Basic Text Processing

Regular Expressions

Slides from Jurafsky & Martin

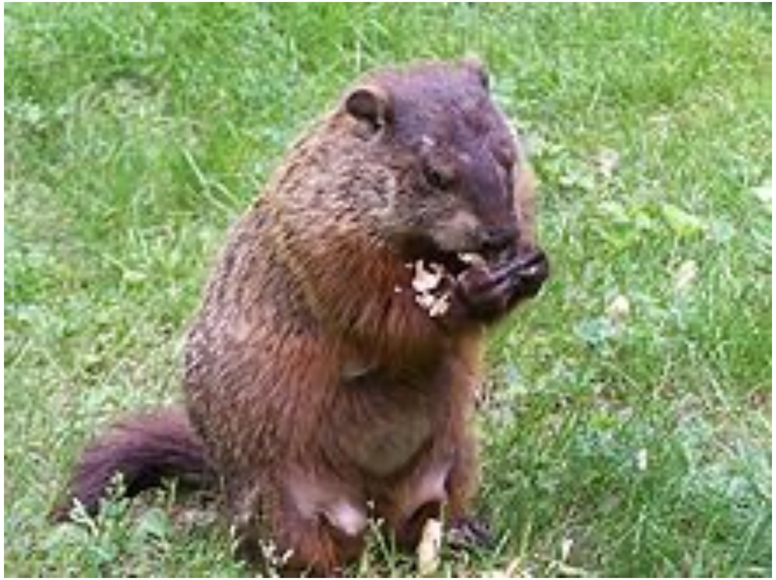
edited by RB with Sed and Python

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 []

Pattern	Matches
[wW]oodchuck	Woodchuck, woodchuck
[1234567890]	Any digit

Ranges [A-Z]

Pattern	Matches	
[A-Z]	An upper case letter	Drenched Blossoms
[a-z]	A lower case letter	my beans were impatient
[0-9]	A single digit	Chapter 1: Down the Rabbit Hole

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 []

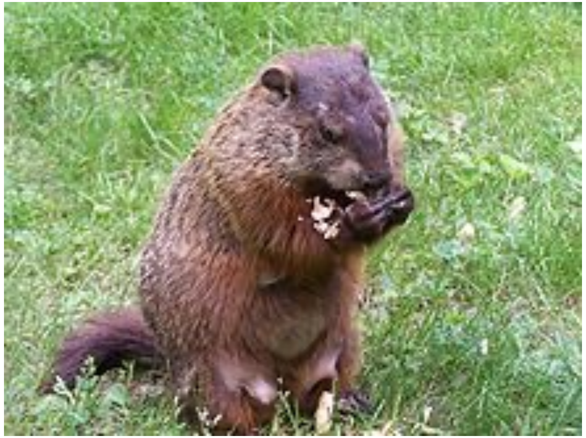
Pattern	Matches	
`[^A-Z]`	Not an upper case letter	`Oyfn pripetchik`
`[^Ss]`	Neither 'S' nor 's'	`I have no exquisite reason"`
`[^e^]`	Neither e nor ^	`Look here`
`a^b`	The pattern a carat b	`Look up a^b now`

Regular Expressions: More Disjunction

Woodchuck is another name for groundhog!

Use the pipe | for disjunction

Pattern	Matches
groundhog|woodchuck	woodchuck
yours|mine	yours
a|b|c	= [abc]
[gG]roundhog|[Ww]oodchuck	Woodchuck



```
p = re.compile('[Ww]oodchucks?|[Gg]roundhogs?')
p.findall('Woodchucks, by any other name, such as groundhog, '
          'wouldchuck the same.')
```

Regular Expressions: ? *+.

Pattern	Matches	
colou?r	Optional previous char	color colour
oo*h!	0 or more of previous char	oh! ooh! oooh! ooooh!
o+h!	1 or more of previous char	oh! ooh! oooh! ooooh!
baa+		baa baaa baaaa baaaaa
beg.n		begin begun begun beg3n

Stephen C Kleene

Kleene *, Kleene +

Regular Expressions: Anchors ^ $

Pattern	Matches
^[A-Z]	Palo Alto
^[^A-Za-z]	1 "Hello"
\.$	The end.
.$	The end? The end!

Python RE: Finding matches

- **match()**: Determine if the RE matches at the **beginning** of the string.
  - Returns a **match object**.
- **search()**: Scan through a string, looking for **any location** matching the RE.
  - Returns a **match object**.
- **findall()**: Find **all substrings** where the RE matches.
  - Returns them as a **list**.
- **finditer()**: Find **all substrings** where the RE matches.
  - Returns them as an **iterator**.

Python RE: Match Objects

Match objects have 4 main methods:

- group():
- start():
- end():
- span()

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 Python:

```
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:

  the 35 boxes → *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:

```
sed -E ‘s/([0-9]+)/<\1> extra/g’
```

Python:

```
p = re.compile('( [0-9]+ )', re.VERBOSE)
p.sub(r'<\1> extra’, ‘the 35 boxes')
```

Capture groups: multiple registers

```
s/the (.*)er they (.*), the \1er we \2/g
```

Matches ‘*the faster they ran, the faster we ran*‘
But not ‘*the faster they ran, the faster we ate*’

`Python:`

```
p = re.compile(r'the (.*)er they (.*), r'the \1er we \2 ')
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
```

Capture groups: multiple registers

```
s/the (.*)er they (.*)/the \1er we \2/g
```

Substitutions:

the faster they ran => *the faster we ran*

the slower they wrote => *the slower we wrote*

```
Python:
    p = re.compile(r'the (.*)er they (.*)')
    p.sub(r'the \1er we \2', 'the faster they ran’)      => the faster we ran
    p.sub(r'the \1er we \2', 'the slower they wrote’) => 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.group() => 'some cats like some cats’
m = p.match('some cats like some people’)
print(m) => None
```

Lookahead and Lookbehind assertions

`(?= pattern)` is true if pattern matches ahead, 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’.

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

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

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
   - https://web.stanford.edu/~jurafsky/slp3/2.pdf

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