Web Crawlers

Lecture 5

Topics

- Deciding What to Search
- Web Crawler
  - Retrieving Web Pages
  - Crawling Steps
  - Focused Crawling
  - Sitemaps
  - Distributed Crawling
  - Storing the Documents
  - Removing Noise
- Building Crawlers with `urllib`
  - Importing Libraries
  - Fetching the resources
  - Handling exceptions
  - Additional libraries

Deciding What to Search

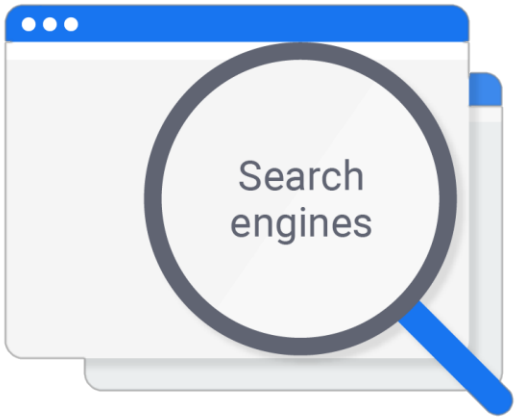
- Although we focus heavily on the technology that makes search engines work, it is the information in the document collection that makes search engines useful.
- Every time a search engine adds another document, the number of questions it can answer increases.

Deciding What to Search

- On the other hand, adding many poor-quality documents increases the burden on the ranking process to find only relevant information for the user.

Deciding What to Search

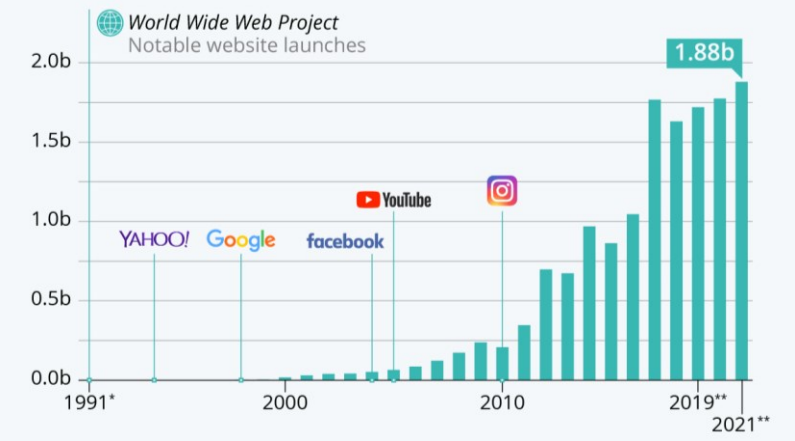
- Web search engines, however, show how successful search engines can be, even when they contain billions of low-quality documents with little useful content.



Web Crawler

- Finding and downloading web pages automatically is called *crawling*, and a program that downloads pages is called a *web crawler*
  - provides the collection for searching
- Challenges:
  - Web is huge (~5 billion pages) and constantly growing.

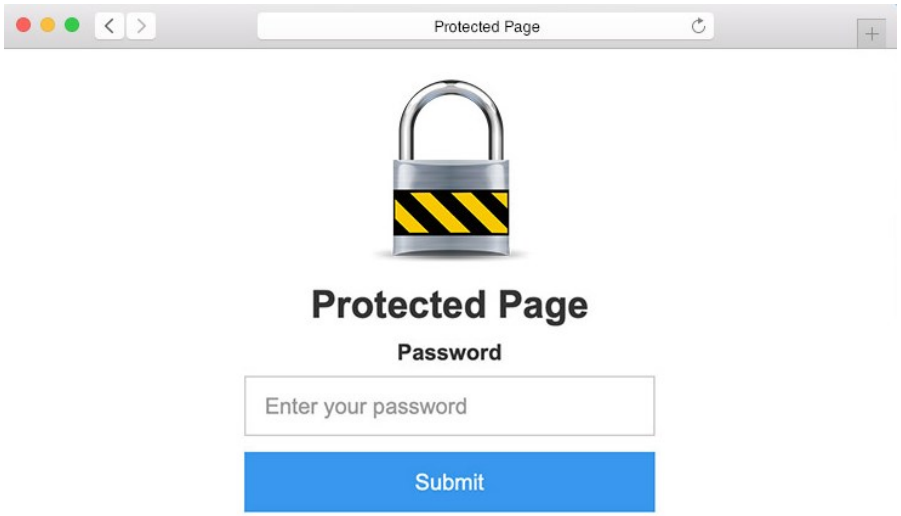
Most organizations do not have enough storage space to store even a large fraction of the Web.



CalPolyPomona

Web Crawler

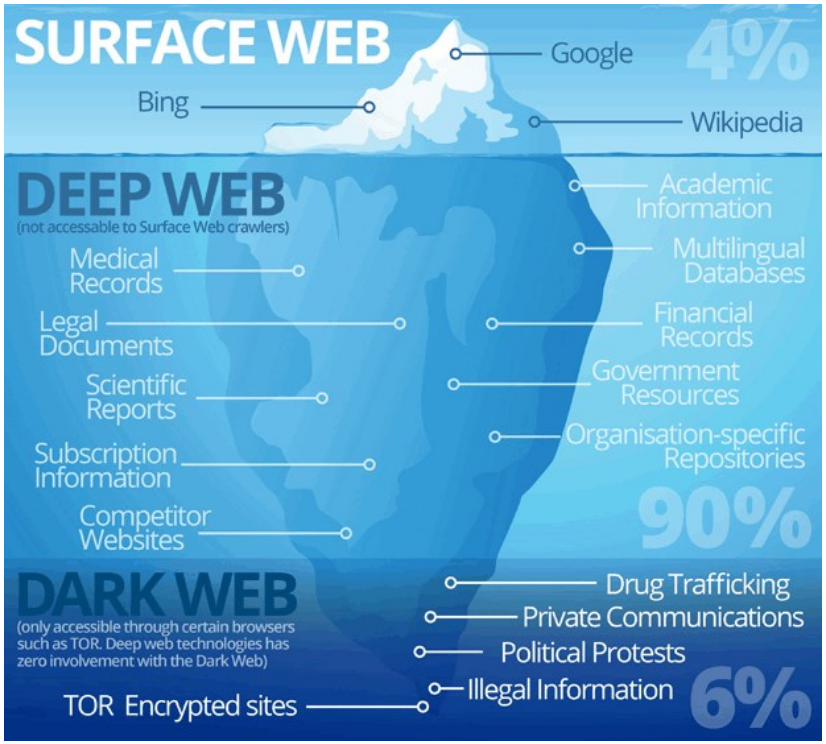
- Challenges:
  - Web is not under the control of search engine providers – data might be available only by typing a request into a form (deep web pages) and there is no easy way to find out how many pages there are on the site.
  - Crawlers used for other types of data



CalPolyPomona

Web Crawler

- The multiple Webs:



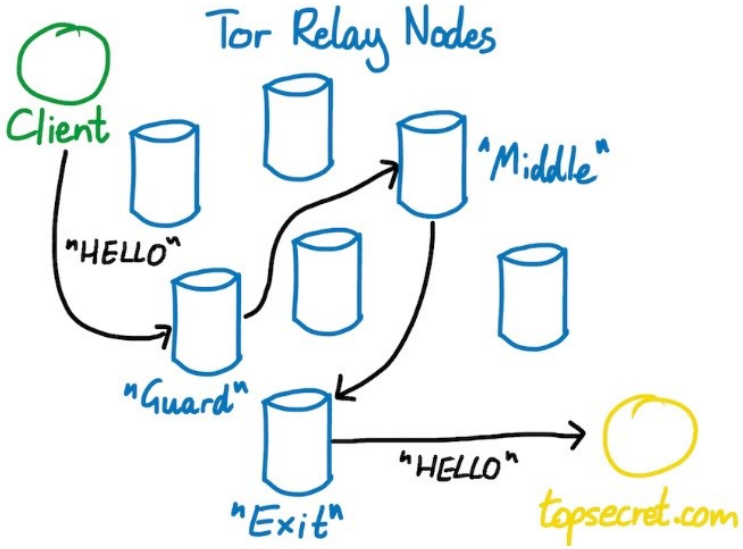
Web Crawler

- The Deep Web:
  - The deep web is any part of the web that is not part of the surface web – the Web part that is indexed by standard search engines.
  - Why cannot Google index those pages?
    - Google cannot submit forms
    - Google cannot find pages that have not been linked to by a top-level domain
    - Google cannot investigate sites where robots.txt - a file that tells search engine crawlers which URLs the crawler can access - prohibits
  - Estimates vary widely, but the deep web almost certainly makes up about 90% of the internet.

Web Crawler

- The Dark Web:
  - The dark web, also known as the darknet, uses addition protocols such as Tor - The Onion Router that runs on top of HTTP, providing an anonymous channel to exchange information.
  - It relies on specific browsers such as the Tor browser.

TorProject.org

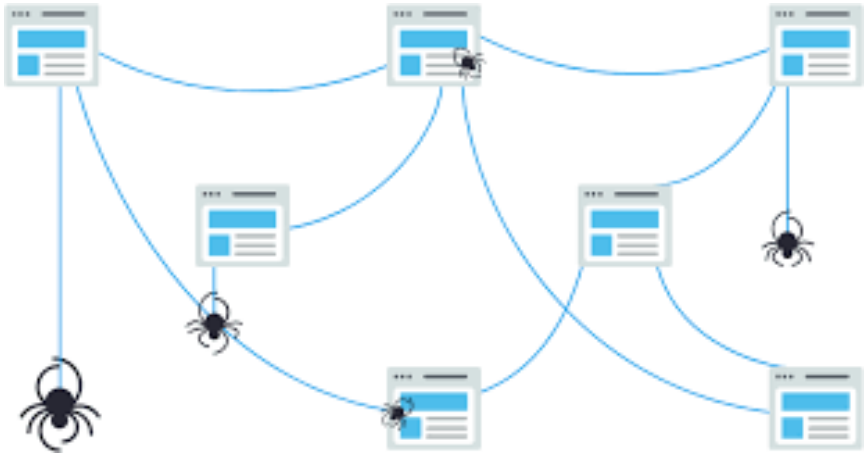


Web Crawler

Point to Ponder #1

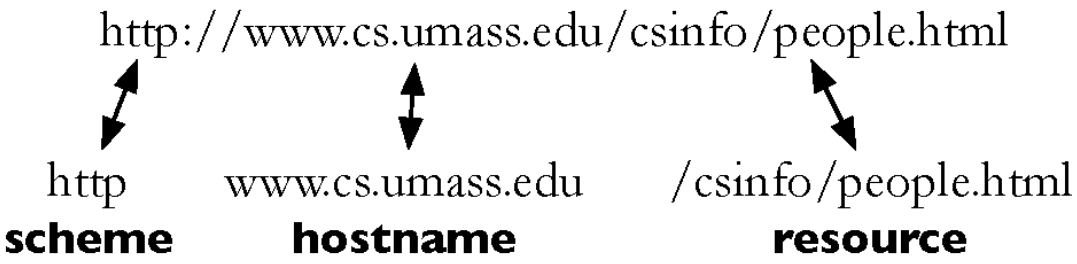
Do you know the difference between a Web Crawler and a Web Spider?

No difference. Crawling is also occasionally referred to as spidering, and a crawler is sometimes called a spider or a spiderbot.



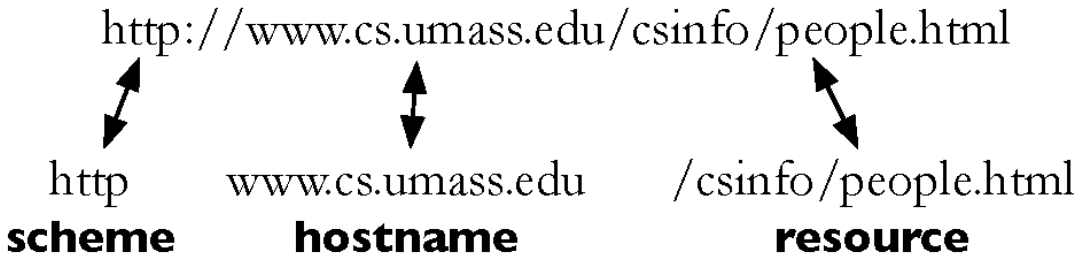
Retrieving Web Pages

- Every web page has a unique *uniform resource locator* (URL) to describe it with three parts.
  - The scheme indicating how the resource can be retrieved. Most URLs used on the Web start with the scheme HTTP (Hypertext Transfer Protocol) or HTTPS (Hypertext Transfer Protocol Secure) to exchange information with client software.



Retrieving Web Pages

- The *hostname* follows, which is the name of the computer that is running the Web server that holds this Web page, e.g., *www.cs.umass.edu*.
- This URL refers to a page on that computer called /csinfo/people.html.



Retrieving Web Pages

- Web browsers and web crawlers are two different kinds of Web clients, but both fetch Web pages in the same way.
  1. First, the client program connects to a domain name system (DNS) server.
  2. The DNS server translates the hostname into a numeric internet protocol (IP) address.
  3. The program then attempts to connect to a server computer with that IP address.
  4. Since that server might have many different programs running on it, with each one listening to the network for new connections, each program listens on a different port. A port is a number that identifies a particular service. By convention, requests for web pages are sent to port 80 unless specified otherwise in the URL.

Retrieving Web Pages

- Web browsers and web crawlers are two different kinds of web clients, but both fetch web pages in the same way.
  5. Once the connection is established, the client program sends an HTTP request to the web server to request a page. The most common HTTP request type is a GET request, for example:

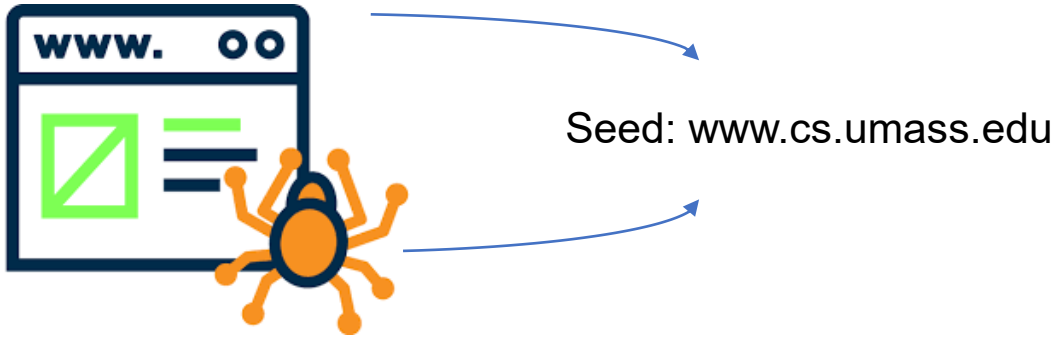
     ```
     GET /csinfo/people.html HTTP/1.0
     ```

  6. This simple request asks the server to send the page called /csinfo/people.html back to the client, using version 1.0 of the HTTP protocol specification.
  7. After sending a short header, the server sends the contents of that file back to the client. If the client wants more pages, it can send additional requests; otherwise, the client closes the connection.

CalPolyPomona

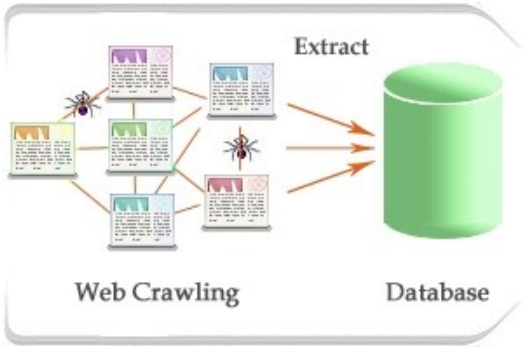
Retrieving Web Pages

- Although Web browsers are handy for executing JavaScript, displaying images, and arranging objects in a more human-readable format (among other things), Web crawlers are excellent at gathering and processing large amounts of data quickly.
- Rather than viewing one page at a time through the narrow window of a monitor, you can view databases spanning thousands or even millions of pages at once.

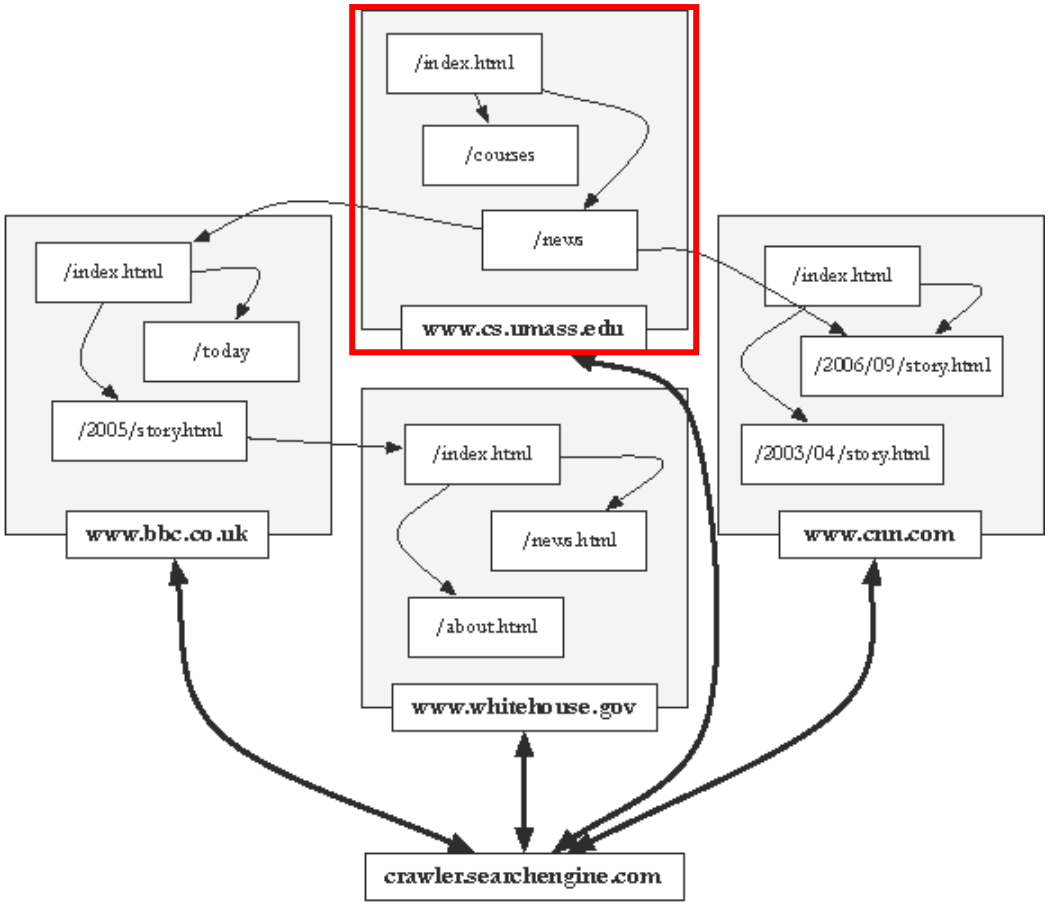


Retrieving Web Pages

- Even when an API does exist, the request volume and rate limits, the types of data, or the format of data that it provides might be insufficient for your purposes.
- This is where crawling steps in. With few exceptions, if you can view data in your browser, you can access it via a Python script. If you can access it in a script, you can store it in a database. And if you can store it in a database, you can do virtually anything with that data.



Crawling Steps



Seed: www.cs.umass.edu

1. Starts with a set of *seeds*, which are a set of URLs given to it as parameters. Seeds are added to a URL request queue. Ex: www.cs.umass.edu
2. Crawler starts fetching pages from the request queue.
3. Downloaded pages are parsed to find link tags that might contain other useful URLs to fetch.
4. If the crawler finds a new URL that it has not seen before, it is added to the crawler's request queue, or *frontier*.
5. Continue until no more new URLs or disk full.

Crawling Steps

- Web crawlers spend a lot of time waiting for responses to requests.
- To reduce this inefficiency, web crawlers use threads and fetch hundreds of pages at once.
- Crawlers could potentially flood sites with requests for pages.
- To avoid this problem, web crawlers use *politeness policies*
  - e.g., delay between requests to same web server

Crawling Steps

- Even crawling a site slowly will anger some web server administrators, who object to any copying of their data.
- Robots.txt file can be used to control crawlers

```
User-agent: *
Disallow: /private/
Disallow: /confidential/
Disallow: /other/
Allow: /other/public/

User-agent: FavoredCrawler
Disallow:

Sitemap: http://mysite.com/sitemap.xml.gz
```

Crawling Steps

- Simple Crawler Thread

```
procedure CrawlerThread(frontier)
    while not frontier.done() do
        website ← frontier.nextSite()
        url ← website.nextURL()
        if website.permitsCrawl(url) then
            text ← retrieveURL(url)
            storeDocument(url, text)
            for each url in parse(text) do
                frontier.addURL(url)
            end for
        end if
        frontier.releaseSite(website)
    end while
end procedure
```

Seeds

Most time-consuming part

enforce politeness policy

Focused Crawling

- Attempts to download only those pages that are about a particular topic
  - used by *vertical search* applications
- Rely on the fact that pages about a topic tend to have links to other pages on the same topic
  - popular pages for a topic are typically used as seeds
- Crawler uses *text classifier* to decide whether a page is on topic

Focused Crawling

Instead of crawling a full copy of the pages and then throw out all unrelated ones, a less expensive approach is focused, or topical, crawling.

