Regular Expressions, Web Scraping, APIs

JSC 370: Data Science II

Today's goals

- Introduction to Regular Expressions
- Understand the fundamentals of Web Scrapping
- Learn how to use an API

Regular Expressions: What is it?

A regular expression (shortened as regex or regexp; also referred to as rational expression) is a sequence of characters that define a search pattern. -- Wikipedia

Regular Expressions: Why should you care?

We can use Regular Expressions for:

- Validating data fields, email address, numbers, etc.
- Searching text in various formats, e.g., addresses, there are many ways to write an address.
- Replace text, e.g., different spellings, `Storm`, `Stôrm`, `Stórm` to `Storm`.
- Remove text, e.g., tags from an HTML text, `<name>George</name>` to `George`.

Regular Expressions 101: Metacharacters

What makes *regex* special is metacharacters. While we can always use *regex* to match literals like `dog`, `human`, `1999`, we only make use of all *regex* power when using metacharacters:

- `.` Any character except new line
- `^` beginning of the text
- `$` end of the text

- `[regex]` Match a single character in `regex`, e.g.
  - `[0123456789]` Any number
  - `[0-9]` Any number in the range 0-9
  - `[a-z]` Lower-case letters
  - `[A-Z]` Upper-case letters
  - `[a-zA-Z]` Lower or upper case letters.
  - `[a-zA-Z0-9]` Any alpha-numeric

- `[^regex]` Match any except those in `regex`, e.g.
  - `[^0123456789]` Match any except a number
  - `[^0-9]` Match anything except in the range 0-9
  - `[^./ ]` any except dot, slash, and space.

Regular Expressions 101: Metacharacters (cont. 1)

Ranges, e.g., `0-9` or `a-z`, are locale- and implementation-dependent, meaning that the range of lower case letters may vary depending on the OS's language. To solve for this problem, you could use Character classes. Some examples:

- `[:lower:]` lower case letters in the current locale, could be `[a-z]`
- `[:upper:]` upper case letters in the current locale, could be `[A-Z]`
- `[:alpha:]` upper and lower case letters in the current locale, could be `[a-zA-Z]`
- `[:digit:]` Digits: 0 1 2 3 4 5 6 7 8 9
- `[:alnum:]` Alpha numeric characters `[:alpha:]` and `[:digit:]`.
- `[:punct:]` Punctuation characters: ! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [\ ] ^ _ ` { | } ~.

For example, in the locale `en_US`, the word `Ḧóla` IS NOT fully matched by `[a-zA-Z]+`, but IT IS fully matched by `[[:alpha:]]+`.

Other important Metacharacters:

- `\s` white space, equivalent to `[\r\n\t\f\v ]`
- | or (logical or).

Regular Expressions 101: Metacharacters (cont. 2)

These usually come together with specifying how many times (repetition):

- `regex?` Zero or one match.
- `regex*` Zero or more matches
- `regex+` One or more matches
- `regex{n,}` At least `n` matches
- `regex{,m}` at most `m` matches
- `regex{n,m}` Between `n` and `m` matches.

Where `regex` is a regular expression

Regular Expressions 101: Metacharacters (cont. 3)

There are other operators that can be very useful,

- `(regex)` Group capture.
- `(?:regex)` Group operation without capture.
- `(?=regex)` Look ahead (match)
- `(?!regex)` Look ahead (don't match)
- `(?<=regex)` Look behind (match)
- `(?<!regex)` Look behind (don't match)

Group captures can be reused with `\1`, `\2`, ..., `\n`.

More (great) information here https://regex101.com/

Regular Expressions 101: Examples

Here we are extracting the first occurrence of the following regular expressions (using `stringr::str_extract()`):

regex	Hanna Perez [name]	The 年 year was 1999	HaHa, @abc said that	GoGo trojans #2020!
.{5}	Hanna	The 年	HaHa,	GoGo
n{2}	nn			
[0-9]+		1999		2020
\s[a-zA-Z]+\s	Perez	year	said	trojans
\s[[:alpha:]]+\s	Perez	年	said	trojans
[a-zA-Z]+ [a-zA-Z]+	Hanna Perez	year was	abc said	GoGo trojans
([a-zA-Z]+\s?){2}	Hanna Perez	The	HaHa	GoGo trojans
([a-zA-Z]+)\1	nn		HaHa	GoGo
(@|#)[a-z0-9]+			@abc	#2020
(?<=#|@)[a-z0-9]+			abc	2020
$[a-z]+$	[name]			

