

Bayesian Filtering Implementation for Web Content

Jonathan Bulava
Department of Computer Science
The College of New Jersey
Ewing, NJ 08628
(609) 637-6323

bulava2@tcnj.edu

ABSTRACT

This paper focuses on evaluating and optimizing the application used to filter web content. It will cover the many steps and complications along the way as well as how they were overcome to arrive at the project's current state. These steps include programming language choice, redesigning the existing filter application, and integrating the new application into our custom Apache module.

1. INTRODUCTION

1.1 Background

The amount of information available via the Internet is growing exponentially every day. The discussion on website censorship becomes increasingly important at the same rate and it is the responsibility of those who use the Internet to control or monitor this content. There are several commercial products on the market to sift through web information and decide for the consumer what is "good" or "bad," but some of these products present issues. For example, most of the filtering process is left to the software without user involvement. In addition, the consumers are not directly connected to the other consumers of the same product. This limits the amount of discussion regarding individual purposes for filtering or assistance between those with similar objectives.

1.2 Solution and Goals

A solution to overcome these issues can be created using Bayesian filtering methods that are mostly used today for email spam detection¹. The overall project goal is to design, implement and deploy a Bayesian filtering system used to calculate the suitability of web content in an online community of similar-minded users. It is in this environment where community leaders can create web filters that prevent access to content deemed inappropriate by the community (e.g. corporate offices, church groups, etc.). Our current research focuses on producing an efficient technique to enable a filter, and acquire swift performance when calculating the validity of a web page.

To begin creating a system that will block select web pages, an Apache 2 server² was set up with its Apache proxy module

enabled. The design is to proxy the client web browser through the server and filter the content as it passes through the server. The filtering process is done in a custom Apache module that we developed. The module receives web page data from the proxy module, extracts text and other useful information from the web page and then calculates a rating for the appropriateness of the web page based on the information extracted. At this point, our prototype returns either the original page to the requesting user or an error page which states that the page has been blocked.

2. REDEVELOPMENT

The Bayesian filtering application was previously built in Java and PHP. The Java implementation functioned quickly, but didn't fit into the architecture of our filtering system. The PHP implementation was very easy to incorporate into the overall filtering system, but because it's not precompiled there was too much overhead for a web page to be returned to the user. We have since decided on a C/C++ implementation of the filtering system because of speed and relative ease of incorporation. At the time, however, we would not know how bandwidth and web page size would affect the system until it was fully integrated. Nevertheless, these factors will be considered when tweaking the system for performance.

2.1 How a Bayesian Filter Works

A Bayesian filter uses statistical analysis to evaluate the validity of a website instead of keywords and phrases like most filters. This method has a higher success rate to the keyword method simply because it calculates a probability of being "bad." If this probability is greater than a desired threshold, the page will be blocked. In a way, it can be thought of as a fuzzy logic approach to filtering web content.

Before it can calculate this probability, the filter needed to have possession of all of the words found on the site being questioned. A parser is used to extract all of the textual content, disregarding any HTML tags or code. These words are then given to the filter and placed in a hash table. These words are compared to content stored within two other hash tables. One of these contains an acceptable content set while the other contains inappropriate content, both predefined by the publisher of the filter. Each word from the website is assigned a probability of unsuitability. With a few formulas given by Bayes' theorem, the entire collection of probabilities is used to calculate an overall probability for the site as a whole.

¹ To learn about Bayesian filtering and its use against email spam, see "A Plan for Spam" by Paul Graham. <http://www.paulgraham.com/spam.html>

² See Apache's website. <http://www.apache.org/>

2.2 A Look at Past Development

After C++ was decided to be used for the application, the next step was to evaluate how the filter was built and implemented in the other two languages. Without any previous knowledge of how such a filtering system works, it was important to use the language closest to C++ and contained the most documented code. Since Java and C++ are both object oriented programming languages, I decided to focus most of my attention attempting to understand how the Java version functioned.

Unfortunately, a majority of the methodology used in the Java application was not as easily available for C++. Many components had to be programmed from scratch and include anything from sorting algorithms to the hash tables. It was quite noticeable that more time would be needed to complete the same task that was previously achieved in Java. Java was most likely the first language chosen to build the filter due to the massive documentation that is so well-kept and easily accessible. On the other hand, however, the new C++ version could very well prove to work much faster than its predecessors. There are also some components built into the Java program structure that are not possible in C++ (e.g., A class returning an instance of itself or two classes contains instances of each other). For these reasons and more, the program structure had to be altered for this new version written in another programming language. All of these complexities lead to a new design for the filtering system.