Sitemaps

- One of the challenges with Web crawling arises because site owners cannot adequately tell crawlers about their sites
- Generated by web server administrators, sitemaps contain lists of URLs and data about them, with useful information for the crawlers such as
  - modification time and modification frequency
  - indication of resources the crawlers might not otherwise find
  - hints about when to check a page for changes

Sitemap Example

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://www.company.com/</loc>
    <lastmod>2008-01-15</lastmod>
    <changefreq>monthly</changefreq>
    <priority>0.7</priority>
  </url>
  <url>
    <loc>http://www.company.com/items?item=truck</loc>
    <changefreq>weekly</changefreq>
  </url>
  <url>
    <loc>http://www.company.com/items?item=bicycle</loc>
    <changefreq>daily</changefreq>
  </url>
</urlset>
```

URL

Frequency of changes

Last time it was changed

Importance of this page. 0.5 is the default value.

Distributed Crawling

- Three reasons to use multiple computers for crawling
  - helps to put the crawler closer to the sites it crawls (increase *throughput* and decrease *latency*)
  - reduces the number of sites the crawler has to remember (no enough RAM)
  - reduces computing resources required
- Distributed crawler uses a hash function to assign URLs to crawling computers
  - usually, all the URLs for a particular host are assigned to a single crawler.

Storing the Documents

- Many reasons to store document text
  - provides efficient access to text for snippet generation, information extraction, etc.
  - it makes sense to keep copies of the documents around instead of trying to fetch them again the next time you want to build an index
  - saves crawling time when page is not updated
- Database systems can provide document storage for some applications
  - Web search engines use customized document storage systems

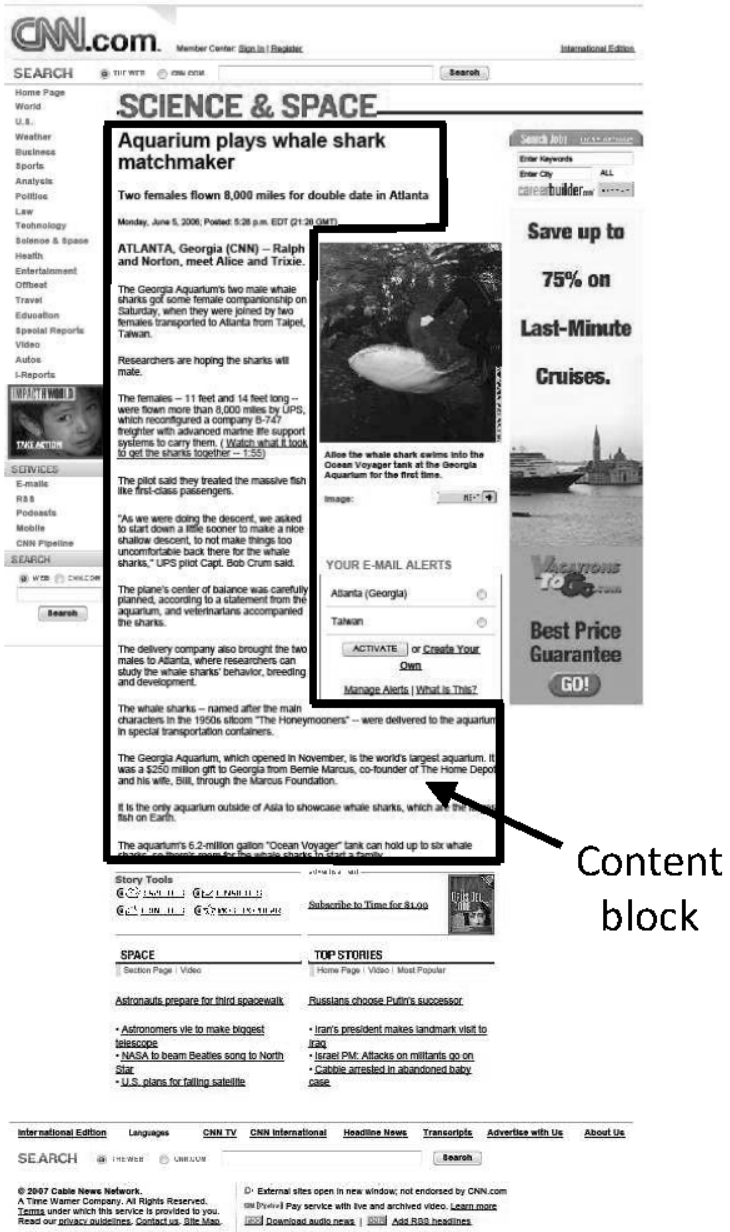
Removing Noise

- Many web pages contain text (e.g., copyright data), links (navigation links), and pictures (e.g., banners, advertisements) that are not directly related to the main content of the page
- This additional material is mostly *noise* that could negatively affect the ranking of the page
- Techniques have been developed to detect the *content blocks* in a web page
  - non-content material is either ignored or reduced in importance in the indexing process

CalPolyPomona

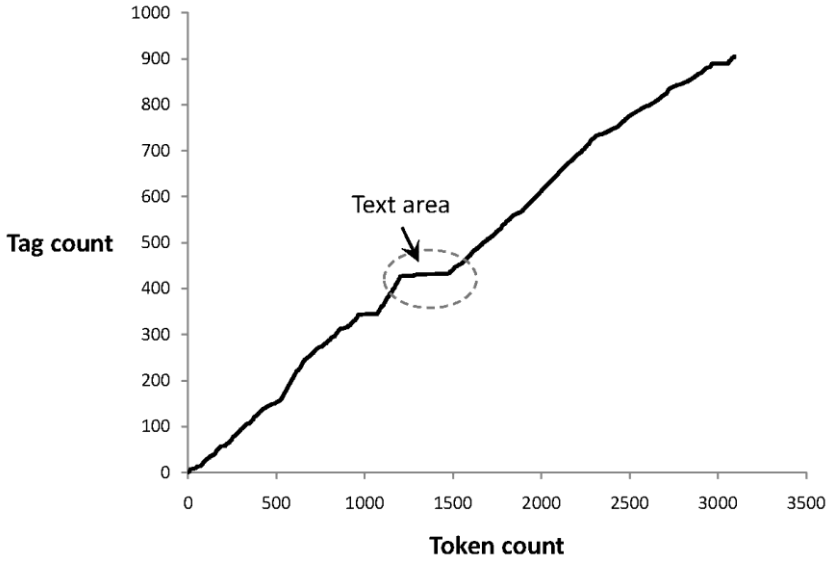
Removing Noise

- Noise Example



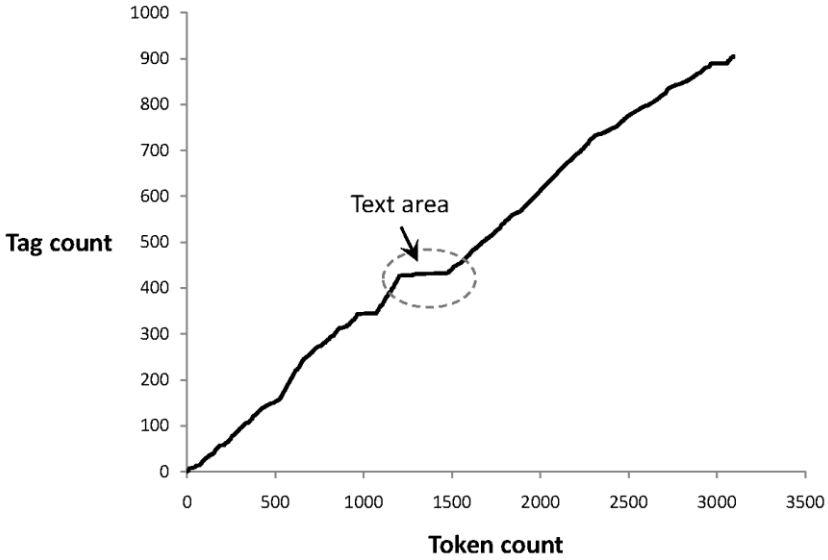
Removing Noise

- Finding Content Blocks
  - Cumulative distribution of tags in the example web page as a function of the total number of tokens (words or other non-tag strings) in the page



Removing Noise

- Finding Content Blocks



Point to Ponder #2

What does the "plateau“ correspond to?

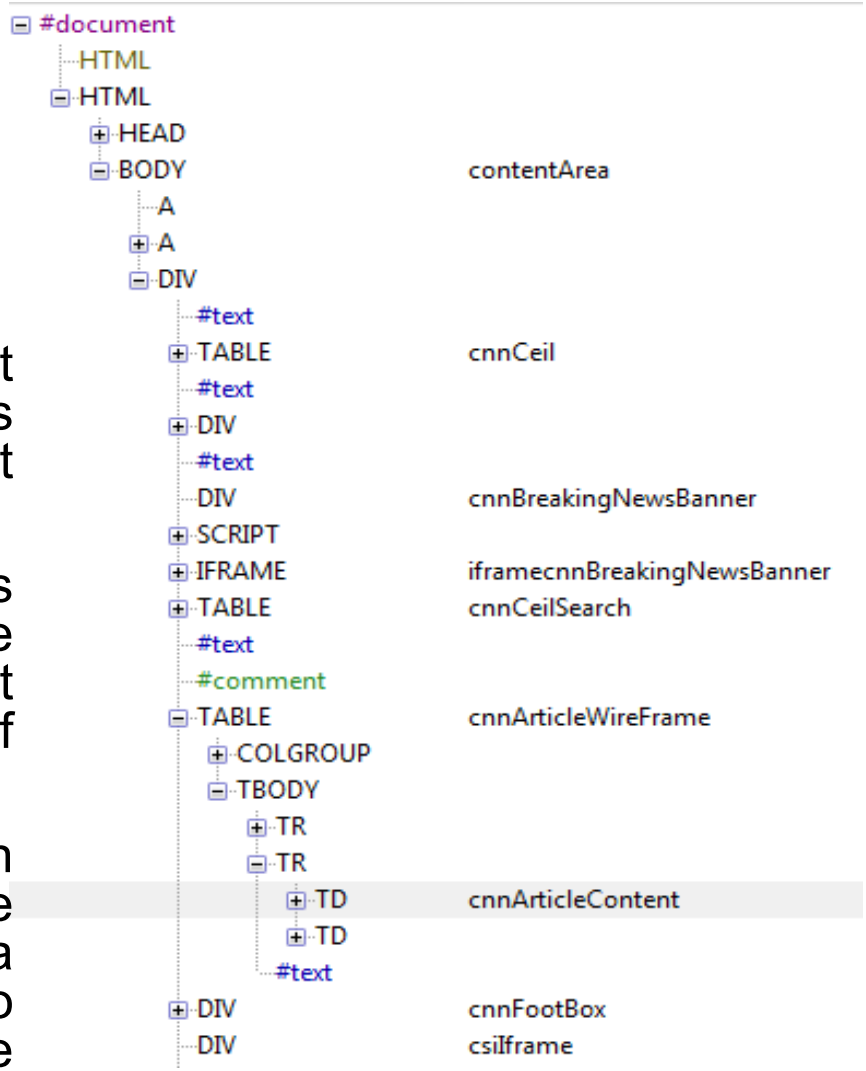
The main text content of the page

Removing Noise

- Finding Content Blocks
  - Other approaches use DOM structure and visual (layout) features.
    - To display a web page using a browser, an HTML parser interprets the structure of the page specified using the tags and creates a Document Object Model (DOM) representation.
    - The tree-like structure represented by the DOM can be used to identify the major components of the web page.

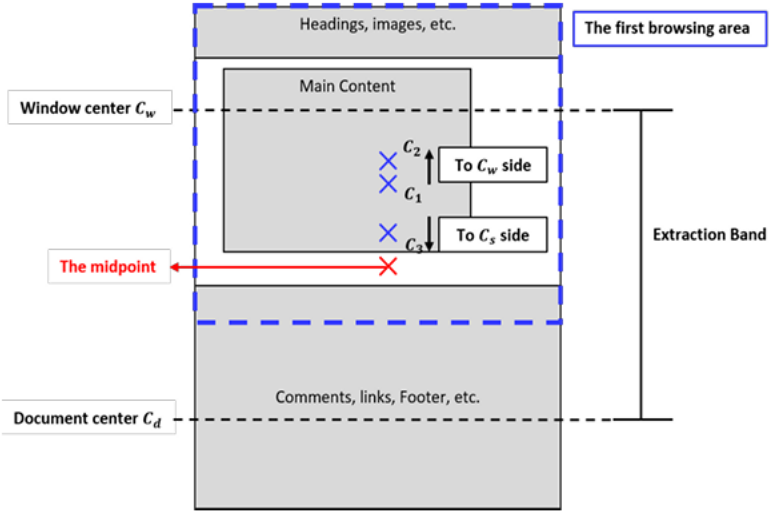
Removing Noise

- Finding Content Blocks
  - The part of the structure that contains the text of the story is indicated by the comment `cnnArticleContent`.
  - The DOM structure provides useful information about the components of a web page, but it is complex and is a mixture of logical and layout components.
  - Gupta et al. (2003) describe an approach that navigates the DOM tree recursively, using a variety of filtering techniques to remove and modify nodes in the tree and leave only content.



Removing Noise

- Finding Content Blocks
  - Another approach to identifying the content blocks in a page focuses on the layout and presentation of the web page.
    - Visual features—such as the position of the block, the size of the font used, the background and font colors, and the presence of separators (such as lines and spaces)—are used to define blocks of information on the page.
    - Yu et al. (2003) describe an algorithm that constructs a hierarchy of visual blocks from the DOM tree and visual features.



Building Crawlers with Python

- Importing Libraries
  - A Web browser is a useful application for creating packets of information, telling your operating system to send them off, and interpreting the data you get back as pretty pictures, sounds, videos, and text.
  - However, you can do the same thing in Python with a few lines of code:

```
from urllib.request import urlopen
```

  - `urllib` is a standard Python library containing functions for requesting data across the Web, handling cookies, and even changing metadata such as headers and your user agent.

CalPolyPomona

Building Crawlers with Python

- Fetching the resources

```
html = urlopen('http://pythonscraping.com/pages/page1.html')
print(html.read())
```

- Output:

```
Python Scraping book:b'<html>\n<head>\n<title>A Useful Page</title>\n</head>\n<body>\n<h1>An Interesting Title</h1>\n<div>\nLorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.\n</div>\n</body>\n</html>\n'
```

Building Crawlers with Python

- Fetching the resources
  - This command outputs the complete HTML code for `page1` located at the URL

    `http://pythonscraping.com/pages/page1.html`

  - More accurately, this outputs the HTML file `page1.html`, found in the directory `<web root>/pages`. Together, they form the resource to be located on the `http://pythonscraping.com` server.
  - `urlopen` is used to open a remote object across a network and read it. It is a generic function able to read HTML files, image files, or any other file stream with ease.

Building Crawlers with Python

- Fetching the resources

Point to ponder #3

Why is it important to start thinking of these addresses as “files” in addition to “pages”?

Most modern web pages have many resource files associated with them. These could be image files, JavaScript files, CSS files, or any other content that the page you are requesting is linked to. When a web browser hits a tag such as <img src="cuteKitten.jpg">, the browser knows that it needs to make another request to the server to get the data at the file cuteKitten.jpg to fully render the page for the user.

Building Crawlers with Python

- Fetching the resources
  - Another example:

```
html = urlopen('https://www.cpp.edu/')
print("CPP: " + str(html.read()))
```

  - Output:

```
CPP: <!DOCTYPE html>\n<html
xmlns="http://www.w3.org/1999/xhtml"
lang="en">\n\t<head>\n\t\t<meta content="IE=edge" http-
equiv="X-UA-Compatible"/>\n\t\t<title>\n\t\t\tCal Poly
Pomona\n\t\t</title>\n\t\t<!-- Required meta tags
...
```

Building Crawlers with Python

- Handling Exceptions

  - The Web is messy. Data is poorly formatted, websites go down, and closing tags go missing.
  - One of the most frustrating experiences in Web crawling is to go to sleep with a crawler running, dreaming of all the data you will have in your database the next day—only to find that the crawler hit an error on some unexpected data format and stopped execution shortly after you stopped looking at the screen.

Building Crawlers with Python

- Handling Exceptions:

```
html = urlopen('https://www.cpp.edu/hi.html')
```

  - Two main things can go wrong in this line:
    - The page is not found on the server or there was an error in retrieving it – “HTTP 404 Page Not Found”.
    - The server is not found - “HTTP 500 Internal Server Error”.
  - In all these cases, the `urlopen` function will throw the generic exception `HTTPError`.

Building Crawlers with Python

- Handling Exceptions:

You can handle this exception in the following way:

```
from urllib.request import urlopen
from urllib.error import HTTPError
try:
   html = urlopen('http://www.pythonscraping.com/pages/page1.html')
except HTTPError as e:
   print(e)
   # return null, break, or do some other "Plan B"
else:
   # program continues.
```

CalPolyPomona

Building Crawlers with Python

- Handling Exceptions:

Catching the `HTTP 404` exception:

Resource does not exist.

```
from urllib.request import urlopen
from urllib.error import HTTPError
try:
   html = urlopen('http://www.pythonscraping.com/pages/page100.html')
except HTTPError as e:
   print(e)
   # return null, break, or do some other "Plan B"
else:
   # program continues.
```

Building Crawlers with Python

- Handling Exceptions:

Catching the `HTTP 500` exception:

User is not authenticated.

```
from urllib.request import urlopen
from urllib.error import HTTPError
try:
   html = urlopen('https://canvas.cpp.edu/courses/74459')
except HTTPError as e:
   print(e)
   # return null, break, or do some other "Plan B"
else:
   # program continues.
```

Building Crawlers with Python

- Handling Exceptions:
  - If the server is not found at all (if, say, `http://www.pythonscraping.com` is down, or the URL is mistyped), `urlopen` will throw an `URLError`.
  - Because the remote server is responsible for returning HTTP status codes, an `HTTPError` cannot be thrown, and the more serious `URLError` must be caught. You can add a check to see whether this is the case:

