Web Data Mining

Lecture 3: Data Access and Acquisition Methods 2

Jaroslav Kuchař & Milan Dojčinovski
jaroslav.kuchar@fit.cvut.cz, milan.dojchinovski@fit.cvut.cz

Czech Technical University in Prague - Faculty of Information Technologies - Software and Web Engineering

OPP A PRAHA PRAGUE PRAGA PRAG
EVROPSKÁ UNIE

Summer semester 2019/2020
Humla v0.3

Overview

- Parsing and Extracting
- Web Scraping
- Accessing Specific Content

Data Types

- Unstructured
  - *Lacks any structure*
  - *e.g. free texts, documents, multimedia, …*
- Semistructured
  - *Inconsistent structure*
  - *Often self-describing (e.g. key-value pairs), flexibility*
  - *e.g. CSV, XML, …*
- Structured
  - *Conform to the predefined model*
  - *Predefined schema*
  - *Highly structured*
  - *e.g. numbers, dates, structured entities, …*

Recall: Parsing

- Parsing the content of the HTTP payload
  - *extracting content for indexing*
  - *extracting links to be added to the frontier*
  - *extracting additional crawling and indexing directives*
  - *headers* Cache-Control, Content-Type, X-Robots-Tag, …
- HTML code very often contains invalid markup
  - *unclosed elements, unencoded special characters, missing required attributes, improperly nested tags, missing quotes, …*
- Bad HTML markup should be fixed
  - *a preprocessing step is required co clean up the HTML*
  - *many tools available,* tidy *- a tool provided by W3C*

HTML markup fix

- Tidy:

```python
from tidylib import tidy_document
document, errors = tidy_document('''<p>f&otilde;o <img src="bar.jpg">''',
    options={'numeric-entities':1})
print(document)
print(errors)
```

- Output:

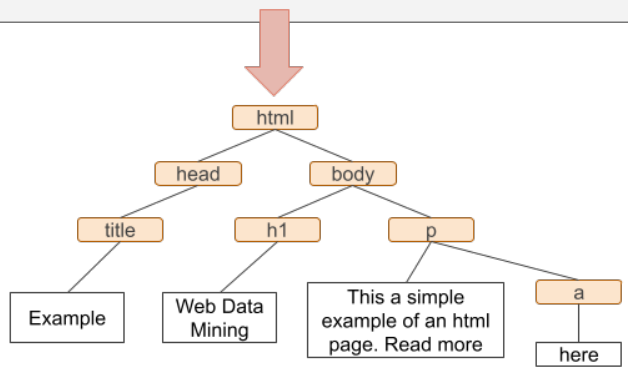
```html
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01//EN">
<html>
  <head>
    <title></title>
  </head>
  <body>
    <p>
      fõo <img src="bar.jpg" alt="">
    </p>
  </body>
</html>
```

```
line 1 column 1 - Warning: missing <!DOCTYPE> declaration
line 1 column 1 - Warning: inserting missing 'title' element
line 1 column 18 - Warning: <img> lacks "alt" attribute
```


Illustration of the DOM tree model

- Parsing of a simple HTML page

```html
<html>
   <head>
      <title>Example</title>
   </head>
      <body>
         <h1>Web Data Mining course</h1>
         <p>This a simple example of an html page. Read more
            <a
href="http://www.w3schools.com/html/html_examples.asp">here</a>
         </p>
   </body>
</html>
```



Extraction of Relevant Information

- Web Information Extraction
  - *Natural language text processing*
  - *Extracting structured data from Web pages*
- Basic approaches
  - *Text fields identification*
    - → *Title, meta information, text blocks*
    - → *Each can have different importance (e.g. HTML title, h1, h3, …)*
  - *Anchors' texts extraction*
    - → *They are acting as a short description of the target.*
    - → *Important for search engines.*
- Detection of relevant structures/information
  - *Recognizers*
  - *Wrappers*
  - *Automatic approaches*
- Other approaches
  - *Machine readable annotations, JS variables, API calls, …*

Recognizers

- Follow a procedure to find a piece of information based on its appearance
  - *e.g. e-mail addresses, phone numbers or street addresses*
- Most of nowadays crawlers and search engines are able to automatically collect such information.
- Can be executed as a set of regular expression patterns:

```python
import re
phonePattern = re.compile(r'^(\d{3})-(\d{3})-(\d{4})$')
phonePattern.search('800-555-1212').groups()
# ('800', '555', '1212')
```

