

Improving Search Relevancy through Human Indexing and Data Mining

<http://www.adiunnithan.com/flashcard/>

2006-2007 Honors Project

Adi Unnithan, Advisor: Dr. Diane Pozefsky

Abstract

Popular search engines today index pages on a defined heuristic, such as the number of links to a page. We can improve upon this measure and display more relevant results by utilizing social networks and their repositories of information and applying data mining techniques to them, such as association rule finding and clustering. In addition, we can improve the searching experience by providing non-linearity in the user interface.

Background

In the past decade and a half, we have witnessed the acceleration of both the supply and demand of information on the internet. However, the main problems with the World Wide Web today are that (1) although it is fairly easy to add new content to the web, there is no organization and (2) there is hardly any good meta-information that we can efficiently search against to discover information that is useful to us. The former problem is simply a result of the web's design and the latter problem is an issue that is being addressed today through popular search engines such as Google [<http://www.google.com>] and the intelligent extrapolation of meta-information using techniques like indexing. Today Google has indexed over 8 billion web pages. Search engine success is judged by three key facets: the speed, the simplicity, and the precision. We use precision (as opposed to accuracy) here mainly to point out one flaw with Google.

The interface to Google is one simple text box. It requires a user to take all of their ideas or all of their questions and attempt to trim them down into a set of carefully-placed keywords possibly surrounded by Boolean logic. There is no intelligence in the search engine to distinguish whether, when we search for "banana pancakes", to look for recipes for banana pancakes or stores that sell banana pancakes or what foods banana pancakes go well with. In addition, the search engine itself is excruciatingly frustrating when we search for something and fail. We have to try again and attempt to search against a different set of keywords. Today there have been some notable attempts at solving this problem, particularly through clustered searching, popularized by Vivisimo [<http://vivisimo.com/>]. While we may not have an exact idea of what we want, the search engine can provide us with options to help us trim our ideas down and give us relevant results, instead of simply presenting us with a blank textbox. Another idea that has been making the rounds on the internet has been heavily promoted by the World Wide Web's original creator, Tim Berners-Lee. This idea is the Semantic Web (Herman, 2007): a project that essentially "gives" metadata to web pages and allows them to be searched more efficiently.

Another issue that complicates search engines today is the notion of the Deep Web. The Deep Web consists of web pages that exist behind login-based systems, are driven by private databases, or do not allow indexing through search engines. Some approximate the Deep Web to be 5 to 500 times larger than the Shallow Web (Ratzen, 2006).

Proposal

We are proposing a project that combines the benefits of the semantic web as well as the guided relevancy process of clustered search engines by leveraging repositories that contain *human-indexed* pages. The basic idea is that if we already have a repository of useful, relevant web pages that are, for example, tagged by humans, we can easily and semantically understand the page through the tags and use them in searching by building query-tag relationships.

Moreover, we can obtain a guided relevancy through another recent paradigm: social networking. There have been many instances where we often cannot find something, yet one of our friends or family members do. This doesn't help us at all unless there is a way in which our searching system could understand what we were looking for and realize that our friend Kevin had looked for that too and found it. Not only would we cut down on search time, but we could drastically improve our relevancy. We could even filter these results more by associating a "profile" to every user. For example, more often than not, two computer science teachers might be looking for the same types of things. By allowing them to communicate the successes and failures of their searches, we can dramatically improve our search behavior.

The key is figuring out how to realize that what we are searching for is similar to what Kevin is searching for. This is the other facet of tags. Suppose that when we search for "banana pancakes", we inherently get a set of tags back via a series of primitive or complex algorithms: pancakes, bananas. In searching, the tags should only be visible to the backend of the system. The user only cares about the query and the results. These are relevant tags, and they were merely extracted from the query itself. Now, since we have a general "tag space" of all the tags in the system, we can utilize another series of algorithms to discover similar tags. Eventually we find the tag we are looking for: "recipes". We notice that Kevin had a very similar query-tag relationship to ours and looked for banana pancake recipes. He found a page that had a "recipes" tag on it for banana pancakes. The system can then suggest this page to us. If another user is searching for nutrition and comes across the recipes tag, we would not display this page, since the "query" part of the query-tag relationship does not match. This is a simple example but it displays in its most trivial form the problem with trying to figure out what a user wants to search for and not being able to find it given the vocabulary they utilize.

Finally, we propose a novel search interface that allows the user to view related concepts to their original query, as well as "turn off" the influence of related concepts on the resulting pages.

Theoretical Implementation

There are other directions to take this project. In order to understand if Kevin actually found a page successful, we could monitor the path he takes across the web and find the "leaf" of the final branch, but we are still not sure if this was a useful page or if he simply gave up. The most straightforward way to achieve this is to provide a simple yes/no, non-intrusive box. This can essentially act as an experience metric for a machine learning system. When a user simply clicks on a link in the search results, this inherently means that the link title peaked the user's interest or curiosity.

The final project involves a weighting system that determines related concepts to a user's original query as well as a naïve machine learning system that improves the relevancy of results as users click on search result links. The implementation is described as follows.

