CSCE604135 | Perolehan Informasi (Information Retrieval)

Text Processing pt. 2

Radityo Eko Prasojo, PhD

Contents

- Regular Expression

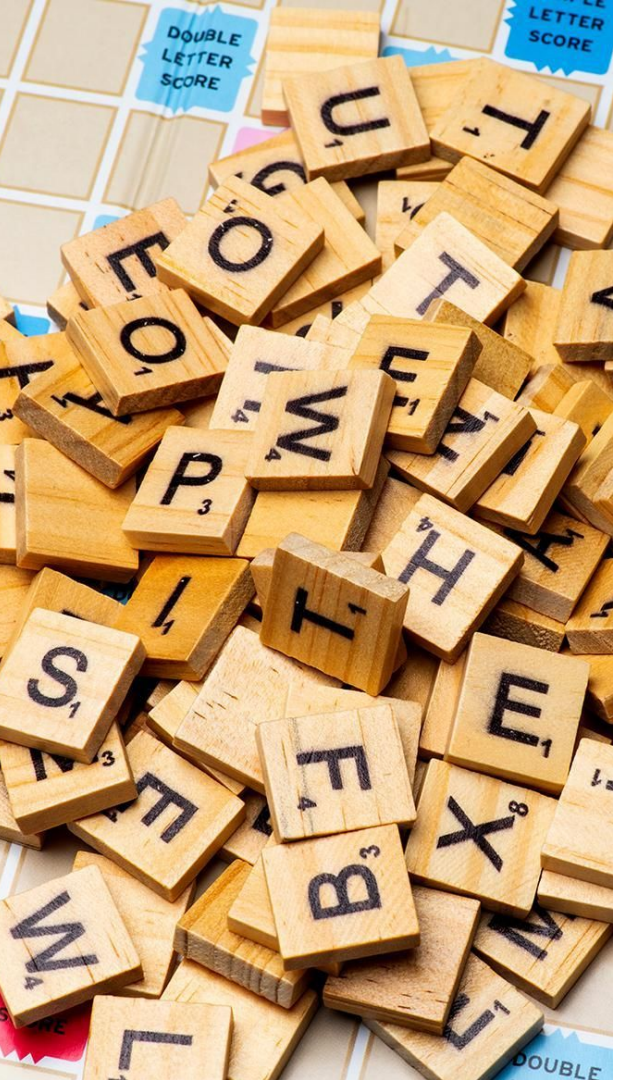
Follow the pattern!

- Many text processing steps can be solved by following some patterns
- E.g., (1) we split paragraphs into sentences by looking at the ending punctuation, but with some exceptions, e.g. dots in website and email

  Website is always <somesubdomain>.<somedomain>.<sometld>
  Email is always <somename>@<somewebsite>

- (2) we split sentences into tokens, typically by spaces, but with some exceptions like phone number

  `<countrycode>(<optionalspace><4digits>) repeat 4 times`



Regular expression

- Or often referred to as “**Regex**”
- A **string** that encodes a **search pattern**
- The search pattern can then be applied to **a target string**, that is to:
  (1) find substrings that match the pattern
  (2) optionally, replace them with another string, or
  (3) to validate the whole target string whether it matches a certain criteria

Use case example (beyond IR!)

- Find user mention on tweets

  Hey @realDonaldTrump, what do you think of @elonmusk?

- Replace username with a mask for anonymization

  Hey <user1>, what do you think of <user2>?

- Validate an email address

  ridho@kata.ai → valid
  ridho@kata@ai → invalid

Regex: Building Blocks

A regex string comprises of:

Metacharacters: `{}[]()^$.|*+?\`

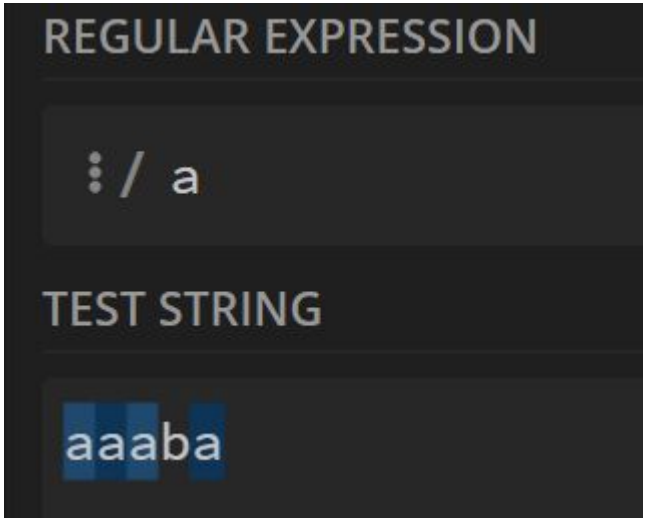
Literals: all other characters

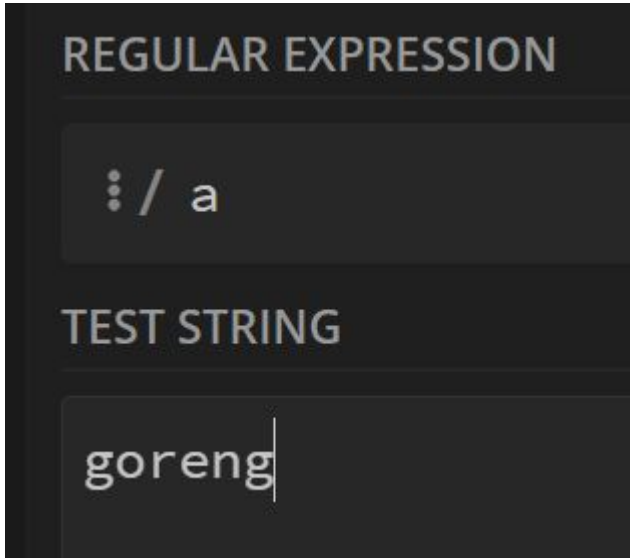
Some characters can be literal or meta depending on the situation

Regex: Literals

- The string ‘a’ is a valid regex, which contain one literal
- Literal means that when applied to a target string, it tries to match the regex **literally**
- The regex matches the following target strings:
  - ‘a’, once
  - ‘aa’, twice
  - ‘aaaa’, four times
  - ‘aaaba’ four times
- But doesn’t match strings not containing ‘a’ at all

Testing regex using tools e.g. regex101.com





Regex: Literals, two more examples

- ‘aa’ is a valid regex with 2 literals
- When applied to the following target strings:
  - ‘a’, doesn’t match
  - ‘aa’, matches once
  - ‘aaaa’, matches twice
  - ‘aaaba’, matches once
- ‘ab1’ is a valid regex with 3 literals
- The regex matches the following target strings:
  - ‘ab1’, once
  - ‘ab1ab1, twice
  - ‘ab1xxxab1GG’, twice

Sure, literals are boring

- What’s the difference with normal string comparisons?
- That’s where **metacharacters** come in order to build more complex and useful patterns
- We will see all the their usages below

Character set

- So far, we have seen regex exactly matching character(s) using some literal(s). What if we do not know the exact string?
- **Example:** you work with art magazine documents, you want to know all documents about art using the color “grey”, but sometimes it is spelled “gray” (British vs American English)
- In regex, you can use the pattern string: `‘gr[ae]y’` to match with either possibilities!
- This is called character set

Character set

- Defined by characters within square brackets
- Regex matches **any single** character within the square brackets
- So the regex `'[abcdefghijklmnopqrstuvwxyz]'` matches with any target string with a lowercase alphabet
  - With 'z', once (with z)
  - With 'abc', three times (with a, b, and then c)
  - With 'ab1cH', three times (with a, b, and then c)
- The order of the characters within the square bracket does not matter
  - I.e. `gr[ae]y` and `gr[ea]y` are the same

Character set - dash

- The **dash character** `'-'` can be used as a shorthand
- `'[abcdefghizklmnopqrstuvwxyz]'` can be shortened into `'[a-z]'`
- Can be combined, e.g. `` `[a-zA-Z0-9]` `` matches all alphanumeric characters
- `` `[a-zA-Z0-9_]` `` matches all alphanumeric characters and underscores
- **Note**: outside the angle brackets, the dash is treated as a literal