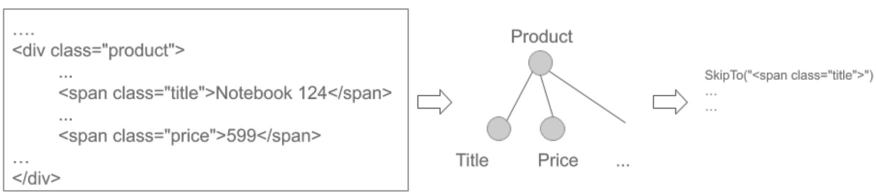
```python
import re
emailPattern = re.compile(r'([a-zA-Z0-9._%+-]+)@([a-zA-Z0-9.-]+\.[a-zA-Z]{2,
emailPattern.search('user@sub.example.com').groups()
# ('user', 'sub.example.com')
```

Wrappers

- Allows to a semi-structured web data source be consulted as if it was a common database.
- Approaches
  - *Manual*
    - → *Observing the source code, finding patterns and writing the crawler*
    - → *Not scalable*
  - *Induction*
    - → *Supervised learning approach, semiautomatic*
    - → *Set of manually labeled pages*
  - *Automatic*
    - → *Unsupervised approach*
    - → *Automatically finds patterns (e.g. repetitive structures, visual aspects, ...)*
    - → *Can scale*

Wrapper Induction Approaches

- Learning from a set of labeled instances
  - *Manual annotations of informations that are required for extraction.*
  - *e.g. product name, price and associated attributes*
- Approach
  - *Tree based representations of annotated information*
    - → *Possibilities for generalizations/pruning*
  - *Learning of extraction rules:*
    - → *e.g.* SkipTo("<span class="title">")



- Limitations
  - *Manual annotation is not possible in large scale.*
  - *Web is dynamic. Cost of maintenance.*

Automatic Extraction

- Based on mining repeated patterns in multiple structures.
- Uses matching approaches
  - *String matching*
    - → *e.g. edit distances - minimum number of changes to change one string to another one ("Page 1" vs "Page 2", "Notebook" vs "CPU")*
  - *Tree matching*
    - → *minimum set of operation to transform trees*
- Building and processing trees
  - *Pure DOM based tree and tree pruning*
  - *Visual aspects can influence the importance of parts*
    - → *e.g. size, colors, visibility, ...*
- Patterns
  - *Repetitive tree structures with the same content can represent headers, footers, menus or ads*
  - *Varying subtrees across multiple pages identify main content*
    - → *High density of textual nodes can represent the main textual content*
  - *Structures close to each other but different colors represents different things*
  - *...*
- Tools
  - *boilerpipe, readability,*
  - *diffbot - analyze the visual layout, dragnet - rule-based models, octoparse*
  - *mercury, fathom*

Overview

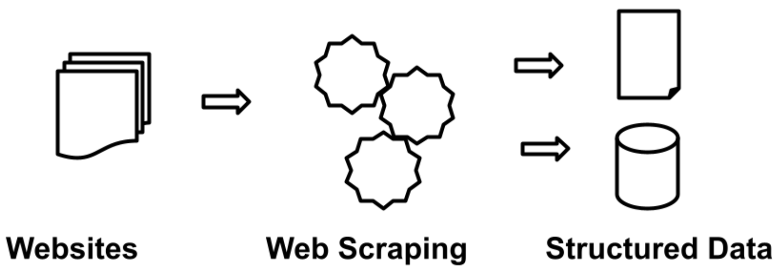
- Parsing and Extracting
- Web Scraping
- Accessing Specific Content

Web Scraping

- Web Scraping vs Crawling
  - *Web Crawling*
    - → *usually provides data for indexing, search engines etc.*
    - → *following links, finding new pages and collecting content*
  - *Web Scraping*
    - → *for pages without API*
    - → *extracting structured information from a web page*
    - → *usually specific for the target website and structured data*
    - → *more focused process*
  - *Many overlapping actions*
    - → *execute JavaScript, emulate human user behavior, submit forms, log in to a website etc.*

Web Scraping (cont.)