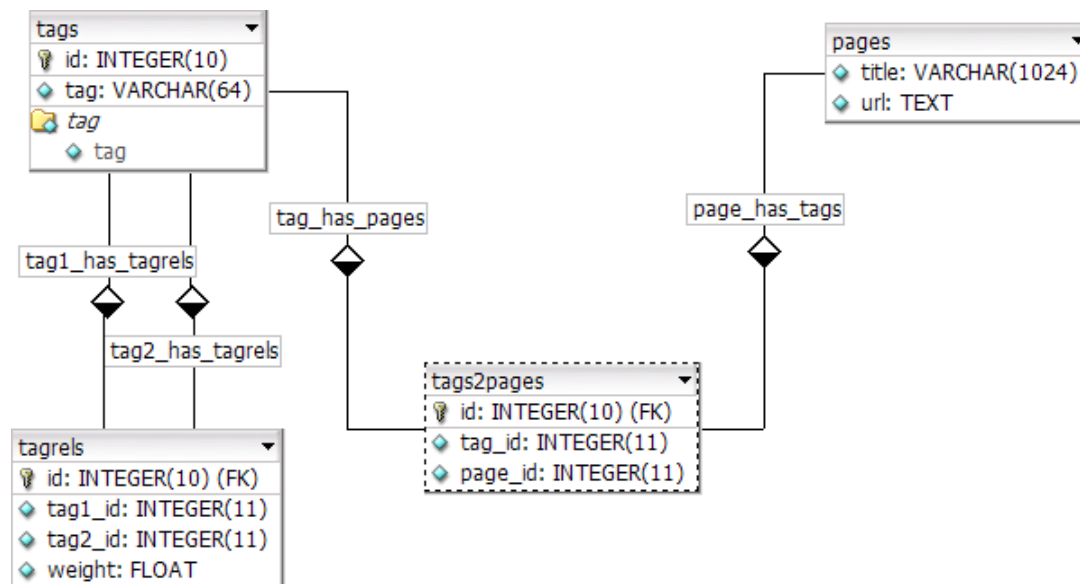
The aforementioned repository of tagged web pages does indeed exist and is very popular today. It is known as del.icio.us. [<http://del.icio.us>]. Users bookmark pages as they travel the web with sites that they find interesting and useful to them. When they bookmark these pages, they also tag them and

possibly provide notes about them. Del.icio.us provides various methods that developers can use to gain access to bookmarks and their tags. The repository does provide a standard API, but this requires user authentication – which is useless for an anonymous search engine. Two of the other access methods – JSON [<http://www.json.org>] and HTML feeds – also require authentication, or only viewing a single user's tags/bookmarks. The final access method is through RSS feeds – these are accessed in much the same way as an anonymous user can view all bookmarks of a certain tag on the web pages in del.icio.us.

When a user performs a search, the following steps are performed by the search engine, as described through a general user scenario:

1. User types in a query; ideally this would consist of a question.
2. Parse the query and create a “structured query”, which just consists of the useful information in the original query (see enhancements) and ignoring “noise”, such as articles.
3. Discover tags from the structured query. For the purposes of understanding, we will call these “contexts”, since they will aid in narrowing the user's context
4. Search the local database to determine if we have enough data.
 - a. If we do not, then leverage a tagging service (del.icio.us) to find related tags and push them, along with their corresponding pages, into a local repository.
 - b. If we are synchronizing with the tagging service to find new tags, upgrade a weight for each tag relationship (tag to tag) that corresponds to how powerful that relationship is.
5. Pull pages from the local database that have the most number of related tags, put priority on those pages that display a combination of the highest tags, and then display the pages in the order of their tag relationship weight.

Data Model



In this data model we obviously have to store tags. A tag is nothing more than a unique identifier and a word. The other major component is a page. We store the name of the page as well as the url. There is a many-to-many relationship between tags and pages; this relationship is contained by a pair table known as 'tags2pages'. In this first iteration we also store the tag relationships through a 'tagrels' table, which defines a relationship between two tags and the weight – an integer that defines how strong that relationship is. The separation of the data model allows us to update the name of a tag or page without affecting each of these parts directly. It also allows us to modify the weight of a relationship without changing anything in the table that holds the tags.

Technical Implementation

The implementation of the process described above is a PHP-based web application [<http://www.php.net>]. Since we are analyzing the web, it would be natural to be able to implement the application on the web. There are a number of factors that play into this: development ease-of-use, ability to iterate/follow an agile model of development, and testing (since we can easily allow a random sampling of users to test this). Ruby on Rails [<http://www.rubyonrails.com>] was attempted, but it soon became complex as the data queries became more complex and configuration was required. Inevitably the application did not abide by the ActiveRecord Rails design pattern, and its use was pointless and complex. PHP5 was chosen for its large support and enterprise scalability, as well as its object-oriented nature. It also provides separation of business logic and presentation logic through the Smarty templating engine [<http://smarty.php.net>] as well as excellent connectivity options to the database engine, MySQL [<http://www.mysql.com>].

A Model-View-Controller design pattern was used so that we could concentrate on the business logic and not have to control the changes to the view (the html/css/javascript) and the model (the objects/code that represent the MySQL database). The view was controlled through the Smarty framework and we simply had to change the template residing in the code. The model leveraged a PHP database abstraction package known as ADODB [<http://adodb.sourceforge.net>]; however, in order to maintain simplicity, readability, and decoupling in the controller layer, we implemented a simple ActiveRecord design pattern in the model layer.