2.3 Design

One of the initial problems that arose in the Java program structure, was the ability to have classes contains instances of one another. The Probability class possessed NuHashtable objects and the NuHashtable class contained a Probability object. This cycle of class structure is not compliant with the C++ programming language. Upon closer inspection, it became evident that the NuHashtable class contained a Probability object for use within one method, fillProb(). Its purpose is to extract all of the words that were parsed from a website and place them each in a Probability object as a WordData type. Each new WordData type keeps track of a word and that words initial probability of appropriateness.

Now, since this the only place that Probability is used in the NuHashtable class, it is quite possible to move this functionality to the driver program where these numbers are heavily used. Once fillProb() is removed from the NuHashtable class, the two objects are no longer dependant on one another. The amount of code used in the driver to achieve the same results actually decreases. The number of line from the Java version to the C++ version decreased to less than half.

The availability of a hash table class for C++ was another issue that appeared during the design phase of this filter. Unfortunately, the only hash table class that could be found for the language was supported by Microsoft's .NET Framework. Therefore, it was evident that I should create my own hash table class. The actual hashing algorithm is conducted by the map.h predefined class. My hash table class is an extension of this class and is not templated, but designed solely for operations that deal with Word_data objects. This should spend up any functions used to view, edit or retrieve data.

In order to avoid memory allocation problems at this point, the number of words retrieved from a website is limited to one thousand words. This will most likely block the page accurately regardless of not containing all the words. In the majority of web pages, similar content will be dispersed through the text and should, therefore, be evaluated correctly by the filter. The worst case scenario would be if a site's unsuitable content is only present after the one thousandth word.

In order to solve this problem in the future, I believe that a second statistical analysis should be built into the filtering system. As would be expected, as the number of words contained on a page increase, so does the execution time to calculate the overall probability. However, if we find that a website has more than a certain number of words, say one thousand to be consistent with the problem stated above, it would be beneficial to do a systematic random sample of words found on the page and use those one thousand words to calculate the probability. With such a large random sample of textual elements, the accuracy of the filter can only increase.

A UML diagram of the filter's structure in C++ can be found in Appendix A.

3. IMPLEMENTATION

Before the filter can be tested as a standalone application, particular conditions need to be in place that will simulate its existence in the actual integrated system. There are three necessary elements for the system to function properly. The filter will need a set of words to be placed in the "good" hash table, a set of words placed in the "bad" hash table, and a set of words from the site being analyzed.

3.1 Preparation for Future Use

When the filter is initialized, the good and bad lists of words are being loaded into the filter via text files. When the system is tied into the online community, every user will have their own lists, just like these, which they can add or delete select words or whole websites. Then the list of words parsed from the disputed web page is loaded into the filter as well. This list will eventually come from a parser within the module, but for the meantime, I am using a web application I created in PHP³ that uses parsing methods previously designed under this project⁴.

3.2 Testing Accuracy

With the working prototype, the next step is to evaluate and test the efficiency of the filter. The first aspect to test is the accuracy of the returned probability. It may seem impossible to detect the filter's level of correctness without doing the calculations by hand. Instead of choosing random websites and seeing if a page is blocked or allowed, certain concrete tests are set up. There are a few situations where a specific outcome is expected.

³ <http://www.php.net/>

⁴ A copy of the code can be found in Appendix C.

For example, in the first test, all of the textual content located on the ACM (Association for Computing Machinery)⁵ front page is used in the filter to denote acceptable content. If we run the ACM through the filter, then we should expect the lowest possible probability. And in fact our tests return an extremely low probability of 5.38653e-35.

The second test to perform on the filter would be the exact opposite idea of the one above. Now the bad content stored in the filter should be compared to its source. The list of inappropriate words for our filter comes from the website of ESPN, Inc. (Entertainment and Sports Programming Network)⁶. If we run this site through the filter, then a probability of 1 should be expected as the result since the site and the bad word list are an exact match. After running the program, we see that 1 is returned every time.

Now, the third type of test is to utilize other websites of which the outcome can be humanly determined. In other words, sites that seem to lean toward good content or lean toward bad content. In order to witness how well our filter actually works, the Yahoo! Sports⁷ page was run through the filter. It should be expected for the site's probability to be quite high. In fact, yahoo sports also returns a probability of 1. This result is most likely the case because ESPN contains more words than Yahoo!'s page and the majority of Yahoo!'s content is probably a sub-set of the words found on ESPN's website.

3.3 Testing Time

The execution time is the most relevant factor to the user requesting the web page. The user shouldn't have to wait excessively longer than it would normally take to load a page without the filter in place. The less time it takes for the filter to examine a web page via the proxy server and return a result the better. The best case scenario would be to create an implementation of the filter where the overhead is undetectable.

The standalone filter application returned results extremely fast for the amount of computation necessary. Although an exact measurement of the execution time was not measured, most results would be returned within one or two seconds. It was not essential to obtain these numbers since we are more concerned about the turn around time once the application is integrated within the Apache 2 web server.

