

Artificial Intelligence Programming *Comments on Project 1*

Chris Brooks

Department of Computer Science
University of San Francisco

9-2: Web crawlers

- A web crawler is a program that crawls the web.
- Starts with a set of root pages.
- Extracts links to other pages.
- Fetch each of those pages.
- Extract their links and repeat.

9-3: Web crawling as search

- We can consider web crawling as a form of search.
- Documents are states.
- By controlling the order in which new pages are explored, we can achieve different search strategies (BFS, DFS, etc.)

9-4: Focused crawlers

- A focused crawler is a crawler that looks for pages that satisfy a particular set of criteria.
- In this project, we'll build a focused crawler.
- Two methods for users to indicate criteria:
 - Keywords
 - Example pages

9-5: The Crawler in a nutshell

```
queue = [rootpage]
while not done :
    dequeue best URL in queue.
    if not visited previously :
        fetch contents of URL
        extract outward links
        score page
    enqueue all outward links according to the page's score.
```

9-6: The Document class

- The Document class will serve as a representation of a Web document.
- Should contain:
 - URL
 - outwardLinks
 - text
 - score
 - title
 - h1 and h2 text

9-7: How to do this?

- You'll want to build a helper class that can parse HTML.
- I recommend subclassing the SGMLParser class.
 - Dive Into Python shows how to do this
 - You may have done this in homework 1.
- This will let you extract anchors, headings, title and text.

9-8: Filtering content

- You'll then remove non-useful content
 - Any words with a non-alphabetic character
 - Any stop words.

9-9: Outward Links

- Extract the URLs from all anchor tags and add them to the list of outward links.
- You may discard links to docs other than text or HTML (can do this either here or later).
- You'll also need to deal with relative URLs.
 - No 'http' or hostname, only a relative path to the document.
 - Before adding them to the list of outward links, fix them to work as standalone URLs.

9-10: Hints and Issues

- You'll need to deal gracefully with malformed HTML.
- You'll need to deal gracefully with remote server timeouts.
- You may need to deal with authentication.
- Remember Python's motto: "Batteries included"

9-11: the FocusedCrawler class

- The FocusedCrawler will manage the process of creating Documents according to their expected value.
 - Much like the 'search()' function in homework 3.
- Maintain a heapq of (score, URL) tuples.
- Dequeue the best URL, check to see if we've visited it before, create a Document, score it, enqueue (score, URL) tuples for its outward Links.

9-12: Closed dictionary

- We'll use a simple closed list (dictionary, actually)
- Keep a dictionary of previously visited URLs.
- Before creating a new document, look to see if that URL has been previously visited.
- What are some limitations of this approach?

9-13: Dealing with robots.txt

- Your crawler should also respect robots.txt
- Before fetching a URL, use the robotparser module to determine if that URL is allowed.

9-14: Scoring a Document

- Now we know how to fetch Documents, process them, and collect their outward links.
- How to determine the 'goodness' of a document?
- Assumption: the estimated score of a document will be determined by the quality of documents that link to it.

9-15: Query-driven search

- To begin, we'll implement a `KeywordQueryScorer`.
- This will allow the user to specify combinations of keywords that should appear in documents that are collected.
- We'll use the following query format:
 - "word1 word2 (word3 word4 word5) word6 word7 (word8 word9) ..."
 - Where combinations inside parens are OR clauses, and other words are ANDed together.

9-16: Query-driven search

- To score a document, count the number of ANDed words that are contained within the document, along with the number of ORed clauses that have any word in the document.
- To normalize this, we then divide by the length of the query, which is the number of ANDed terms plus the number of ORed clauses.
- This gives us a number between 0 and 1.

9-17: Example

- Let's say our document contains 'cat cat dog dog bunny squirrel fish'
- Our query is 'dog (cat snake) bunny lion'
- Our score is $3/4 = 0.75$.
- Note that multiple occurrences of a query term count as one match.