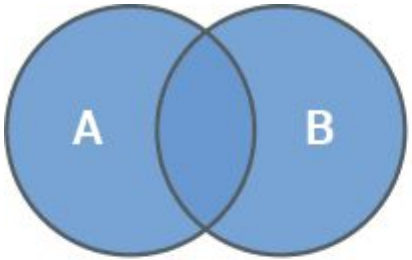
Character set - operators: negation

- Negation ‘^’, matches all characters except those inside the bracket
- ‘`[^abc]`’ matches anything beside a, b, or c, whereas `` `[^0-9]` `` matches any non-numeric character
- Example: `` `[^abc][^0-9]` ``, when applied to the following target strings
  - ‘9A’, matches once
  - ‘aa90’, doesn’t match
  - ‘da90k’, matches twice

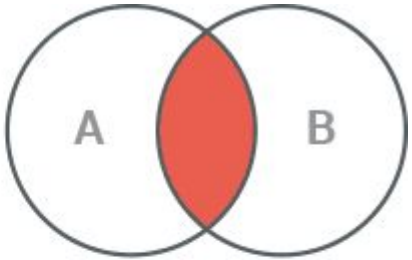
Character set - operators: subtraction/difference

- Overloads the character dash '`-`', with syntax '`[class-[diff]]`'
- Example: '`[a-z-[aiueo]]`' matches any lowercased alphabet that is not a vowel, i.e. any consonant
- Can be nested, e.g.: `` `[0-9-[0-6-[0-3]]]` ``, first subtract 0-3 from 0-6, yielding 4-6, then subtract 4-6 from 0-9, yielding `[0-37-9]`, i.e. it will match with 0123789
- Negation takes precedence over subtraction: e.g. `` `[^1234-[5678]]` `` means (not 1234) minus 5678

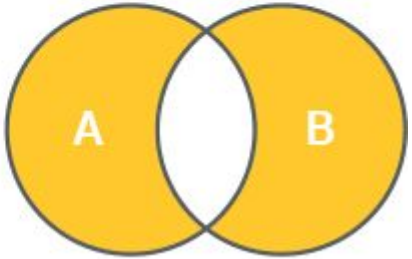
Character set - operators: union, intersection, symmetric difference



'[A||[B]]'



'[A&&[B]]'



'[A~~[B]]'

Character set - shorthands

- `\d` = all digits
- `\D` = `[^\d]` = all non digits
- `\w` = `[A-Za-z0-9_]` = all “alphanumeric characters”
- `\W` = `[^\w]` = all “non-alphanumeric characters”
- `\s` = all whitespaces (space, tab, carriage returns)
- `\S` = all non-whitespaces

Wildcard dot (.)

- Matches any character
- Similar: `\N` matches any character except newlines
- Five dots ‘ . . . . . ’ matches any five characters
- If we want to match a literal dot (period), add backslash before it ’ \ . ’
- For example, the regex ‘ . . . ’ matches both the target strings **Dr.** and **LOL**, but the regex ‘ . . \ . ’ matches only the first one.

Optional items (?)

- ‘harbou?r’ matches with either harbor or harbour
- ‘[a-z][a-z]?’ matches with one or two lowercase letters

Repetitions

- `'*'` repeats 0 or more times, `'+'` repeats 1 or more times
  - `'a*'` matches a, aaa, and bbbb
  - `'a+'` matches a and aaa but not bbbb
- `'{n}'` repeats exactly n times, `'{n,}'` repeats at least n times
- `'{n,m}'` repeats at least n times and at most m times
- `'a{0,1}'` is the same as `'a?'`, `'a{0,}'` is the same as `'a*'`, `'a{1,}'` is the same as `'a+'`

Repetitions - greediness

- ‘<.+>’ matches with everything that starts with < and ends with >, like HTML tags, greedily
- However, this often works against what we want
  - i.e., the regex greedily matches the whole `<div>The paragraph text</div>`, but sometimes we want to exclusively get the tags, i.e. `<div>`The paragraph text`</div>`
- To enforce **lazy search behaviour**, we can add ? after the repetition meta, that is: ‘<.+**?**>’
- This way, it stops on the first enclosing bracket > it founds
- This lazy search can be applied to any repetition terms (using options, stars, and curly brackets)

Repetitions - lazy search is slow!