```
except HTTPError as e:
        print(e)
except URLError as e:
        print('The server could not be found!')
```

See crawler.py

Building Crawlers with Python

- Additional libraries
  - `Scrapy` is a free and open-source web-crawling framework. Originally designed for web scraping, it can also be used to extract data using APIs or as a general-purpose web crawler. `Scrapy` provides a complete package for developers to customize their crawlers/scrapers, making it a good choice for larger and more complex data extraction projects
  - `Selenium` is a powerful web scraping tool developed originally for website testing. These days, it's also used when the accurate portrayal of websites—as they appear in a browser—is required. Selenium works by automating headful or headless browsers to load the website, retrieve the required data, and even take screenshots or assert that certain actions happen on the website.

CalPolyPomona

Text Transformation

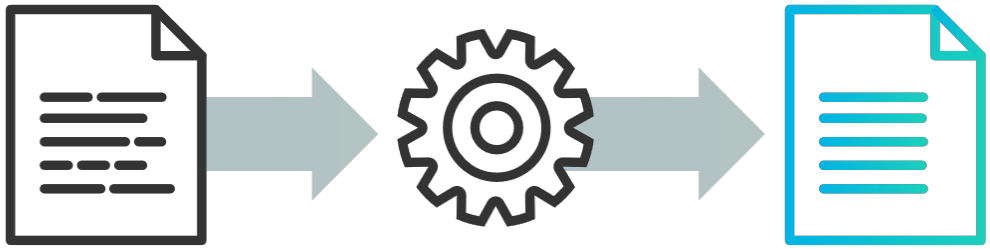
Lecture 6

Topics

- Transforming Text
- Parsing
  - Regular Expressions
    - PostgreSQL and MongoDB
- Tokenizing
- Stopping
- Stemming/Lemmatizing
- Phrases and N-grams
- Document Structure and Markup
- Text Transformation in Python using `BeautifulSoup` and `scikit-learn`

Transforming Text

- After gathering the text we want to search, the next step is to decide whether it should be modified or restructured in some way to simplify searching.
- The types of changes that are made at this stage are called *text transformation* or, more often, *text processing*.
- The goal of text processing in search engines is to convert the many forms in which words can occur into consistent *index terms*.
- Index terms are the representation of the content of a document that are used for searching.



Transforming Text

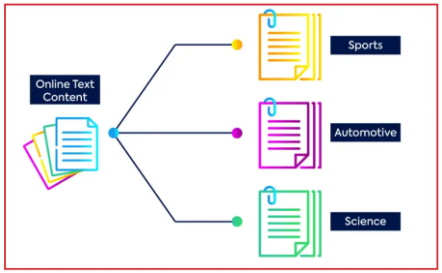
- The simplest decision about text processing would be to NOT DO IT AT ALL. A good example of this is the "find" feature in your favorite word processor that scans the document and tries to find the exact sequence of letters that you just search.
- Problems:
  - Matching the exact string of characters typed by the user is too restrictive
    - i.e., it doesn't work very well in terms of effectiveness
  - Not all words are of equal value in a search
  - Sometimes not clear where words begin and end
  - Not even clear what a word is in some languages
    - e.g., Chinese, Korean

Transforming Text

- Many search engines do not distinguish between uppercase and lowercase letters. However, they go much further by
  - stripping punctuation from words.
  - splitting words apart in a process called tokenization.
  - ignoring entirely some words in a process called stopping.
  - matching similar words to each other (like "run" and "running") in a process called stemming.
  - recognizing document structural elements such as titles, figures, links, and headings.
- All those features contributes to a more effective and efficient search process.

Transforming Text

- In addition, techniques involving more complex text processing are being used to identify additional index terms or features for search.
  - Information extraction techniques for identifying people's names, organization names, addresses, and many other special types of features
  - classification, which can be used to identify semantic categories



Parsing

- Document parsing involves the recognition of the content and structure of text documents to facilitate data comprehension.
- By applying *tokenization* or *lexical analysis,* a parser recognizes each word occurrence in the sequence of characters in a document.
- Apart from these words, there can be many other types of content in a document, such as metadata, images, graphics, code, and tables a parser needs to identify.
  - Metadata content includes document attributes such as date and author, and, most importantly, the tags that are used by markup languages to identify document components.

Parsing

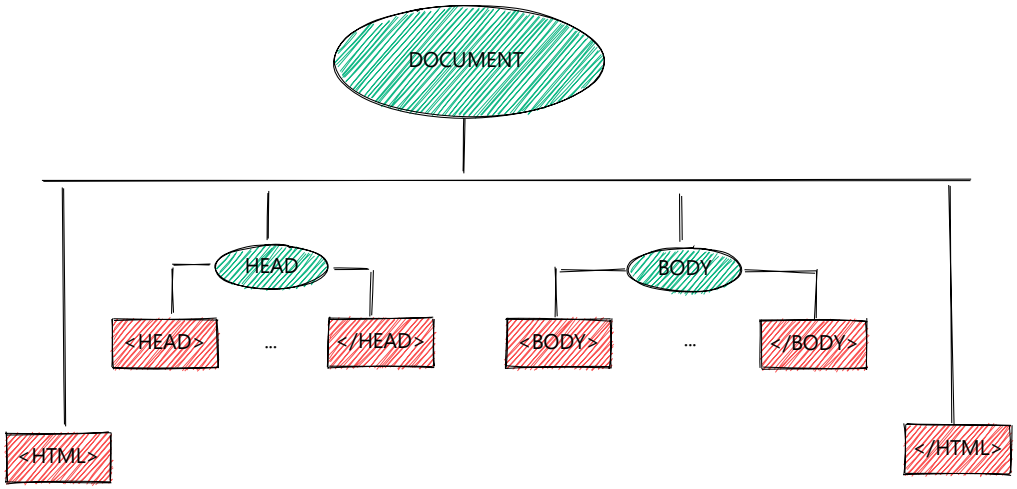
- Point to ponder #1

What are the two most popular markup languages?

HTML (Hypertext Markup Language) and XML (Extensible Markup Language)

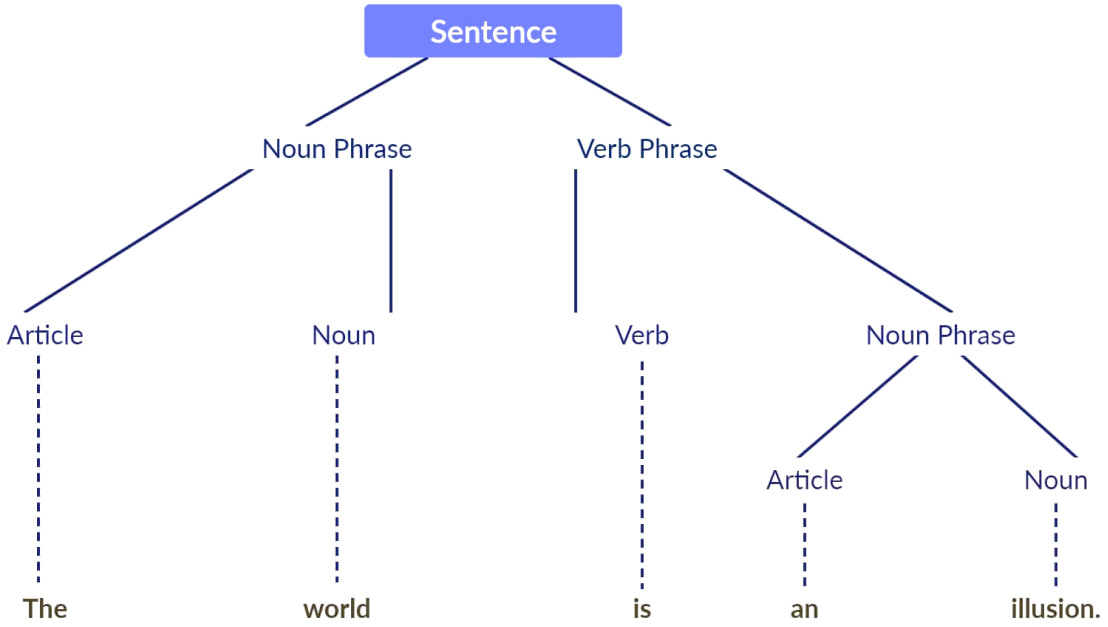
Parsing

- The parser uses the tags and other metadata recognized in the document to interpret the document's structure based on the *syntax* of the markup language (*syntactic analysis*) and to produce a representation of the document that includes both the structure and content. For example, an HTML parser interprets the HTML tags and creates a Document Object Model (DOM) representation of the page that is used by a web browser.



Parsing

- Parsers are also used in other domains such as **natural languages**. The term parsing comes from Latin pars (orationis), meaning part (of speech).



Parsing

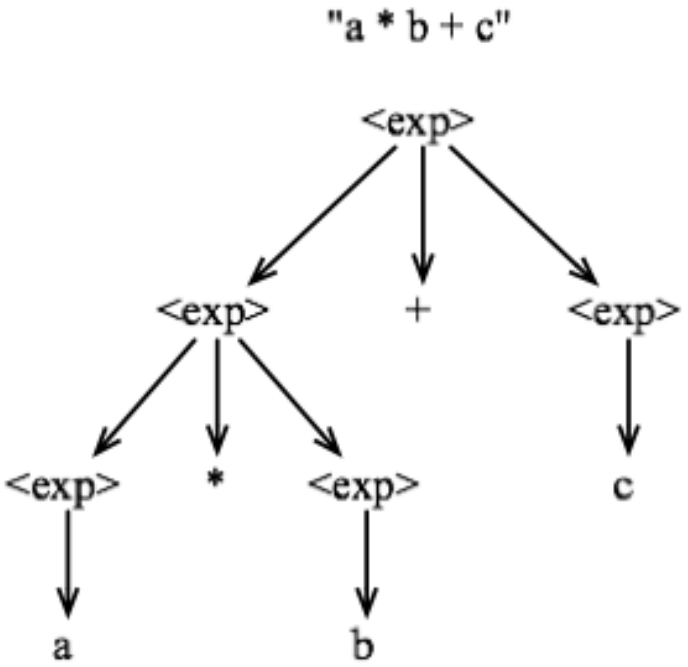
- Point to ponder #2

Give a practical application of parsing in natural language processing?

Predict the most probable word or sequence of words that follows a given input context.

Parsing

- Or **computer languages**.

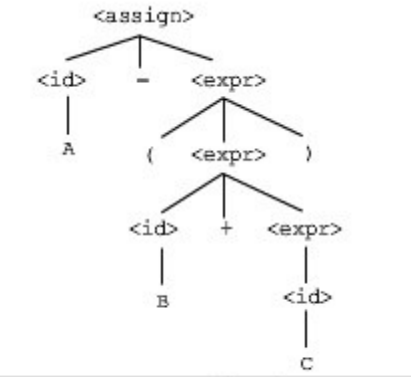


Parsing

- Point to ponder #3

Give a practical application of parsing in computer languages?

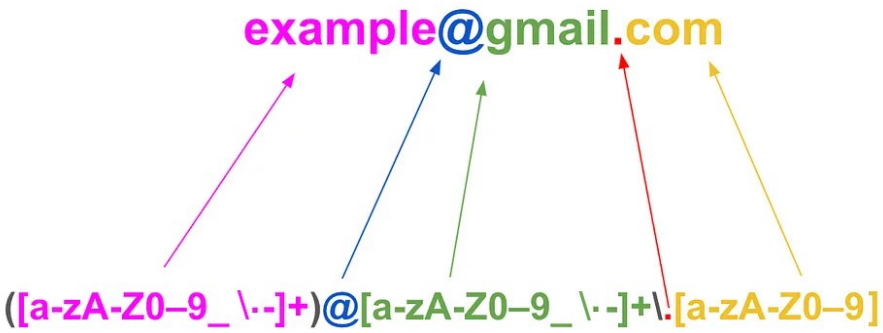
Design compilers and interpreters.



Parse Tree for the Statement A = (B + C)

Regular Expressions

- Regular expressions (often shortened to regex) are so called because they are used to identify regular strings.
- They can definitively say, “Yes, this string you’ve given me follows the rules, and I’ll return it,” or “This string does not follow the rules, and I’ll discard it.” We use regex for describing a search pattern.
- This can be exceptionally handy for quickly scanning large documents to look for strings that look like phone numbers or email addresses.



Regular Expressions

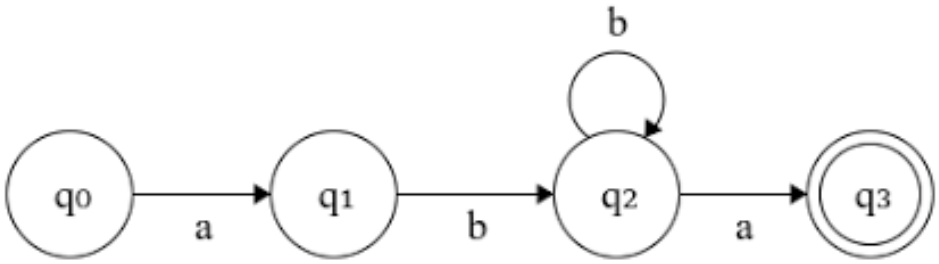
- But what is a regular string? It's any string that can be generated by a series of linear rules, such as these:
  1. Write the letter a at least once.
  2. Append to this the letter b exactly five times.
  3. Append to this the letter c any even number of times.
  4. Write either the letter d or e at the end.
- Strings that follow these rules are aaaabbbbbccccd, aabbbbbcce, and so on (there are an infinite number of variations).
- Regular expressions are merely a shorthand way of expressing these sets of rules. For instance, here's the regular expression for the series of steps just described:

  aa*bbbbb(cc)*(d|e)

Regular Expressions

- Point to ponder #4

What does this process remind you?



- Regular expressions and finite automata have equivalent expressive power: For every regular expression R, there is a corresponding FA that accepts the set of strings generated by R.

Regular Expressions

- The string `aa*bbbbb(cc)*(d|e)` might seem a little daunting at first, but it becomes clearer when you break it into its components:
- aa*
  - The letter a is written, followed by a* (read as a star), which means “any number of as, including 0 of them.” In this way, you can guarantee that the letter a is written at least once.
- bbbbb
  - No special effects here—just five bs in a row.
- (cc)*
  - Any even number of things can be grouped into pairs, so in order to enforce this rule about even things, you can write two cs, surround them in parentheses, and write an asterisk after it, meaning that you can have any number of pairs of cs (note that this can mean 0 pairs, as well).
- (d|e)
  - Adding a bar in the middle of two expressions means that it can be “this thing or that thing.” In this case, you are saying “add a d or an e.” In this way, you can guarantee that there is exactly one of either of these two characters.

Regular Expressions

See regex3.py

Symbol(s)	Meaning	Example	Matches
?	Indicates zero or one occurrence of the character, subexpression, or bracketed character.	a?bb	bb, abb
*	Matches the preceding character, subexpression, or bracketed character, 0 or more times.	a*b*	aaaaaaaa, aaabbbbb, bbbbbb
+	Matches the preceding character, subexpression, or bracketed character, 1 or more times.	a+b+	aaaaaaaab, aaabbbbb, abbbbbb
[]	Matches any character within the brackets (i.e., “Pick any one of these things”).	[A-Z]*	APPLE, CAPITALS, QWERTY
()	A grouped subexpression (these are evaluated first, in the “order of operations” of regular expressions).	(a*b)*	aaabaab, abaaab, ababaaaaab
{m, n}	Matches the preceding character, subexpression, or bracketed character between m and n times (inclusive).	a{2,3}b{2,3}	aabbb, aaabbb, aabb

Regular Expressions

Symbol(s)	Meaning	Example	Matches
[^]	Matches any single character that is not in the brackets.	[^A-Z]*	apple, lowercase, qwerty
|	Matches any character, string of characters, or subexpression, separated by the I (note that this is a vertical bar, or pipe, not a capital i).	b(a|i|e)d	bad, bid, bed
.	Matches any single character (including symbols, numbers, a space, etc.).	b.d	bad, bzd, b$d, b d
\d	Matches any digit. Equivalent to [0-9].	\da or [0-9]a	2a, 12a
\D	Matches any character that is not a digit. Equivalent to [^0-9].	\Da or [^0-9]a	ba, bba
\w	Matches any alphanumeric character from the basic Latin alphabet, including the underscore. Equivalent to [A-Za-z0-9_].	\wp or [A-Za-z0-9_]p	Apple, April, 2p

Regular Expressions

Symbol(s)	Meaning	Example	Matches
\W	Matches any character that is not a word character from the basic Latin alphabet. Equivalent to [^A-Za-z0-9_].	\W or [^A-Za-z0-9_]	#party
\s	Matches a single white space character.	\s[Ww]	Hello World!
\S	Matches a single character other than white space.	\S[Ww]	First-World!
^	Indicates that a character or subexpression occurs at the beginning of a string.	^a	apple, asdf, a
\	An escape character (this allows you to use special characters as their literal meanings).	\. | \\	. | \
$	Often used at the end of a regular expression, it means “match this up to the end of the string.” Without it, every regular expression has a de facto “.*” at the end of it, accepting strings where only the first part of the string matches. This can be thought of as analogous to the ^ symbol.	[A-Z]*[a-z]*$	ABCabc, zzyx, Bob

Regular Expressions

- One classic example of regular expressions can be found in the practice of identifying email addresses. Although the exact rules governing email addresses vary slightly from mail server to mail server, we can create a few general rules. The corresponding regular expression for each of these rules is shown in the second column:

  - Rule 1: The first part of an email address contains at least one of the following: uppercase letters, lowercase letters, the numbers 0–9, periods (.), dashes (-), or underscores (_).
    - [A-Za-z0-9\._-]+
  - Rule 2: After this, the email address contains the @ symbol.
    - @
  - Rule 3: The email address then must contain at least one uppercase or lowercase letter.
    - [A-Za-z]+
  - Rule 4: This is followed by a period (.).
    - \.
  - Rule 5: Finally, the email address ends with com, org, edu, or net (in reality, there are many possible top-level domains, but these four should suffice for the sake of example).
    - (com|org|edu|net)

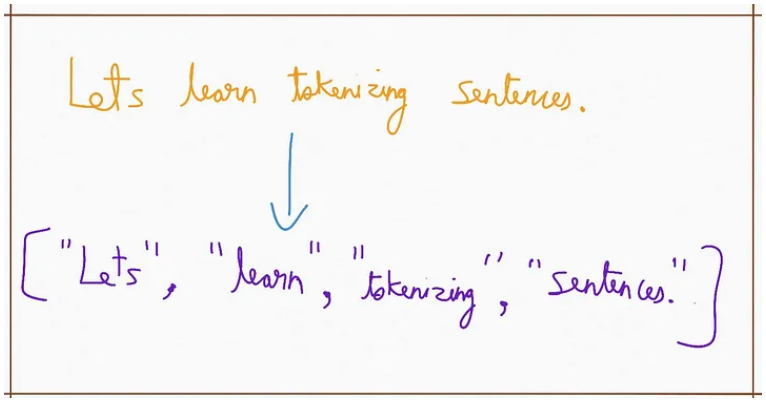
Regular Expressions

- By concatenating all of the rules, you arrive at this regular expression:
  - [A-Za-z0-9\._]+@[A-Za-z]+\.(com|org|edu|net)

- When attempting to write any regular expression from scratch, it's best to first make a list of steps that concretely outlines what your target string looks like.
- Pay attention to edge cases. For instance, if you're identifying phone numbers, are you considering country codes and extensions?

Tokenizing

- Forming words from sequence of characters
- Surprisingly complex in English, can be harder in other languages
- Early IR systems:
  - any sequence of alphanumeric characters of length 3 or more
  - terminated by a space or other special character
  - upper-case changed to lower-case



Tokenizing

- Example:
  - “Bigcorp's 2007 bi-annual report showed profits rose 10%.” becomes
  - “bigcorp 2007 annual report showed profits rose”
- Too simple for search applications or even large-scale experiments
- Why? Too much information lost
  - Small decisions in tokenizing can have major impact on effectiveness of some queries

Tokenizing

- Small words can be important in some queries, usually in combinations
  - xp, ma, pm, ben e king, el paso, master p, gm, j lo, world war II
- Both hyphenated and non-hyphenated forms of many words are common
  - Sometimes hyphen is not needed
    - e-bay, wal-mart, active-x, cd-rom, t-shirts
  - At other times, hyphens should be considered either as part of the word or a word separator
    - winston-salem, mazda rx-7, e-cards, pre-diabetes, t-mobile, spanish-speaking

Tokenizing

- Special characters are an important part of tags, URLs, code in documents
- Capitalized words can have different meaning from lower case words
  - Bush, Apple
- Apostrophes can be a part of a word, a part of a possessive, or just a mistake
  - rosie o'donnell, can't, don't, 80's, 1890's, men's straw hats, master's degree, england's ten largest cities, shriner's

Tokenizing

- Numbers can be important, including decimals
  - nokia 3250, top 10 courses, united 93, quicktime 6.5 pro, 92.3 the beat, 288358
- Periods can occur in numbers, abbreviations, URLs, ends of sentences, and other situations
  - I.B.M., Ph.D., cs.umass.edu, F.E.A.R.
- Notes:
  - 1) tokenizing steps for *queries* must be identical to steps for *documents*
  - *2)* important decisions can be deferred to other components such as *information extraction* and *query transformation*

Stopping

- Function words (determiners, prepositions) have little meaning on their own
- High occurrence frequencies
- Treated as *stopwords* (i.e., removed)
  - reduce index space, improve response time, improve effectiveness
- Throwing out these words decreases index size, increases retrieval efficiency, and generally improves retrieval effectiveness.

Stopping

- Constructing a stopword list must be done with caution. Removing too many words will hurt retrieval effectiveness in particularly frustrating ways for the user. For instance, the query "to be or not to be" consists entirely of words that are usually considered stopwords. Although not removing stopwords may cause some problems in ranking, removing stopwords can cause perfectly valid queries to return no results.

Stopping

- Stopword list can be created from high-frequency words (e.g., top 50) or based on a standard list
  - words might be included that are important for some queries.
- Lists are customized for applications, domains, and even parts of documents
  - e.g., “click” is a good stopword for anchor text
- If storage space requirements allow, it can also be considered to index all words in documents, make decisions about which words to use at query time.
  - By keeping the stopwords in the index, there will be a number of possible ways to execute a query with stopwords in it.

Stemming

- There are many morphological variations of words
  - *inflectional* (plurals, tenses)
  - *derivational* (making verbs nouns etc.)
- In most cases, these have the same or very similar meanings
- Stemmers attempt to reduce morphological variations of words to a common stem
  - usually involves removing suffixes
- Can be done at indexing time or as part of query processing (like stopwords)

Stemming

- Generally, a small but significant effectiveness improvement
  - can be crucial for highly inflected languages (e.g., Arabic or Russian)
  - e.g., 5-10% improvement for English, up to 50% in Arabic

kitab	*a book*
kitabi	*my book*
al**kitab**	*the book*
kitabuki	*your book* (f)
kitabuka	*your book (m)*
kitabuhu	*his book*
kat**ab**a	*to write*
ma**ktab**a	*library, bookstore*
ma**ktab**	*office*

Words with the Arabic root **ktb**

Stemming

- Two basic types
  - Dictionary-based: uses lists of related words
  - Algorithmic: uses program to determine related words
- Algorithmic stemmers
  - *suffix-s:* remove 's' endings assuming plural
    - e.g., cats → cat, lakes → lake, wiis → wii
    - Many *false negatives*: centuries → centurie
    - Some *false positives*: is → I
- Note: Although stemming only removes the last few characters from a word, often leading to incorrect meanings and spelling, *lemmatizing* considers the context and uses a corpus to supply a lemma - the meaningful base form of a word.

Phrases

- Many queries are 2–3-word phrases
- Phrases are
  - More precise than single words
    - e.g., documents containing “black sea” vs. two words “black” and “sea”
  - Less ambiguous
    - e.g., “big apple” vs. “apple”
- Can be difficult for ranking
  - e.g., Given query “fishing supplies”, how do we score documents with
    - exact phrase many times, exact phrase just once, individual words in same sentence, same paragraph, whole document, variations on words?

Phrases

- Text processing issue – how are phrases recognized?
- Three possible approaches:
  - Identify syntactic structure of sentences using a *part-of-speech* (POS) tagger
  - Use word *n-grams*
  - Store word positions in indexes and use *proximity operators* in queries
- POS taggers use statistical models of text to predict syntactic tags of words. They are often used as a building block for language models.
  - Example tags:
    - NN (singular noun), NNS (plural noun), VB (verb), VBD (verb, past tense), VBN (verb, past participle), IN (preposition), JJ (adjective), CC (conjunction, e.g., “and”, “or”), PRP (pronoun), and MD (modal auxiliary, e.g., “can”, “will”).
    - Phrases can then be defined as simple noun groups (noun phrases), for example

Phrases

- Pos Tagging Example

Original text:

Document will describe marketing strategies carried out by U.S. companies for their agricultural chemicals, report predictions for market share of such chemicals, or report market statistics for agrochemicals, pesticide, herbicide, fungicide, insecticide, fertilizer, predicted sales, market share, stimulate demand, price cut, volume of sales.

Brill tagger:

Document/NN will/MD describe/VB marketing/NN strategies/NNS carried/VBD out/IN by/IN U.S./NNP companies/NNS for/IN their/PRP agricultural/JJ chemicals/NNS ,/, report/NN predictions/NNS for/IN market/NN share/NN of/IN such/JJ chemicals/NNS ,/, or/CC report/NN market/NN statistics/NNS for/IN agrochemicals/NNS ,/, pesticide/NN ,/, herbicide/NN ,/, fungicide/NN ,/, insecticide/NN ,/, fertilizer/NN ,/, predicted/VBN sales/NNS ,/, market/NN share/NN ,/, stimulate/VB demand/NN ,/, price/NN cut/NN ,/, volume/NN of/IN sales/NNS ./.

Phrases

- High-frequency simple noun phrases from a TREC corpus consisting mainly of news stories and a corpus of comparable size consisting of all the 1996 patents issued by the United States Patent and Trademark Office (PTO).

TREC data		Patent data	
Frequency	*Phrase*	*Frequency*	*Phrase*
65824	united states	975362	present invention
61327	article type	191625	u.s. pat
33864	los angeles	147352	preferred embodiment
18062	hong kong	95097	carbon atoms
17788	north korea	87903	group consisting
17308	new york	81809	room temperature
15513	san diego	78458	seq id
15009	orange county	75850	brief description
12869	prime minister	66407	prior art
12799	first time	59828	perspective view
12067	soviet union	58724	first embodiment
10811	russian federation	56715	reaction mixture
9912	united nations	54619	detailed description
8127	southern california	54117	ethyl acetate
7640	south korea	52195	example 1
7620	end recording	52003	block diagram
7524	european union	46299	second embodiment
7436	south africa	41694	accompanying drawings
7362	san francisco	40554	output signal
7086	news conference	37911	first end
6792	city council	35827	second end
6348	middle east	34881	appended claims
6157	peace process	33947	distal end
5955	human rights	32338	cross-sectional view
5837	white house	30193	outer surface

Phrases

- Although POS tagging produces reasonable phrases and is used in a number of applications, in general it is too slow to be used as the basis for phrase indexing of large collections.
- There are simpler and faster alternatives that are just as effective. One approach is to store word position information in the indexes and use this information to identify phrases only when a query is processed.
- The identification of syntactic phrases is replaced by testing word proximity constraints, such as whether two words occur within a specified text window.

Word N-Grams

- POS tagging or testing word proximities at query time are too slow for large collections
- Simpler definition – phrase is any sequence of *n* words – known as *n-grams*
  - *bigram*: 2-word sequence, *trigram*: 3-word sequence, *unigram*: single words
  - N-grams also used at character level for applications such as OCR where the text is "noisy" and word matching can be difficult.
    - Character n-grams are also used for indexing languages such as Chinese that have no word breaks.

Word N-Grams

- N-grams, both character and word, are generated by choosing a particular value for n and then moving that "window" forward one unit (character or word) at a time. In other words, n-grams overlap.
- For example, the word "tropical" contains the following character bigrams: tr, ro, op, pi, ic, ca, and al. Indexes based on n-grams are obviously larger than word indexes.
- The more frequently a word n-gram occurs, the more likely it is to correspond to a meaningful phrase in the language.
- Could index all n-grams up to specified length
  - Much faster than POS tagging
  - Uses a lot of storage
    - e.g., document containing 1,000 words would contain 3,990 instances of word n-grams of length $2 \leq n \leq 5$

Word N-Grams

- Web search engines index n-grams
- Google sample:

Number of tokens:	1,024,908,267,229
Number of sentences:	95,119,665,584
Number of unigrams:	13,588,391
Number of bigrams:	314,843,401
Number of trigrams:	977,069,902
Number of fourgrams:	1,313,818,354
Number of fivegrams:	1,176,470,663

- Most frequent trigram in English is “all rights reserved”
  - In Chinese, “limited liability corporation”

Document Structure and Markup

- Some parts of documents are more important than others
- Document parser recognizes structure using markup, such as HTML tags
  - Headers, anchor text, bolded text all likely to be important
  - Metadata can also be important
  - Links used for *link analysis*

Document Structure and Markup

- Example of a Web Page

Tropical fish

From Wikipedia, the free encyclopedia

Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species. Fishkeepers often use the term *tropical fish* to refer only those requiring fresh water, with saltwater tropical fish referred to as *marine fish*.

Tropical fish are popular aquarium fish , due to their often bright coloration. In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

- The page has some structure that could be used in a ranking algorithm. The main heading for the page, "tropical fish", indicates that this phrase is particularly important. The same phrase is also in bold and italics in the body of the text, which is further evidence of its importance. Other words and phrases are used as the anchor text for links and are likely to be good terms to represent the page content.

Automate Text Transformation with `BeautifulSoup`

"Beautiful Soup, so rich and green,

Waiting in a hot tureen!

Who for such dainties would not stoop?

Soup of the evening, beautiful Soup!"

- The `BeautifulSoup` library was named after a Lewis Carroll poem of the same name in Alice's Adventures in Wonderland.
- Like its Wonderland namesake, `BeautifulSoup` tries to make sense of the nonsensical; it pulls data out of HTML and XML files and provide idiomatic ways of navigating, searching, and modifying the parse tree.

Installing `BeautifulSoup`

- To install `BeautifulSoup`, opening a Python terminal and run:

```
pip install beautifulsoup4
```

- `BeautifulSoup` will be recognized as a Python library on your machine.
- You can test the installation by importing the library:

```
from bs4 import BeautifulSoup
```

- The import should complete without errors.

Running `BeautifulSoup`

- The most commonly used object in the BeautifulSoup library is, appropriately, the `BeautifulSoup` object.

```
from urllib.request import urlopen
from bs4 import BeautifulSoup
html =  urlopen('http://www.pythonscraping.com/pages/page1.html')
bs = BeautifulSoup(html.read(), 'html.parser')
print(bs.h1)
```

- The output is as follows:

See parser.py

```
<h1>An Interesting Title</h1>
```

Running `BeautifulSoup`

- `urlopen()` returns a http.client.HTTPResponse object that wraps the HTTP response from the server providing access to the request *headers* and the entity *body*. Example of an HTML header:
  - `Server: nginx`
  - `Date: Tue, 07 Nov 2023 21:17:27 GMT`
  - `Content-Type: text/html`
  - `Content-Length: 564`
  - `Connection: close`
  - `X-Accel-Version: 0.01`
  - `Last-Modified: Sat, 09 Jun 2018 19:15:58 GMT`
  - `ETag: "234-56e3a58a63780"`
  - `Accept-Ranges: bytes`
  - `Vary: Accept-Encoding`
  - `X-Powered-By: PleskLin`
- The `.read()` method reads and returns the response body only.

Running `BeautifulSoup`

- Note that this returns only the first instance of the `h1` tag found.
- By convention, only one `h1` tag should be used on a single page, but conventions are often broken on the Web, so you should be aware that this will retrieve the first instance of the tag only, and not necessarily the one that you're looking for.
  - For instance:

```
bs = BeautifulSoup("<HTML><body><h1>Trash</h1><h1>Title</h1></body><HTML>", 'html.parser')
print(bs.h1)
```

See paser2.py

  - The output is as follows:

```
<h1>Trash</h1>
```

Running `BeautifulSoup`

- The `prettify()` method will turn a `BeautifulSoup` parse tree into a nicely formatted string, with a separate line for each tag and each string:

```
bs = BeautifulSoup("<HTML><body><p>Some text</p></body><HTML>", 'html.parser')
print(bs.body.prettify())
```

- The output is as follows:

```
<body>
 <p>
  Some text
 </p>
</body>
```

See paser3.py

CalPolyPomona

Running `BeautifulSoup`

- Printing the entire parse tree.

```
bs = BeautifulSoup(html.read(), 'html.parser')
print(bs)
```

- The output is as follows:

```
<html>
<head> <title>A Useful Page</title> </head>
<body>
<h1>An Interesting Title</h1>
<div>
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.
</div>
</body>
</html>
```

See paser4.py

CalPolyPomona

Running `BeautifulSoup`

- Previously, we imported the `urlopen` function and called `html.read()` to get the HTML content of the page.
- In addition to the text string, `BeautifulSoup` can also use the file object directly returned by `urlopen`, without needing to call `.read()` first:

```
bs = BeautifulSoup(html, 'html.parser')
```

- This HTML content is then transformed into a `BeautifulSoup` object, with the following structure:

html → *<html><head>...</head><body>...</body></html>*

head → *<head><title>A Useful Page<title></head>*

title → *<title>A Useful Page</title>*

body → *<body><h1>An Int...</h1> Lorem ip... </body>*

h1 → *<h1>An Interesting Title</h1>*

div → * Lorem Ipsum dolor... *

Running `BeautifulSoup`

- Note that the `h1` tag that you extract from the page is nested two layers deep into your `BeautifulSoup` object `structure (html → body → h1)`.
- However, when you actually fetch it from the object, you call the `h1` tag directly:

```
bs.h1
```

- In fact, any of the following function calls would produce the same output:

```
bs.html.body.h1
bs.body.h1
bs.html.h1
```

See paser5.py

Running `BeautifulSoup`

- When you create a `BeautifulSoup` object, two arguments are passed in:

```
bs = BeautifulSoup(html.read(), 'html.parser')
```

- The first is the HTML text the object is based on, and the second specifies the parser that you want `BeautifulSoup` to use in order to create that object.
- In the majority of cases, it makes no difference which parser you choose. `html.parser` is a parser that is included with Python 3 and requires no extra installations in order to use.

Running `BeautifulSoup`

- Another popular parser is `lxml`.
- This can be installed through `pip`:

```
pip3 install lxml
```

- `lxml` can be used with `BeautifulSoup` by changing the parser string provided:

```
bs = BeautifulSoup(html.read(), 'lxml')
```

- `lxml` has some advantages over `html.parser` in that it is generally better at parsing “messy” or malformed HTML code. It is forgiving and fixes problems like unclosed tags, tags that are improperly nested, and missing head or body tags.
- One of the disadvantages of `lxml` is that it has to be installed separately and depends on third-party C libraries to function. This can cause problems for portability and ease of use.

Handling Exceptions

- Every time you access a tag in a BeautifulSoup object, it is smart to add a check to make sure the tag actually exists.
- If you attempt to access a tag that does not exist, BeautifulSoup will return a None object.

```
print(bs.nonExistentTag)
```

- The problem is, attempting to access a tag on a None object itself will result in an AttributeError being thrown.

```
print(bs.nonExistentTag.someTag)
```

- This returns an exception:

AttributeError: 'NoneType' object has no attribute 'someTag'

Handling Exceptions

- To avoid the exception, you can use try block:

```
bs = BeautifulSoup(html, 'html.parser')
try:
    badContent = bs.nonExistentTag.someTag
except AttributeError as e:
    print('Tag was not found')
```

- Or check if the tag exists:

```
badContent = bs.nonExistentTag
if badContent != None:
   print(badContent.someTag)
else:
    print('Tag was not found')
```

See paser6.py

Handling Exceptions

- Function getTitle, which returns either the title of the page or a None object

```
def getTitle(url):
    try:
      html = urlopen(url)
    except HTTPError as e:
      return None
    try:
      bs = BeautifulSoup(html.read(), 'html.parser')
      title = bs.body.h1
    except AttributeError as e:
      return None
    return title

title = getTitle('http://www.pythonscraping.com/pages/page1.html')
if title == None:
  print('Title could not be found')
else:
  print(title)
```

See paser7.py

Advanced HTML Parsing

- Nearly every website you encounter contains stylesheets. Although you might think that a layer of styling on websites that is designed specifically for browser and human interpretation might be a bad thing, the advent of CSS is a boon for parsers.
- CSS relies on the differentiation of HTML elements that might otherwise have the exact same markup in order to style them differently. Some tags might look like this:

```
<span class="green"></span>
<span class="red"></span>
```

- Parsers can easily separate these two tags based on their class; for example, they might use BeautifulSoup to grab all the red text but none of the green text.
- Because CSS relies on these identifying attributes to style sites appropriately, you are almost guaranteed that these class and ID attributes will be plentiful on most modern websites.

Advanced HTML Parsing

- The page located at

  http://www.pythonscraping.com/pages/warandpeace.html.

the lines spoken by characters in the story are in red, whereas the names of characters are in green. You can see the span tags, which reference the appropriate CSS classes, in the following sample of the page's source code:

```
<span class="red">Heavens! what a virulent attack!</span> replied

<span class="green">the prince</span>, not in the least disconcerted by this reception.
```

Advanced HTML Parsing

- Using this BeautifulSoup object, you can use the `find_all` function to extract a Python list of proper nouns found by selecting only the text within `<span class="green"></span>` tags:

```
from urllib.request import urlopen
from bs4 import BeautifulSoup

html = urlopen('http://www.pythonscraping.com/pages/warandpeace.html')
bs = BeautifulSoup(html.read(), 'html.parser')

nameList = bs.find_all('span', {'class':'green'})
for name in nameList:
    print(name.get_text())
```

See paser8.py

```
or by using single-line loops leveraging list comprehension

print([ tag.get_text() for tag in bs.find_all('span', {'class':'green'}) ])
```

Advanced HTML Parsing

- When run, it should list all the proper nouns in the text, in the order they appear in War and Peace.
- Previously, you've called `bs.tagName` to get the first occurrence of that tag on the page.
- Now, you're calling `bs.find_all(tagName, tagAttributes)` to get a list of all of the tags on the page, rather than just the first.
- After getting a list of names, the program iterates through all names in the list, and prints `name.get_text()` in order to separate the content from the tags.
- Keep in mind that it's much easier to find what you're looking for in a BeautifulSoup object than in a block of text. Calling `.get_text()` should always be the last thing you do, immediately before you print, store, or manipulate your final data.

Advanced HTML Parsing

- BeautifulSoup's `find()` and `find_all()` are the two functions you will likely use the most. With them, you can easily filter HTML pages to find lists of desired tags, or a single tag, based on their various attributes.
- The two functions are extremely similar:

```
find_all(tag, attributes, recursive, text, limit, keywords)
find(tag, attributes, recursive, text, keywords)
```

- In all likelihood, 95% of the time you will need to use only the first two arguments: `tag` and `attributes`.

Advanced HTML Parsing

- The `tag` argument is one that you've seen before; you can pass a string name of a tag or even a Python list of string tag names. For example, the following returns a list of all the header tags in a document:

```
find_all(['h1','h2','h3','h4','h5','h6'])
```

- The `attributes` argument takes a Python dictionary of attributes and matches tags that contain any one of those attributes. For example, the following function would return both the green and red span tags in the HTML document:

```
find_all('span', {'class':{'green', 'red'}})
```

See paser9.py and paser10.py

Advanced HTML Parsing

- The `recursive` argument is a Boolean that if sets to True, the `find_all` function looks into children, and children's children, for tags that match your parameters.
- If `recursive` is False, it will look only at the top-level tags in your document. By default, `find_all` works recursively (recursive is set to True); it's generally a good idea to leave this as is, unless you really know what you need to do, and performance is an issue.
- The `text` argument is unusual in that it matches based on the text content of the tags, rather than properties of the tags themselves. For instance, if you want to find the number of times "the prince" is surrounded by tags on the example page, you could replace your `.find_all()` function in the previous example with the following:

```
print(len(bs.find_all(text='the prince')))
```

The output of this is 7.

See paser11.py

Advanced HTML Parsing

- The `limit` argument, of course, is used only in the `find_all` method; `find` is equivalent to the same `find_all` call, with a limit of 1. You might set this if you're interested only in retrieving the first x items from the page. Be aware, however, that this gives you the first items on the page in the order that they occur, not necessarily the first ones that you want.
- The `keyword` argument allows you to select tags that contain a particular attribute or set of attributes. For example:

```
title = bs.find_all(id='title')
```

- This returns the first tag with the word "title" in the `id` attribute. It is equivalent to:

```
bs.find_all('', {'id':'title'})
```

See paser12.py

Advanced HTML Parsing

- Chaining Searches.
  - When using `find` and `find_all` to get a Tag object, you can also use this object to search further. Chaining searches can lead to performance increases as you reduce the search space for each step. For example:

```
html = urlopen('http://www.pythonscraping.com/pages/page3.html')
bs = BeautifulSoup(html, 'html.parser')

gifts = bs.find(id='giftList').find_all('tr',{'class':'gift'})
print(len(gifts))
```

The output of this is 5.

See paser13.py

Regular Expressions and BeautifulSoup

- BeautifulSoup and regular expressions go hand in hand when it comes to scraping the Web. In fact, most functions that take in a string argument (e.g., `find(id="aTagIdHere"))` will also take in a regular expression.
- If we check the page found at http://www.pythonscraping.com/pages/page3.html, we note that the site has many product images, which take the following form:

```
<img src="../img/gifts/img3.jpg">
```

- If you wanted to grab URLs to all of the product images, it might seem fairly straightforward at first: just grab all the image tags by using `.find_all("img").`

Regular Expressions and BeautifulSoup

- But there's a problem. In addition to the obvious "extra" images (e.g., logos), modern websites often have hidden images, blank images used for spacing and aligning elements, and other random image tags you might not be aware of. Certainly, you can't count on the only images on the page being product images.
- Also, let's assume that the layout of the page might change, or that, for whatever reason, you don't want to depend on the position of the image in the page in order to find the correct tag. This might be the case when you are trying to grab specific elements or pieces of data that are scattered randomly throughout a website.

```
<td>
    <img src="../img/gifts/img2.jpg">
</td>
```

```
<div>
    <img src="../img/gifts/img3.jpg">
</div>
```

Regular Expressions and BeautifulSoup

- The solution is to look for something identifying about the tag itself. In this case, you can look at the file path of the product images:

```
import re
html = urlopen('http://www.pythonscraping.com/pages/page3.html')
bs = BeautifulSoup(html, 'html.parser')

images = bs.find_all('img',{'src':re.compile('\.\./img/gifts/img.*\.jpg')})

for image in images:
    print(image['src'])
    print(image.get('src'))
    # both ways we get the text from an element.
```

See regex1.py

Regular Expressions and BeautifulSoup

- This prints only the relative image paths that start with ../img/gifts/img and end in .jpg, the output of which is the following:

```
../img/gifts/img1.jpg
../img/gifts/img2.jpg
../img/gifts/img3.jpg
../img/gifts/img4.jpg
../img/gifts/img6.jpg
```

A regular expression can be inserted as any argument in a BeautifulSoup expression, allowing you a great deal of flexibility in finding target elements.

Regular Expressions and BeautifulSoup

- Another simple example

```
import re
html = urlopen('http://www.pythonscraping.com/pages/page3.html')
bs = BeautifulSoup(html, 'html.parser')

regex = re.compile('\$(\d+,)?\d+(.)\d+\\n')
prices = bs.find_all(text=regex)
for price in prices:
    print(price.strip())
```

- You can also use the `search` method to find a pattern in a String

```
x = re.search("^The.*Spain$", "The rain in Spain")
print(x.string)
```

See regex2.py

Regular Expressions in PostgreSQL

- The filtering condition with the `LIKE` operator is limited to finding patterns by including wildcards (`%`) only.
- To overcome this, PostgreSQL provides an advanced way of pattern matching using Regular Expressions which are implemented using the TILDE (~) operator.