We will now step through the logical process and detail the implementation of the process described above. Every **bold section** identifies a point at which the “weight” of a tag relationship is upgraded. Every *italicized phrase* identifies an important term:

1. User types in a structured query, currently consisting of a set of *contexts*, into index.php
2. We bypass this step.
3. Extrapolate the *contexts* from the structured query.
4. Search.php is called; it iterates through the *contexts* in the query. If a tag does not exist, or if there are less than 10 tag relationships that contain the tag in the local database, we will add the tag to the table and “sync” with del.icio.us.
 - a. The syncing with del.icio.us is performed through RSS parsing. We use the MagpieRSS PHP package to transform the XML into an associative array. For the del.icio.us RSS feed for each tag, for each item in the feed, there exists a dc:subject element, which consists of space-delimited tags, for example:

```
<item rdf:about="http://www.aharef.info/static/htmlgraph/">
```

```

<title>Websites as graphs - an HTML DOM Visualizer</title>
<link>http://www.aharef.info/static/htmlgraph/</link>
<dc:creator>vorkronor</dc:creator>
<dc:date>2007-04-26T15:38:37Z</dc:date>
<dc:subject>
    applet html processing visualization service css
</dc:subject>
</item>

<item rdf:about="http://www.456bereastreet.com/">
    <title>Prevent HTML tables from becoming too wide</title>
    <link>http://www.456bereastreet.com/</link>
    <dc:creator>slakistan</dc:creator>
    <dc:date>2007-04-27T03:59:57Z</dc:date>
    <dc:subject>css html layout tables tips</dc:subject>
</item>

```

We aggregate all of the tags in the dc:subject element as well as the <title> element into an array; in this previous example the result would be: [applet, html, processing, visualization, service, css, websites, graphs, html, dom, visualizer, css, html, layout, tables, tips, html, tables].

- b. Flatten the array – this is done by checking for tags that exist multiple times, storing only one array element for that tag, and then **incrementing the default weight (1) by 1 for each time** we find a duplicate tag. Each tag in this array can be called a *related tag*. In the previous example, the result would be: [Applet => 1, Html => 4, Processing => 1, Visualization => 1, Service => 1, Css => 2, Websites => 1, Graphs => 1, Dom => 1, Visualizer => 1, Layout => 1, Tables => 2, Tips => 1]
- c. If we have to perform a *sync* with the server, then after a sync, we will “filter” a group out of the associative array, in which we add every related tag into the local database, create a tag relationship object, and create an associative array of TagRel objects, known as a TagRelGroup. We then “save” the array of these objects – if the tag relationship already exists in our local database then we will **upgrade the weight of the tag relationship**.
- d. Also, if we have to perform a sync, then we will grab the pages that have the original tag and store them in the database as well.
- e. After syncing, regardless of whether the application needed to sync or not, it will grab all the tag relationships from the database where tag1 is the *context* or tag2 is the *context*. The application then constructs a related tags array where each bucket is in the form [relatedTag => weight]
- f. For each *context*, we push this related tags array as a bucket into a 2-dimensional array of related tag arrays.
- g. Loop through all the tags in each array of the related tags array and push each tag into a new associative array (*weighted related tags array*). As we loop through each tag, if the tag exists in *weighted related tags array* while in the process of populating it, then **add the weight to the current weight of the related tag**. For example, if we determined that “html” has the following flattened related tags array: [webdesign => 2, css => 2, tables =>

1]. We also assume that “reference” has the following flattened related tags array: [tutorials => 3, guide => 2, tables => 2]. Therefore, the final flattened related tags will contain the same tags, except “tables” will have a tag weight of 3. Finally, sort the associative array by weight from highest to lowest.

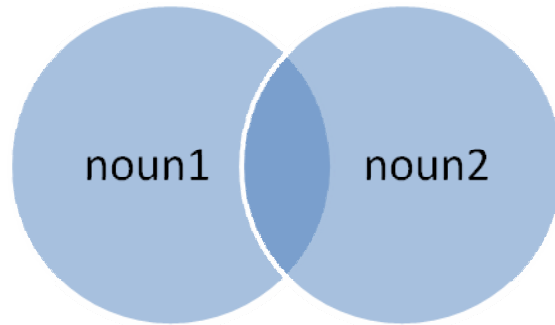
- h. If the *context* doesn’t have the highest weight after we merge all the tags, slip in the original tag into the related tags array and **set the original tag’s weight to one that is 1.25 times higher than the max weight of the array.**
5. We have a constant, `NUMBER_OF_RELATED_TAGS`, which defines how many related tags to display on the results page and consequently how many related tags we should use to pull pages from. So for each x (where $x = \text{NUMBER_OF_RELATED_TAGS}$) amount of related tags, the application will pull pages from the database that have that related tag as one of their tags (as per an SQL INNER JOIN on the `tags2pages` table, performed by `PageModel::findByTag($tagname)`) into an array.
- a. The array has a lot of duplicate pages that are associated with multiple tags, so we have to flatten this page array – for each of the pages, read the weight of each tag and add it to compile a final weight.

The application can improve itself as well, strengthening weights over time as users click on subsequent links. The theory draws upon a basic technique of machine learning, wherein we improve *performance* at a *task* following a certain *experience*. When a user clicks a link in the search results, a SQL query upgrades the relationship between all original tags and all the tags on that page by $1.25 \times$ the current weight. Unlike the determination of page results, this is simply a static number but it preserves fairness; the weight is upgraded by 25%, so a relationship that isn’t so strong will get upgraded a little bit, but if more users click on it then it will quickly accelerate in strength.