- Main process
  1. *Pick a URL*
     - *e.g. one from the predefined or collected list of URLs*
  2. *Collect the URL content*
     - *HTTP request/response*
  3. *Parse and extract desired information from the structure*
     - *using predefined rules or automatic detection mechanisms*



Web Scraping Example - Fetching Data

- Sending HTTP requests

```
import requests

url="https://en.wikipedia.org/wiki/Web_scraping"

# Make a GET request to fetch the raw HTML content
html_content = requests.get(url).text

print(html_content)
```

- Output:

```
<!DOCTYPE html>
<html class="client-nojs" lang="en" dir="ltr">
<head>
<meta charset="UTF-8"/>
<title>Web scraping - Wikipedia</title>
...
```


Parsing Data - Beautifulsoup

- Beautifulsoup
  - *Python library for pulling data out of HTML and XML files*
- Parsers
  - lxml
    - → *fast*
    - → *parses broken HTML*
    - → *external dependency*
  - html.parser
    - → *internal dependency in Python*
  - html5lib
    - → *parses broken HTML*
    - → *external dependency*
- Parsing
  - *navigating HTML tree*
    - → children, next_siblings, previous_sibling, parent
  - *selecting elements*
    - → find, find_all
  - *using CSS selectors, ...*

Web Scraping Example - Parsing Data

```
# Parse the html content
soup = BeautifulSoup(html_content, "lxml")
# soup = BeautifulSoup(html_content, "html.parser")
print(soup.prettify()) # print the parsed data of html

# accessing DOM tree elements
print(soup.title)
print(soup.title.text)

# finding elements
print(soup.find_all('section', class_='categories'))
print(soup.find(id='product-price').children)

# using CSS selectors
print([e.text for e in soup.select("h2:has(> span#See_also) + div > ul > li
```

```
....

['Archive.is', 'Comparison of feed aggregators', 'Data scraping',
'Data wrangling', 'Importer', 'Job wrapping', 'Knowledge extraction',
'OpenSocial', 'Scraper site', 'Fake news website', 'Blog scraping',
'Spamdexing', 'Domain name drop list', 'Text corpus',
'Web archiving', 'Blog network', 'Search Engine Scraping',
'Web crawlers']
```


Web Scraping Example - Parsing Data 2

```
import re
# Parse the html content
soup = BeautifulSoup(html_content, "lxml")
# soup = BeautifulSoup(html_content, "html.parser")

soup.find_all('a', {'href':re.compile('*.html') }).attrs['href']

soup.find_all(lambda t: len(t.attrs)==3)
```

```
....

[ ... ]
```

Overview

- Parsing and Extracting
- Web Scraping
- Accessing Specific Content

Crawling Deep Web

- Deep Web
  - *The content hidden for crawling engines - e.g. behind HTML forms.*
  - *Usually user databases, registration-required web forums, web mail pages, online banking, pages behind paywalls (video on demand), …*

https://www.cisoplatform.com/profiles/blogs/surface-web-deep-web-and-dark-web-are-they-different

Crawling Deep Web (cont.)

- Categories
  - *Contextual web - content dependent on the context (e.g. location)*
  - *Dynamic pages - response to a query (e.g. search)*
  - *Restricted access - technical restrictions (e.g. robots.txt, CAPTCHA)*
  - *Non-HTML content - multimedia or other file formats*
  - *Private Web - login required pages*
  - *Scripted content - AJAX, Flash, ...*
  - *Unlinked - no backlinks*
  - *Hidden - using only specific software, protocols (e.g. Tor)*

Crawling Dynamic pages

- Detection of forms
  - *e.g. search form*

```
<input type="search">
```

- Query generation
  - *pre-computing submissions for each HTML form*
    → *Using known entities from a knowledge base or search engine query logs*
- URL generation
  - *replacing parameters in URLs*
    → *e.g. https://www.google.com/search?q=deep+web*
- Empty page filtering
  - *for detection of irrelevant generations*
- Deduplications of results

Selenium

- Selenium
  - *primarily for automating web applications for testing purposes*
  - *can be used for scraping/crawling as well*

```python
# Example for forms
from selenium.webdriver import Firefox
from selenium.webdriver.firefox.options import Options
opts = Options()
opts.set_headless()
browser = Firefox(options=opts)
browser.get('https://duckduckgo.com')
search_form = browser.find_element_by_id('search_form_input_homepage')
search_form.send_keys('python')
search_form.submit()
results = browser.find_elements_by_class_name('result__a')
print(results[0].text)
browser.close()
```