```
Select * from documents where text ~'^Months.*';
```

Regular Expressions in MongoDB

- In MongoDB, we can do pattern matching in two different ways:
  - With `$regex` Operator

```
db.documents.find({text : {$regex : "^Months"}}).pretty()
```

  - Without `$regex` Operator using `//` delimiters

```
db.documents.find({text : /^Months/}).pretty()
```

Tokenizing with `scikit-learn`

- The `CountVectorizer` provides a simple way to both tokenize a collection of text documents and build a vocabulary of known words, but also to encode new documents using that vocabulary.
- You can use it as follows:
  - Create an instance of the `CountVectorizer` class.
  - Call the `fit()` function in order to learn a vocabulary from one or more documents.
  - Call the `transform()` function on one or more documents as needed to encode each as a vector.
    - An encoded vector is returned with a length of the entire vocabulary and an integer count for the number of times each word appeared in the document.
    - The vectors returned from a call to `transform()` will be sparse vectors, and you can transform them back to `numpy` arrays to look and better understand what is going on by calling the `toarray()` function.

Tokenizing with `scikit-learn`

- Example:

```
#list of text documents
text = ["The quick brown fox jumped over the lazy dog."]

# create the transform
vectorizer = CountVectorizer()

# tokenize and build vocab
vectorizer.fit(text)

# summarize
print(vectorizer.vocabulary_)
```

See tokenizing.py

Output:

{'the': 7, 'quick': 6, 'brown': 0, 'fox': 2, 'jumped': 3, 'over': 5, 'lazy': 4, 'dog': 1}

Tokenizing with `scikit-learn`

- All words were made lowercase by default and the punctuation was ignored. These and other aspects of tokenizing can be configured in the API documentation.

```
# encode document
vector = vectorizer.transform(text)

# summarize encoded vector
print(vector.shape)
print(vector.toarray())
```

Output:

(1, 8)

[[1 1 1 1 1 1 1 2]]

See tokenizing.py

Tokenizing with `scikit-learn`

- Running the example first prints the vocabulary, then the shape of the encoded document.
- There are 8 words in the vocab, and therefore encoded vectors have a length of 8. Remember that words need to be encoded as integers or floating-point values for use as input to a search or machine learning process.
- We can see an array version of the encoded vector showing a count of 1 occurrence for each word except the (index and id 7) that has an occurrence of 2.

```
{'the': 7, 'quick': 6, 'brown': 0, 'fox': 2, 'jumped': 3, 'over': 5, 'lazy': 4, 'dog': 1}
[[1 1 1 1 1 1 1 2]]
```

Tokenizing with `scikit-learn`

- Importantly, the same vectorizer can be used on documents that contain words not included in the vocabulary. These words are ignored, and no count is given in the resulting vector.
- Example:

```
# encode another document
text2 = ["the puppy"]

vector = vectorizer.transform(text2)
print(vector.shape)
print(vector.toarray())
```

See tokenizing.py

Output:

{'the': 7, 'quick': 6, 'brown': 0, 'fox': 2, 'jumped': 3, 'over': 5, 'lazy': 4, 'dog': 1}

[[0 0 0 0 0 0 0 1]]

Stooping with `scikit-learn`

- You can pass the `stop_words` list as an argument to the `CountVectorizer` transformer.
- The stop words can be passed as a custom list or a predefined list of stop words can be used by specifying the language. In this case, we are using English stopwords.

```
sw=['the','over']
vectorizer = CountVectorizer(stop_words=sw)
```

- To check the stop words used, we can use the following code:

```
print(vectorizer.stop_words)
```

Output:

['the', 'over']

See stopping.py

Stooping with `scikit-learn`

- Using a Predefined set of stopwords:

```
vectorizer = CountVectorizer(stop_words='english')
vectorizer.fit(text)
print(vectorizer.vocabulary_)
vector = vectorizer.transform(text)
print(vector.shape)
print(vector.toarray())
```

Output:

{'quick': 5, 'brown': 0, 'fox': 2, 'jumped': 3, 'lazy': 4, 'dog': 1}

(1, 6)

[[1 1 1 1 1 1]]

See stopping.py

Lemmatizing with `scikit-learn/nltk`

- Fancy token-level analysis such as stemming or lemmatizing are not included in the scikit-learn codebase but can be added by customizing either the tokenizer or the analyzer.
- Here's a `CountVectorizer` with a tokenizer and `lemmatizer` using NLTK:

```
from nltk import word_tokenize
from nltk.stem import WordNetLemmatizer

class LemmaTokenizer:
      def __init__(self):
          self.wnl = WordNetLemmatizer()
      def __call__(self, doc):
          return [self.wnl.lemmatize(t) for t in word_tokenize(doc)]

vectorizer = CountVectorizer(tokenizer=LemmaTokenizer())
```

WordNet is a lexical database of semantic relations between words that links words into semantic relations including synonyms, hyponyms, and meronyms.

Lemmatizing with `scikit-learn`

```
# To install nltk
import nltk
nltk.download() #download and install all available packages

text = ["The dogs slept behind the churches."]
vectorizer.fit(text)
print(vectorizer.vocabulary_)

vector = vectorizer.transform(text)
print(vector.shape)
print(vector.toarray())
```

Output:

{'the': 5, 'dog': 3, 'slept': 4, 'behind': 1, 'church': 2, '.': 0}

(1, 6)

[[1 1 1 1 1 2]]

See lemmatizing.py

N-grams with `scikit-learn`

- `CountVectorizer` is able to tokenize the data and split it into n-grams, of which we can define the type and the length by passing a string ('word', 'char', 'char_wb') to the `analyzer` argument and a tuple to the `ngram_range` argument.
- For example, 'word' and (1,1) would give us unigrams or 1-grams such as "whey" and "protein", while 'word' and (2,2) would give us bigrams or 2-grams, such as "whey protein".

```
vectorizer = CountVectorizer(analyzer='word', ngram_range = (1, 1))
```

See ngrams.py

N-grams with `scikit-learn`

```
text = ["The dogs slept behind the churches."]
vectorizer = CountVectorizer(analyzer='word', ngram_range = (1, 1))

vectorizer.fit(text)
print(vectorizer.vocabulary_)

vector = vectorizer.transform(text)
print(vector.shape)
print(vector.toarray())
```

Output:

{'the': 4, 'dogs': 2, 'slept': 3, 'behind': 0, 'churches': 1}

(1, 5)

[[1 1 1 1 2]]

N-grams with `scikit-learn`

```
text = ["The dogs slept behind the churches."]
vectorizer = CountVectorizer(analyzer='word', ngram_range = (1, 2))


vectorizer.fit(text)
print(vectorizer.vocabulary_)


vector = vectorizer.transform(text)
print(vector.shape)
print(vector.toarray())
```

{'the': 7, 'dogs': 3, 'slept': 5, 'behind': 0, 'churches': 2, 'the dogs': 9, 'dogs slept': 4, 'slept behind': 6, 'behind the': 1, 'the churches': 8}

(1, 10)

[[1 1 1 1 1 1 1 2 1 1]]

N-grams with `scikit-learn`

```
text = ["The dogs slept behind the churches."]
vectorizer = CountVectorizer(analyzer=char_wb', ngram_range = (3, 3))


vectorizer.fit(text)
print(vectorizer.vocabulary_)