9-18: Searching by topic

- Our KeywordQuery scorer has some weaknesses.
 - Doesn't account for likelihood of words occurring.
 - Doesn't count frequency of a query's occurrence
 - Requires our users to be able to write Boolean queries.

9-19: Searching by topic

- Our user may want to say “find more documents like these!”
- We’ll call a set of documents that are topically related a *similarity set*
- Task: construct a model of these documents and use this to compare to other documents.

9-20: Documents as vectors

- To do this, we'll use a *vector representation* of a document.
- Imagine an n -dimensional vector, where n is all words of interest.
- The value in the n th place is the number of occurrences of word n in that document.
 - Note: implement this using a dictionary, not a list.
- This vector gives us a representation of a document that can be compared to other documents using linear algebra.

9-21: Term frequency

- This count of the number of times each word occurs in a document of interest is called the *term frequency*.
- For the similarity set, you'll want to count each word in every document.
- This is helpful, but not the whole story.
 - We want to score highly those words that occur frequently in the similarity set, but rarely in randomly-chosen documents.
 - This means that we need to estimate how rare each word is.

9-22: Document frequency

- The estimate of how commonly a word occurs in a randomly-chosen page is called the document frequency.
 - Sorry for the confusing terminology - it's the standard usage in information retrieval.
- To address this, construct a *corpus*.
- for each word in the corpus, count the number of documents that it appears in (ignoring multiple appearances).
- This is the document frequency.

9-23: TFIDF

- We can now modify the values for each word in the vector representing our similarity set.
- Give each word a TFIDF score.
- $TFIDF(word) = TF(word) * \log\left(\frac{|corpus|}{DF(word)}\right)$
- If words occur in the similarity set, but not in the corpus, give them a DF of 1.
- We now have a vector containing each word in our similarity set, along with its TFIDF score.

9-24: Including document structure

- We'll give additional weight to words in title, h1, or h2.
- 2x for h2, 3x for h1, 5x for title.
- Intuition: these words were considered important by the document's author.

9-25: Scoring pages

- We can construct a similar vector for each Document we want to score.
 - Get TF for each word in the page.
 - Use the same DFs from the corpus.
 - Build a vector that maps each word in a Document to its TFIDF score.

9-26: Scoring pages

- We can then compare two vectors by measuring the cosine of the angle between them.
 - Identical documents: $\cos = 1$.
 - Completely different: $\cos = 0$.
- Formula:

$$\text{sim}(v1, v2) = \frac{\sum_{(word \text{ in } v1 \cup v2)} v1[word] * v2[word]}{\sqrt{\sum_{(word \text{ in } v1)} v1[word]^2} \sqrt{\sum_{(word \text{ in } v2)} v2[word]^2}} \quad (1)$$

9-27: Scoring pages

- As with the `KeywordQueryScorer`, we can use this to estimate the value of a Document's outward links.
- Enqueue URLs according to this score.

9-28: Hints

- Use dictionaries to implement vectors.
- You only need to compute document frequencies once ever.
- List comprehensions make the computation of cosine similarity much easier.
- The toughest part of the TFIDF scorer is understanding what to do (the actual code is pretty straightforward). This means that you need to schedule time to figure out what you're doing.

9-29: How to do this project

- Start early. Work on this while it's fresh in your mind.
- Read. There is a lot of useful information in the project spec, Dive Into Python, and the python.org website.
- Divide and Conquer. Break each piece into smaller pieces until you find yourself with problems you can solve directly. Start by solving the simple cases, then worry about the exceptions.
- Communicate. You are welcome to use the cs662 mailing list to discuss techniques for solving some of the sticky problems that come up when dealing with real-world problems. You may not share actual code for the project, but you are welcome to share high-level information (“take a look at urllib2”) or sample code that demonstrates a particular module or technique.
- Leave yourself time to prepare decent answers to the written questions.
- Put one foot in front of the other ...