Web Crawling as an AI Project

Christopher H. Brooks
Department of Computer Science
University of San Francisco
2130 Fulton St.
San Francisco, CA 94117-1080
cbrooks@cs.usfca.edu

Abstract

This paper argues for the introduction of real-world programming projects into AI curricula, specifically using Python as an implementation language. We describe a modular set of projects centered around a focused web crawler, along with potential extensions. The author's experiences using this project in a class of undergraduates and Master's students are also discussed.

Introduction

In teaching Artificial Intelligence to undergraduates and Master's students at a liberal arts university for the past five years, I have found that these students have a different set of needs than I had as a Ph.D. student, and that an AI course that successfully meets their learning outcomes should look very different from one designed for first-year Ph.D. students.

In particular, our students need (and are interested in) a great deal more programming and hands-on implementation. As a result, I have developed a number of large, flexible programming projects that I use to help students learn both basic AI principles and also real-world applications of AI techniques.

In this paper, I discuss one such project: a focused web crawler. The crawler begins as a basic exposure to search algorithms and then can be extended in a nummber of directions to include information retrieval, Bayesian Learning, unsupervised learning, natural language processing, and knowledge representation. I begin with a discussion of the needs of the students at the University of San Francisco, and a discussion of Python as a language for teaching AI, and the Web as an application domain. This is followed by a description of the basic web crawler project, along with a set of potential extensions. I conclude with a discussion of the project's strengths and weaknesses.

Course description

At the University of San Francisco, our AI course is taken by two groups of students: seniors who are taking it as an upper-division elective and first-year Master's students who are taking the course as part of their Master's degree. As a result, there are a wide variety of educational goals the course must satisfy:

- It should provide students with an overview of relevant AI techniques and technologies. For almost all students, this class will be their only exposure to AI, so it's important that the class be self-contained and provide real-world, practical examples of how AI techniques can be used in problem-solving. A common challenge in teaching AI is an overreliance on toy problems such as 8-puzzle or Monkey and Bananas; while these are excellent for explaining how an algorithm works, without an introduction of how these techniques are applied to more complex problems, students are often left with the impression that AI has no successful real-world applications.
- It should provide programming experience. For our undergraduates, this class is a chance to further develop their programming ability before graduation. More importantly, it provides a chance for our Master's students to improve their programming skills. We have a large population of international Master's students, many of whom have not had significant programming experience as undergraduates, so it is essential that their early classes provide this. Within our curriculum, the AI class serves the goal of getting students "up to speed" on fairly complex programming assignments.
- It should provide relevant job skills. Nearly all of our students go on to industry (as opposed to academia) after graduation, so we are always conscious of the fact that their educational experience must provide them with marketable skills. Many of our students go on to work in Web-related fields, so providing them with experience processing and classifying Web data potentially makes them more attractive to potential employers.

These needs call for an AI class with strong emphases on on both theory and programming and implementation. *Artificial Intelligence: A Modern Approach (AIMA)* (Russell and Norvig 2002) is an outstanding and comprehensive textbook, used in over 90% of university courses (including mine). AIMA provides a well-written, comprehensive, thorough introduction to AI, but is focused on providing students with a theoretical understanding of AI algorithms and techniques. AIMA co-author Peter Norvig's

Paradigms of AI Programming (Norvig 1991) is an excellent text that describes the issues involved in building non-trivial AI programs, but focuses primarily on historically interesting problems such as Eliza (Weizenbaum 1966) and Mycin (Buchanan and Shortliffe 1984).

I have found that our students learn this material best by doing; much of it is abstract enough, or sufficiently separate from their past experience, that just presenting it in lecture and giving pencil-and-paper homework problems is not sufficient for students to grasp the material. They need to actually implement it in order to truly understand the details. In order to fit the needs of my students, I have supplemented AIMA with self-developed material describing how to apply these techniques to real-world problems.

The class follows a fairly typical structure for an introductory AI course. We begin with a discussion of agency, followed by search, then knowledge representation and logic, followed by probability and Bayesian reasoning. Learning is integrated through the curriculum, rather than treated separately; for example, decision trees are presented immediately after learning about propositional inference as a means of constructing rulesets, and Bayesian learning is presented in conjunction with belief networks. The crawler described in this paper is presented in conjunction with search. Students have approximately two weeks to complete it. Later in the semester, extensions such as Bayesian classification are added in.