vector = vectorizer.transform(text)
print(vector.shape)
print(vector.toarray())
```

{' th': 4, 'the': 25, 'he ': 13, ' do': 2, 'dog': 8, 'ogs': 20, 'gs ': 12, ' sl': 3, 'sle': 24, 'lep': 18, 'ept': 10, 'pt ': 21, ' be': 0, 'beh': 5, 'ehi': 9, 'hin': 15, 'ind': 17, 'nd ': 19, ' ch': 1, 'chu': 7, 'hur': 16, 'urc': 26, 'rch': 22, 'che': 6, 'hes': 14, 'es.': 11, 's. ': 23}

(1, 27)

[[1 1 1 1 2 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 2 1]]

CalPolyPomona

Indexes

Lecture 7

Topics

- Data Structures
- Abstract Model of Ranking
  - A More Concrete Model
- Inverted Index
  - Documents
  - Counts
  - Positions
  - Proximity Matches
  - Fields
  - Extent Lists
  - Scores
  - Ordering
- Using MongoDB References

Data Structures

- If you want to store a list of items, linked lists and arrays are good choices.
- If you want to quickly find an item based on an attribute, a hash table is a better choice.
- More complicated tasks require more complicated structures, such as B-trees or priority queues.

Data Structures

- Why are all these data structures necessary? Strictly speaking, they aren't.
- Most things you want to do with a computer can be done with arrays alone. However, arrays have drawbacks: unsorted arrays are slow to search, and sorted arrays are slow at insertion.
- By contrast, hash tables and trees are fast for both search and insertion. These structures are more complicated than arrays, but the speed difference is compelling.

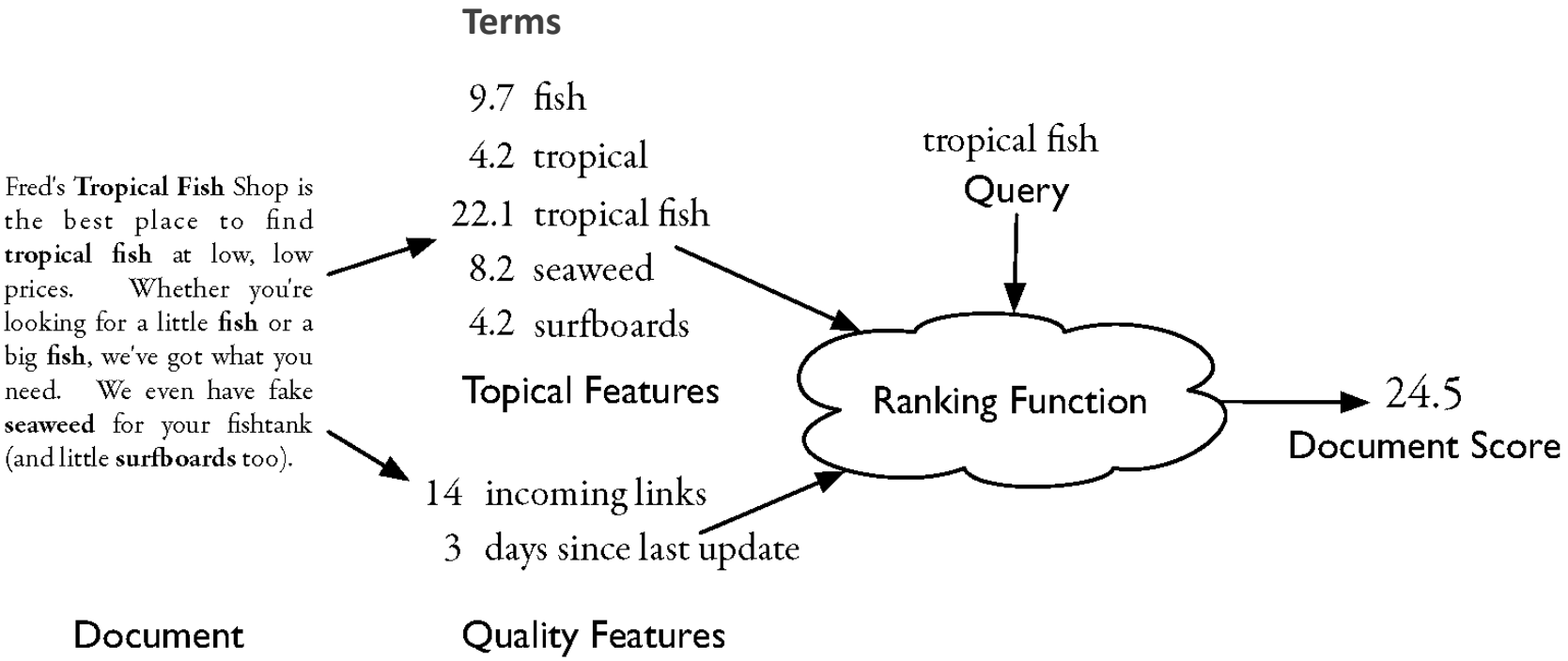
Data Structures

- Text search has unique requirements, so it calls for its own kind of data structure, the inverted index.
- The name "inverted index" is really an umbrella term for many different kinds of structures that share the same general philosophy - documents associated with words, rather than words with documents to support search with fast response time and updates.
- The specific kind of data structure used depends on the ranking function.

Abstract Model of Ranking

- Text search engines use a particular form of search: *ranking*
  - documents are retrieved in sorted order according to a score computed using the document representation, the query, and a *ranking algorithm*
- To dive into the index construction for a search system, we will consider an abstract model of ranking
  - this will allow us to skip the details of retrieval models

Abstract Model of Ranking



- For now, the contents of the Ranking Function component are unimportant, except for the fact that these components ignore many of the document features, focusing only on the small subset that relate to the query.
- This fact makes **inverted indexes** appealing data structures for search.

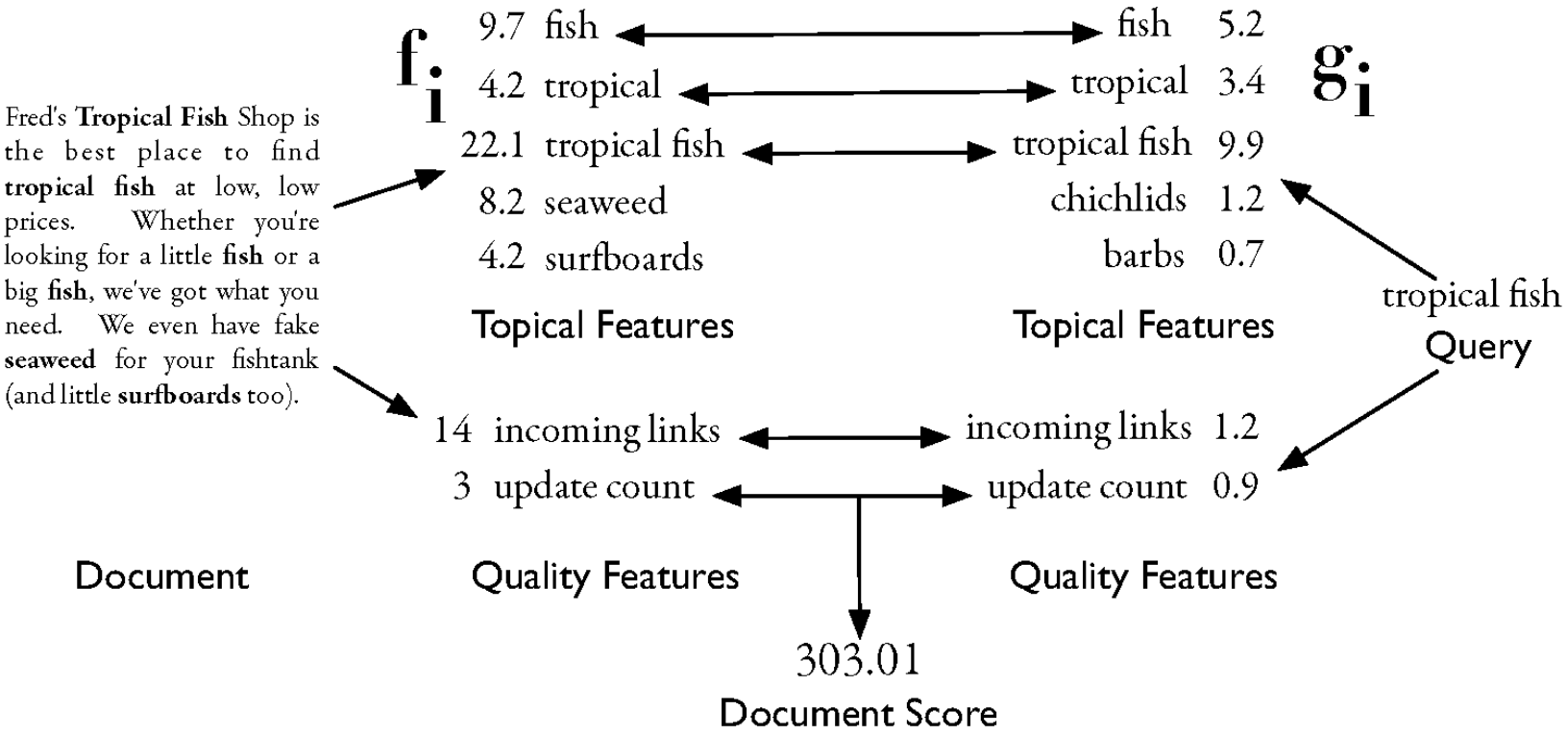
Abstract Model of Ranking

- On the left side of the figure is a sample document.
- The text is transformed into index terms or document features.
- There are two kinds of features. On top, we have topical features, which estimate the degree to which the document is about a particular subject, and on the bottom, we have two possible document quality features.
- A feature function is just a mathematical expression that generates numbers from document text (e.g., binary, tf, tf-idf).
- On the right side of the figure, we see a cloud representing the ranking function. The ranking function takes data from document features combined with the query and produces a score.
- A high score means a good match for the query.

A More Concrete Model

$$R(Q, D) = \sum_i g_i(Q) f_i(D)$$

f_i is a document feature function
g_i is a query feature function



A More Concrete Model

Point to ponder #1

$f_{tropical(D)}$ and $f_{ish(D)}$ will be high for each documents?

These values will be larger for documents that contain the words "tropical" or "fish" more often or more prominently.

A More Concrete Model

- The query also has some feature values that aren't topical, such as the update count feature - how important document updates are to relevance.
  - For instance, the query "today's weather in london“ would be higher for documents updated frequently.
- If a retrieval system had to perform a sum over millions of features for every document, text search systems would not be practical.
- In practice, the query features $g_i(Q)$ are mostly zero. This means that the sum for each document is only over the non-zero $g_i(Q)$ values.

Inverted Index

- All modern search engine indexes are based on inverted indexes.
- An inverted index is organized by index term.
- The index is inverted because usually, we think of words being a part of documents, but if we invert this idea, documents are associated with words.

Documents	
DocId	Docs
1	To be or not to be
2	To be right
3	Not to be left

Terms	
TermId	Term
1	be
2	left
3	not
4	or
5	right
6	to

Index	
Terms	Docs
1	1 2 3
2	3
3	1 3
4	1
5	2
6	1 2 3

Inverted Index

- Index terms are often alphabetized like a traditional book index, but they need not be, since they are often found directly using a hash table.
- Each index term has its own inverted list that holds the relevant data for that term.
- In an index for a book, the relevant data is a list of page numbers, while in a search engine, the data might be a list of documents or a list of word occurrences.
- Each list entry is called a posting, and the part of the posting that refers to a specific document or location is often called a pointer.
- Each document in the collection is given a unique number to make it efficient for storing document pointers.
- Inverted lists are ordered by document number, which makes certain kinds of query processing more efficient.

CalPolyPomona

Inverted Index

- Indexes in books store more than just location information. For important words, often one of the page numbers is marked in boldface, indicating that this page contains a definition or extended discussion about the term.
- Inverted lists can also have extended information, where postings can contain a range of information other than just locations. By storing the right information along with each posting, the feature functions can be computed efficiently.

Documents	
DocId	Docs
1	To be or not to be
2	To be right
3	Not to be left

Terms	
TermId	Term
1	be
2	left
3	not
4	or
5	right
6	to

Index	
Terms	Docs
1	1:2:[2,6], 2:1:[2], 3:1:[3]
2	3:1:[4]
3	1:1:[4], 3:1:[1]
4	1:1:[3]
5	2:1:[3]
6	1:2:[1,5], 2:1:[1], 3:1:[2]

DocId:termCount:[idx,idx]

Documents

- The simplest form of an inverted list stores just the documents that contain each word, and no additional information. This kind of list is similar to the kind of index found in books.
- Consider the documents (sentences):

S_1 Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

S_2 Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.

S_3 Tropical fish are popular aquarium fish, due to their often bright coloration.

S_4 In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

CalPolyPomona

Documents

Simple Inverted Index

and	1														
aquarium	3														
are	3	4													
around	1														
as	2														
both	1														
bright	3														
coloration	3	4													
derives	4														
due	3														
environments	1														
fish	1	2	3	4											
fishkeepers	2														
found	1														
fresh	2														
freshwater	1	4													
from	4														
generally	4														
in	1	4													
include	1														
including	1														
iridescence	4														
marine	2														
often	2	3													
only	2														
pigmented	4														
popular	3														
refer	2														
referred	2														
requiring	2														
salt	1	4													
saltwater	2														
species	1														
term	2														
the	1	2													
their	3														
this	4														
those	2														
to	2	3													
tropical	1	2	3												
typically	4														
use	2														
water	1	2	4												
while	4														
with	2														
world	1														

Documents

- Notice that this simple index does not record the number of times each word appears;
- It only records the documents in which each word appears. For instance, S2 contains the word "fish" twice, whereas S1 contains "fish" only once.
- The inverted list for "fish" shows no distinction between sentences 1 and 2; both are listed in the same way.
- For instance, consider the query "tropical fish". Three sentences match this query: S1, S2, and S3. The data in the document-based index gives us no reason to prefer any of these sentences over any other.

Documents

- Inverted lists become more interesting when we consider their intersection.
- Suppose we want to find the sentence that contains the words "coloration" and "freshwater". The inverted index tells us that "coloration" appears in S3 and S4, while "freshwater" appears in S1 and S4. We can quickly tell that only S4 contains both "coloration" and "freshwater".
- Since each list is sorted by sentence number, finding the intersection of these lists takes O(min(m, n)).

Counts

- In an inverted index that contains only document information, the features are binary - 1 if the document contains a term, 0 otherwise. This is important information, but it is too coarse to find the best few documents when there are a lot of possible matches.
- Now, postings can also include a second number to record the number of times the word appears in the document. This extra data allows us to prefer S2 over S1 and S3 for the query "tropical fish", since S2 contains "tropical“ twice and "fish" three times.
- Word counts can help distinguish documents that are about a particular subject from those that discuss that subject in passing, being a powerful predictor of document relevance

CalPolyPomona

Counts

Inverted Index with counts

- supports better ranking algorithms

and	1:1			
aquarium	3:1			
are	3:1	4:1		
around	1:1			
as	2:1			
both	1:1			
bright	3:1			
coloration	3:1	4:1		
derives	4:1			
due	3:1			
environments	1:1			
fish	1:2	2:3	3:2	4:2
fishkeepers	2:1			
found	1:1			
fresh	2:1			
freshwater	1:1	4:1		
from	4:1			
generally	4:1			
in	1:1	4:1		
include	1:1			
including	1:1			
iridescence	4:1			
marine	2:1			
often	2:1	3:1		
only	2:1			
pigmented	4:1			
popular	3:1			
refer	2:1			
referred	2:1			
requiring	2:1			
salt	1:1	4:1		
saltwater	2:1			
species	1:1			
term	2:1			
the	1:1	2:1		
their	3:1			
this	4:1			
those	2:1			
to	2:2	3:1		
tropical	1:2	2:2	3:1	
typically	4:1			
use	2:1			
water	1:1	2:1	4:1	
while	4:1			
with	2:1			
world	1:1			

Positions

- When looking for matches for a query like "tropical fish", the location of the words in the document is an important predictor of relevance.
- Although a document that contains the words "tropical" and "fish" is likely to be relevant, we really want to know if the document contains the exact phrase "tropical fish".
- To determine this, we can add position information to our index. Then, each posting would contain two numbers: a document number first, followed by a word position. In the previous indexes, there was just one posting per document. Now there is one posting per word occurrence.

CalPolyPomona

Positions

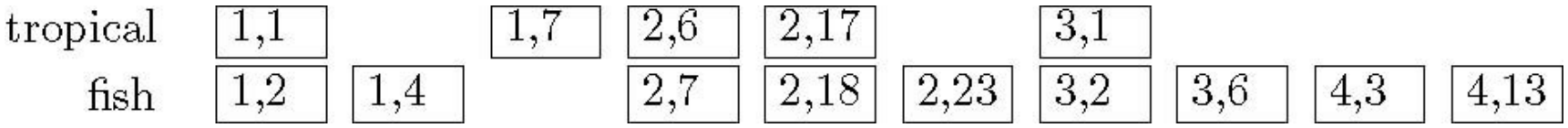
Inverted Index with positions

- supports proximity matches

and	1,15					
aquarium	3,5					
are	3,3	4,14				
around	1,9					
as	2,21					
both	1,13					
bright	3,11					
coloration	3,12	4,5				
derives	4,7					
due	3,7					
environments	1,8					
fish	1,2	1,4	2,7	2,18	2,23	
			3,2	3,6	4,3	
			4,13			
fishkeepers	2,1					
found	1,5					
fresh	2,13					
freshwater	1,14	4,2				
from	4,8					
generally	4,15					
in	1,6	4,1				
include	1,3					
including	1,12					
iridescence	4,9					
marine	2,22					
often	2,2	3,10				
only	2,10					
pigmented	4,16					
popular	3,4					
refer	2,9					
referred	2,19					
requiring	2,12					
salt	1,16	4,11				
saltwater	2,16					
species	1,18					
term	2,5					
the	1,10	2,4				
their	3,9					
this	4,4					
those	2,11					
to	2,8	2,20	3,8			
tropical	1,1	1,7	2,6	2,17	3,1	
typically	4,6					
use	2,3					
water	1,17	2,14	4,12			
while	4,10					
with	2,15					
world	1,11					

Proximity Matches

- The word position information is most interesting when we look at intersections with other posting lists.
- For instance, two inverted lists are lined up next to each other below. We see that the word "tropical" is the first word in S1, and "fish" is the second word in S1, which means that S1 must start with the phrase "tropical fish". The word "tropical" appears again as the seventh word in S1, but "fish" does not appear as the eighth word, so this is not a phrase match.
- There are four occurrences of the phrase "tropical fish" in the four sentences. The phrase matches are easy to see in the figure; they happen at the points where the postings are lined up in columns.

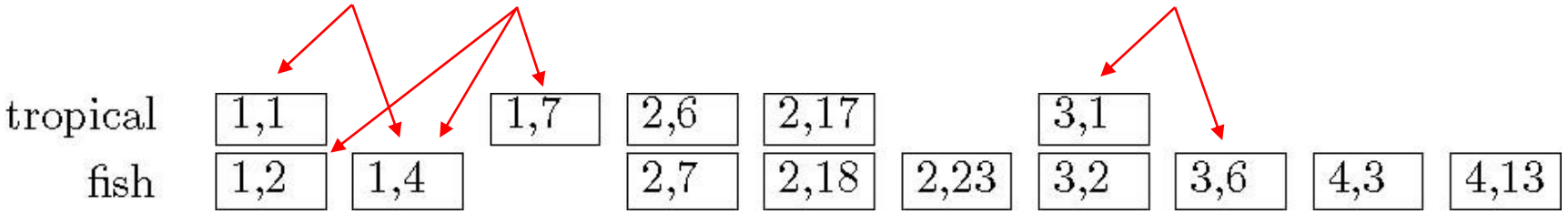


Proximity Matches

- This same technique can be extended to find longer phrases or more general proximity expressions, such as "find tropical within 5 words of fish.“
- Suppose that the word "tropical" appears at position *p.* We can then look in the inverted list for "fish" for any occurrences between position *p* - 5 and *p* + 5.
- Any of those occurrences would constitute a match.

Point to ponder #2

Where are the matches below according to this definition?



Fields

- Real documents are not just lists of words. They have sentences and paragraphs that separate concepts into logical units. Some documents have titles and headings that provide short summaries of the rest of the content.
- Special types of documents have their own sections; for example, every email contains sender information and a subject line. All of these are instances of what we will call document fields, which are sections of documents that carry some kind of semantic meaning.

Fields

- To improve search, search engines need to be able to determine whether a word is in a particular field.
- One option is to make separate inverted lists for each kind of document field. Essentially, you could build one index for document titles, one for document headings, and one for body text.
  - finding a word in any section of the document is tricky since you need to fetch inverted lists from many different indexes

<title>

mars	1.txt
planet	1.txt, 2.txt
water	2.txt

<body>

mars	1.txt, 2.txt
planet	1.txt, 2.txt
water	1.txt, 2.txt
red	1.txt, 3.txt
air	3.txt

Fields

- Another option is to store information in each word posting about where the word occurred. For instance, we could specify that the number 0 indicates a title and 1 indicates body text, or a combination of both.
  - if you have more fields than just a title, the representation will grow.

<title>

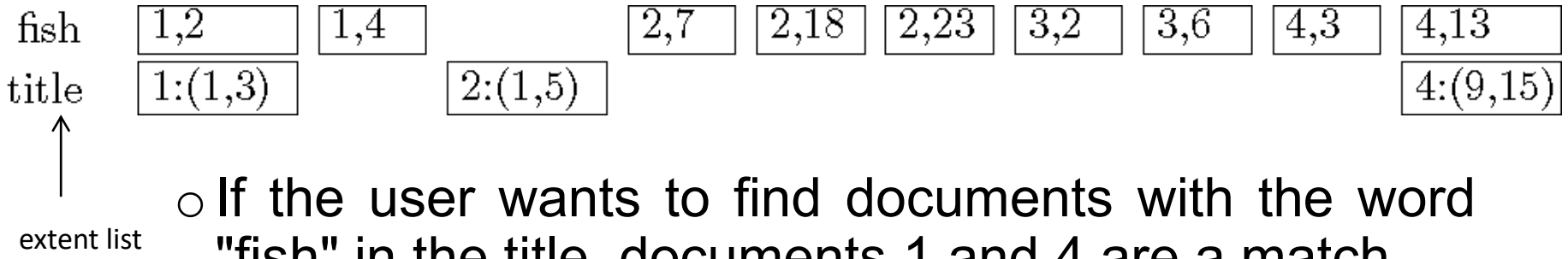
mars	1:0, 2:0
planet	1:0, 2:0
water	1:0, 2:0
red	1:1, 3:1
air	3:1

Extent Lists

- Now, suppose we want to index books. Some books, like the one below, have more than one author. Somewhere in the XML description of this book, you might find:
  - <author>W. Bruce Croft</author>,
  - <author>Donald Metzler</author>, and
  - <author>Trevor Strohman</author>
- Suppose you would like to find books by an author named Croft Donald. If you type the phrase query "croft donald" into a search engine, should this book match? No, they are in two distinct author fields.

Extent Lists

- This is where extent lists come in. An *extent* is a contiguous region of a document that represents extents using word positions
  - For example, if the title of a book started on the fifth word and ended *just before* the ninth word, we could encode that as (5,9).



  - If the user wants to find documents with the word "fish" in the title, documents 1 and 4 are a match.

Scores

- If the inverted lists are going to be used to generate feature function values, why not just store the value of the feature function?
- This approach makes it possible to store feature function values that would be too computationally intensive to compute during the query processing phase.
- It also moves complexity out of the query processing engine and into the indexing code, where it may be more tolerable.

Scores

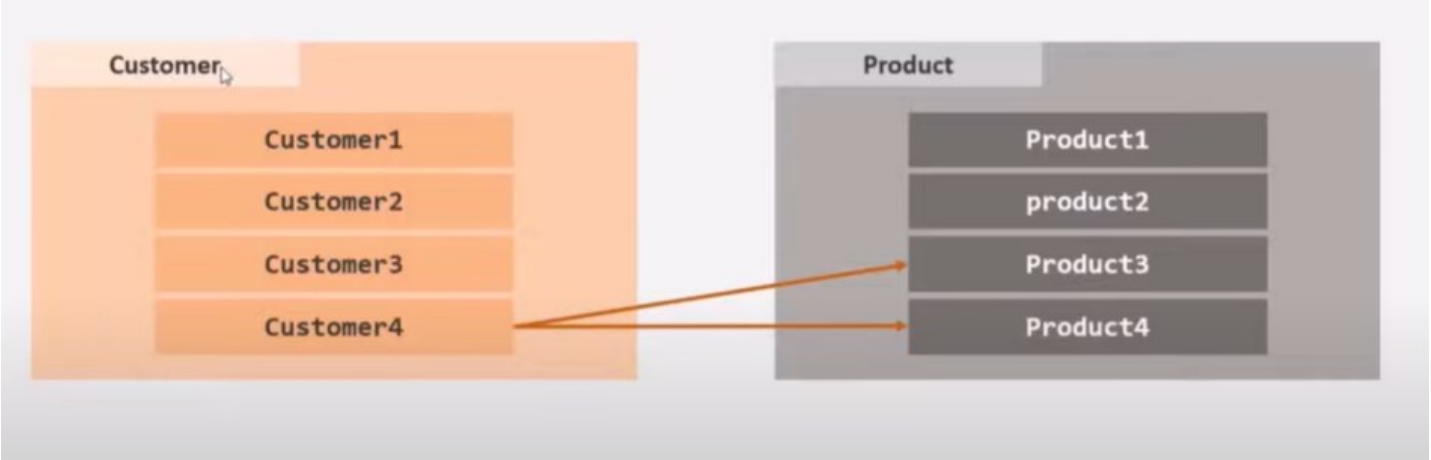
- We can store the final value right in the inverted list.
  - e.g., list for “fish” [(1:3.6), (3:2.2)], where 3.6 is the total feature value for document 1
  - this improves speed but reduces flexibility since we can no longer change the scoring mechanism once the index is built.
- More importantly, information about word proximity is gone in this model, meaning that we can't include phrase information in scoring unless we build inverted lists for phrases, too. These pre-computed phrase lists require considerable additional space.

Ordering

- So far, we have assumed that the postings of each inverted list would be ordered by document number. Although this is the most popular option, this is not the only way to order an inverted list. An inverted list can also be ordered by score so that the highest-scoring documents come first.
  - e.g., list for “fish” [(2:4.2), (1:3.6), (3:2.2), (4:1.8)]
- Query processing engine can focus only on the top part of each inverted list, where the highest-scoring documents are recorded
  - very efficient for single-word queries

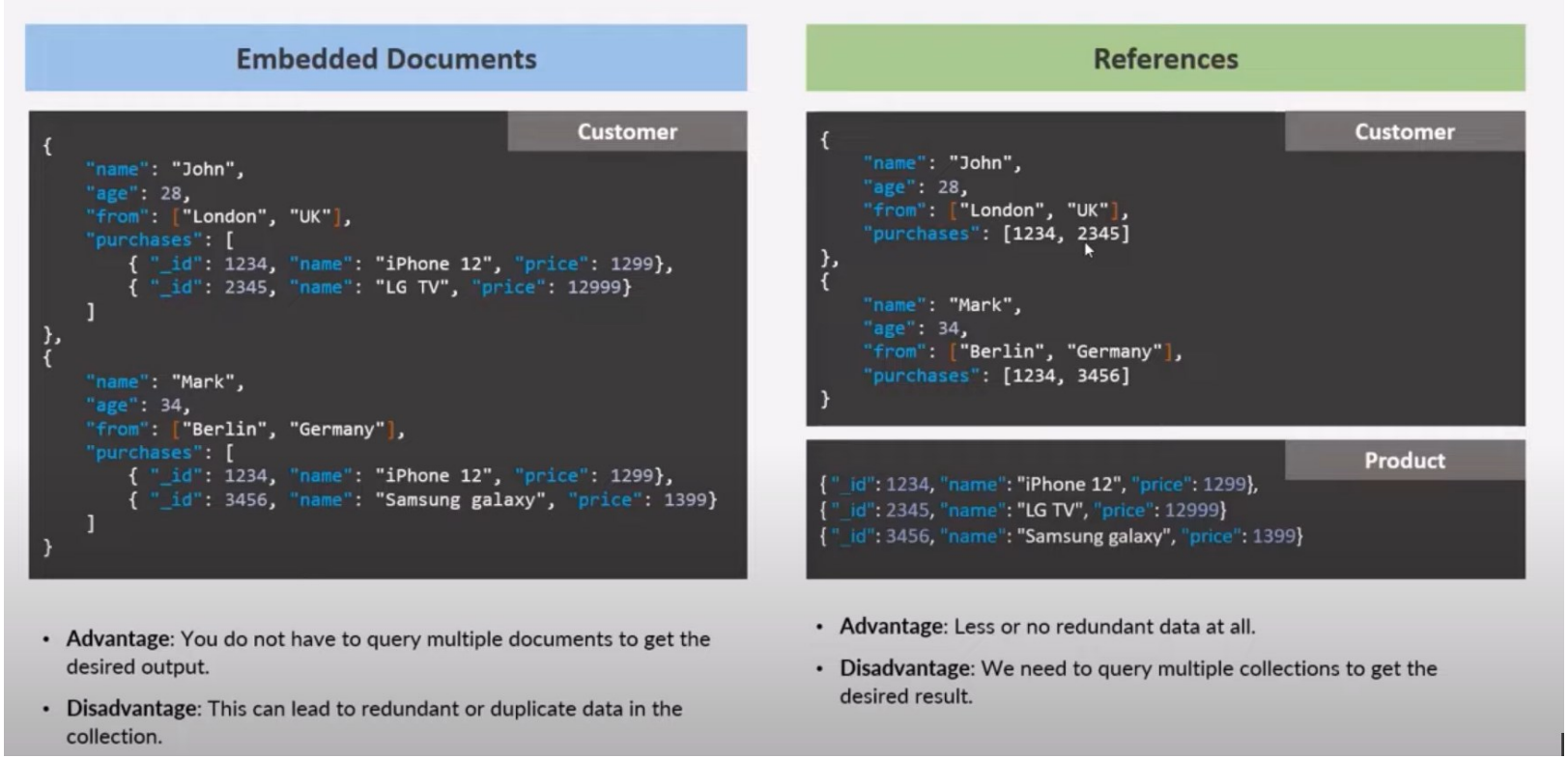
Using MongoDB References

- So far, we have worked with one collection in Mongo DB. But what if we need multiple collections to represent/store our data?
- For instance, we might have customer and product collections in the database and these collections might have a relation – the products that someone has purchased.



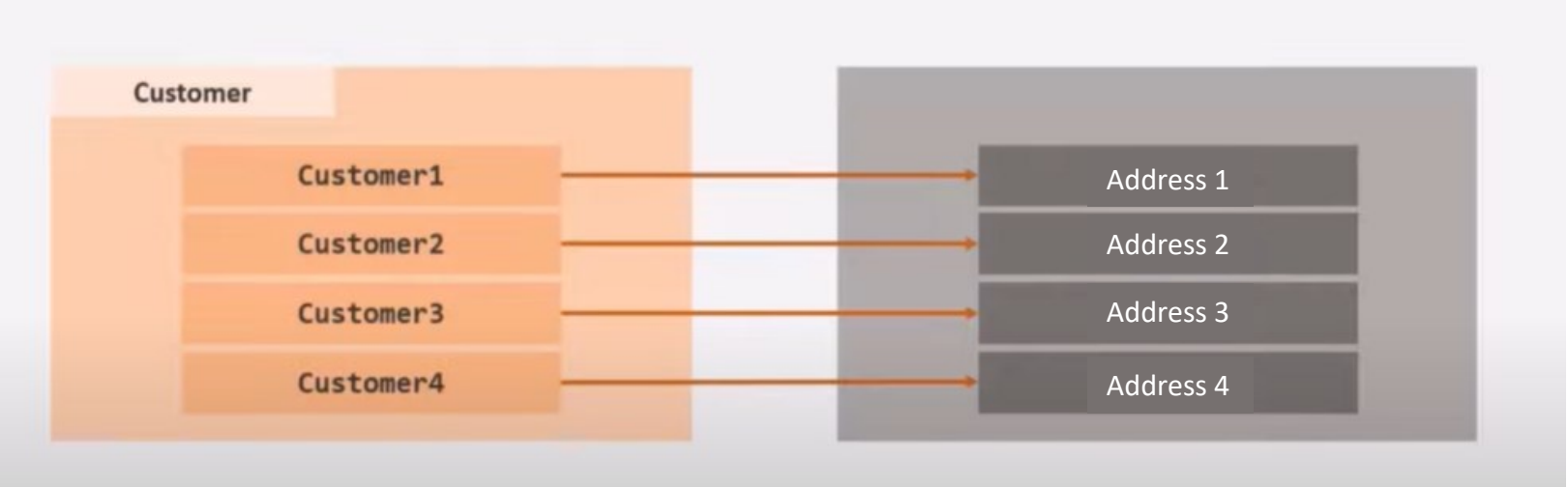
Using MongoDB References

- In addition to embeddings there is another way to specify a relation in Mongo DB: using references.



Using MongoDB References

- One-to-one relation



CalPolyPomona

Using MongoDB References

- One-to-one relation

```
ecommerce> db.customer.find()
[
  {
    _id: ObjectId("6568278fe3c2cb64d66360a4"),
    name: 'John',
    age: 28,
    contact: { email: 'john@abc.com', phone: Long("9989786567") },
    address: ObjectId("65682cc9e3c2cb64d66360a7")
  },
  {
    _id: ObjectId("65682893e3c2cb64d66360a5"),
    name: 'Mark',
    age: 38,
    contact: { email: 'mark@abc.com', phone: Long("9989786457") },
    address: ObjectId("65682cfbe3c2cb64d66360aa")
  }
]
```

Using MongoDB References

- One-to-one relation
  - By using the `$lookup` operator, we can fetch data from two related documents merged into one resulting document.
  - This *aggregation* stage performs a left outer join to a collection in the same database.
  - There are four required fields:
    - `from`: The collection to use for `lookup` in the same database
    - `localField`: The field in the primary collection that can be used as a unique identifier in the from collection.
    - `foreignField`: The field in the from collection that can be used as a unique identifier in the primary collection.
    - `as`: The name of the new field that will contain the matching documents from the from collection.

Using MongoDB References

- One-to-one relation
  - Example:

```
db.customer.aggregate([
{$lookup: {from: "address",
           localField: "address",
           foreignField: "_id",
           as: "addr"}},
{$project: {address: 0}}
])
```

CalPolyPomona

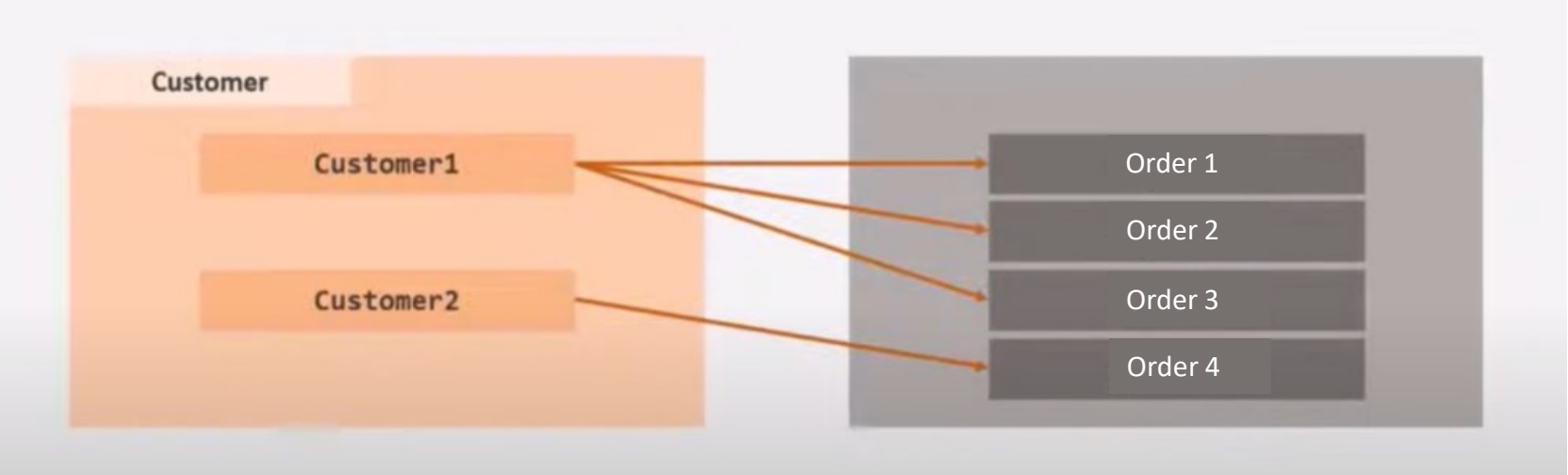
Using MongoDB References

- One-to-one relation
  - Output:

```
[
  {
    _id: ObjectId("6568278fe3c2cb64d66360a4"),
    name: 'John',
    age: 28,
    contact: { email: 'john@abc.com', phone: Long("9989786567") },
    addr: [
      {
        _id: ObjectId("65682cc9e3c2cb64d66360a7"),
        street: 'park avenue',
        houseNo: 78,
        PIN: 556655,
        city: 'London',
        country: 'UK'
      }
    ]
  },
  {
    _id: ObjectId("65682893e3c2cb64d66360a5"),
    name: 'Mark',
    age: 38,
    contact: { email: 'mark@abc.com', phone: Long("9989786457") },
    addr: [
      {
        _id: ObjectId("65682cfbe3c2cb64d66360aa"),
        street: 'salt lake',
        houseNo: 38,
        PIN: 556699,
        city: 'Berlin',
        country: 'Germany'
      }
    ]
  }
]
```

Using MongoDB References

- One-to-many relation



CalPolyPomona

Using MongoDB References

- One-to-many relation

```
ecommerce> db.customer.find()
[
  {
    _id: ObjectId("6568278fe3c2cb64d66360a4"),
    name: 'John',
    age: 28,
    contact: { email: 'john@abc.com', phone: Long("9989786567") },
    address: ObjectId("65682cc9e3c2cb64d66360a7"),
    orders: [ 3 ]
  },
  {
    _id: ObjectId("65682893e3c2cb64d66360a5"),
    name: 'Mark',
    age: 38,
    contact: { email: 'mark@abc.com', phone: Long("9989786457") },
    address: ObjectId("65682cfbe3c2cb64d66360aa"),
    orders: [ 1, 2, 4 ]
  }
]
```

Using MongoDB References

- One-to-many relation
  - Example:

```
db.customer.aggregate([
{$lookup: {from:"orders",
           localField:"orders",
           foreignField:"_id",
           as: "orderDetails"}},
{$project: {orders: 0}}
])
```

CalPolyPomona

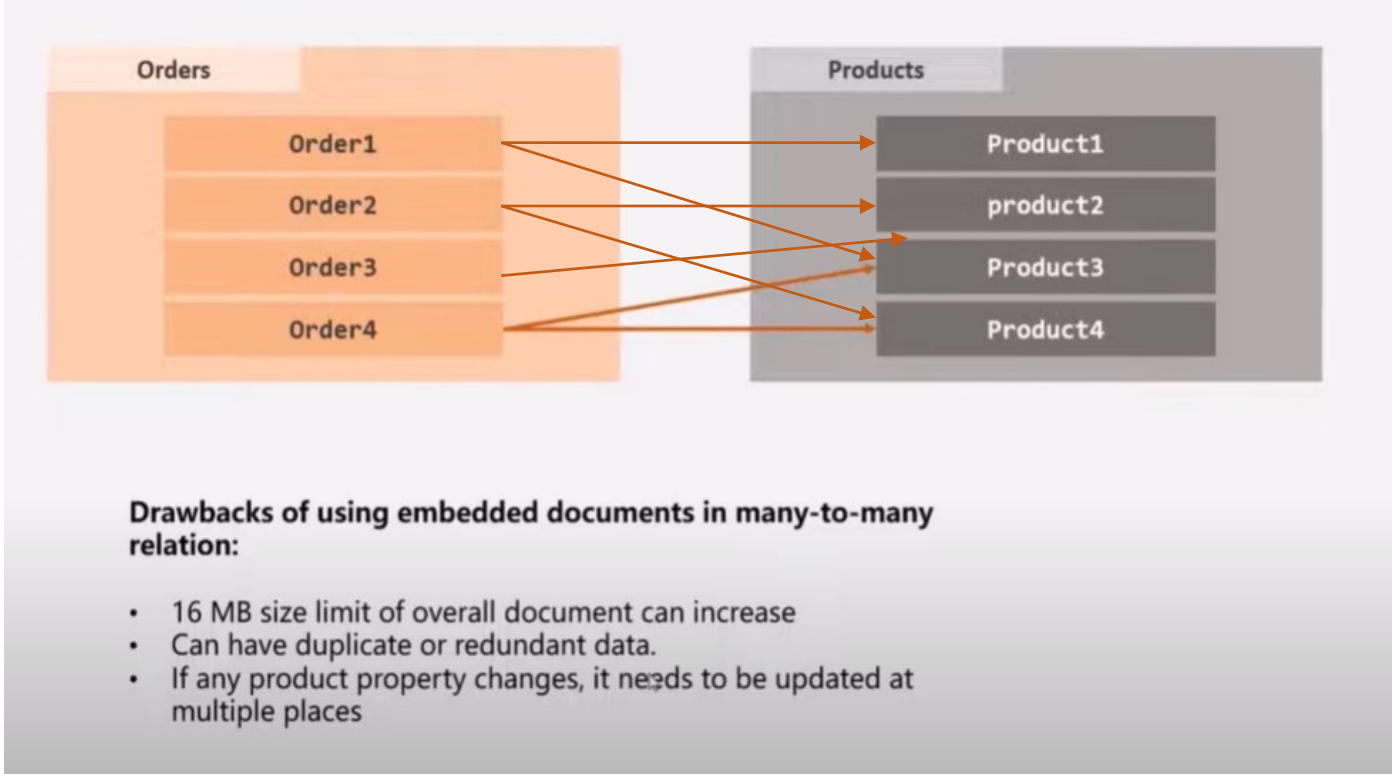
Using MongoDB References

- One-to-many relation
  - Output:

```
[
  {
    _id: ObjectId("6568278fe3c2cb64d66360a4"),
    name: 'John',
    age: 28,
    contact: { email: 'john@abc.com', phone: Long("9989786567") },
    address: ObjectId("65682cc9e3c2cb64d66360a7"),
    orderDetails: [
      {
        _id: 3,
        date: ISODate("2011-06-07T08:00:00.000Z"),
        pId: [ 345, 787, 456 ]
      }
    ]
  },
  {
    _id: ObjectId("65682893e3c2cb64d66360a5"),
    name: 'Mark',
    age: 38,
    contact: { email: 'mark@abc.com', phone: Long("9989786457") },
    address: ObjectId("65682cfbe3c2cb64d66360aa"),
    orderDetails: [
      {
        _id: 1,
        date: ISODate("2011-03-02T08:00:00.000Z"),
        pId: [ 123, 456 ]
      },
      {
        _id: 2,
        date: ISODate("2011-03-04T08:00:00.000Z"),
        pId: [ 123, 787 ]
      },
      {
        _id: 4,
        date: ISODate("2011-06-07T08:00:00.000Z"),
        pId: [ 563 ]
      }
    ]
  }
]
```

Using MongoDB References

- Many-to-many relation



Using MongoDB References

- Many-to-many relation

```
ecommerce> db.orders.find()
[
  {
    _id: 1,
    date: ISODate("2011-03-02T08:00:00.000Z"),
    pId: [ 123, 456 ]
  },
  {
    _id: 2,
    date: ISODate("2011-03-04T08:00:00.000Z"),
    pId: [ 123, 787 ]
  },
  {
    _id: 3,
    date: ISODate("2011-06-07T08:00:00.000Z"),
    pId: [ 345, 787, 456 ]
  },
  { _id: 4, date: ISODate("2011-06-07T08:00:00.000Z"), pId: [ 563 ] }
]
```

Using MongoDB References

- Many-to-many relation
  - Example:

```
db.orders.aggregate([
{$lookup: {from: "products",
           localField: "pId",
           foreignField: "_id",
           as: "products"}},
{$project: {pId: 0}}
])
```

CalPolyPomona

Using MongoDB References

- Many-to-many relation
  - Output:

```
[
  {
    _id: 1,
    date: ISODate("2011-03-02T08:00:00.000Z"),
    products: [
      { _id: 456, name: 'LG TV', brand: 'LG', price: 1999 },
      {
        _id: 123,
        name: 'A book',
        author: [ 'John Smith' ],
        price: 129
      }
    ]
  },
  {
    _id: 2,
    date: ISODate("2011-03-04T08:00:00.000Z"),
    products: [
      {
        _id: 123,
        name: 'A book',
        author: [ 'John Smith' ],
        price: 129
      },
      {
        _id: 787,
        name: 'Another book',
        author: [ 'Sarah King', 'Kelly M' ],
        price: 119
      }
    ]
  },
  {
    _id: 3,
    date: ISODate("2011-06-07T08:00:00.000Z"),
    products: [
      { _id: 345, name: 'iPhone 13', brand: 'Apple', price: 1299 },
      {
        _id: 787,
        name: 'Another book',
        author: [ 'Sarah King', 'Kelly M' ],
        price: 119
      },
      { _id: 456, name: 'LG TV', brand: 'LG', price: 1999 }
    ]
  },
  {
    _id: 4,
    date: ISODate("2011-06-07T08:00:00.000Z"),
    products: [
      {
        _id: 563,
        name: 'Washing Machine',
        brand: 'Bosch',
        price: 11299
      }
    ]
  }
]
```

CalPolyPomona

Retrieval Models

Lecture 8

Topics

- Overview
- Basic Models
  - Boolean Retrieval
  - Vector Space model
    - `scikit-learn` Python implementation
- Advanced Models
  - Probabilistic model
  - Language modeling

Overview

- Provide a mathematical framework for defining the search process
  - includes explanation of assumptions
  - basis of many ranking algorithms
  - can be implicit through trial and error
- Progress in retrieval models has corresponded with improvements in effectiveness

Overview

- Relevance is a key but complex concept
  - Many factors to consider
  - People often disagree when making relevance judgments
- Retrieval models make various assumptions about relevance to simplify problem
  - e.g., *topical* vs. *user* relevance
  - e.g., *binary* vs. *multi-valued* relevance

Overview

- Basic Models
  - Boolean Retrieval
  - Vector Space Model

Boolean Retrieval

- Two possible outcomes for query processing
  - TRUE and FALSE
  - “exact-match” retrieval
  - simplest form of ranking
- Query usually specified using Boolean operators
  - AND, OR, NOT
  - proximity operators also used

Boolean Retrieval

- Advantages
  - Results are predictable, relatively easy to explain
  - Many different features can be incorporated
  - Efficient processing since many documents can be eliminated from search
- Disadvantages
  - Effectiveness depends entirely on user (no way to distinguish between retrieved documents)
  - Simple queries usually don't work well
  - Complex queries are difficult

Boolean Retrieval

- Sequence of queries driven by number of retrieved documents
  - “lincoln”
  - “president AND lincoln"
  - “president AND lincoln AND NOT (automobile OR car)”
  - “president AND lincoln AND biography AND life AND birthplace AND gettysburg AND NOT (automobile OR car)”
  - “president AND lincoln AND (biography OR life OR birthplace OR gettysburg) AND NOT (automobile OR car)”

Vector Space Model

- Documents and query represented by a vector of term weights with t dimensions. Note that terms can be words, stems, phrases, sequence of characters, etc.
- A document D_i is represented by a vector of index terms, were d_{ij} represents the weight of the jth term:

$$D_i = (d_{i1}, d_{i2}, \ldots, d_{it})$$

- A collection containing n documents can be represented by a matrix of term weights

	$Term_1$	$Term_2$	$\ldots$	$Term_t$
Doc_1	d_{11}	d_{12}	$\ldots$	d_{1t}
Doc_2	d_{21}	d_{22}	$\ldots$	d_{2t}
$\vdots$	$\vdots$			
Doc_n	d_{n1}	d_{n2}	$\ldots$	d_{nt}

Vector Space Model

- Queries are represented the same way as documents. That is, a query Q is represented by a vector of t weights, were q_j represents the weight of the jth term:

$$Q = (q_1, q_2, \ldots, q_t)$$

CalPolyPomona

Vector Space Model

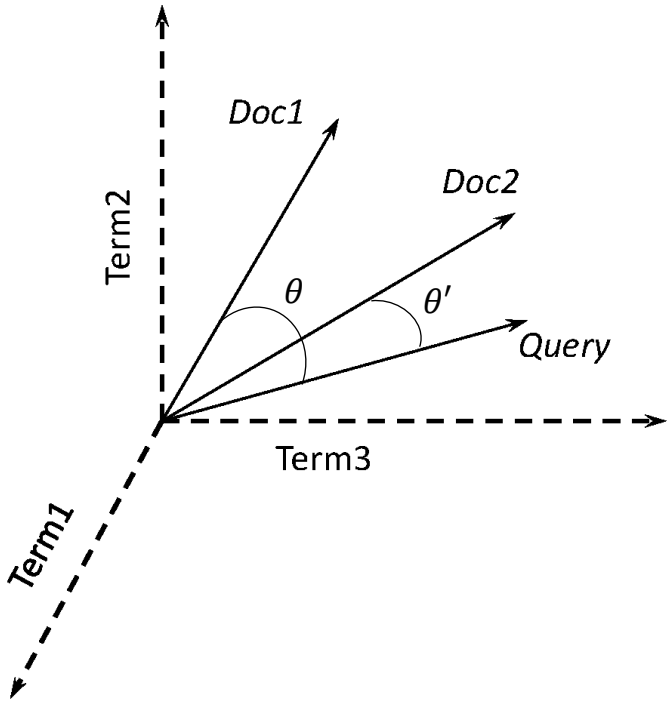
D_1 Tropical Freshwater Aquarium Fish.
D_2 Tropical Fish, Aquarium Care, Tank Setup.
D_3 Keeping Tropical Fish and Goldfish in Aquariums, and Fish Bowls.
D_4 The Tropical Tank Homepage - Tropical Fish and Aquariums.

Terms	Documents			
	D_1	D_2	D_3	D_4
aquarium	1	1	1	1
bowl	0	0	1	0
care	0	1	0	0
fish	1	1	2	1
freshwater	1	0	0	0
goldfish	0	0	1	0
homepage	0	0	0	1
keep	0	0	1	0
setup	0	1	0	0
tank	0	1	0	1
tropical	1	1	1	2

- Document D_3, for example, is represented by: (1,1,0,2,0,1,0,1,0,0,1).
- Query "tropical fish", for example, is represented by: (0,0,0,1,0,0,0,0,0,0,1).

Vector Space Model

- One of the appealing aspects of the vector space model is the use of simple diagrams to visualize the documents and queries.
- Examples of a 3-D space.



Vector Space Model

- Documents are ranked by the distance between points representing query and documents
  - *Similarity* measure more common than a distance or *dissimilarity* measure
  - e.g., Cosine similarity – the most successful one. It measures the cosine of the angle between the query and the document vectors.

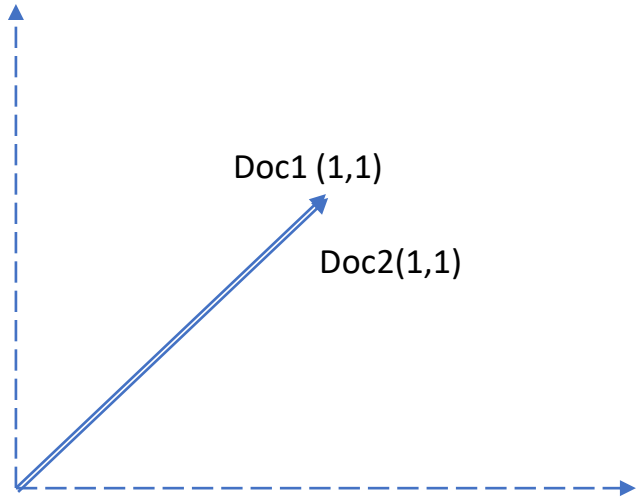
$$Cosine(D_i, Q) = \frac{\sum_{j=1}^{t} d_{ij} \cdot q_j}{\sqrt{\sum_{j=1}^{t} {d_{ij}}^2 \cdot \sum_{j=1}^{t} {q_j}^2}} = Cosine\,(\theta)$$

Vector Space Model

- The numerator of this measure is the sum of the products of the term weights for the *matching query* and document terms (known as the dot product or inner product).
- The denominator normalizes this score by dividing by the product of the lengths of the two vectors.

Vector Space Model

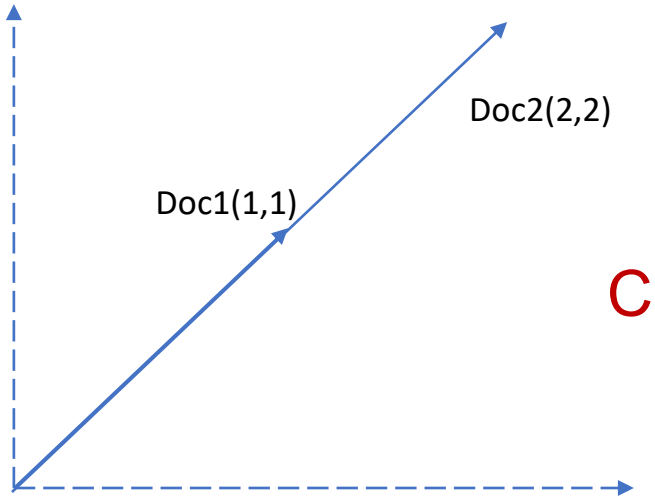
- Point to ponder #1
- When the vectors are normalized so that all documents and queries are represented by vectors of equal length, what is the cosine of the angle between two identical vectors?



Cosine(Doc1,Doc2) = 1

Vector Space Model

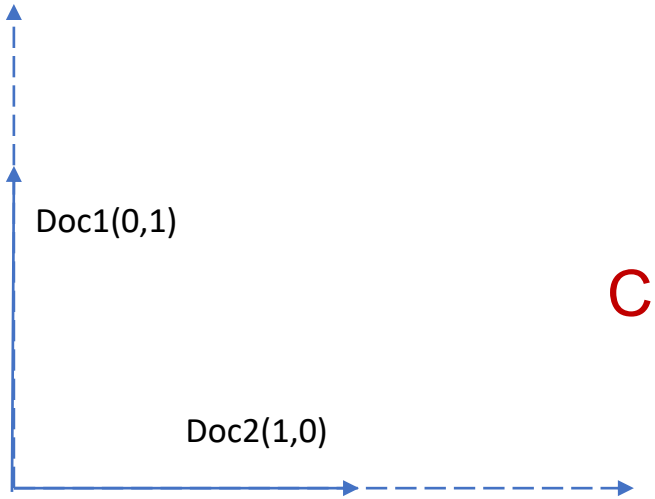
- Point to ponder #2
- How about vectors that are scalar multiple of each other?



Cosine(Doc1,Doc2) = 1

Vector Space Model

- Point to ponder #3
- How about vectors that do not share any non-zero terms?



Cosine(Doc1,Doc2) = 0

Vector Space Model

- Cosine similarity takeaway: it is effective because it captures the orientation (or "direction") of the vectors and not their magnitude, making it sensitive to the similarities in the pattern of the data, and insensitive to the magnitude of the vectors.
- In other words, does this query and document point in the same direction?

Vector Space Model

- Consider two documents D_1, D_2 and a query Q *being represented by tf-idf weights*

  D_1 = (0.5, 0.8, 0.3), D_2 = (0.9, 0.4, 0.2), Q = (1.5, 1.0, 0)

$$Cosine(D_1, Q) = \frac{(0.5 \times 1.5) + (0.8 \times 1.0)}{\sqrt{(0.5^2 + 0.8^2 + 0.3^2)(1.5^2 + 1.0^2)}}$$

$$= \frac{1.55}{\sqrt{(0.98 \times 3.25)}} = 0.87$$

$$Cosine(D_2, Q) = \frac{(0.9 \times 1.5) + (0.4 \times 1.0)}{\sqrt{(0.9^2 + 0.4^2 + 0.2^2)(1.5^2 + 1.0^2)}}$$

$$= \frac{1.75}{\sqrt{(1.01 \times 3.25)}} = 0.97$$

Ranking (Docs closer to the query)

Vector Space Model

- We treat the query as if it were a new document but only consider terms previously identified and indexed for the collection of documents.
- *tf* will depend on the query while *idf* will be the same computed earlier for documents.
- If a query contains a new “term”, this “term” will be discarded.

Vector Space Model

- Advantages
  - Simple computational framework for ranking
  - Any similarity measure or term weighting scheme could be used
- Disadvantages
  - Assumption of term independence
  - Long documents might suffer from poor similarity values (the curse of dimensionally)

`scikit-learn` Python implementation

- TF-IDF Vectorizer `scikit-learn` – two options
  - `CountVectorizer` and `TfidfTransformer`
  - `TfidfVectorizer`

CountVectorizer and TfidfTransformer

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
import pandas as pd

# set of documents
train = ['The sky is blue.','The sun is bright.']
test = ['The sun in the sky is bright', 'We can see the shining sun, the bright sun.']

# instantiate the vectorizer object
countvectorizer = CountVectorizer(analyzer= 'word', stop_words='english')

# convert the training set into a matrix. This can be done in one step by using the function .fit_transform()
countvectorizer.fit(train)
training_v = countvectorizer.transform(train)
```

CountVectorizer and TfidfTransformer

```
#retrieve the terms found in the corpora
count_tokens = countvectorizer.get_feature_names_out()


# showing the training term matrix in an organized way by using a data frame
print("Count Vectorizer Training\n")
print(pd.DataFrame(data = training_v.toarray(),index = ['Doc1','Doc2'],columns = count_tokens))


Output:
       blue  bright  sky  sun
Doc1      1       0    1    0
Doc2      0       1    0    1
```

Only term frequency.

