Classificoogle:
High Level Feature-Based HTML Document Classification

Darren Lewis
dalewis@stanford.edu

Ursula Chen
urse@stanford.edu

ABSTRACT
In this paper, we describe the design and implementation of a document classification system. By utilizing both high-level features of HTML documents and traditional term frequency information, we hope to achieve better classification results than systems that rely solely on text or document structure.

1. INTRODUCTION
As the World Wide Web increases in size and complexity, the importance and difficulty of determining its structure increases. Search engines and information retrieval systems require more sophisticated algorithms to find relevant data among the vast web of documents.

Classification systems help bring order to the web by determining the type or subject matter of a document. While traditional information retrieval methods were designed for plain text, the material available on the web is much richer and requires a whole new set of classification techniques.

This project aims to leverage the full amount of information contained within an HTML document in order to classify it. For each document, we compute a vector based on traditional tf.idf measures weighted according to the high-level document structure. We use these vectors to classify each document into one of ten predetermined categories from the Open Directory Project (ODP).[1]

2. PREVIOUS WORK
A number of papers use a subset of the terms in the corpus to determine the vector representation of each document. For example, Aggarwal, Gates, and Yu [3], in their supervised clustering algorithm, represent cluster centroids using only the terms most unique to the category. In an alternate approach, Wulfekuhler and Punch[8] used clustering techniques to find groups of terms that often occurred together across the corpus, regardless of category. Document vectors therefore indicate the number of occurrences of any word within each term cluster. These techniques differ from our approach in that they are fundamentally "flat text" based algorithms and do not take advantage of the document structure encoded in hypertext documents.

Other papers focused purely on document structure. Asirvatham and Ravi[4] chose a few structural elements of a document such as presence of mathematical formulae and percentage of a document occupied by link text as features. They then used machine learning to classify the documents into the broad categories of information, research, or personal web pages. Riboni[6] experiments with various text-based classification methods on hypertext documents, but he takes into account the enclosing tag of the text. For example, he finds that classifying documents based solely on the text found within META and TITLE tags and ignoring the BODY text results in higher accuracy than a classification based on all three sections of text.

3. DATA
We chose ten mid-size categories with about 500 to 2000 documents each from the Top/Computers section of the Open Directory Project to use as our corpus. These categories are listed below.

Categories
Artificial_Intelligence
Software/Operating_Systems/Linux
Software/Operating_Systems/Mac_OS
Software/Operating_Systems/Windows
Programming/Internet
Security/Products_and_Tools
Systems/Handhelds
Robotics
Internet/Protocols/
Internet/Searching/Directories/Open_Directory_Project

4. METHODOLOGY

The work for this classifier is divided into four stages: pre-processing, indexing, clustering, and classifying new documents. A high-level overview of each stage is given, along with detailed descriptions of each step.

4.1 PRE-PROCESSING

The pre-processing stage involved the following steps:

1) Parsing XML content from the ODP to retrieve links for the ten categories.
2) Crawling the links from the parsing stage in order to build the document corpus.
3) Cleaning the files from the crawl to remove non-HTML files or malformed HTML documents that we will not be able to parse.
4) Splitting the files into a test set and a training set.

The details of each stage are below:

4.1.1 Parsing XML from the ODP

We downloaded the 1.24GB "content" file from the ODP website. This is a single XML file that contains the categories and associated links for all of the categories that are present in the ODP. Since we were just interested in a very small subset of these links, we ran two parsing stages over this file. In the first stage, we parsed out all of the categories and links under Top/Computers into a separate file. In the next stage, we extracted only those links that were in the ten categories that our classifier recognizes.

4.1.2 Crawling

After the desired links were parsed, we wrote a crawler to download the HTML for these links into a separate file for each category.

4.1.3 Cleaning files

The crawl files were cleaned to remove non-HTML documents, since our classifier only works on HTML pages.

4.1.4 File splitting

Each of the cleaned crawl files were split into two files, a training set and a test set. We decided to make the test sets 100 documents each, leaving the resulting documents as the training set for each category. The documents in the test set were randomly chosen from the documents in each crawl file. Since each crawl file had a different number of documents, the training sets were of different sizes. This is a realistic model of the web, because different categories on the Internet are bound to have different numbers of documents.