4. INTEGRATION

Once the filter was tested and proved to work efficiently, it came time to integrate it with the custom Apache module that my research partner had been constructing. The idea behind this next step was to include the code that I written and use it in the module as a library of functions.

Unfortunately, this is where the project took an unexpected turn. We were under the impression from the beginning of the project that it would be easy to integrate C++ generated code into the

⁵ <http://www.acm.org/>

⁶ <http://www.espn.com>

⁷ <http://sports.yahoo.com>

module that is written in C. For several weeks, several ideas were passed around and different approaches were taken. All of our attempts to accomplish the incorporation of an object oriented programming language and a linear programming language were unsuccessful.

After looking back and evaluating the overall concept of the project, we realized that our goal was to develop a filtering prototype through the use of a web server and it was not necessarily to be accomplished by way of our current methods. It came apparent that the best solution would be to port the C++ code to C so it could be used directly from within the module without having to deal with a different programming language. At first we were hesitate of the idea only because we would not have enough time to possible include all of the additional functionality that the C++ version offers, but it was a trade off we were prepared to make for the time being in order to prove that the filtering system will function as planned.

The main components of the filter, mostly consisting of the Probability class, were converted into C code. A standalone C file was created to gradually make the shift in language and compile it after new segments were added. Variables that are used throughout the program were temporarily initialized at the top of this file so that the compilation could successfully run. After the conversion was completed, this section of initialization could be eliminated and the rest of the code would therefore be able to be pasted directly into the module and work accordingly. Once this code was in the hands of my research partner who is the one in charge of the Apache module, a few additional changes were made on his part to make the filter work efficiently.

5. SUMMARY AND CONCLUSIONS

Until this semester, the idea of a Bayesian filtering system functioning accurately on a web server was just that; an idea. Now after much research and hard work, we have discovered a method to make this concept possible. The most tedious process along the way was certainly producing code for the C++ application due to the lack of extensive documentation. Reengineering is a concept that is very common in computer science and it is surely the next step in this project. We have proven that this can be done and now the process needs to be examined more closely. Although the foundation of the project has been created, it needs to be scrutinized in order to add necessary functionality to our solution or to find other methods to achieve the same results only faster and more efficient.

6. FUTURE WORK

There are three essential components still remaining in the project that need to be addressed. The first of these has been added onto the project because of the way the filter and the web server eventually were integrated. The C++ filter application has far more functionality than the translated C version that is located on the web server. Our concentration this semester was to prove that the filter could actually work in this manner and now additional elements found in the standalone application can be included to create a more flexible and efficient environment.

Now that filters can be enabled in the system, the next step would be to integrate the online community with the development on the

web server. It is from this site that the filters should be created and maintained. The Bayesian Clearinghouse creates clean user interface for all the members of the site to easily edit their filtering needs.

The final component in this project would be to actually question our current solution to the problem. Is there a better way to achieve the same goals? Better yet, is there a more efficient way to achieve our goals and even add more to the functionality to the filtering system? Questioning the solution can lead to other options and developments that may be quite beneficial to the project.