Regular Expressions 101: Examples (cont. 1)

1. .{5} Match **any character** (except line end) five times.
2. n{2} Match the letter **n** twice.
3. [0-9]+ Match **any number** at least once
4. \s[a-zA-Z]+\s Match a **space**, **any lower or upper case letter** at least once, and a **space**.
5. \s[[:alpha:]]+\s Same as before but this time .
6. [a-zA-Z]+ [a-zA-Z]+ Match two sets of letters separated by one space.
7. ([a-zA-Z]+\s?){2} Match **any lower or upper case letter** at least once, maybe followed by a white space, twice.
8. ([a-zA-Z]+)\1 Match **any lower or upper case letter** at least once, and then match the same pattern again.
9. (@|#)[a-z0-9]+ Match either the `@` or `#` symbol, followed by one or more **lower case letter** or **number**.
10. (?<=#|@)[a-z0-9]+ Match one or more **lower case letter** or **number** that follows either the `@` or `#` symbol.
11. \ [[a-z]+\ ] Match the symbol `[`, at least one **lower case letter**, and the symbol `]`.

Regular Expressions 101: Functions in R

1. Lookup text: `base::grepl()`, `stringr::str_detect()`.
2. Similar to `which()`, which elements are `TRUE` `base::grep()`, `stringr::str_which()`
3. Replace the first instance: `base::sub()`, `stringr::str_replace()`
4. Replace all instances: `base::gsub()`, `stringr::str_replace_all()`
5. Extract text: `base::regmatches()`, `stringr::str_extract()` and `stringr::str_extract_all()`.

Regular Expressions 101: Functions in R (cont.)

For example, like in Twitter, let's create a regex that matches usernames or hashtags with the following pattern:

`(@|#)([[:alnum:]]+)`

Code	@Hanna Perez [name] #html	The @年 year was 1999	HaHa, @abc said that @z
`str_detect(text, pattern)` or `grepl(pattern, text)`	TRUE	TRUE	TRUE
`str_extract(text, pattern)`	@Hanna	@年	@abc
`str_extract_all(text, pattern)`	[@Hanna, #html]	[@年]	[@abc, @z]
`str_replace(text, pattern, "\1justinbieber")`	@justinbieber Perez [name] #html	The @justinbieber year was 1999	HaHa, @justinbieber said that @z
`str_replace_all(text, pattern, "\1justinbieber")`	@justinbieber Perez [name] #justinbieber	The @justinbieber year was 1999	HaHa, @justinbieber said that @justinbieber

Note: While it is not showing in the table, the group replacement was scaped, i.e., `\\1` instead of `\1` in the code.

Data

This week we will use a dataset consisting of medical transcriptions from https://www.mtsamples.com/. See the readme on the course git. The dataset consists of 4999 rows and 6 columns: “X”, “description”, “medical_specialty”, “sample_name”, “transcription” and “keywords”.

```
library(data.table)
library(stringr)

fn <- "mtsamples.csv"
if (!file.exists(fn))
  download.file(
    url = "https://github.com/JSC370/jsc370-2022/blob/main/data/medical_transcriptions/mtsamples.
    destfile = fn
  )
mtsamples <- fread(fn, sep = ",", header = TRUE)
```

Regex to Lookup Text: Tumor

We would like to see if this is a tumor-related entry. For that we can search through the "description" using grepl in the following code:

```
# How many entries contain the word tumor
mtsamples[grepl("tumor", description, ignore.case = TRUE), .N]
```

```
## [1] 67
```

```
# Generating a column tagging tumor
mtsamples[, tumor_related := grepl("tumor", description, ignore.case = TRUE)]

# Taking a look at a few examples
mtsamples[tumor_related == TRUE, .(description)][1:3,]
```

```
##                                                                                                                c
## 1:                                    Transurethral resection of a medium bladder tumor (TURBT), left lat
## 2:                                                  Transurethral resection of the bladder tumor (TURE
## 3:  Cystoscopy, transurethral resection of medium bladder tumor (4.0 cm in diameter), and direct bladc
```

Notice the `ignore.case = TRUE`. This is equivalent to transforming the text to lower case using `tolower()` before passing the text to the regular expression function.