Python as an AI Language

Python has received a great deal of attention as an introductory language (Guzdial 2005; Downey 2007); in my experience, it has also proved to be an excellent language for a project-based AI course for several reasons:

- It has a clean, readable syntax that looks very much like pseudocode. This makes the mapping from description to implementation much simpler. It is also easy to learn; students who have used Java, C#, or C++ can pick up the basics in a couple of lectures.
- It is cross-platform, freely available, and easy for students to set up.
- It allows for the ability to use a number of different programming styles, such as imperative, object-oriented, functional, and applicative. It has much of the functionality of Lisp (except for Lisp's macros), such as anonymous and higher-order functions, functions as first-class objects, mapping and list operations, and closures, but with a much shallower learning curve.
- It has extensive libraries for working with networks, HTML, XML, RDF, and other relevant technologies.
- It is a very high-level language, which allows for the rapid prototyping and development of programs. This makes it possible for students to complete relatively complex programs in a week or two.

Web Focus

Many of the projects in my class have some sort of Web-oriented application. The web has proved to be an excellent domain for several reasons: students are familiar with it, there is a huge source of freely available data, and there is a large job market for students who have experience working with Web data. The success of the AI and the Web track at AAAI would indicate that there is a wide variety of research being developed in this area as well.

Wikipedia

I typically have students start out by working with a locally-hosted mirror of Wikipedia. This has several advantages: of reasons:

- Pages are all uniformly constructed, contain valid HTML, and link to other Wikipedia pages. This makes it much easier for students to parse them and extract links, so that they can spend time on AI-related tasks, rather than dealing with malformed URLs or handling parser exceptions.
- Pages have a large amount of text, all in a single language, on a variety of subjects. This makes it appropriate for tasks involving topic detection, search, or natural language.
- Hosting a mirror locally allows students to avoid dealing with DNS problems, 404 errors, or accidentally crawling a remote site that excludes robots and spiders. [1]

Focused Web Crawler

The Web crawler is the first "large" project I give the students. It comes as we are discussing search, and serves several needs: it gives students a hands-on introduction to search algorithms, it provides a real-world application of search, and it also scales up programming difficulty from previous assignments, which are small tasks designed to help them learn Python.

In this project, students implement a focused Web crawler as a way of learning about different search algorithms [2]. This is a program that starts at a given Web page and searches outward from this page, looking for other pages that meet a user's needs or match a search query. In the basic project, students crawl a locally-hosted copy of Wikipedia. To begin, they are asked to build a crawler that can start at a given URL and select a fixed number of pages.

Students are provided with a basic crawler as a starting point. This is a small piece of Python code that can fetch a webpage, instantiate a Wikipage class to hold it, and enqueue each of the URLs (outward links) in the page. The Wikipage class contains code to extract text from HTML and split it into individual words and capture all outward links.

The crawler uses a Scorer object to determine where to place each page in a search queue (implemented as a heap). By using depth in the search tree as a score and using either a min-heap or a max-heap, students can implement breadth-first or depth-first search. One nice feature of this architecture is that students see the separation between the search

[1] Several years ago, USF access to Slashdot was blocked for a couple of days after a student ran his crawler on their site, in violation of their Robot Exclusion policy.

[2] The project description and provided code can be found here: http://www.cs.usfca.edu/∼brooks/F07classes/cs662/assignments/assignment3.html

```
def crawl(self, nGoalpages=10, ntotalPages=500, threshold=0.5,
          startURL, scorer=None) :
    pagesFound = 0
    ntotalPages = 0
    startpage = wikipage.Wikipage(startURL)
    scorer.score(startpage)
    heappush(self.queue, startpage)
    while pagesFound < npages and pagesCrawled < ntotalPages:
        nextpage = heappop(self.queue)
        if nextpage.score > threshold :
            self.pages.append(newpage)
            pagesFound += 1
        for link in nextpage.outwardLinks :
            newpage = wikipage.Wikipage(base+link)
            scorer.score(newpage)
            heappush(self.queue, newpage)
    return self.pages
```

Figure 1: Python code implementing a generic search function for the web crawler.

architecture and the specific algorithm being implemented. While they change the scoring function for each task, the `crawl` method remains unchanged. This illustrates the power of reusable code, and of separating knowledge from algorithm, as is discussed in AIMA.

Students are then asked to implement a closed list by storing a dictionary of previously visited URLs and removing them from all outwardLinks.

Once this basic architecture is built, students focus on the problem of searching for pages that meet a user's needs, through the implementation of a series of progressively complex Scorer classes. The scorer provides a metric of how well a particular page matches a user's information needs. Pages with high scores have their outward links placed to the front of the queue; low-scoring pages have their outward links enqueued at the rear. This provides students with exposure to heuristic search.

