Project Report (MTH 524)

Automated Information Extraction From Web Pages

Ripu Singla Y6387 (ripu@iitk.ac.in)

18^{th} **April, 2011**

I consider the problem of automated information extraction from a previously unseen web page. In this project, I restrict myself to extract names, designation and company names etc., of people on a conference website or a similar website hosting sufficient structured data. However, the code I have written can be easily adjusted to extract any other structured information (not just names etc.) by just changing the first step. I would like to mention here that I have not used any information or priority ordering of html tags. The only assumption I make is that there is lot of data structured in a similar manner in the given web page.

1 Approach

I use a several step process to achieve the above stated goal. I have used Python programming language and other open source libraries to implement the algorithm. I used Tornado web server to deploy the code for demonstration purpose and made an interface that allow us to see the various steps of execution for easy understanding and debugging. Total code is about 400 lines. The first step is to download the source of html page for which I use urllib.

1.1 Lynx Command Line Browser

Lynx is an excellent command line browser and another intelligent use of lynx is that it can be used to remove all the html tags and get hold of raw text. Then I use some post processing steps to clean up the text i.e., remove quotes, stars and another junk using python RE.

1.2 NLTK Library

NLTK stands for *Natural Language Toolkit* which is a open source project for natural language processing. I use nltk library to extract all the proper nouns from raw text. I use *word tokenizer* and *pos tagger* to do above task. Most of the time taken by my code is due to this library, I have tried to find an alternative to nltk but I haven't come across any yet.

1.3 lxml Library

Then I use open source lxml XML toolkit which is a pythonic bindings for C libraries libxml2 and libxslt to find out the xpaths for the nouns extracted in above step. From now on, I will call this collection of xpaths as *xpathlist*. Xpath is the address of a given node in an html page (for example, /html/body/div/p[1]/b).

1.4 Removing Indices

Now the idea is that if a web page is sufficiently structured, we can find a pattern in the xpaths extracted above. There are two ways to extract the pattern, one involves removing the nodes one by one starting from the end of the xpath. For example, removing 'b' from the example xpath given above. Second way is to remove the index from the end of xpath. For example, removing '[1]' from 'p[1]' in the above example xpath. After a lot of experimenting, I decided to use the second way of extracting the pattern. Second way captures the fact that in general when there is a lot of structured data, it will follow a sequence of indexing independent of the node i.e., div or tr,td or any other node. If we apply this process to all the xpaths in xpathlist, then because of the assumed structure of web page, a lot of xpaths will reduce to a common xpath giving a pattern. Now it depends on the structure of web page that if we will get our pattern after a single iteration of removing indices or multiple iteration. After each iteration, I analyse the output using the resulted pattern and decide on whether to stop or not.

1.5 Core XPath

After removing the indices once or multiple times, We get a new list of xpaths. Then we do a count of various xpaths in this list. Now the most important xpath is probably the xpath with the highest count (let us call it xpath*) because most of xpaths reduced to this xpath*. So probably most of information we are after, hangs from this xpath*. If count of xpath* is sufficiently higher than other counts, above assumption is reasonable. However, if second or third highest counts are comparable to first highest. Then it may be the case that all of these xpaths are pointing to the same dataset. An example is, suppose first highest is '/html/body/div/tr/td/a', the second highest is '/html/body/div/tr/td/b' and third highest is '/html/body/div/tr/td/c', all with comparable counts. In this case, it is clear that actual data hangs from '/html/body/div/tr/td' and all of these xpaths are a particular case of a more general xpath. Basically we need to apply the first way of extracting the pattern from last section. After lot of experimenting with real data, I have devised the following algorithm:

- if second highest is less than 0.6 (experimental constant) times the first highest, we simply accept the first highest after removing the last node as our core xpath.
- if not, we remove last nodes from both the xpaths and see if second one is more general than the first i.e., if second includes the data represented by first. if yes, then we accept the second highest as our core xpath.

1.6 Backtracking

Although we have a core xpath in hand from which all the information hangs, but we do not want to extract the information as a big blurb and neither we want the junk information which may be included in core xpath. We want the structured information which can be further used by some automated scripts if required. So we apply the backtracking, it involves finding the xpaths in our

original xpathlist that contributed in generating this core xpath. At this stage, we know how much iterations we used for removing indices, using that information we map (meaning that we apply backtracking as given above) the xpaths in xpathlist to the core xpath and generate a new list of xpaths (we call it ypathlist). Using, ypathlist we can find unique information above individuals rather than a big blurb. We again use lxml library to get text from xpaths.