4.2 INDEXING

We used Jakarta's Lucene [9] to build an inverted index over the documents in our corpus. We built six different types of indices for testing our system, each based on a unique way of parsing the HTML documents. Following are the details of the indexing along with a description of the index types.

4.2.1 Implementation

In order to build an index, we built a feeder to retrieve documents from our crawl files. These documents were then parsed with an HTML parser in combination with a filtered string tokenizer that extracted terms to be put into the index.

- **HTML parser** – Each index type has an associated HTML parser. These parsers walk the HTML document structure and invoke callback functions upon entering and exiting tags, which allows us to determine when we are parsing high-level features or just plain text. See below for a discussion of the high-level features and the index types.
- **String tokenizer** – After some of the initial indexing, we discovered that the index was overpopulated with useless terms. To decrease the number of terms in the index, we filtered by making all terms lowercase, removing common stop words, and using a Porter's stemmer to delete suffixes from the terms.

4.2.2 High-level document features

The high-level document features that we looked at are listed below:

- Document title – this is the text that is located within the <title> tags of the HTML document.
- Link text – this is the text located within the anchor (<a>) tags of the page.
- Heading text – this is the text located within <H1> through <H4> tags.
- Bold text – this is the text located between <b> tags.

4.2.3 Index representation of high-level features

After deciding to use an inverted index to aid our clustering stage, we needed to come up with a way to store the high-level feature information in the index.

Our original design was to just store standard term information in the index, and then during the clustering stage, compute the high-level features of each document and add those features to that document's vector. However, the issue with this approach is that it involves processing each document twice, and also requires retrieving the term information for each document during the vector construction stage. This is expensive and also negates the value of the inverted index, which is designed to be used for looking up document information by term rather than term information by document. To overcome this obstacle, we devised a clever method to use the inverted index to store high-level feature information along with traditional term information. Specifically, we assigned a weight to each of the high-level features. In those indices where we used high-level feature information, we simply put each of the high level terms into the index the number of times equal to its weight. So, for example, every word in the title was entered into the index ten times for that document. This was accomplished during the HTML parsing stage, which generates a "page text" string and an "important text" string with the appropriate term weighting. The benefits of this approach are multifold: First, it allows us to use the inverted index and only the inverted index during the clustering algorithm, the details of which are in the next section. Second, it greatly simplifies the construction of our document vectors. By weighting a term in the index, we increase its tf.idf value, thereby enabling documents to cluster more strongly around the high-level feature terms, which we subsequently call the "important" terms. Third, it allowed us to use our existing infrastructure for computing tf.idf vectors rather than developing new code for constructing vectors that are split between traditional tf.idf and other values. It also eliminated the need to determine a range of values for our high-level features, because the high-level feature values are now implicitly and automatically computed along with the rest of the terms. The weights we used for our high-level features are as follows:

Feature	Weight
Title	10
Links	3
H1	5
H2	4
H3	3
H4-H6	2
Bold	2
Default text	1

Table 1: Document features and weights

4.2.4 Index types

Below are the six different types of indices we built:

- *Simple index* – All terms (this includes regular text along with the text in the high-level feature tags) are put into the index, no weighting.
- *Type 1 High-level feature index* – Same as the simple index, but with term weighting for the "important" terms
- *Type 2 High-level feature index* – Only "important" terms are indexed, no weighting.
- *Type 3 High-level feature index* – Same as 2, but with term weighting.
- *Type 4 High-level feature index* – The general "text" terms and the "important" terms are indexed as separate term types, no weighting.
- *Type 5 High-level feature index* – Same as 4, but with term weighting.

4.3 CLUSTERING

In the clustering stage, we used the inverted index described above to compute document vectors for each document. Then we ran two different algorithms to compute the centroids for each category. Finally, we assigned the centroids to categories. The first clustering algorithm is the method proposed by Han and Karypis[5]. For this case, we simply computed the centroids of all the document vectors from each category's training set. The second clustering algorithm we explored was the well-known k-means algorithm. [2],[6]. In both cases we computed document vectors in the same way, which is described below.