- The lazy search works by first trying to fulfill the minimum requirements of the repetition term, then try to match the remaining pattern. If it fails, it backtracks. Repeat until match.
- In the case of the target string using the pattern ‘<.+?>’, first it matches ‘<’, then the dot matches with the ‘d’. Because + matches at least once, then it already satisfies the minimum requirement. Next, it tries to match the closing parenthesis ‘>’, but the next character in the target string is ‘i’, so it does not match. Then it backtracks and matches the i with the repetition. Repeats until it reaches the first ‘>’.
- Very slow because of the repeating backtracks!

Repetitions - alternative to lazy

- Use the negation!
- ‘<[^>]+>’ matches exactly the same, but faster because no backtracking necessary!

Anchors (^ and $)

- ^ matches the beginning of a target string, whereas $ matches the end
- ^a matches the a in target string abc, but ^b does not match at all
- c$ matches the c in target string abc, but b$ does not match at all
- ^abc$ matches only the string abc and not others
- ^[A-Z].+[\.?!]$ is a naive, simple pattern to verify whether a text is always properly capitalized at the beginning and ends with a proper punctuation.

Groups

- `'\w+ Potter'` matches with any character from the Potter family* mentioned in a document
- We can add grouping as brackets, e.g.: `'(\w+) Potter'`. In this case the group then **can be accessed** to obtain the character first name
- We can add many groups and subgroups in a pattern, e.g. if a survey form data is stored in a text document and we want to extract the birth place and date, we can use the pattern

  `'Birth: (\w+), (\d{1,2}-\d{1,2}-(\d{4}))'`

  where the groups represent the place, full date, and the year

Groups - options

- You may want to search for two options, e.g. finding cat or dog from a target string. In Regex, you can use `'(cat|dog)'` to match either literals
- Another example: `'(Mo|Tu|We|Th|Fr|Sa|Su)'` matches any possible two-character day codes

Groups - naming

- We can name groups, so instead of having

```
'Birth: (\w+), (\d{1,2}-\d{1,2}-(\d{4}))'
```

- We can have

```
'Birth: (?P<p>\w+), (?P<d>\d{1,2}-\d{1,2}-(?P<y>\d{4}))'
```

Where the string between brackets <> are the name. The group name can be used to make **accessing group** programmatically easier, and also for **backreference**.

Groups - backreference

- Assuming no attributes in the html tags, we can capture HTML tags with

  ```
  '<[^>]>.+<\\[^>]>'
  ```

  But this does not ensure the opening and closing tags contain the same tag.

- One way to ensure that is by using backreference:

  ```
  '<([^>])>.+<\\(\1)>'
  ```

  The `\1` refers to the first group detected by the regex. This way, the opening and closing tags are ensured to be the same.

Groups - backreference with name

- Alternatively, the backreference can utilize names:

```
'<(?P<opening>[^>])>.+<\\(?P=opening)>'
```

The `?=opening` refers to the first group which we named `opening`

Conditional matching - lookahead and lookbehind

- **Negative lookahead**: `'Red(?!Carpet)'` matches Red as long as it is not followed by Carpet
- **Positive lookahead**: `'Red(?=Carpet)'` matches Red as long as it is followed by Carpet. Note: `'RedCarpet'` would also matches the same string, but will return the word Carpet as well
- **Negative lookbehind**: `'(?<!Red)Carpet'` matches Carpet as long as it is not preceded by Red
- **Positive lookbehind**: `'(?<=Red)Carpet'` matches Carpet as long as it is preceded by Red. Note: `'RedCarpet'` would also matches the same string, but will return the word Red as well

Regex ‘Flavor’

- There are small differences in regex implementation between different programming languages.
- Differences include small syntax differences and functionality supports
- The complete list of functionalities are maintained at https://www.regular-expressions.info/, while syntax differences need to be studied from each language documentation

Further references

- https://www.regular-expressions.info/
- https://docs.python.org/3/library/re.html - Official, built in regex functionality in python
- https://bitbucket.org/mrabarnett/mrab-regex/src/hg/ - 3rd party regex in python, with more functionalities (e.g. set operators)
- ‘Mastering Regular Expressions’ book by Jeffrey Friedl

Supported by the Kampus Merdeka grant of
Ministry of Education, Culture, Research, and Technology
of Republic of Indonesia