Regex Lookup text: Pronoun of the patient

Now, let's try to guess the pronoun of the patient. To do so, we could tag by using the words *he*, *his*, *him*, *they*, *them*, *theirs*, *ze*, *hir*, *hirs*, *she*, *hers*, *her* (see this article on sexist text):

```
mtsamples[, pronoun := str_extract(
  string  = tolower(transcription),
  pattern = "he|his|him|they|them|theirs|ze|hir|hirs|she|hers|her"
)]
```

What is the problem with this approach?

Regex Lookup text: Pronoun of the patient (cont. 1)

For this we use the following regular expression:

`(?<=\W|^)(he|his|him|they|them|theirs|ze|hir|hirs|she|hers|her)(?=\W|$)`

Bit by bit this is:

- `(?<=regex)` lookback search.
  - `\W` any non alpha numeric character, this is equivalent to `[^[:alnum:]]`, | or
  - `^` the beginning of the text,
- `he|his|him...` any of these words,
- `(?=regex)` followed by,
  - `\W` any no alpha numeric character, this is equivalent to `[^[:alnum:]]`, | or
  - `$` the end of the text.

```
mtsamples[, pronoun := str_extract(
  string  = tolower(transcription),
  pattern = "(?<=\\W|^)(he|his|him|they|them|theirs|ze|hir|hirs|she|hers|her)(?=\\W|$)"
  )]
mtsamples[1:10, pronoun]
```

```
##  [1] "she" "he"  "he"  NA    NA    "she" "she" NA    NA    NA
```

Regex Lookup text: Pronoun of the patient (cont. 2)

```
mtsamples[, table(pronoun, useNA = "always")]
```

```
## pronoun
##   he  her  him  his  she them they <NA>
##  767  394   29  361  870   18   67 1176
```

Regex Extract Text: Type of Cancer

- Imagine now that you need to see the types of cancer mentioned in the data.
- For simplicity, let's assume that, if specified, it is in the form of `TYPE cancer`, i.e. single word.
- We are interested in the word before cancer, how can we capture this?

Regex Extract Text: Type of Cancer (cont 1.)

We can just try to **extract** the phrase "`[some word] cancer`", in particular, we could use the following regular expression

`[[:alnum:]-_]{4,}\s*cancer`

Where

- `[[:alnum:]-_]{4,}` captures any alphanumeric character, including `-` and `_`. Furthermore, for this match to work there must be at least 4 characters,
- `\s*` captures 0 or more white-spaces, and
- `cancer` captures the word cancer:

```
mtsamples[, cancer_type := str_extract(tolower(keywords), "[[:alnum:]-_]{4,}\\s*cancer")]
mtsamples[, table(cancer_type)]
```

```
## cancer_type
##        anal cancer     bladder cancer      breast cancer       colon cancer
##                  1                  6                 16                 12
## endometrial cancer  esophageal cancer        lung cancer     ovarian cancer
##                  5                  1                  8                  1
##   papillary cancer    prostate cancer     uterine cancer
##                  2                 14                  4
```

Fundamentals of Web Scrapping

What?

> Web scraping, web harvesting, or web data extraction is data scraping used for extracting data from websites -- Wikipedia

How?

- The `rvest` R package provides various tools for reading and processing web data.
- Under-the-hood, `rvest` is a wrapper of the `xml2` and `httr` R packages.

(in the case of dynamic websites, take a look at selenium))

Web scraping raw HTML: Example

We would like to capture the table of COVID-19 death rates per country directly from Wikipedia.

```
library(rvest)
library(xml2)

# Reading the HTML table with the function xml2::read_html
covid <- read_html(
  x = "https://en.wikipedia.org/wiki/COVID-19_pandemic_death_rates_by_country"
  )

# Let's the the output
covid
```

```
## {html_document}
## <html class="client-nojs" lang="en" dir="ltr">
## [1] <head>\n<meta http-equiv="Content-Type" content="text/html; charset=UTF-8 ...
## [2] <body class="mediawiki ltr sitedir-ltr mw-hide-empty-elt ns-0 ns-subject  ...
```

Web scraping raw HTML: Example (cont 1.)