Crawling Restricted/Private pages

- Limited possibility to access the data
  - *Login forms with email confirmations*
  - *CAPTCHA, reCAPTCHA, ...*
    - → *Completely Automated Public Turing test to tell Computers and Humans Apart (CAPTCHA)*
    - → *Visuals, sounds, ...*
    - → *Secondary goals - image annotations, digitizing texts, ...*
- Approaches
  - *Special agents for session/cookie based approaches*
    - → *Automatic or manual registrations/logins*
  - *CAPTCHA solvers, ...*
  - *Simulations of real human behavior, ...*

Web Scraping Example - "Real Browser"

- Sending HTTP requests

```python
import requests

headers = {'user-agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_1
url="https://en.wikipedia.org/wiki/Web_scraping"

# Make a GET request to fetch the raw HTML content
html_content = requests.get(url, headers=headers).text

print(html_content)
```

- Output:

```html
<!DOCTYPE html>
<html class="client-nojs" lang="en" dir="ltr">
<head>
<meta charset="UTF-8"/>
<title>Web scraping - Wikipedia</title>
...
```


Web Scraping Example - Session

- Sending HTTP requests

```python
import requests

# requests.get(url, cookies={...}})

session = requests.Session()
login_data = {'user': 'user', 'password': 'password'}
session.post('https://example.com/login', login_data)

result = s.get('https://example.com/data')

print(result.text)
```

- Output:

```html
<!DOCTYPE html>
<html class="client-nojs" lang="en" dir="ltr">
<head>
<meta charset="UTF-8"/>
<title>Web scraping - Wikipedia</title>
...
```

Crawling AJAX applications

- Modern applications produce content dynamically, faster and richer

```html
<div id="main">
</div>
<script>
for(var i=0; i<=5; i++){
var data = "<a href=\"#\">Link"+i+"</a><br />";
    document.getElementById('main').innerHTML += data;
}
</script>
```

Link0
Link1
Link2
Link3
Link4
Link5

Crawling AJAX applications

- AJAX application's content is produced dynamically by the browser
  - *dynamically fetch data from the server using JavaScript and display to the user*
  - *e.g., by clicking on button or link*
  - *this dynamically created content is not visible to crawlers*
- Typical scenario
  - *user opens a web page, e.g., http://example.com/*
  - *user clicks on link products*
  - *a JavaScript code is executed, which dynamically changes the URL to a pretty URL http://example.com/#products and content of the current page*
- Problem: a browser can execute JavaScript and produce content on-the-fly, the crawler generally can have limitations
- Headless browsers solutions
  - *No GUI, emulated DOM, …*
  - *can be expensive, resource-wise, …*

Crawling AJAX applications

- Another problem: only the URL part before the hashtag (#) is processed from the server
  - *hash fragments are not part of the HTTP requests*
  - *hash fragments are not sent to the server*
- HTTP request containing hash fragment

```
curl -v https://mail.google.com/mail/u/0/#inbox

> GET /mail/u/0/ HTTP/1.1
> Host: mail.google.com
> User-Agent: curl/7.30.0
> Accept: */*

< ...HTTP response...
```

- Note the path /mail/u/0/ in the HTTP request
  - *it does not contain the hash fragment*

Making AJAX Applications Crawlable

- Solution:
  - *The crawler detects a pretty URL*
    - → *a URL containing hash fragment beginning with !*
    - → *e.g., https://mail.google.com/mail/u/0/#!inbox*
  - *The crawler transforms the pretty URL to a ugly URL*
    - → *the #! is replaced with _escaped_fragment_*
    - → *the _escaped_fragment_ becomes part of the query parameters*
    - → *e.g., https://mail.google.com/mail/u/0/#!inbox becomes https://mail.google.com/mail/u/0/?_escaped_fragment_=inbox*
  - *The crawler requests the ugly URL*
  - *The server interprets the ugly URL and serves the content*
  - *The crawler processes the content*
- Reference
  - *see Google AJAX Crawling for more details (officially deprecated as of October 2015)*