4.3.1 Constructing document vectors

The first stage in constructing the document vectors was to put all "useful" terms from the index into a hash table. Our original goal was to use all the terms from the index for the document vectors, but even with the stemming and stop word elimination as described

above, the index was still filled with useless terms, misspellings, and garbage. We used two heuristics in order to decrease the length of our document vectors to a manageable size. The first is that we only used terms that matched the regular expression ^[a-z]*, that is, all terms that start with and are composed entirely of letters. The second heuristic is to only include terms that are found in at least a lower bound number of documents. We arbitrarily set this lower bound to 10. Terms that passed these two heuristics were considered "useful" and were the terms that the document vectors were computed on. After all useful terms were entered into the hash table, we iterated over the terms, and for each term/document pairing in the index, we computed the tf.idf value for that term and that document and set the appropriate value in that document's vector, using the index of the term in the list of values from the hash table as the index for the term in the document vectors. We used the standard tf.idf value of:

$$tf.idf_{t,d} = tf_{t,d} * \log(n / df_t),$$

where $tf_{t,d}$ is the frequency of term t in document d, df_t is the number of documents that contain term t, and n is the total number of documents in the corpus.[10]

4.3.2 Computing centroids

In both cases, the first step is to normalize the document vectors to unit length. We then performed the following operations:

- Category-based centroids – For the category-based centroids, we simply added up all the documents in each category and divided by the number of documents in the category. Since we stored the URL of each document in the index along with that document's terms, determining which category each document vector belonged to was easy.
- K-means centroids – We ran the normal k-means algorithm until the maximum difference between the old cluster centers and the new cluster centers was less than an epsilon value, which we chose to be .001.

4.3.3 Assigning centroids

- Category-based centroids – Assigning the category-based centroids was trivial since the category was known before the computation.
- K-means centroids – Since we planned to assign the k-means centroids to the ODP categories, we needed an algorithm to make the assignment. We decided to make the assignments such that the sum of the document similarities between the k-means centroids and the category-based centroids (which are an input to our k-means algorithm) was a maximum. Note that the similarity between two unit-normalized vectors is simply their dot product. In order to maximize this assignment, we were fortunate that our number of categories was low, and we simply calculated every permutation of assignments and selected the assignment with the maximum value.[12]

4.4 CLASSIFYING NEW DOCUMENTS

The final stage of this project involved classifying new documents based on our centroids from the clustering stage. We classified new documents by computing the new document's vector, and then assigning the document to the category of the centroid that has the highest similarity with the new document. The key to constructing the new document's vector was to use the same HTML parser and string tokenizer that were used in the indexing stage. Specifically, we used the parser to compute the general text and term-weighted "important" text, and then used the string tokenizer to parse out the terms from the document. We then put the "useful" terms from these into a hash table to compute the frequency of each term in the new document, using the same heuristic for term "usefulness" as described in Section 4.3.1. We used the inverted index to find the document frequency of each term, which allowed us to compute the tf.idf values for each term in the new document's vector. We then computed the dot product of the new vector with each of the centroids and assigned the document to the category with the highest score

5. RESULTS

We evaluated our classification schemes by classifying each of the documents in our test sets and then determining the accuracy of each category centroid by dividing the number of correct classifications by the number of documents in the test set. These calculations were done for all ten categories in each of the six index types. The results are summarized in the tables below.

	Index Type					
Category	Simple	1	2	3	4	5
AI	0.85	0.85	0.76	0.83	0.84	0.85
Linux	0.83	0.84	0.77	0.77	0.84	0.83
Mac	0.69	0.76	0.74	0.78	0.71	0.77
Windows	0.73	0.79	0.72	0.76	0.72	0.74
Programming	0.78	0.8	0.69	0.73	0.76	0.77
Security	0.75	0.78	0.7	0.71	0.77	0.76
Handhelds	0.76	0.78	0.72	0.74	0.79	0.75
Robotics	0.66	0.68	0.64	0.62	0.66	0.65
Protocols	0.84	0.82	0.67	0.72	0.84	0.79
ODP	0.67	0.72	0.71	0.72	0.68	0.7
Sum	7.56	7.82	7.12	7.38	7.61	7.61
Stddev	0.070	0.052	0.040	0.055	0.067	0.058
Mean	**0.756**	**0.782**	**0.712**	**0.738**	**0.761**	**0.761**