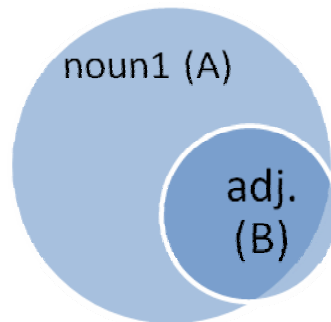
Barriers to Relevancy

Once an initial algorithm was implemented, there were a few interesting observations that eventually led to important tweaks in the implementation. Often times, the weight of a relationship can describe a lot regarding the semantic relationship between two tags. Here are some observations:

1. If a tag is inherently broad, there is a quick dropoff in the weight of the tag. For example, “syndrome” often shows up in the final related tags array along with several actual names of syndromes. However, the weight of “syndrome” will be much higher than the other tags, mainly because “syndrome” will show up in every page that contains any of the other actual names of syndromes. This is not necessarily bad – a user can give a modifier to “syndrome”, such as “rett syndrome” or “bowel syndrome” and any irrelevant syndromes will automatically become suppressed in the page results.
 - a. The problem inherently stems from being unable to identify parts of speech. Pages containing noun/noun pairs and noun/adjective pairs have varying overlappings. For example, if we have pages resulting from two nouns, the overlapping of pages that contain both can be visually described as follows:



For a noun and an adjective, the overlapping can be described as follows:



Based on these two hypotheses, we can construct a logical statement: if $\text{size}(A \& B) \gg \text{size}(!A \& B)$ then put a priority on pages that have “B”. In evaluating this statement, we eventually find that if A’s page-space is much greater than B’s, then we have to “punish” the weight of A.

2. According to the initial implementation, a page that had a majority of the contexts and the highest-weighted related tags would be pushed to the top of the page results. However, in del.icio.us, some pages have lots of tags associated with them. In some cases, many of these tags aren’t related to each other (which makes it obviously irrelevant to any result) and in other cases the tags are related (this can still be considered irrelevant because this page does not provide content that is specific to whatever the user was originally asking for). Regardless, an early iteration of the application previously did not take into account the number of tags. Two ideas were tested and the results were compared.

- a. The problem occurs when flattening the items in the page array; the naïve method is to simply add the previous weight to the new weight when we come across a duplicate page. The next obvious step would be to set the weight of each page as an average – the sum of all the weights (where “all the weights” means all the weights associated with each duplicate page) divided by the number of tags on that page. So when we come across a duplicate page, the calculation would be:

$$\text{weight}_{\text{curr}} = \frac{(\text{weight}_{\text{curr}} * \text{numtags}) + \text{weight}_{\text{new}}}{\text{numtags}}$$

The results of this tweak were dramatic. Figure 1 shows the previous technique; the pages at the top of the results had around 15 to 20 tags. Figure 2 demonstrates the newer technique, where much more relevant pages are pushed to the top and pages with lots of tags are punished by being pulled to the bottom.

- b. Although this produced drastically better results, there were still some issues. Sometimes a page in the results had two original tags yet it is still buried at the bottom of the results simply because of the fact that we divided by the number of total tags on that page. In addition, another consequence of this previous tweak was that most of the results at the

top had only one of the original tags and *no other tags*. The highest ranked results were not very trustworthy. One idea to avoid this was to increase the amount that the weight gets increased when we find a duplicate tag, such that there is more “weight distance” between each related tag. However, this still prevented that page from reaching the top of the results. Let us assume that our structured query is ‘html webdesign’. Let page A have a tag ‘html’ and page B have the two tags ‘html’ and ‘webdesign’. After our related tags extrapolation, we find that both ‘html’ and ‘webdesign’ have the same weight, “41”. However, page A and page B will have the same weight. Page A’s weight will be $41 / 1 \text{ tag} = 41$. Page B’s weight will be $41 * 2 / 2 = 41$. Essentially, we want the fact that there are two original tags in the query to have much more priority than the fact that page B has more tags than page A.

- c. A re-evaluation of the goals resulted in a much better, final technique. Based on the previous example, we can state that a page with one tag should have the same punishment (almost none, in fact) as a page with two tags. However, as the number of tags on a page grows, we should have a sort of exponentially growing punishment. Therefore, as opposed to the previous case in which the punishment was the number of tags (we divided the total weight by it), now we can state that the punishment should be an exponential function. A function was in fact extrapolated based on the fact that if a page has five tags, it should still be given no punishment. After that, the exponential function starts to take effect; when there are 8 tags on a page, we want there to be a punishment of ‘2’ – take the total weight and divide it by 2. When there are 10 tags on a page, we want there to be a punishment of ‘9’. The punishment was now not just ‘numtags’, but:

$$\text{punishment} = 2.5^{(\text{numtags} - 5)} + 1$$

The difference can be seen in figures 3 and 4.

