COMP20008
Elements of data processing

Semester 1 2020

Lecture 5: Web crawling and text search

Web – the repository of data

A large amount of text data is on the Web.

Web crawling is a method to get the data from the web.

We will cover:

- Web crawling and scraping; getting text data from the web

Then, we will cover common techniques for text data processing

- Search and matching in text
- Text preprocessing (next lecture)

Web crawling

- The data to be searched needs to be gathered from the Web
- Web crawlers are also known as spiders, robots, and bots.
- Crawlers attempt to visit every page of interest and retrieve them for processing and indexing
- Basic challenge: there is no central index of URLs of interest.
- Secondary challenges:
  - same content as a new URL
  - never return status ‘done’ on access
  - websites that are not intended to be crawled
  - content generated on-the-fly from databases → costly for the content provider → excessive visits unwelcome
  - Some content has a short lifespan

Crawling

The web is a highly linked graph

If a web page is of interest, there will be a link to it from another page.

In principle:

- Create a prioritised list L of URLs to visit (seed URLs)
- Create a list V of URLs that have been visited and when.

Repeat forever:

1. Choose a URL u from L and fetch the page $p(u)$ at location u.
2. Parse and index $p(u)$
   Extract URLs $\{u'\}$ from $p(u)$.
3. Add u to V and remove it from L

   Add $\{u'\} - V$ to L.
4. Process V to move expired or 'old' URLs to L.

Crawling

- Every page is visited eventually.
- Synonym URLs are disregarded.
- Significant or dynamic pages are visited more frequently.
- The crawler musn't cycling indefinitely in a single web site.

Crawling vs Scraping

- Crawling starts with seed URLs; scraping specifies a fixed list of URLs
- Crawling visits all of a linked graph; scraping extracts data from the fixed list

Crawling – challenges

- **Crawler traps** are surprisingly common. For example, a ‘next month’ link on a calendar can potentially be followed until the end of time.
- The **Robots Exclusion Standard:** protocol that all crawlers are supposed to observe. It allows website managers to restrict access to crawlers while allowing web browsing.

Simple crawlers are available, for example python `scrapy`

Parsing

Once a document has been fetched, it must be **parsed**.

- Web documents can usually be segmented into discrete zones such as **title, anchor text, headings**, and so on.
- Data can be extracted from specific zones.
- Information such as **links and anchors** can be analysed, **formats** such as PDF or Postscript or Word can be translated, and so on.

- Python `BeautifulSoup`

Parsing – cont.

- Preprocessing for text – more details on this in the next lecture
- The result is data in a more structured format
  - xml
  - Json
  - csv

Practice in workshop

basic crawling and scraping in the workshop `BeautifulSoup`

Text (string) search

Exact string search

- Given a string, is some substring contained within it?
- Given a string, find all occurrences of some substring.

For example, find `Exxon` in:

```
In exes for foxes rex dux mixes a pox of waxed luxes. An
axe, and an axon, to exo Exxon max oxen. Grexit or
Brexit as quixotic haxxers with buxom rex taxation.
```

Exact string search

- Given a string, is some substring contained within it?
- Given a string, find all occurrences of some substring.

For example, find `Exxon` in:

```
In exes for foxes rex dux mixes a pox of waxed luxes. An
axe, and an axon, to exo Exxon max oxen. Grexit or
Brexit as quixotic haxxers with buxom rex taxation.
```

Approximate string search

Find exon in:

```
In exes for foxes rex dux mixes a pox of waxed luxes. An
axe, and an axon, to exo Exxon max oxen. Grexit or
Brexit as quixotic haxxers with buxom rex taxation.
```

Approximate string search

Find `exon` in:

```
In exes for foxes rex dux mixes a pox of waxed luxes. An
axe, and an axon, to exo Exxon max oxen. Grexit or
Brexit as quixotic haxxers with buxom rex taxation.
```

Not present!

...But what is the “closest” or “best” match?

Application – spelling correction

Need the notion of a **dictionary**:

- Here, a list of words (entries) that are "correct" with respect to our (expectations of our) language
- We can break our input into words (substrings) that we wish to match, and compare each of them against the entries in the dictionary
- A word (item) in the input that *doesn't* appear in the dictionary is *misspelled*
- A word (item) in the input that *does* appear in the dictionary might be correctly spelled *or might be* misspelled (beyond the scope of this subject)

Application – spelling correction

Therefore, the problem here:

Given some item of interest — which does not appear in our dictionary — which entry from the dictionary was truly intended?

Depends on the person who wrote the original string!

Application – detecting novel words

Word Blending:

- forming novel words by “blending” two other words
- not simple concatenation (e.g., football is not a blend word)

```
breakfast + lunch → brunch
fork + spoon → spork
Britain + exit → Brexit
```

- Language changes continuously
- New terms are often coined in colloquial language (e.g., Twitter)
- Social media is fertile grounds for linguistic innovation.

Other applications

Name matching

street and place name conventions

```
boulevard|blvd|bd|bde|blv|bl|blvde|blvrd|boulavard|boul|bvd
apartment|apt|ap|aprt|aptmnt
village|vil|vge|vill|villag|villg|vlg|vlge|vllg
```

- Data cleaning (e.g., deduplication)
- Query repair

Approximate string search/matching

What's a "best" match?