Table 2: Accuracy of Category-based Centroids

	Index Type					
Category	Simple	1	2	3	4	5
AI	0.69	0.64	0.03	0.76	0.63	0
Linux	0.86	0.45	0.57	0.66	0.31	0.83
Mac	0.1	0.27	0	0	0.03	0.06
Windows	0	0.77	0.63	0.83	0.7	0.02
Programming	0.05	0.01	0.63	0.1	0.14	0.7
Security	0	0	0.76	0.79	0	0.75
Handhelds	0.6	0.61	0.07	0.06	0.04	0.6
Robotics	0.55	0.63	0.65	0.58	0.51	0.63
Protocols	0.61	0.44	0.03	0.3	0.22	0.26
ODP	0.62	0.6	0.59	0.61	0.61	0.6
Sum	4.08	4.42	3.96	4.69	3.19	4.45
Stddev	0.330	0.268	0.317	0.323	0.272	0.325
Mean	**0.408**	**0.442**	**0.396**	**0.469**	**0.319**	**0.445**

Table 3: Accuracy of K-means-based Centroids

For category-based centroids, we attained our highest accuracy by considering both plain text and important text to be composed of the same type of terms, and weighting the important text according to the method described in Section 4.2.3. With our weighting scheme, inserting all terms into the index performed better than inserting just the important terms, which can be seen by observing the accuracies for index types 2 and 3. Indexing general text and important text separately was better than the simple indexing case and just using important terms, but still not quite as accurate as index type 1 which considers all terms the same type but weights important terms.

For k-means-based centroids, the accuracy was much more erratic than the category-based centroids, with some categories having extremely high accuracy while others have zero percent accuracy. The most accurate index was type 3, which indexes important terms with weighting and not general text. The erratic nature of the k-means centroids can be attributed to the fact that from a term frequency standpoint, the documents from the ten categories that we used from the ODP do not cluster exactly into their assigned categories. K-means is not sophisticated enough to determine why the documents were classified by the ODP editors as they were. The reason is that the ODP human classifiers take a holistic and content-centered approach to classification, while our K-means implementation classifies based on feature-weighted tf.idf values.

Listed below are the top five terms by tf.idf for each centroid from the category-based centroids of index 1 and the k-means-based centroids of index 3, the best performers for each type of centroid computation:

Category	Terms
AI	`intellig, agent, research, confer, artifici`
Linux	`linux, i, kernel, project, page`
Mac	`mac, macintosh, os, x, softwar`
Windows	`window, xp, Microsoft, tip, nt`
Programming	`asp, cgi, script, host, soap`
Security	`secur, password, encrypt, pgp, protect`
Handhelds	`palm, pocket, pda, pilot, pc`
Robotics	`robot, team, home, simul, research`
Protocols	`rfc, ip, protocol, rout, dn`
ODP	`restaur, chefmoz, state, unit, result`

Table 4. Top five terms from index 1 category-based centroids. (number of terms: 9161)

Category	Terms
AI	agent, i, ip, network, intellig
Linux	linux, instal, user, the, debian
Mac	filter, theme, environ, gnu, desktop
Windows	window, palm, asp, xp, more
Programming	soap, uddi, xml, servic, section
Security	secur, password, softwar, recoveri, inc
Handhelds	http, re, info, user, dongl
Robotics	rfc, robot, xform, protocol, faq
Protocols	draft, mpl, ietf, bibliographi, rsvp
ODP	unit, state, result, chefmoz, restaur

Table 5. Top five terms from index 3 k-means-based centroids. (number of terms: 4280)

6. INTERFACE

We decided to put our classifier to good use by hooking it up to a web interface so that users from all over the Internet can determine which of our ten categories best match their search results. We did this using Java Servlet technology in tandem with the Google API [11] Our web interface works as follows: Users enter a search query into a Google-like search bar. The servlet sends the query to Google using the Google API and receives the top ten search results. Since we do not possess the resources to store and pre-classify all the documents that a Google search might return, we took the more conservative approach and simply download each of the search results, send the HTML to the Classifier, and display the result and category that the Classifier returns. Google actually uses the ODP categories in its search engine, but rather than classifying search results, they classify the query. We believe that having a classification for the search results is more useful, because it allows the user to determine with just a quick glance what category each of their results belong in, giving them the ability to quickly ignore results from unwanted categories and pursue results from pertinent categories.