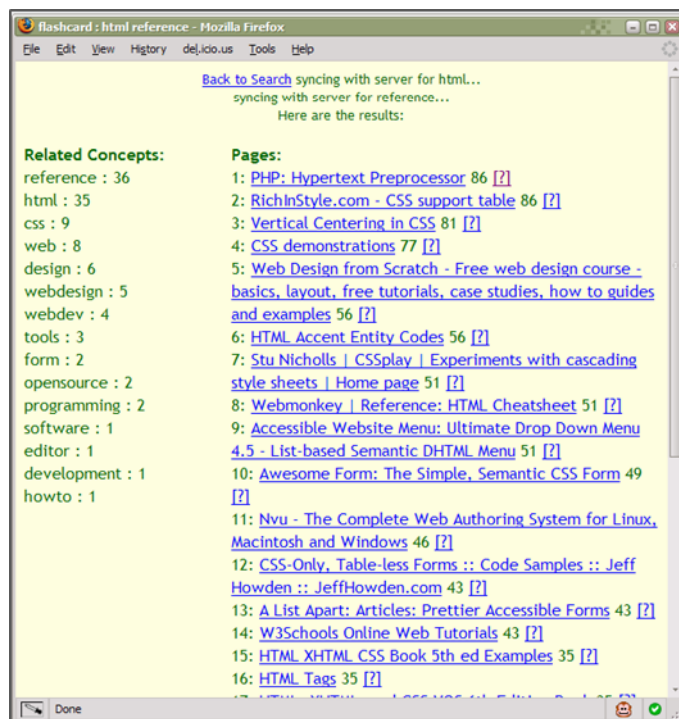


Figure 1: Naive page flattening with no tag count

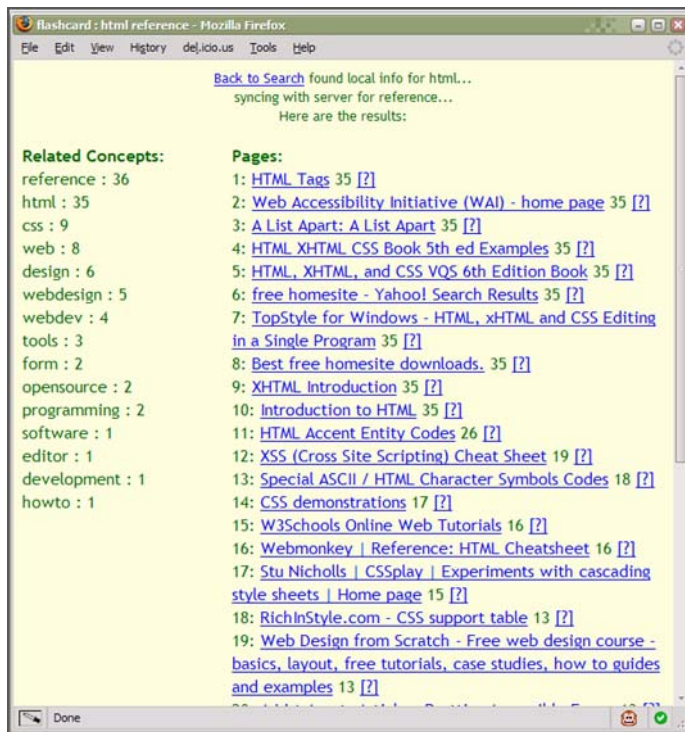


Figure 2: page flattening with averaged tag count

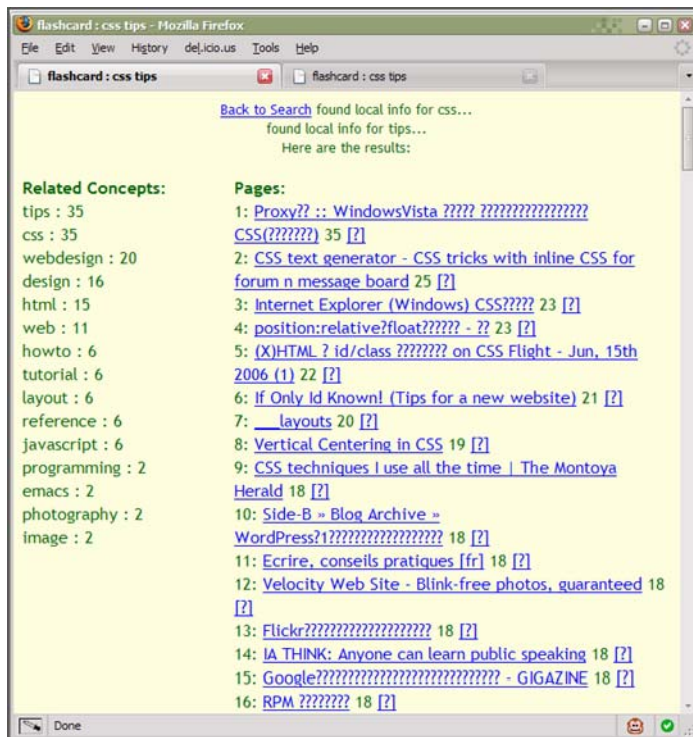


Figure 3: Linear tag bounding (punishment is always numTags)