- We want to get the HTML table that shows up in the doc. To do this, we can use the function `xml2::xml_find_all()` and `rvest::html_table()`
- The first will locate the place in the document that matches a given **XPath** expression.
- XPath, XML Path Language, is a query language to select nodes in a XML document.
- A nice tutorial can be found here
- Modern Web browsers make it easy to use XPath!

Live Example! (inspect elements in Google Chrome, Mozilla Firefox, Internet Explorer, and Safari)

Web scraping with `xml2` and the `rvest` package (cont. 2)

Now that we know what is the path, let's use that and extract

```
table <- xml_find_all(covid, xpath = "/html/body/div[3]/div[3]/div[5]/div[1]/table[2]")
table <- html_table(table) # This returns a list of tables
head(table[[1]])
```

```
## # A tibble: 1 × 2
##   X1    X2
##   <lgl> <chr>
## 1 NA    Wikimedia Commons has media related to COVID-19 death rates per populat…
```

Web APIs

What?

> A Web API is an application programming interface for either a web server or a web browser. -- Wikipedia

Some examples include: twitter API, facebook API, Gene Ontology API

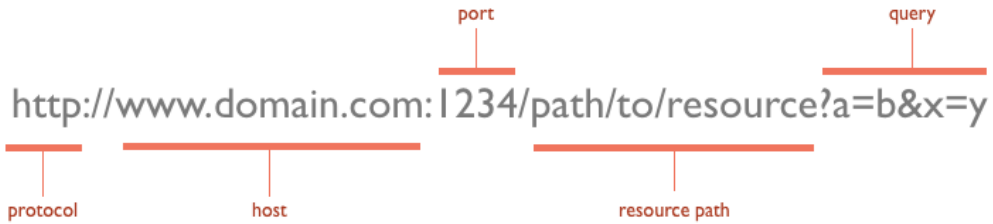
How?

You can request data, the **GET method**, post data, the **POST method**, and do many other things using the HTTP protocol.

How in R?

For this part, we will be using the `httr()` package, which is a wrapper of the `curl()` package, which in turn provides access to the `curl` library that is used to communicate with APIs.

Web APIs with curl



Structure of a URL (source: "HTTP: The Protocol Every Web Developer Must Know - Part 1")

Web APIs with curl

Under-the-hood, the `httr` (and thus `curl`) sends request somewhat like this

```
curl -X GET https://google.com -w "%{content_type}\n%{http_code}\n"
```

A get request (`-X GET`) to `https://google.com`, which also includes (`-w`) the following: `content_type` and `http_code`:

```
<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>301 Moved</TITLE></HEAD><BODY>
<H1>301 Moved</H1>
The document has moved
<A HREF="https://www.google.com/">here</A>.
</BODY></HTML>
text/html; charset=UTF-8
301
```

We use the `httr` R package to make life easier.

Web API Example 1: Gene Ontology

- We will make use of the Gene Ontology API.
- We want to know what genes (human or not) are **involved in** the function **antiviral innate immune response** (go term GO:0140374), looking only at those annotations that have evidence code ECO:0000006 (experimental evidence):

```
library(httr)
go_query <- GET(
  url   = "http://api.geneontology.org/",
  path  = "api/bioentity/function/GO:0140374/genes",
  query = list(
    evidence          = "ECO:0000006",
    relationship_type = "involved_in"
  ),
  # May need to pass this option to curl to allow to wait for at least
  # 60 seconds before returning error.
  config = config(
    connecttimeout = 60
    )
)
```

We could have also passed the full URL directly...

Web API Example 1: Gene Ontology (cont. 1)

Let's take a look at the curl call:

```
curl -X GET "http://api.geneontology.org/api/bioentity/function/GO:0140374/genes?evidence=ECO%3A0
```

What `httr::GET()` does:

```
> go_query$request
## <request>
## GET http://api.geneontology.org/api/bioentity/function/GO:0140374/genes?evidence=ECO%3A000000
## Output: write_memory
## Options:
## * useragent: libcurl/7.58.0 r-curl/4.3 httr/1.4.1
## * connecttimeout: 60
## * httpget: TRUE
## Headers:
## * Accept: application/json, text/xml, application/xml, */*
```

Web API Example 1: Gene Ontology (cont. 2)