Spider Traps

- Spider Trap - a set of web pages that create an infinite number of URLs for crawling
  - *finding new previously unvisited URLs for infinite amount of time*
- Harmful for crawlers
  - *waste of bandwidth and storage to download and store duplicate and useless data*
  - *waste of server's bandwidth and space with fake information*
- Creation of spider's traps for good causes
  - *to catch spambots, which send emails on behalf of others*
  - *to catch crawlers, which waste a website's bandwidth*
- Example - Calendar with "next day/month/week/year" link
  - *creation of dynamic pages with links, which point to next year or month*
  - *http://cal.org/01/2014/ has "next" link pointing to http://cal.org/01/2015/*
  - *again http://cal.org/01/2015/ has "next" link pointing to http://cal.org/01/2016/*

Honeypot Traps

- Honeypot Traps
  - *In the form of links which are not visible to the typical user on the browser*
    - → *e.g. setting the CSS as* display: none
  - *hidden form inputs*
    - → *e.g. predefined random/empty value expected (e.g. GitHub login page)*
    - → *specific name, that the crawler fill in e.g.* username, password
- Crawler
  - *can try to scrape the information from the link*
  - *can be detected and owner blocks the source IP address*
- Detection is not easy

Web Scraping Example - Proxies

- Sending HTTP requests
  - *https://free-proxy-list.net/*

```python
import requests
url = 'https://httpbin.org/ip'
proxies = {
    "http": 'http://62.244.49.202:52323',
    "https": 'http://62.244.49.202:52323'
}
response = requests.get(url,proxies=proxies)
print(response.json())
```

- Output:

```
{'origin': '62.244.49.202'}
```


Other Tools, Demos, Examples

- Headless browsers tools
  - *https://github.com/GoogleChrome/puppeteer*
  - *https://github.com/emadehsan/thal*
  - *https://nickjs.org/*

```javascript
const nick = new Nick()
;(async () => {
    const tab = await nick.newTab()
    await tab.open("news.ycombinator.com")
    await tab.untilVisible("#hnmain")
    await tab.inject("../injectables/jquery-3.0.0.min.js")
    const hackerNewsLinks = await tab.evaluate((arg, callback) => {
        const data = []
        $(".athing").each((index, element) => {
            data.push({
                title: $(element).find(".storylink").text(),
                url: $(element).find(".storylink").attr("href")
            })
        })
        callback(null, data)
    })
    console.log(JSON.stringify(hackerNewsLinks, null, 2))
})()
.then(() => {
    console.log("Job done!")
    nick.exit()
})
.catch((err) => {
    console.log(`Something went wrong: ${err}`)
    nick.exit(1)
})
```

Other Tools, Demos, Examples

- https://blog.phantombuster.com/web-scraping-in-2017-headless-chrome-tips-tricks-4d6521d695e8

```
if (await tab.isVisible(".captchaImage")) {
  // Get the URL of the generated CAPTCHA image
  // Note that we could also get its base64-encoded value and solve it too
  const captchaImageLink = await tab.evaluate((arg, callback) => {
    callback(null, $(".captchaImage").attr("src"))
  })

  // Make a call to a CAPTCHA solving service
  const captchaAnswer = await buster.solveCaptchaImage(captchaImageLink)

  // Fill the form with our solution
  await tab.fill(".captchaForm",
    { "captcha-answer": captchaAnswer },
    { submit: true })
}
```


Machine Readable Annotations

- Significantly reduce the effort to properly extract several entities
  – *Standard based*
  – *Limited amount of entities*
  – *Limited amount of relevant information*
- Content annotated by providers in any existing format
  – *Content-level*
    → *Microformats*
    → *Microdata*
    → *RDFa*
  – *Page-level*
    → *OpenGraph*
    → *JSON-LD*

Content-Level Annotations

- Microformats

```
<p class="h-card">
  <img class="u-photo" src="http://example.org/photo.png" alt="" />
  <a class="p-name u-url" href="http://example.org">Joe Bloggs</a>
  <a class="u-email" href="mailto:joebloggs@example.com">joebloggs@example.c
  <span class="p-street-address">17 Austerstræti</span>
  <span class="p-locality">Reykjavík</span>
  <span class="p-country-name">Iceland</span>
</p>
```