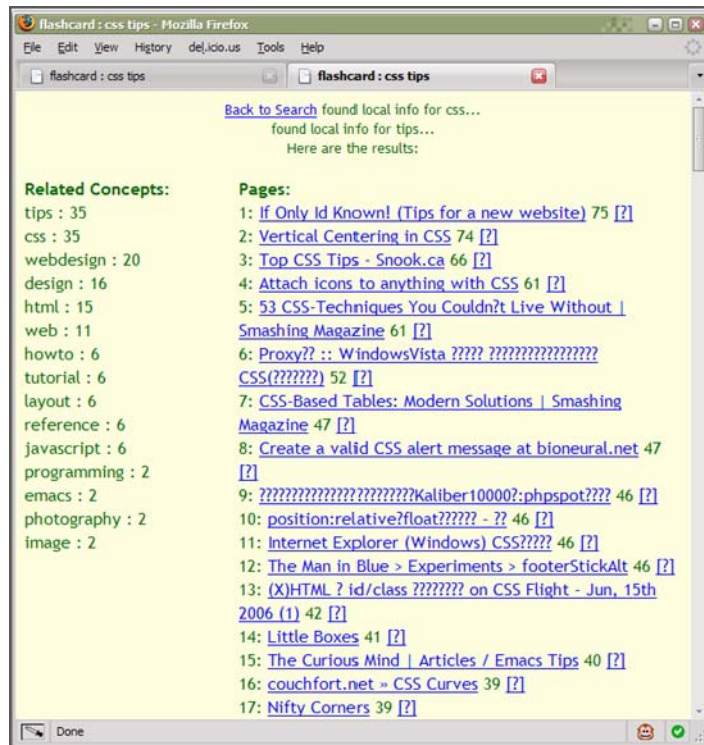


Figure 4: Exponential tag bounding (punishment is exponential function based on numTags)

Crawling/Indexing

A problem resulting from the dependence on del.icio.us is simply the lack of data and tags. Users typically have complex queries that cannot be satisfied by the dataset resulting from del.icio.us. One simple solution is to use the pages in del.icio.us as starting points for a crawling/indexing scheme. We maintain human indexing as the backbone of the system, allowing it to find relationships among words and drive the results of users' queries. The general process is as follows:

1. We visit each page that has been human-indexed and extract the full content
2. Using regular expressions, we discover all of the outbound links on the page
3. We visit each link on the page and extract the full content
4. The full content is flattened – we find the words with the highest frequency -- and we use the variance of word frequencies on the page to determine how many tags to associate with the page.
5. Finally, we insert the extracted tags and the page into the database

The implementation of this idea centers around the use of a service that is persisted on the backend of the site that can run independently of the PHP based search engine. The service instantiates a set of java threads that perform the various portions of the aforementioned algorithm:

- A Database Listener thread selects all the pages that haven't been "crawled" since the last time the Database Listener looked at the database. We maintain this invariant through a timestamp that the Database Listener updates after it has selected all the entries in the database. After the Database Listener has extracted the subset of del.icio.us pages from the database, it puts each

page into a static *pagesFedQueue*, an instance of a bounded buffer which has synchronized *getNext()* and *put()* methods.

- A Crawler thread utilizes the *getNext()* method to select each element of the *pagesFedQueue* and visit the URL. Using regular expressions, we discover all of the outbound links on the page. An issue is that many of the outbound links could be relative URLs – URLs that are missing the protocol, host name, or port number. We convert these relative URLs to complete URLs by using methods provided in Java's *java.net.URL* class and place this URL in a *pagesToCrawlQueue* which, like the *pagesFedQueue*, is also an instance of a bounded buffer.
- An Indexer thread uses *pagesToCrawlQueue.getNext()* and reads the entire content of the resulting webpage. It first extrapolates the title from the title tags on the page. It then uses a *StringTokenizer* to iterate through all of the words on the page and store it in a *HashMap*. A tag filter is used to ignore any surrounding markup or html idiosyncrasies. The *HashMap* stores the frequencies of each word; if a collision occurs, then the value (frequency of the word) associated with the hash key (the word) is incremented by 1. After we have constructed the hash of word frequencies, an insertion sort is performed on the keys and we store each word in an array, sorted by frequency. While performing the insertion sort, we also calculate the sum of all the frequencies and the squared sum of all the frequencies to determine the variance of frequencies. From the variance, the standard deviation is calculated and we only insert into the database the words that are within three standard deviations of the highest frequency that we have discovered.

Data Mining

Data Mining was introduced as a superset to the machine learning piece that is driven by human feedback. It was introduced as a proof of concept that it is possible to generate efficient association rules and/or clustering based on human-generated data. The goal of association rule finding in this application's context is to discover hidden, strong associations between more than two items. The basic idea of association rule finding can be attributed to "market-basket analysis". For example, a legendary survey of supermarket customers found the hidden consequence that customers who tend to buy diapers (who are typically young men) also buy beer.

A basic but efficient algorithm for association rule finding was described by Agrawal, Imielinski, and Swami and is known as Apriori (Agrawal, 1993). The implemented algorithm is described as follows:

1. We describe an "itemset" as a set of items, or tags, in our implementation. A "1-itemset" is defined as simply an array of all the tags in the system.
2. "Minimum support" in our implementation is described as the minimum percent of pages that an item can be in.
3. We begin the algorithm by creating a 1-itemset
4. For each k-1 itemset, we generate itemsets of size k by attempting to find two itemsets A and B that have the same k-2 tags; we add these along with the third item of A and the third item of B to create a new k-itemset.
5. We then test that the k-itemset satisfies the minimum-support property; if it does then we add the newly generated itemset to our final list of k-itemsets.

An issue is that since the database is constantly being updated, the minimum support will constantly decrease. If the minimum support is very small, the complexity of the algorithm increases and performance degrades. Therefore, a solution to this problem is still being researched.