Find approximate match(es) for exon in:

```
In exes for foxes rex dux mixes a pox of waxed luxes.
An axe, and an axon, to exo Exxon max oxen.
Grexit or Brexit as quixotic haxxers with buxom rex taxation.
```

Approximate string search/matching

Find approximate match(es) for exon in:

```
In exes for foxes rex dux mixes a pox of waxed luxes.
An axe, and an axon, to exo Exxon max oxen.
Grexit or Brexit as quixotic haxxers with buxom rex taxation.
```

exon → Exxon **Insert** x
exon → exo **Delete** n
exon → axon **Replace** e with a
exon → oxen **Transpose** e and o (not covered)

Method – neighbourhood search

For a given string *w* of interest:

- Generate all variants of *w* that utilise at most *k* changes (Insertions/Deletions/Replacements) — **neighbours**
- Check whether generated variants exist in dictionary
- **All** results found in dictionary are returned

For example:

`... proceed if you can see no` **`ther`** `option ...`

`the their there tier` **`other`** `mther tnher thpr ...`

Unix command-line utility agrep is an efficient tool for finding these.

Method – edit distance

Levenshtein distance (a type of edit distance)

Scan through each dictionary entry looking for the "best" match

Intuition:

- Transform the string of interest into each dictionary entry
- **Operations**: Insert, Delete, Replace, and Match
- Each Insert, Delete, Replace operation incurs a score;
- Best match is the dictionary entry with best (lowest) aggregate mismatch **score;**

The details of the algorithm are not covered in the subject

Edit-distance to similarity

The distance is divided by the length of the longer string to get a **similarity**

$$sim_{edit}(s_1, s_2) = 1.0 - \frac{d(s_1,s_2)}{\max(|s_1|,|s_2|)}$$

- cart –> arts?

c	a	r	t	
	a	r	t	s

$$sim_{edit}(cart, arts) = 1.0 - \frac{2}{4} = 0.5$$

Jaro-Winkler similarity

- Based on edit-distance (also character-based similarity)
- Give more weight to strings with matching prefixes
- Useful for matching (approximate) prefixes / suffixes.
- The details of the algorithm are not covered in the subject

Method – N-gram distance

Another method for finding best approximate string match

What is an (character) n-gram? A (character) substring of length *n*

- 2-grams of crat: cr, ra, at; **or**
- 2-grams of crat: #c, cr, ra, at, t# (sometimes)
- 3-grams of crat: ##c, #cr, cra, rat, at#, t##

A sequence of characters is converted into a set of n-grams; it becomes a vector in a vector space.

N-gram distance – example

- 2-grams of crat: $G_2(\texttt{crat}) = \texttt{\#c, cr, ra, at, t\#}$
- 2-grams of cart: $G_2(\texttt{cart}) = \texttt{\#c, ca, ar, rt, t\#}$
- 2-grams of arts: $G_2(\texttt{arts}) = \texttt{\#a, ar, rt, ts, s\#}$

N-gram distance between $G_n(x)$ and $G_n(y)$:

$$|G_n(x)| + |G_n(y)| - 2 \times |G_n(x) \cap G_n(y)|$$

crat and cart:

$$|G_2(\text{crat})| + |G_2(\text{cart})| - 2 \times |G_2(\text{crat}) \cap G_2(\text{cart})| = 5+5-2\times2= 6$$

crat and arts:

$$|G_2(\text{crat})| + |G_2(\text{art})| - 2 \times |G_2(\text{crat}) \cap G_2(\text{art})| = 5+5-2\times0= 10$$

N-gram distance

- More sensitive to long substring matches, less sensitive to relative ordering of strings (matches can be anywhere!)
- Despite its simplicity, takes roughly the same time to compare entire dictionary
- Quite useless for very long strings and/or very small alphabets (Why?)
- **Potentially useful for (approximate) prefixes / suffixes**, e.g., Street → St; or smog → smoke

N-gram distance to similarity

- Cosine

$$cos(S_1, S_2) = \frac{S_1 \cdot S_2}{|S_1| \times |S_2|}$$

- Jaccard similarity

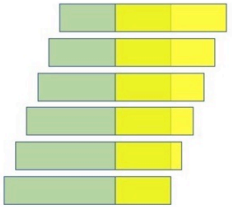
overlap coefficient

$$sim_{jacc}(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$$

- Sørensen-Dice similarity

$$sim_{dice}(S_1, S_2) = \frac{2 \times |S_1 \cap S_2|}{|S_1| + |S_2|}$$

In my opinion, the Jaccard Similarity is a very powerful analysis technique, but it has a major drawback when the two sets being compared have different sizes. Consider two sets, A and B, where both sets contain 100 elements. Now assume that 50 of those elements are common across the two sets. The Jaccard Similarity is js(A, B) = 50 / (100 + 100 - 50) = 0.33. Now if we increase set A by 10 elements and decrease set B by the same amount, all while maintaining 50 elements in common, the Jaccard Similarity remains the same. And there is where I think Jaccard fails: it has no sensitivity to the sizes of the sets. The following figure highlights how the Jaccard and Overlap Coefficient change as the set sizes are change but the intersection size remains that same.



A	B	Jaccard	Overlap
100	100	0.333	0.500
110	90	0.333	0.556
120	80	0.333	0.625
130	70	0.333	0.714
140	60	0.333	0.833
150	50	0.333	1.000

Figure 4: Similarity Scores as Set Sizes Change

N-gram distance to similarity

S1: G_2(crat) = {#c, cr, ra, at, t#}

S2: G_2(cart) = {#c, ca, ar, rt, t#}

{#c, cr, ra, at, t#, ca, ar, rt}

Crat: $[1, 1, 1, 1, 1, 0, 0, 0]$

Cart: $[1, 0, 0, 0, 1, 1, 1, 1]$

- $|S_1| = 5$
- $|S_2| = 5$
- $|S_1 \cup S_2| = 8$
- $|S_1 \cap S_2| = 2$
- $S_1 \cdot S_2 = 1 + 0 + 0 + 0 + 1 + 0 + 0 + 0 = 2$

Acknowledgement

Some contents of the slides are based on materials created by Lea Frermann and Justin Zobel and Karin Verspoor, CIS