1.7 Partitioning Data

Now ypathlist gives us the xpaths for unique information. But ypathlist may contain different xpaths for the information related to a single individual. We would like to findout what information belongs to a single person and merge that together. To accomplish that we would like to find pattern in ypathlist, i.e. which of the xpaths in this list belong to a single individual. We first sort the ypathlist and then by comparing the various xpaths in ypathlist, we calculate a number (say patterncount) which gives us how many xpaths represent information about single person and a list (say taillist) which tell us how these xpaths differ from one another in a single group. So in a way it generates a partition. For example, suppose the length of ypathlist is 9, then the patterncount may be 3 and the taillist may look like ['node a','node b','node c']. So there is information about 3 individuals. As we are dealing with real data, so there may be a lot of noise in ypathlist. Hence we calculate the patterncount 5 times (another experimental constant) and take the highest count. We also apply the condition that patterncount should not be more than 5 to avoid separate data mapped to one individual.

1.8 Output

Using ypathlist, patterncount and taillist, we extract the required information using lxml library. Because there may be some xpaths missing due to noise in real life data, we do not simply extract the text for xpaths in ypathlist. We use the patterncount and taillist to construct the missing xpaths and extract data which may not be present in original xpathlist. One reason of missing data may be that nltk library is not 100% accurate and some nouns may have been missed. Sometimes, we may get xpaths that points to paragraphs containing a lot of text, to avoid outputting paragraphs we use a condition that if text is more than length of 250 characters, we assume it is paragraph and we skip it.

1.9 Stopping Criterion

For stopping criterion, we apply the following logic. We are removing indices one by one from end of the xpaths in xpathlist. If we remove more indices than required, we will generate a more general xpath than required which will produce a ypathlist having no definite pattern because different information xpaths are intersecting with each other. For example, suppose the ypathlist looks like following:

- /html/body/div/tr[1]/td/a

- /html/body/div/tr[1]/td/b
- /html/body/div/tr[2]/td/strong
- /html/body/div/tr[3]/td/a
- /html/body/div/tr[3]/td/b

and so on. So in that case patterncount and taillist will come out to be wrong and will try to construct nodes that do not exist or in other language do not have any text associated with them. So the text in output will decrease and that is precisely my stopping criterion. Formally, with each iteration we will count words in output, if count is increasing we will keep increasing the iterations, if count becomes constant or start decreasing we will stop and revert to previous iteration.

2 Results

I have tested the program for many different websites, the results are good as long as site is sufficiently structured. It even works for university department websites which have relatively less data and less structure. Today most of the websites are constructed using softwares which preserve the structure and hence this program is very effective. Another thing is that we can extract any other information by using other markers than nouns, the rest of the program will still work as it doesn't assume any thing else than relative dominant structure.