A more advanced solution to intelligently finding related concepts involves clustering methods. Based on an initial survey of clustering methods (Berkhin, 2002), it was determined that density-based clustering

methods would be most viable for the weight-based relevancy engine. Partition-based algorithms, such as k-means are useful in this case since they are iterative-based, but the disadvantages encompass having to determine the number of clusters ahead of time and only circular shapes, with respect to the spatial layout of data determined by weights, can be identified as clusters. Hierarchical clustering algorithms are useful in that we can “zoom in” on how specific our cluster should be, but the drawback is that once a data item has been clustered, it cannot be assigned to a different cluster later. Density-based algorithms can detect arbitrary sizes of clusters and can filter out any noise, which is useful in our application. Therefore, a popular density-based algorithm, DBSCAN (Ester, 1996), was attempted; here is the basic algorithm:

1. For each point in the total set of points
 - a. If the cluster ID of the point is unclassified
 - i. If we can expand the cluster from the point(see following algorithm), then:
 1. Set the cluster ID equal to the next cluster ID

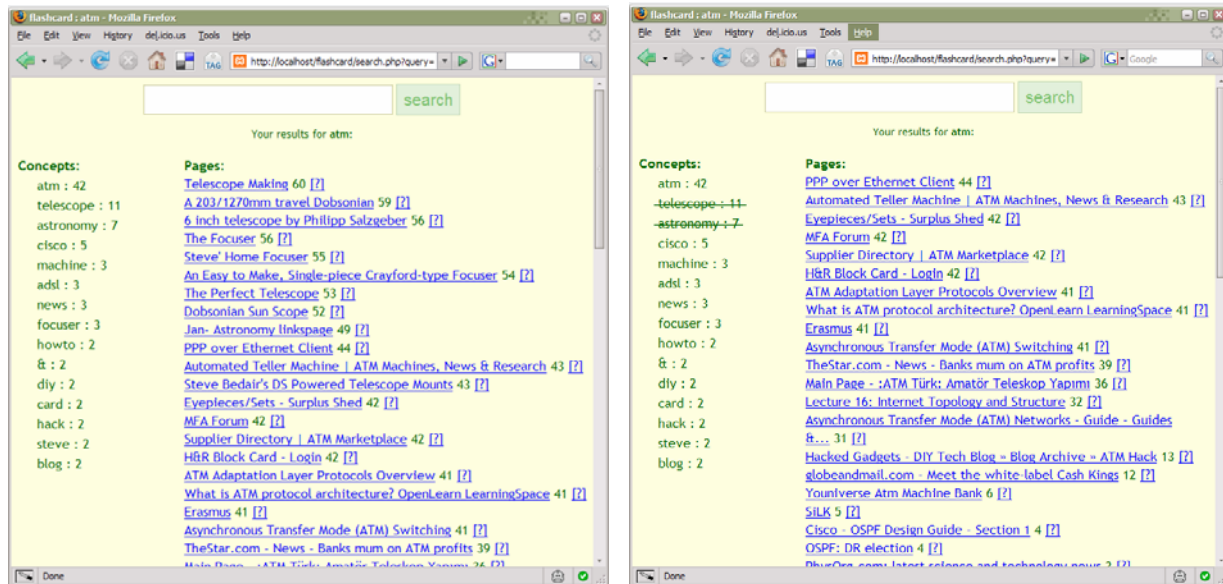
The following is the algorithm for expanding the cluster:

1. We find all points that are within a minimum distance of the point; these are called “seeds”
2. If the number of seeds is less than a given minimum size, then:
 - a. The point isn’t a “core point”, so we treat it as noise and return false
3. Otherwise:
 - a. Change all the cluster IDs of all the seed points to the given cluster ID
 - b. Delete the original, “core point” from the seeds array
 - c. For each seed in the seeds array:
 - i. Find all the points that are within a minimum distance of the first element in the seeds array; store these in a ‘result’ array
 - ii. If the number of points in the result array is greater than the minimum given size, then:
 1. For each point in the result array:
 - a. If the cluster ID is either unclassified or noise, then change the cluster ID of the point to the cluster ID
 - b. If the cluster ID of the point is unclassified, append the point to the seeds array
 - iii. Delete the first element from the seeds array
 - d. Return true

The original DBSCAN algorithm utilized the spatial data structure R* tree (Beckmann, 1990), which quickly finds region queries in $O(\log n)$ time based on minimum bounding rectangles. However, our database is not a spatial one. Region queries still have to be very fast. Before performing the DBSCAN algorithm, the application pulls all the tags and tag relationships from the database and translates them into a “SetOfPoints”, a class which associates each point with all of its neighbors, ordered by weights. This would not be feasible in the original DBSCAN algorithm since each point has some weight with every other point. However, in our system, this is not the case; there are strict relationships that exist in the relational database, so SetOfPoints contains a hash table of Points, where each Point has a “neighbors” hash table so it takes us $O(1)$ to find the all the Points within a minimum distance (known as an “eps-neighborhood” of a given Point.

Searching Experience

We can leverage the functionality provided by the discovery of related concepts in a more real-time fashion. Once the results page has been presented to the user along with related concepts, we can use this to allow the user to toggle/turn off/turn on other related concepts. Using a mixture of various client-side technologies and server-side preparation, we can translate what the user wants to do in terms of stepping out of or switching to a related concept into a user interface experience.



