Data Foundations: Regular Expressions

Some slides from Stanford NLP Course

Instructor: Anthony Rios

Outline

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`

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`

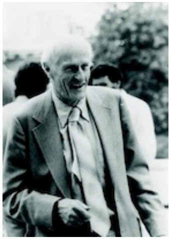
Regular Expressions: More Disjunction

- Woodchucks is another name for groundhog!
- The pipe | for disjunction

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

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!

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]`

Regular Expressions with Python

A **regular expression** is a sequence of characters that forms a search pattern, mainly for use in pattern matching with strings, or string matching.

Regular Expressions and Python

```
>>> myString = ‘test string abcd’

>>> import re

>>> re.search(‘From:’, myString) # re.search(PATTERN, STRING)

>>> re.search(‘From:’, myString) is None
True

>>> re.search(‘test’, myString)
<_sre.SRE_Match object; span=(0, 4), match='test'>
```

If Statements and re.search

example.py

```
import re
myList = ['line 1', 'test 1', 'line 2', 'test 2', 'line 3']
for item in myList:
        if re.search('test', item): # Short for "re.search is not None"
                print(item)
```

```
anthony@MacBook:~$ python example.py
test 1
test 2
```

Exercise 1

Write code to loop over mbox.txt and counts the number of lines that contain the “From:” substring. Use the **re** package for this exercise.

5 minutes

Regular Expressions Question

If we can just use **“if ‘test’ in item”**, why should we use **re.search**?

Basic Notation

- (a | b) – a **OR** b
- (a | b)***** – **zero or more** occurrences of a OR b
- (a | b)**+** – **one or more** occurrences of a OR b

Basic Notation - Python

```
>>> myString = ‘The guppy, also known as rainbow fish, is one of the world’s most widely distributed tropical fish’

>>> re.search("gupp(y|ies)", myString) #match guppy or guppies
<_sre.SRE_Match object; span=(4, 9), match='guppy'>

>>> myString = ‘I have guppis’
>>> re.search("gupp(y|ies)", myString) # Returns None
```

Basic Notation - Python

```
>>> myString = ‘The Ohio State University’

>>> re.search(“The*”, myString)
<_sre.SRE_Match object; span=(0, 3), match='The'>

>>> myString = ‘Theeeeeee Ohio State University’

>>> re.search(“The*”, myString)
<_sre.SRE_Match object; span=(0, 9), match='Theeeeeee'>

>>> myString = ‘Th Ohio State University’

>>> re.search(“The*”, myString)
<_sre.SRE_Match object; span=(0, 2), match='Th'>
```

Basic Notation - Python

```
>>> myString = 'Th Ohio State University'
```

```
>>> re.search("The+", myString) # Returns None
>>>
```

Basic Notation

- [Bb] – match the character B OR b
- [0-9] – match all numbers 0 to 9
- [a-b] – all alphabets from a to z (lowercase)
- [A-Z] – all alphabets from A to Z (uppercase)
- [A-Za-z] – all alphabets from A to Z (uppercase or lowercase)

Basic Notation - Python

```
>>> myString = 'Woodchuck'

>>> re.search("[Ww]oodchuck", myString)
<_sre.SRE_Match object; span=(0, 9), match='Woodchuck'>

>>> myString = 'woodchuck'

>>> re.search("[Ww]oodchuck", myString)
<_sre.SRE_Match object; span=(0, 9), match='woodchuck'>

>>> myString = 'oodchuck'

>>> re.search("[Ww]oodchuck", myString) # Returns None
>>>
```

Language

A **language** is the countable set of **all possible strings** over a given alphabet.

Regular Expression vs Language

Regular Expression — Language

- a | b — {a,b}
- (a | b)(a | b) — {aa, ab, ba, bb}
- a | a*b — {a, b, ab, aab, a. . . ab}
    - a OR a (one or more) b

More Notation

- $\wedge$ a – strings that start with a
- a\$ – strings that end with a
- . – matches any character except a newline
- \S – Matches any non-whitespace character

re.findall

```
>>> text = “He was carefully disguised but captured quickly by police.”
```

```
>>> re.findall(”\S+ly”,text)
[‘carefully’, ‘quickly’]
```

re.search vs re.findall

re.search searches through string looking for the **first location** where the regular expression pattern produces a match

re.findall returns **all non-overlapping matches** of pattern in string, as a list of strings.

Exercise 2

Write a program to look for lines of the form in the “mbox.txt” file:

New Revision: 39772

Extract the number from each of the lines using a regular expression and the findall() method. Compute the average of the numbers and print out the average.

⏱ 12 minutes

The Regular Expression Workflow

Assume we want to count all the occurrences of the English article “the”. The following is a natural workflow for regex development:

- re.findall(“**the**”, myString)
  - This will ignore capital words (i.e., The)
- re.findall(“**[tT]he**”, myString)
  - This ignores word boundaries. We will match “other” and “theology”.
- re.findall(“**b[tT]he****b** ”, myString)

The Regular Expression Workflow

Suppose we wanted to match “the” in more complex contexts where it may end with an underscore or numbers. For this we need to explicitly state the word boundaries:

- re.findall(“**\b[tT]he\b** ”, myString)
  - Will not match “the_” or “the123”
  - this could be useful when parsing twitter screen names.
- re.findall(“**[∧a-zA-Z][tT]he[∧a-zA-Z]**”, myString)
  - This will not match “the” at the beginning or end of a line.
- re.findall(“**(∧ | [∧a-zA-Z])[tT]he[(dollar | ∧a-zA-Z])**”, myString)

False Negatives/Positives

The process we just went through was based on fixing two kinds of errors: **false positives**, strings that we incorrectly matched like “other” or “there”, and **false negatives**, strings that we incorrectly missed, like “The”

Reducing the overall error rate for an application thus involves two antagonistic efforts:

- Increasing **precision** (minimize false positives)
- Increasing **recall** (minimize false negatives)

Exercise 3

For the following string:

text = "any machine with more than 6 GHz and 500 GB of disk space for less than $999.99"

Develop a regular expression that will extract **all** of the following information:

- 6 GHz
- 500 GB
- Mac
- $999.99

⏱10 minutes

Exercise 4

`https://regexone.com`