\1> extra', 'the 35 boxes')}\]
Capture groups: multiple registers

\texttt{s/the (.*\)er they (.*), the \textbackslash{}1er we \textbackslash{}2}

Matches  \texttt{‘the faster they \textit{ran}, the faster we \textit{ran}’}
But not  \texttt{‘the faster they \textit{ran}, the faster we \textit{ate}’}

Python:
\begin{verbatim}
p = re.compile(r'the \1er they \2, the faster we ran')
m = p.match('the faster they ran, the faster we ran')
m.span() => (0, 38)    m.group() => 'the faster they ran, the faster we ran'

m = p.match('the faster they ran, the faster we ate')
print(m) => None
\end{verbatim}
Capture groups: multiple registers

\texttt{s/the (\.*)er they (\.*)/the \1er we \2/g}

Substitutions:

- \texttt{the faster they ran} \to \texttt{the faster we ran}
- \texttt{the slower they wrote} \to \texttt{the slower we wrote}

Python:

```python
p = re.compile(r'the \1er they \2')
p.sub(r'the \1er we \2', 'the faster they ran') \to the faster we ran
p.sub(r'the \1er we \2', 'the slower they wrote') \to the slower we wrote
```
But suppose we don't want to capture?

Parentheses have a double function: **grouping terms** and **capturing**.

**Non-capturing** groups: add a `?` after parenthesis:

```
/(?:some|a few) (people|cats) like some \1/
```

matches **some cats like some cats**

but not **some cats like some a few**

**Python:**

```
p = p = re.compile(r'(?:some|a few) (people|cats) like some \1')
m = p.match('some cats like some cats')
m.span() => the faster we ran
m = p.match('some cats like some a few')
print(m) => the slower we wrote
```
Lookahead assertions

(?= pattern) is true if pattern matches, but is zero-width; doesn't advance character pointer

- Isaac (?=Asimov) will match 'Isaac ' only if it’s followed by 'Asimov'.

(?! pattern) true if a pattern does not match

- Isaac (?!Asimov) will match 'Isaac ' only if it’s not followed by 'Asimov'.

Simple Application: ELIZA

Early NLP system that imitated a Rogerian psychotherapist (Weizenbaum, 1966).

Uses pattern matching to match, e.g.,:
- “I need X”
and translates them into, e.g.
- “What would it mean to you if you got X?”
Simple Application: ELIZA

Men are all alike.
IN WHAT WAY

They're always bugging us about something or other.
CAN YOU THINK OF A SPECIFIC EXAMPLE

Well, my boyfriend made me come here.
YOUR BOYFRIEND MADE YOU COME HERE

He says I'm depressed much of the time.
I AM SORRY TO HEAR YOU ARE DEPRESSED
How ELIZA works

s/.*/ I’M (depressed|sad) ./ I AM SORRY TO HEAR YOU ARE \
1/
s/.*/ I AM (depressed|sad) ./ WHY DO YOU THINK YOU ARE \
1/
s/.*/ all ./ IN WHAT WAY?/
s/.*/ always ./ CAN YOU THINK OF A SPECIFIC EXAMPLE?/
Summary

Regular expressions play a surprisingly large role:
- Sophisticated sequences of regular expressions are often the first model for any text processing text.

For hard tasks, we use machine learning classifiers:
- But regular expressions are still used for pre-processing, or as features in the classifiers.
- Can be very useful in capturing generalizations.
Supplemental readings

1. Chapter 2 in Jurafsky & Martin

2. Regular expressions in Python:
   ◦ https://docs.python.org/3/howto/regex.html
   ◦ https://docs.python.org/3/library/re.html

3. Regular expressions with Sed:
   ◦ https://www.tutorialspoint.com/unix/unix-regular-expressions.htm