Let's take a look at the response:

```
go_query
```

```
## Response [http://api.geneontology.org/api/bioentity/function/GO:0140374/genes?evidence=ECO%3A0000006&r
##   Date: 2022-02-11 19:50
##   Status: 200
##   Content-Type: application/json
##   Size: 28.3 kB
## {"associations": [{"id": "4d4749094d47493a31393138303436095a64686863310909474...
```

Remember the codes:

- 1xx: Information message
- 2xx: Success
- 3xx: Redirection
- 4xx: Client error
- 5xx: Server error

Web API Example 1: Gene Ontology (cont. 3)

We can extract the results using the `httr::content()` function

```
dat <- content(go_query)
dat <- lapply(dat$associations, function(a) {
  data.frame(
    Gene        = a$subject$id,
    taxon_id    = a$subject$taxon$id,
    taxon_label = a$subject$taxon$label
  )
})
dat <- do.call(rbind, dat)
str(dat)
```

```
## 'data.frame':    23 obs. of  3 variables:
##  $ Gene       : chr  "MGI:1918046" "ZFIN:ZDB-GENE-200316-1" "UniProtKB:Q8IUD6" "UniProtKB:Q8IUD6" ...
##  $ taxon_id   : chr  "NCBITaxon:10090" "NCBITaxon:7955" "NCBITaxon:9606" "NCBITaxon:9606" ...
##  $ taxon_label: chr  "Mus musculus" "Danio rerio" "Homo sapiens" "Homo sapiens" ...
```

Web API Example 1: Gene Ontology (cont. 4)

The structure of the result will depend on the API. In this case, the output was a JSON file, so the content function returns a list in R. In other scenarios it could return an XML object (we will see more in the lab)

```
knitr::kable(head(dat),
  caption = "Genes experimentally annotated with the function\
  **antiviral innate immune response** (GO:0140374)"
  )
```

Table: Genes experimentally annotated with the function **antiviral innate immune response** (GO:0140374)

Gene	taxon_id	taxon_label
MGI:1918046	NCBITaxon:10090	Mus musculus
ZFIN:ZDB-GENE-200316-1	NCBITaxon:7955	Danio rerio
UniProtKB:Q8IUD6	NCBITaxon:9606	Homo sapiens
UniProtKB:Q8IUD6	NCBITaxon:9606	Homo sapiens
UniProtKB:Q92667	NCBITaxon:9606	Homo sapiens
UniProtKB:P09914	NCBITaxon:9606	Homo sapiens

Web API Example 2: Using Tokens

- Sometimes, APIs are not completely open, you need to register.
- The API may require to login (user+password), or pass a token.
- In this example, I'm using a token which I obtained here
- You can find information about the National Centers for Environmental Information API here

Web API Example 2: Using Tokens (cont. 1)

- The way to pass the token will depend on the API service.
- Some require authentication, others need you to pass it as an argument of the query, i.e., directly in the URL.
- In this case, we pass it on the header.

```
stations_api <- GET(
  url    = "https://www.ncdc.noaa.gov",
  path   = "cdo-web/api/v2/stations",
  config = add_headers(
    token = "[YOUR TOKEN HERE]"
    ),
  query  = list(limit = 1000)
)
```

This is equivalent to using the following query

```
curl --header "token: [YOUR TOKEN HERE]" \
  https://www.ncdc.noaa.gov/cdo-web/api/v2/stations?limit=1000
```

Note: This won't run, you need to get your own token

Web API Example 2: Using Tokens (cont. 2)

Again, we can recover the data using the `content()` function:

```
ans <- content(stations_api)
ans$results[[1]]
## $elevation
## [1] 139
##
## $mindate
## [1] "1948-01-01"
##
## $maxdate
## [1] "2014-01-01"
##
## $latitude
## [1] 31.5702
##
## $name
## [1] "ABBEVILLE, AL US"
##
## $datacoverage
## [1] 0.8813
##
## $id
## [1] "COOP:010008"
```

Web API Example 3: HHS health recommendation

Here is a last example. We will use the Department of Health and Human Services API for "[...] demographic-specific health recommendations" (details at health.gov)

```
health_advises <- GET(
  url  = "https://health.gov/",
  path = "myhealthfinder/api/v3/myhealthfinder.json",
  query = list(
    lang = "en",
    age  = "32",
    sex  = "male",
    tobaccoUse = 0
  ),
  config = c(
    add_headers(accept = "application/json"),
    config(connecttimeout = 60)
  )
)
```