The first Scorer the students implement is a keyword scorer: the user provides keywords indicating their search need, and the scorer counts their presence or absence. Pages are scored based on the fraction of keywords they contain; pages with a high score have their outward links placed at the front of the queue. This provides students with a chance to work with Boolean queries, and also an opportunity to discuss the challenge of satisfying a user's needs with a keyword query, given that most keyword searches contain only a few terms.

The students then implement a more complex scorer: the TFIDF scorer. The user provides the crawler with a set of pages that conform to her interests, and the crawler must find similar pages. Each page is treated as a vector, with TFIDF (Baeza-Yates and Ribeiro-Neto 1999) used to compute term weights. Cosine similarity is used to measure the similarity between crawled pages and pages of interest, and provide a score for each page. Students get the opportunity to learn about the idea of documents as vectors, are exposed to basic IR methods for comparing documents, and are forced to think about an alternative modality for meeting a user's information needs: "find me more things like this" as opposed to "find me documents that contain these words."

Later in the semester when we reach probabilistic reasoning, students implement a Bayesian scorer. The user provides a set of "liked" and "not liked" pages, which are used to construct a Naive Bayes classifier. This classifier provides a score for each crawled page, which is its probability of being "liked." The Bayesian scorer also uses a modality in which the user provides a set of documents she likes without explicitly describing what she is looking for in terms of a search query. This provides students with the opportunity to do an experimental comparison: they collect documents with each scorer, examine the top N documents by hand, and determine what fraction of the documents truly met the search criteria. They also examine false positives, and try to identify why those pages were scored incorrectly. This often helps students to identify weaknesses of each algorithm, such as problems with synonymy or polysemy, not working with document structure, and the presence of noise words, and gives them a chance to suggest improvements.

A nice thing about this project is that I can return to it at later points in the semester, or in related classes, and have students extend it in different ways in order to learn about other AI or Web-related topics. Some extensions include:

- Natural Language Processing. Students extend the crawler to use the NLTK toolkit (Loper, Klein, and Bird 2007) to perform chunking on Wikipedia pages, extract noun phrases, and then perform Named Entity Recognition. Students then construct rules that are used to populate a small knowledge base. For example, we often work with The Simpsons as a domain, and students extract knowledge from Wikipedia to automatically learn family relationships between the different characters.
- Document clustering. Students use the crawler to collect a set of Wikipedia pages and then cluster them according to topic, using the vector model described above and a k-nearest neighbor algorithm.
- FOAF search. Students modify the parser to process FOAF (Friend-of-a-Friend) (Brickley and Miller 2006) files. FOAF is an RDF dialect that describes a person and provides references to their friends' FOAF files. Students start with a given FOAF file and crawl outward, building of a networked representation of the relationships in the class. I also ask them to solve "Kevin Bacon" sorts of problems: find the shortest route between two people that involves a third person. The resulting model can then be emitted as RDF and used as input for an ontology editor such as SWOOP (Kalyanpur, Parsia, and Hendler 2005) or Protege (Noy et al. 2001). Students get experience working with RDF, constructing a simple ontology, and working with small-world networks.
- Learning user preferences. Students introduce a feedback loop into the crawler. After collecting a set of documents, the user is given the chance to evaluate each document and rate it as "liked" or "not liked". This data is then fed into a classifer (I typically have students use an off-the-shelf tool such as SVM_{light} (Joachims 1999)) to learn a model of user preferences. This model is then used in the next round of heuristic search; documents that are more likely to be "liked" are placed to the front of the search queue.
- Performance and algorithmic analysis; students are given

a set of pages or topics to discover and measure the performance of different search strategies and scorers. This can be used to measure the number of pages visited and enqueued with depth-first and breadth-first search, as a way to evaluate the precision and recall of different classifiers, and as a way for students to get some exposure to experimental design and hypothesis testing.

- Crawling "in the wild." Students modify their crawler to work on any web pages, rather than just Wikipedia. This produces a host of problems for them to consider, including dealing with robots.txt, handling malformed HTML and Javascript, handling other document types such as Word, PDF and Flash, broken URLs, 404s, spider traps, DNS resolution, working with multiple threads, and detecting non-English pages. This is an excellent extension for students who want experience either with networking protocols or with advanced Python. There are also several AI approaches to detecting language, including Bayesian classification or n-gram based prediction.