- Microdata

```
<p itemscope itemprop="Person" itemtype="http://schema.org/Person">
  <span itemprop="name">Christopher Froome</span> was sponsored by
  <span itemprop="sponsor" itemtype="http://schema.org/Organization">
    <a itemprop="url" href="http://www.skysports.com/">Sky</a></span> in the
</p>
```

- RDFa

```
<p vocab="http://schema.org/" typeof="Person">
  <span property="name">Christopher Froome</span> was sponsored by
  <span property="sponsor" typeof="http://schema.org/Organization">
    <a property="url" href="http://www.skysports.com/">Sky</a></span> in the
</p>
```


Page-Level Annotations

- OpenGraph

```
<head prefix="og: http://ogp.me/ns# fb: http://ogp.me/ns/fb# ">
   <meta property="fb:app_id"      content="45646845135" />
   <meta property="og:url"         content="http://example.com/profile/tom_h
   <meta property="og:type"        content="profile" />
   <meta property="og:title"       content="Tom Hanks" />
   <meta property="og:description" content="Tom Hanks profile" />
   <meta property="og:image"       content="http://example.com/profile/hanks
```

- JSON-LD

```
<script type="application/ld+json">
{
  "@context": "http://schema.org/",
  "@type": "Person",
  "name": "Christopher Froome",
  "sponsor":
  {
    "@type": "Organization",
    "name": "Sky",
    "url": "http://www.skysports.com/"
  }
}
</script>
```

Machine Readable Annotations - Example

- BeautifulSoup

```
url="https://en.wikipedia.org/wiki/Web_scraping"
html_content = requests.get(url).text
soup = BeautifulSoup(html_content, "html.parser")
p = soup.find('script', {'type':'application/ld+json'})
print(p.contents)
```


```
['{"@context":"https:\\/\\/schema.org","@type":"Article","name":"Web scrapin
```


- extruct

```
import extruct
import requests
import pprint
pp = pprint.PrettyPrinter(indent=2)
data = extruct.extract(html_content)
pp.pprint(data)
```


```
{ 'json-ld': [ { '@context': 'https://schema.org',
                 '@type': 'Article',
                 'author': { '@type': 'Organization',
                             'name': 'Contributors to Wikimedia projects'},
                 'dateModified': '2019-12-16T19:58:14Z',
                 'datePublished': '2005-09-17T18:57:30Z',
                 'headline': 'data scraping used for extracting data from '
                             'websites',
```


Other "Sources" for Scraping

- JavaScript variables
  - <script> *tag on the page*

```
from selenium.webdriver import Firefox
from selenium.webdriver.firefox.options import Options
opts = Options()
browser = Firefox(options=opts)
browser.get('https://www.bbc.com/news/world-51235555')

browser.execute_script('return config;')

browser.close()
```


```
{
  'headline': 'Space cookies: First food baked in space by astronauts',
  'iStats_counter_name': 'news.world.story.51235555.page',
  'language': 'en-gb',
  'last_updated': {'date': '2020-01-24 12:26:02',
   'timezone': 'Europe/London',
   'timezone_type': 3},
  'length': 2994,
  'mediaType': 'video',
  ...
  'section': {'id': '99115',
   'name': 'World',
   'uri': '/news/world',
   'urlIdentifier': '/news/world'},
```

Other "Sources" for Scraping (cont.)

- API calls, XHRs
  - *website loads data dynamically*
  - *it typically uses XMLHttpRequests (XHRs)*

```
url = 'https://aukro.cz/backend/api/offers/search?page=0&size=60&sort='
data = {'text': 'notebook', "splitGroupKey": "listing", "splitGroupValue": "
headers = { 'Content-Type': 'application/json'}
requests.post(url, json = data, headers = headers).json()
```

```
...
'content': [{'itemId': 121212121212,
   'itemName': 'Notebook',
   'catalog': {'id': 1, 'name': 'Elektronika'},
...
```


Scraping Strategy

- Scrape Websites Without Being Blocked
  - *Slow down the scraping*
    - *→ too many request*
    - *→ too fast form submit etc.*
  - *Use proxy servers*
  - *Apply different scraping patterns*
  - *Look like a human*
    - *→ Use (switch) correct user-agents*
    - *→ Handle cookies*
  - *Be careful of honeypot traps*
  - *Use headless or real browser*