Web API Example 3: HHS health recommendation (cont. 1)

Let's see the response

```
health_advises
```

```
## Response [https://health.gov/myhealthfinder/api/v3/myhealthfinder.json?lang=en&age=32&sex=male&tobacco
##   Date: 2022-02-11 19:50
##   Status: 200
##   Content-Type: application/json
##   Size: 332 kB
## {
##     "Result": {
##         "Error": "False",
##         "Total": 17,
##         "Query": {
##             "ApiVersion": "3",
##             "ApiType": "myhealthfinder",
##             "TopicId": null,
##             "ToolId": null,
##             "CategoryId": null,
## ...
```

Web API Example 3: HHS health recommendation (cont. 2)

```
# Extracting the content
health_advises_ans <- content(health_advises)

# Getting the titles
txt <- with(health_advises_ans$Result$Resources, c(
  sapply(all$Resource, "[[", "Title"),
  sapply(some$Resource, "[[", "Title"),
  sapply(`You may also be interested in these health topics:`$Resource, "[[", "Title")
))
cat(txt, sep = "; ")
```

Protect Yourself from Seasonal Flu; Hepatitis C Screening: Questions for the doctor; Talk with Your Doctor About Depression; Get Your Blood Pressure Checked; Drink Alcohol Only in Moderation; Get Shots to Protect Your Health (Adults Ages 19 to 49); Get Tested for HIV; Quit Smoking; Watch Your Weight; Testing for Syphilis: Questions for the Doctor; Eat Healthy; Protect Yourself from Hepatitis B; Testing for Latent Tuberculosis: Questions for the Doctor; Manage Stress; Alcohol Use: Conversation Starters; Get Active; Quitting Smoking: Conversation Starters

Summary

- We learned about regular expressions with the package **stringr** (a wrapper of **stringi**)
- We can use regular expressions to detect (`str_detect()`), replace (`str_replace()`), and extract (`str_extract()`) expressions.
- We looked at web scraping using the **rvest** package (a wrapper of **xml2**).
- We extracted elements from the HTML/XML using `xml_find_all()` with XPath expressions.
- We also used the `html_table()` function from rvest to extract tables from HTML documents.
- We took a quick review on Web APIs and the Hyper-text-transfer-protocol (HTTP).
- We used the **httr** R package (wrapper of **curl**) to make `GET` requests to various APIs
- We even showed an example using a token passed via the `header`.
- Once we got the responses, we used the `content()` function to extract the message of the response.

Detour on CURL options

Sometimes you will need to change the default set of options in CURL. You can checkout the list of options in `curl::curl_options()`. A common hack is to extend the time-limit before dropping the conection, e.g.:

Using the **Health IT** API from the US government, we can obtain the **Electronic Prescribing Adoption and Use by County** (see docs here)

The problem is that it usually takes longer to get the data, so we pass the config option `connecttimeout` (which corresponds to the flag `--connect-timeout`) in the curl call (see next slide)

Detour on CURL options (cont.)

```
ans <- httr::GET(
  url    = "https://dashboard.healthit.gov/api/open-api.php",
  query  = list(
    source = "AHA_2008-2015.csv",
    region = "California",
    period = 2015
    ),
  config = config(
    connecttimeout = 60
    )
)
```

```
> ans$request
# <request>
# GET https://dashboard.healthit.gov/api/open-api.php?source=AHA_2008-2015.csv&region=California
# Output: write_memory
# Options:
# * useragent: libcurl/7.58.0 r-curl/4.3 httr/1.4.1
# * connecttimeout: 60
# * httpget: TRUE
# Headers:
# * Accept: application/json, text/xml, application/xml, */*
```

Regular Expressions: Email validation

This is the official regex for email validation implemented by RCF 5322

```
(?:[a-z0-9!#$%&'*+/=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?:[\x01-\x08
\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])*")@(?:(?
:[a-z0-9](?:[a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?:[a-z0-9-]*[a-z0-9])?|\[(?:(?:(2(5[
0-5]|[0-4][0-9])|1[0-9][0-9]|[1-9]?[0-9]))\.){3}(?:(2(5[0-5]|[0-4][0-9])|1[0-9][0
-9]|[1-9]?[0-9])|[a-z0-9-]*[a-z0-9]:(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-\
x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])
```

See the corresponding post in StackOverflow