CountVectorizer and TfidfTransformer

```
# convert the test set into a matrix based on the training set index terms.
test_v  = countvectorizer.transform(test)


# showing the test term matrix in an organized way by using a data frame
print("Count Vectorizer Test\n")
print(pd.DataFrame(data   =   test_v.toarray(),index   =   ['Doc1','Doc2'],columns   =
count_tokens))


Output:
       blue  bright  sky  sun
Doc1      0       1    1    1
Doc2      0       1    0    2
```

Only term frequency.

CountVectorizer and TfidfTransformer

```
# Transfer the training term matrix of Countvectorizer to tf-idf by using TfidfTransformer


tfidf_training = TfidfTransformer()

tfidf_training.fit(training_v)

tfidf_training_matrix = tfidf_training.transform(training_v)


print(pd.DataFrame(data = tfidf_training_matrix.toarray(),index = ['Doc1','Doc2'],columns = count_tokens))


Output:

          blue    bright       sky       sun

Doc1  0.707107  0.000000  0.707107  0.000000

Doc2  0.000000  0.707107  0.000000  0.707107
```

tf-idf values.

CalPolyPomona

CountVectorizer and TfidfTransformer

```
# Transfer the test term matrix of Countvectorizer to tf-idf by using TfidfTransformer


tfidf_test = TfidfTransformer()

tfidf_test.fit(test_v)

tfidf_test_matrix = tfidf_test.transform(test_v)


print(pd.DataFrame(data = tfidf_test_matrix.toarray(),index = ['Doc1','Doc2'],columns
= count_tokens))
```

Output:

	blue	bright	sky	sun
Doc1	0.0	0.501549	0.704909	0.501549
Doc2	0.0	0.447214	0.000000	0.894427

tf-idf values.

TfidfVectorizer

```
from sklearn.feature_extraction.text import TfidfVectorizer
import pandas as pd


# set of documents
train = ['The sky is blue.','The sun is bright.']
test = ['The sun in the sky is bright', 'We can see the shining sun, the bright sun.']


# instantiate the vectorizer object
tfidfvectorizer  = TfidfVectorizer(analyzer= 'word', stop_words='english')


# convert the training set into a matrix. This can be done in one step by using the function .fit_transform()
tfidfvectorizer.fit(train)


training_v = tfidfvectorizer.transform(train)
```

TfidfVectorizer

```
#retrieve the terms found in the corpora
tfidf_tokens = tfidfvectorizer.get_feature_names_out()


# showing the term matrix in an organized way by using a data frame
print("TD-IDF Vectorizer Training\n")
print(pd.DataFrame(data = training_v.toarray(),index = ['Doc1','Doc2'],columns = tfidf_tokens))


Output:
          blue    bright       sky       sun
Doc1  0.707107  0.000000  0.707107  0.000000
Doc2  0.000000  0.707107  0.000000  0.707107
```

tf-idf values.

TfidfVectorizer

```
# convert the test set into a matrix based on the training set index terms.
test_v  = tfidfvectorizer.transform(test)


# showing the term matrix in an organized way by using a data frame


print("TD-IDF Vectorizer Test\n")
print(pd.DataFrame(data   =   test_v.toarray(),index   =   ['Doc1','Doc2'],columns   =
tfidf_tokens))


Output:
      blue    bright       sky       sun
Doc1   0.0  0.577350   0.57735  0.577350
Doc2   0.0  0.447214   0.00000  0.894427
```

tf-idf values.

cosine_similarity

```
from sklearn.metrics.pairwise import cosine_similarity


X = [1,1]
Y = [2,3]
Z = [5,3]


# calculate cosine similarity between X and Y
cos_sim = cosine_similarity([X, Y])
print(cos_sim)

Output:
[[1.         0.98058068]
 [0.98058068 1.        ]]
```

Symmetric matrix.

cosine_similarity

```
# calculate cosine similarity between X and Z
cos_sim = cosine_similarity([X, Z])
print(cos_sim)

Output:
[[1.        0.9701425]
 [0.9701425 1.       ]]


# calculate the entire cosine similarity matrix among X, Y, and Z
cos_sim = cosine_similarity([X, Y, Z])
print(cos_sim)

Output:
[[1.         0.98058068 0.9701425 ]
 [0.98058068 1.         0.90373784]
 [0.9701425  0.90373784 1.        ]]
```

Advanced Models

- Probabilistic model
  - It tries to calculate the probability of relevancy for each query and document pair.
  - This is often done by consider the odds value, instead of the probability itself.
- Language modeling
  - It is a theoretical framework for doing retrieval and is based on the generative probability of each query given a candidate document.
  - To do so, the divergence between the query language model and the document language model is calculated; these two language models are two probabilistic distributions and thus their divergence can be calculated via KL-divergence formula.

Deciding What to Search

- **Importance of Document Quality:**
  - High-quality documents increase the number of questions a search engine can answer.
  - Poor-quality documents **increase** the burden on the ranking process.

Web Crawler

- A program that downloads web pages automatically, providing a collection for searching.
- Crawler == Spider
  Challenges:
  - The vast size of the web (~5 billion pages).
  - Web pages may only be accessible via forms (deep web).
  - web is not under control of search engine providers
  - Different types of data require specialized crawlers.

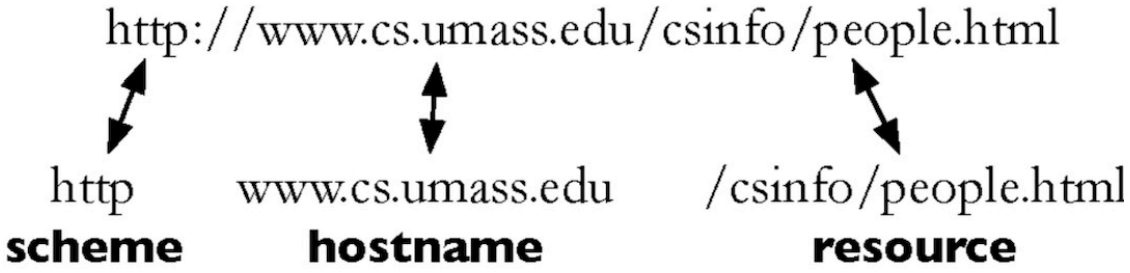
Multiple Webs:

- **Surface Web:**
  The web part that is indexed by standard search engines.
- **Deep Web**:
  - Not indexed by standard search engines.
  - Comprises about 90% of the internet.
  - *Challenges*:
    - Forms submission: Google cannot submit forms
    - Unlinked pages: Google cannot find pages that have not been linked to by a top level domain
    - robots.txt restrictions: is a file that tells search engines crawlers which urls can access/prohibit
  - **Dark Web:**
    - Uses protocols like Tor for anonymity.
    - Requires specific browsers.

----------Retrieving Web Pages ----------

Components of a URL: (Unique resource locator)

- **Scheme:** Indicates how to retrieve the resource (e.g., HTTP, HTTPS).
- **Hostname:** Name of the computer running the web server.
- **Path:** Specifies the location of the page on the server.



Process of Fetching Pages: (for both browsers + crawlers)

1. Client program connect to a DNS server to get the IP address.
2. DNS server translates the hostname into a numeric IP (internet protocol address)
3. Client program connects to the server computer with that IP.
4. Send an HTTP request to fetch the page.
   GET Request - most common HTTP request type:
   `GET /csinfo/people.html HTTP/1.0`
   asks server to send page '/csinfo/people.html ' back to client, using 1.0 version of the HTTP protocol specification.
5. The server sends the page contents back to the client.

> **APIs**
>
> Crawling is an alternative to APIs when APIs have request volume rate limits, irrelevant types of data or inconvenient formatting

Crawling Steps

- **Steps:**
  1. Start with a set of seed URLs (parameters) that are in the URL request queue
  2. Fetch pages from the request queue.
  3. Parse downloaded pages for new URLs.
  4. Add new URLs to the request queue (or frontier).
  5. Continue until no new URLs are found or storage is full.

Efficiency and Politeness:

- Use threads to fetch multiple pages simultaneously.
- Implement politeness policies to avoid flooding servers.
  - e.g., delay between requests
- Use robots.txt to respect server administrators' rules

```
User-agent: *
Disallow: /private/
Disallow: /confidential/
Disallow: /other/
Allow: /other/public/

User-agent: FavoredCrawler
Disallow:

Sitemap: http://mysite.com/sitemap.xml.gz
```

Focused Crawling

- **Objective:**
  - Download pages about a specific topic
    - used by `#vertical_search` applications.
  - Rely on links within topic-related pages.
  - Use ***text classifiers*** to determine relevance.
  - less expensive approach

Deep Web

- Parts of the web not easily accessible to crawlers.
- **Private sites** : require login or are intentionally hidden
- **Form Results**: Accessible only after entering data into forms.
- **Scripted Pages**: Links generated by client-side scripts like JavaScript.

Sitemaps

A challenge with web crawling: site owners cannot adequately tell crawlers about their sites

- **Purpose:**
  - Help crawlers understand site structure.
  - Contain URLs and metadata (modification time, frequency).
  - Provide hints for efficient crawling (when to check a page for changes) .

```
<?xml version="1.0" encoding="UTF-8"?>
<urlset xmlns="http://www.sitemaps.org/schemas/sitemap/0.9">
  <url>
    <loc>http://www.company.com/</loc>          ← URL
    <lastmod>2008-01-15</lastmod>               ← Frequency of changes
    <changefreq>monthly</changefreq>            ← Last time it was changed
    <priority>0.7</priority>                    ← Importance of this page. 0.5
  </url>                                          is the default value.
  <url>
    <loc>http://www.company.com/items?item=truck</loc>
    <changefreq>weekly</changefreq>
  </url>
  <url>
    <loc>http://www.company.com/items?item=bicycle</loc>
    <changefreq>daily</changefreq>
  </url>
</urlset>
```

Distributed Crawling

The use of multiple computers for crawling

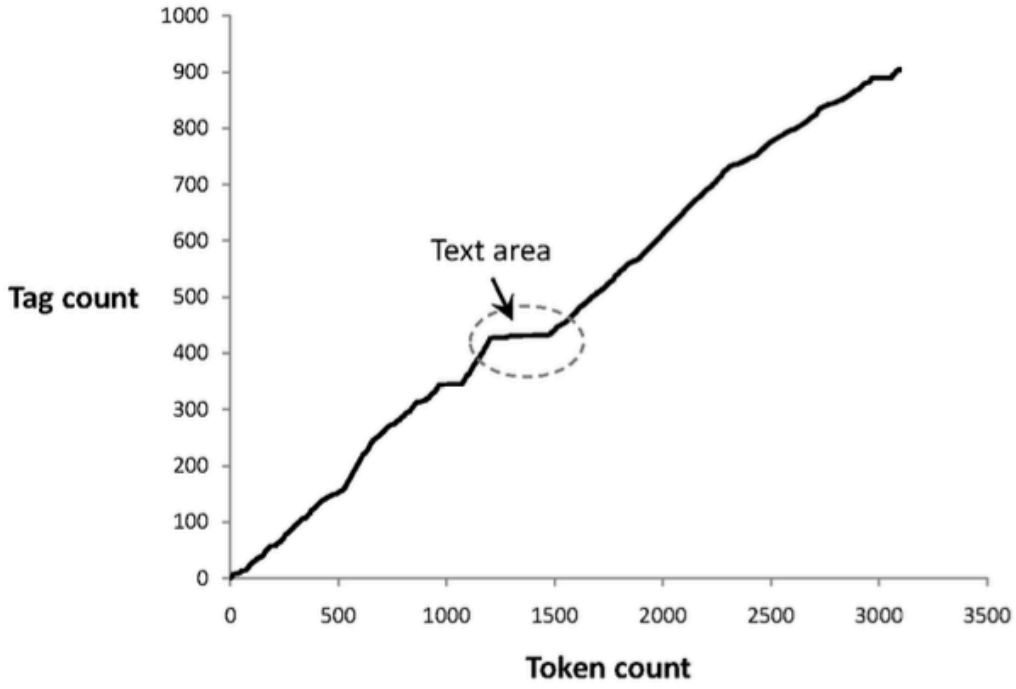
- **Advantages:**
  - Increase throughput and decrease latency by placing crawlers closer to sites.
  - Reduce memory requirements.
  - Distribute computational load.
- **Method:** Use a hash function to assign URLs to specific crawlers.

Storing the Documents

- **Reasons:**
  - Provide efficient access for snippet generation and information extraction.
  - Reduce the need for repeated fetching.
- **Storage Systems:**
  - Customized systems for web search engines.

Removing Noise

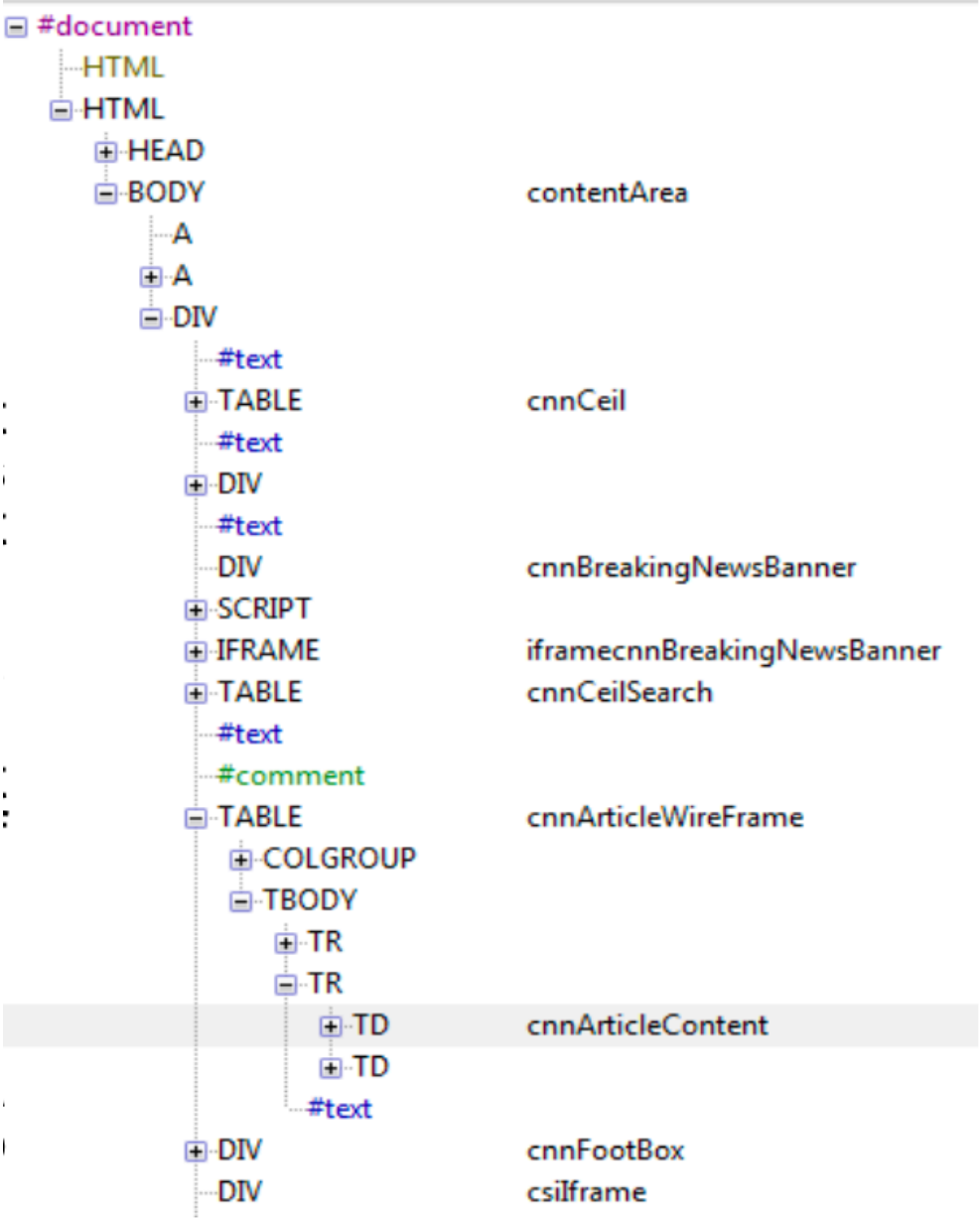
- **Issue:**
  - Web pages often contain unrelated content (e.g., ads, navigation links) which can affect the ranking of the page
- **Techniques:**
  - Detect content blocks to reduce noise.



The 'plateau' corresponds to the main text content of the page.

  - Use the DOM structure and visual features to identify main content.
    - HTML parser interprets structure of the page by using tags and creates a Document Object Model representation

- 'cnnArticleContent' is the part that contains the text of the story



- Algorithms like those by Gupta et al. (2003) and Yu et al. (2003) help in filtering noise.

Building Crawlers with Python

- **Libraries:**
  - `urllib` for requesting data, handling cookies, and modifying headers and your user agent.
- **Fetching Resources:**
  - Use `urlopen` to open and read remote objects.

    ```
    html =
    urlopen('http://pythonscraping.com/pages/page1.html')print(html.
    ```

```
read())
```

Output:

```
Python Scraping book:b'<html>\n<head>\n<title>A Useful Page</title>\n</head>\n<body>\n<h1>An Interesting Title</h1>\n<div>\nLorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.\n</div>\n</body>\n</html>\n'
```

Why is it important to start thinking of these addresses as "files" in addition to "pages"?

Most modern web pages have many resource files associated with them. These could be image files, JavaScript files, CSS files, or any other content that the page you are requesting is linked to. When a web browser hits a tag such as <img src="cuteKitten.jpg">, the browser knows that it needs to make another request to the server to get the data at the file cuteKitten.jpg to fully render the page for the user.

Handling Exceptions:

- Handle `HTTPError` for issues like:
  - "404 Page Not Found" : The page is not found on the server or there was an error in retrieving it
  - "500 Internal Server Error" : The server is not found

```
** Catching the HTTP 500 Error **

from urllib.request import urlopen
from urllib.error import HTTPError
try:
html = urlopen('https://canvas.cpp.edu/courses/74459')
except HTTPError as e:
print(e)
# return null, break, or do some other "Plan B"
else:
# program continues.
```

- Handle `URLError` for server connection issues.
  - Since the server is responsible for returning HTTP status codes, an HTTPError cannot be thrown, which is why URLErrors must be caught!
  - Checks to see if server connection error has occurred:

```
        except HTTPError as e:
                print(e)

        except URLError as e:
                print('The server could not be found!')
```

- **Additional Tools:**
  - **Scrapy:** Framework for web crawling and data extraction.
  - **Selenium:** Tool for automating browsers to scrape dynamic content.

These notes summarize the critical aspects of web crawlers, including their functionality, challenges, and implementation methods using Python.