On the left pane, we see unfiltered results of a search for 'atm'. On the right, we have "turned off" the related concepts "telescope" and "astronomy". We can see the impact on the results. The ability to do this is achieved as follows:

1. On the server side we create an inverted index, where every tag is associated with the pages it resides on. The resulting array appears as follows: [Tag1 => [page1, page2, page3...], tag2=> [page1, page3...].
2. Using a JSON-PHP package, we can automatically convert the PHP associative array into Javascript Object Notation (JSON). Once we have done this, we can pass this string into the view and store it in an HTML element, such as a div block.
3. Javascript can read this string via the DOM and we can utilize the Javascript method eval() to translate the string into a series of Javascript objects; it keeps this structure in memory.
4. When a user "turns off" or "turns on" a related concept, an event handler fires a hash lookup on the javascript object, and it can subsequently use the DOM to find the referenced page IDs and hide or show them.

Not only does this feature enhance the user experience, but it also lifts the weight off the server's and database's shoulders by delivering a full MVC pattern into the client.

Improvements

There are several improvements and extensions we can perform upon the current application mentioned here. There are two main issues in resolving relevancy. The first is figuring out exactly what the user's goal/query/question is and the second is delivering the content that answers that goal/query/question. The first can be aided by viewing a user's content history and determining what pages they viewed in the

attempt of answering their question. We could flatten the content of these pages and determine related concepts in a similar manner to crawling/indexing.

Del.icio.us, as proved by this application, is a very strong data source but could be supported by other data sources that are human driven, for example: Flickr (<http://www.flickr.com>) – a photo tagging service, Blogmarks (<http://www.blogmarks.net>) – a blogging tagging service, Wikipedia (<http://www.wikipedia.com>) – a publicly writable encyclopedia, Yahoo! Answers (<http://answers.yahoo.com>) – a human-driven instant-answers service, and Dmoz (<http://www.dmoz.org>) – a taxonomy for the web.

The modification of search results on the client-side proved to be a success and the inclusion of other user interface technologies and visualizations would help deliver relevant results. The key point is that we don't take on all of the work in terms of delivering the results; in the end the user can decide for themselves through intuitive ways, which supports the entire human-aided theme of this application. Navigating through relationships, switching contexts, and narrowing in on contexts through the support of technologies like hierarchical clustering and density-based clustering would improve the service considerably.

Taking the human-centric approach even further, we can instill social networking within the search engine by segregating results based on a user's profile. For example, teachers could improve teachers' results without having the "noise" that is other's results. Weights in this case would be multi-dimensional and we could leverage more advanced machine learning technologies to deliver relevant results to specific groups of people.

Research has found that people who have friends are encouraged to annotate documents in del.icio.us (Lee, 2006); if we introduce social networking, such as the aforementioned profiles, users might be more inclined towards explicit feedback – clicking a 'yes' or 'no' button corresponding to the usefulness of a page. This can be witnessed in many web applications that exist today, such as Digg (<http://www.digg.com>) or YouTube (<http://www.youtube.com>).

Conclusion

The inclusion of social networking into a search engine that can discover semantic relationships was a success. Until artificial intelligence has advanced further, navigating and semantically defining the web can be effectively driven by humans, with the help of existing artificial intelligence technologies. The internet is by design a democratic technology. The data that exists on the web did not arise empirically or theoretically from any experiment; it sprouted from human creativity and intelligence. Therefore, humans in effect are the ultimate decision-makers in an abstract topic such as relevancy. By allowing humans to indirectly define simple semantic relationships; we can powerfully deliver relevant results in addition to discovering underlying, hidden concepts.

References

- Agrawal, R., & Srikant, R. (1994). Fast Algorithms for Mining Association Rules in Large Databases. *Proceedings of the 20th International Conference on Very Large Data Bases*. San Francisco: Morgan Kaufmann Publishers Inc.
- Agrawal, R., Imielinski, T., & Swami, A. (1993). Mining Association Rules between sets of items in large databases. *ACM SIGMOD international conference on Management of data*. Washington D.C.: ACM Press.
- Beckmann, N., Kriegel, H.-P., Schneider, R., & Seeger, B. (1990). The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. Atlantic City: ACM Press.
- Berkhin, P. (2002). *Survey of Clustering Data Mining Techniques*. San Jose: Accrue Software.
- Ester, M., Kriegel, H.-P., Sander, J., & Xu, X. (1996). A Density-based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. *Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining*. Portland: AAAI Press.
- Herman, I. (2007, February 2). *W3C Semantic Web Activity*. Retrieved February 5, 2007, from W3C: <http://www.w3.org/2001/sw/>
- Ivancsy, R., Babos, A., & Legany, C. (2005). Analysis and Extensions of Popular Clustering Algorithms. *Proceedings of the 6th International Symposium of Hungarian Researchers on Computational Intelligence*. Budapest, Hungary.
- Lee, K. J. (2006). What Goes Around Comes Around: An Analysis of Del.icio.us as a Social Space. *Computer Supported Cooperative Work Conference* (pp. 191-194). New York City: ACM Press.
- Ratzan, L. (2006, December 11). *Mining the Deep Web: Search Strategies that Work*. Retrieved January 12, 2007, from Computerworld: http://www.computerworld.com/action/article.do?command=viewArticleBasic&articleId=9005757&source=rss_news50