7. CHALLENGES AND TRADEOFFS

Working with such vast amounts of information presents a number of interesting challenges. The first and most obvious challenge is that of quantity, in both time and space. Crawling large quantities of data from the web takes a long time, and storing that data takes a great deal of space. We chose to focus on mid-size subcategories from within a top-level category because it allowed our Classifier to be accurate for the documents it knew about rather than less accurate for a larger subset of categories. In retrospect, it might have been interesting to take a random sampling of web pages from all of the top-level categories and attempt to build a Classifier that classifies documents into one of the top-level categories, but we decided instead to stick to small categories within Computers because it presented a more rigorous challenge for our Classifier to determine subtle distinctions between web pages.

Another challenge was memory usage. Deciding how many terms to include in each document vector presented a significant tradeoff in the system, because we wanted to ensure that important terms were included, but our vectors had to be small enough to compute with our modest RAM and disk space. We did all of the large index processing and centroid computation using high-powered 64-bit Sun UltraSparcIIIs with 2048MB of RAM, providing more than adequate CPU power and memory availability for our document vectors after our heuristics were used to limit the number terms in each vector.

Third, servlet technology provided an unexpected obstacle to our Classifier. Our original design had the servlet associated with an index and category centroids, so the servlet would open the index and read the centroids from a file upon loading. The issue with this is that servlet I/O must occur in a very specific manner, and since the I/O for the index file was controlled by Lucene classes that could not easily be subclassed, we discovered that we were unable to open the index file from the servlet. Our workaround was to split up the work of the Classifier into two processes, the servlet and an Evaluation Server. The Evaluation Server is a normal java process that is associated with an index and a centroid file, and thus it is able to access these files in the normal manner. The servlet accepts web queries and sends them along to the Classifier, which in turn acts as a client and uses a TCP/IP socket to communicate with the Evaluation Server. Communication over the socket is achieved via

object serialization, and overall design of the system is better in the sense that it is more modular and the servlet is used for its specialty, web processing, while the Evaluation Server takes care of the classification computations described above.

8. CONCLUSIONS

We have shown a novel method of utilizing high-level document features in an automatic document classifier. By extending an existing tf.idf framework to include higher-level features via term weighting, we have constructed a classifier that is indeed more accurate than a classifier over the same index without term weighting. Future areas of investigation include exploring additional high-level document features such as tables and images, expanding the number of categories that the classifier deals with, and creating an on-the-fly classification paradigm that is faster than our current method of downloading and classifying each search result.

9. REFERENCES

Special thanks to Hinrich Schütze for suggesting using the ODP Computers section as our training set, and Benjamin Grol for suggesting using sockets to overcome the file i/o problem.

[1] Open Directory Project, http://www.dmoz.org

[2] Anderberg, M.R. *Cluster Analysis for Applications.* Academic Press, New York, NY, 1973.

[3] Aggarwal, C.C., Gates, S.C. and Yu, P.S. On the merits of building categorization systems by supervised clustering. In *Proceedings of KDD-99, 5th ACM International Conference on Knowledge Discovery and Data Mining.* (1999), ACM Press, New York, NY, 352-356.

[4] Asirvatham, A.P. and Ravi, K.K. *Web page classification based on document structure.* (2001).

[5] Han, E.-H.S. and Karypis, G. Centroid-based document classification: analysis and experimental results. *Tech. Rep. 00-017*, Computer Science, University of Minnesota. (2000).

[6] MacQueen, J. B. Some methods for classification and analysis of multivariate observations. In *Proceedings of 5th Berkeley symposium on mathematical statistics and Probability.* (1967), University of California Press, Berkeley, CA, 1:281-297.

[7] Riboni, D. Feature selection for web page classification. *Eurasia-ICT 2002 proceedings of the workshops*. (2002).

[8] Wulfekuhler, M.R., and Punch, W.F. Finding salient features for personal web page categories. In *Proceedings of the 6th international world wide web conference.* (1997).

[9] Jakarta Lucene, http://jakarta.apache.org/lucene

[10] Lecture slides, CS276 Lecture 3 (Stanford University)

[11] Google API, http://api.google.com

[12] "Generating the Next Largest Permutation in Lexicographic Order". Rosen, K. Discrete Mathematics and Its Applications, Fourth Edition:298. (1999).