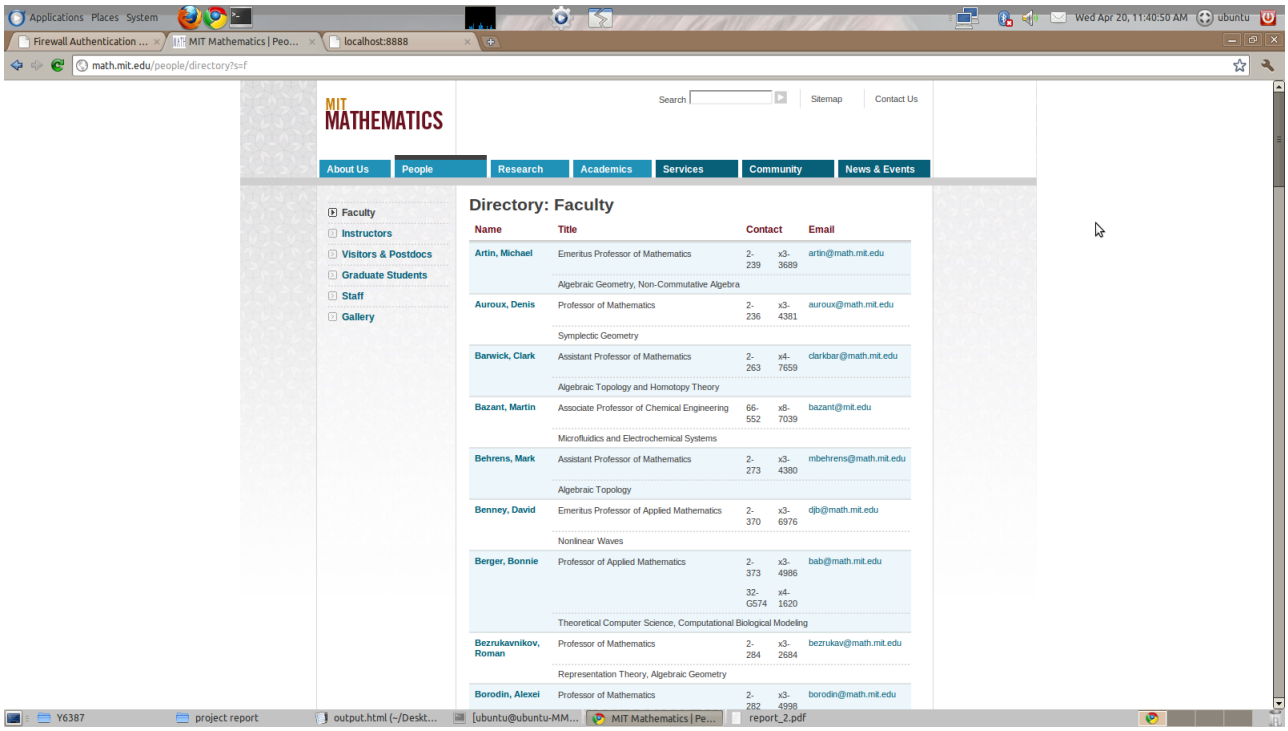

Figure 1: An example webpage

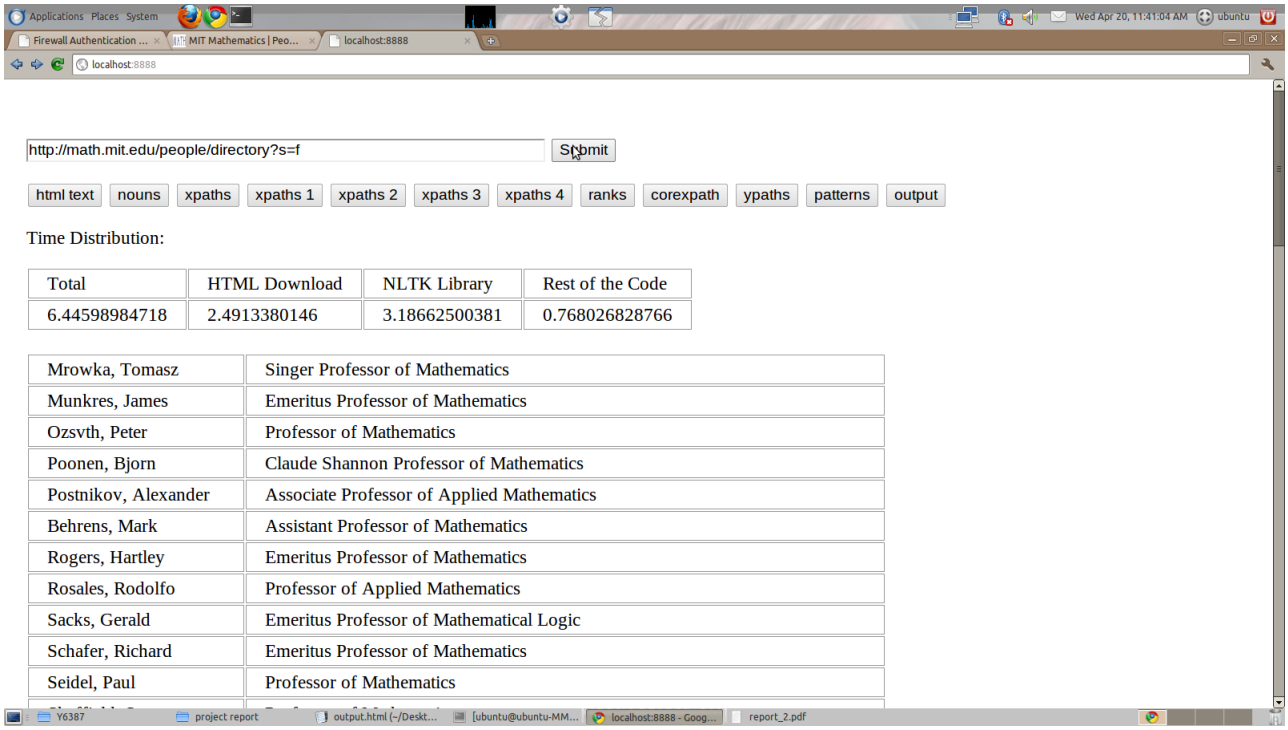

Figure 2: Program output