Discussion

I've used different versions of the focused web crawler for the past five years in my AI class, with a great deal of success. The project has several features that make it work well as a first large assignment:

- It's a modular project; pieces can be added or removed depending upon the focus of the course in a given year. For example, in Fall 2007, I wanted to spend more time talking about Natural Language Processing, and so I had the students extend the scorer to use an off-the-shelf noun phrase chunker to extract relational information.
- It provides students with exposure to real-world applications of a variety of AI algorithms. They see both how these algorithms can be used to solve complex problems, and also how the "vanilla" versions of these algorithms can struggle when faced with the messiness of real-world datasets.
- It can be a jumping-off point for directed studies or student research projects. I have had students develop more scalable versions of the crawler for processing online datasets (Brooks et al. 2006; Brooks and Montanez 2006) and for constructing personalized assistants for discovering research papers (Brooks et al. 2007).
- It forces students to think carefully about the details of these techniques and how they should be implemented. For example, why is a heap more efficient than a list? Why is a dictionary more useful than a list for representing a closed list? The project is just large enough that poor implementation choices will lead to long running times. This also provides an opportunity to talk about profiling, unit testing, and similar topics normally treated in isolation in a software engineering class.

One weakness of the project is that it's not clear how to integrate A* search; it's not obvious how to construct an admissible (and interesting) heuristic in this setting. This can be a challenge for instructors using AIMA, as A* is typically the topic covered directly after uninformed search. I typically give a small assignment where students implement the basic uninformed and heuristic search algorithms on a toy problem such as fox-and-chickens, and then follow this with the crawler.

Conclusion

This paper has presented an argument for the incorporation of programming projects with a real-world focus into an introductory AI class, the advantages of Python as an AI programming language, and a detailed description of a modular focused web crawler project.

Further work includes the continued development of the crawler, along with the further development of other course projects in progress, including a Bayesian spam classifier, tools for unsupervised clustering of email, a genetic algorithm for solving constraint and scheduling problems, and an agent that uses reinforcement learning to help place products in an e-commerce setting.

References

Baeza-Yates, R., and Ribeiro-Neto, B. 1999. *Modern Information Retrieval*. New York: Addison-Wesley.

Brickley, D., and Miller, L. 2006. FOAF Vocabulary Specification. XMLNS Specification. Available at: http://xmlns.com/foaf/0.1/.

Brooks, C. H., and Montanez, N. 2006. Improved Annotation of the Blogosphere via Autotagging and Hierarchical Clustering. In *Proceedings of the 15th International World Wide Web Conference (WWW-2006)*. Edinburgh, UK: ACM Press.

Brooks, C. H.; Agarwal, M.; Endo, J.; King, R.; Montanez, N.; and Stevens, R. 2006. Slashpack: An Integrated Toolkit for Gathering and Filterng Hypertext Data. In *Proceedings of the 21st National Conference on Artificial Intelligence (AAAI-06)*.

Brooks, C. H.; Fang, Y.; Joshi, K.; Okai, P.; and Zhou, X. 2007. Citepack: An Autonomous Agent for Discovering and Integrating Research Sources. In *Proceedings of the 2007 AAAI Workshop on Information Integration on the Web*. Vancouver: AAAI Press.

Buchanan, B. G., and Shortliffe, E. H., eds. 1984. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. AAAI Press. Available at http://www.aaaipress.org/Classic/Buchanan/buchanan.html.

Downey, A. 2007. *How To Think Like a (Python) Programmer*. Green Tea Press. Available at: http://www.greenteapress.com/thinkpython/.

Guzdial, M. 2005. *Introduction to Computing and Programming in Python, A Multimedia Approach*. Prentice Hall.

Joachims, T. 1999. Making Large-scale SVM Learning Practical. In Schlkopf, B.; Burges, C.; and Smola, A., eds., *Advances in Kernel Methods - Support Vector Learning*. MIT Press.

Kalyanpur, A.; Parsia, B.; and Hendler, J. 2005. A Tool for Working with Web Ontologies. *Proceedings of the International Journal on Semantic Web and Information Systems* 1(1).

Loper, E.; Klein, E.; and Bird, S. 2007. *Introduction to Natural Language Processing*. self-published. See also: http://nltk.sourceforge.net/index.php/Book.

Norvig, P. 1991. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kauffman.

Noy, N. F.; Sintek, M.; Decker, S.; Crubezy, M.; Fergerson, R. W.; and Musen, M. A. 2001. Creating Semantic Web Contents with Protege-2000. *IEEE Intelligent Systems* 16(2):60–71.

Russell, S., and Norvig, P. 2002. *Artifical Intelligence: A Modern Approach*. Prentice Hall, 2nd edition.

Weizenbaum, J. 1966. ELIZA–A Computer Program For the Study of Natural Language Communication Between Man and Machine. *Communications of the ACM* 9(1).