7. ACKNOWLEDGMENTS

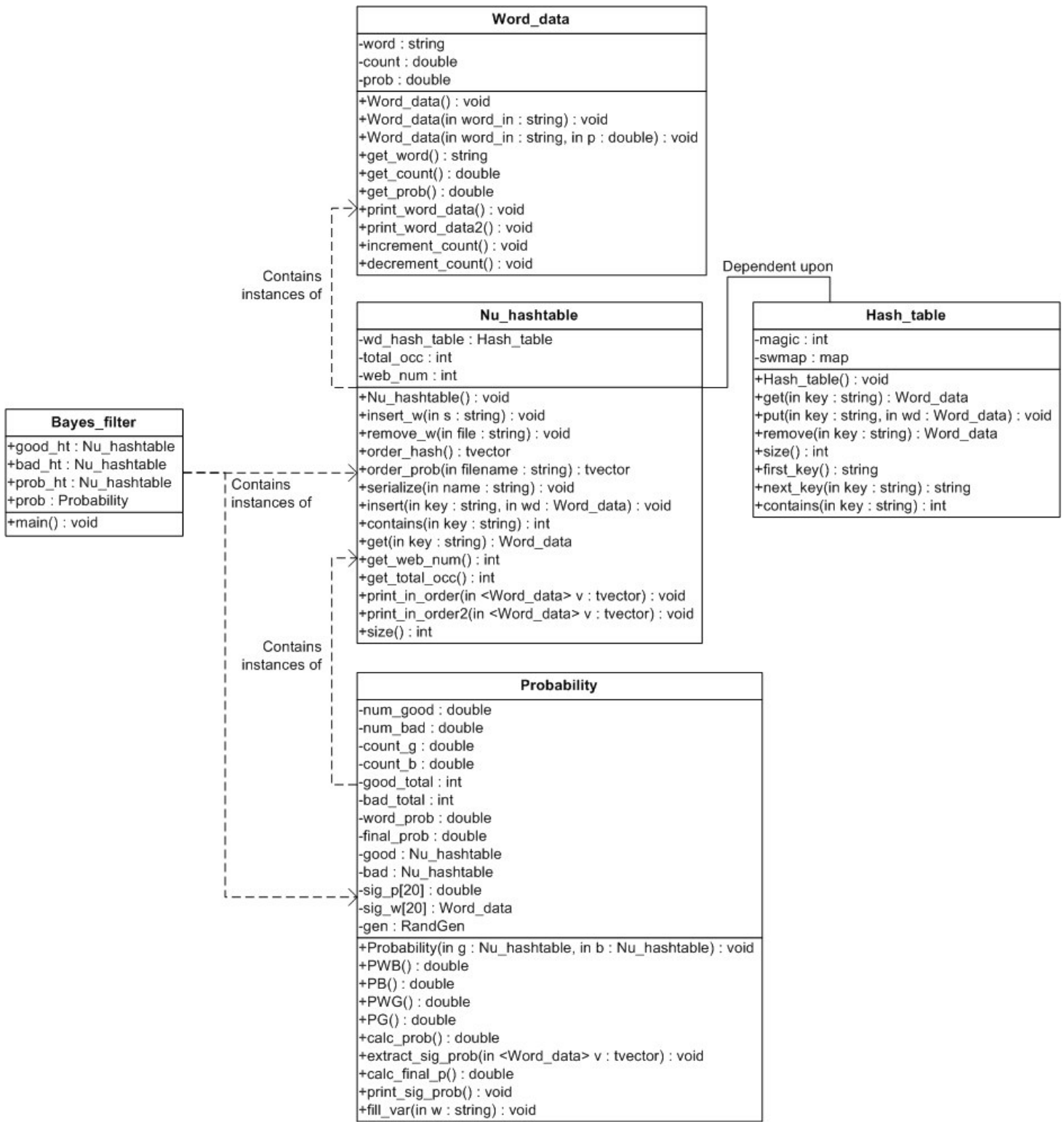
I would like to thank my research mentor, Dr. P. DePasquale, for allowing me to be one of the first students to develop such a novel and remarkable project in computer science. I also want to thank my research partner, Mr. B. Wanner, for possessing the ability for us to work together efficiently in order to attain our goals throughout the term.

8. REFERENCES

- [1] Ullman, Larry. PHP for the World Wide Web, Second Edition. Peachpit Press, Berkeley CA, 2004.
- [2] Astrachan, Owen L. A Computer Science Tapestry, Second Edition. The McGraw-Hill Companies, Inc, Boston, 2000.

APPENDIX A

Below is a UML diagram of the filter application's class structure.



APPENDIX B

B.1 The Driver Program

This program is run to test the results of the Bayesian filter.

```
/******  
Jonathan Bulava  
Filtering Research  
Dr. Peter DePasquale  
Bayes_filter.cpp  
*****/  
  
/******  
This is the driver class used to test the Bayesian  
filter that can be constructed using the other  
included classes. It initializes the good and bad  
hashtables which are dependent on the content the user  
wishes to filter. It then initializes a hashtable of  
words from a text file where the filename is given on  
the command line. These words are from the site that  
the filter is supposed to check. The probability is  
sent to a dat file unless the debug flag has been set  
at the command line. In that case, it is only printed  
on the screen.  
*****/  
  
#include "Nu_hashtable.h"  
#include "Word_data.h"  
#include "Probability.h"  
#include <string>  
#include <iostream>  
#include <fstream>  
//#include "time.h"  
using namespace std;  
  
int main(int argc, char **argv) { //USAGE: Bayes_filter <file> (*|debug)  
  
    if ((argc == 2) || (argc == 3)) { //Make sure there are two or three arguments  
        // entered on the command line  
  
        bool debug = false; //debug value set to false  
  
        if (argc == 3) {  
            string debug_check = argv[2];  
            if(debug_check == "debug") //if debug is typed in on the command line,  
                debug = true; //debug value is set to true  
        }  
  
        if(debug) //MESSAGE: Beginning of program  
            cout << "\nBegin main...\n";  
  
        //Initialize variables  
        string filename = argv[1]; //text file of parsed html text  
        Nu_hashtable good_ht; //good hash table  
        Nu_hashtable bad_ht; //bad hash table  
        Nu_hashtable prob_ht; //probability hash table
```

```

if(debug) //MESSAGE: All variables initialized
    cout << "Initialized hash tables...\n";

good_ht.fill_word("good.txt"); //fill hashtable with good words
bad_ht.fill_word("bad.txt"); //fill hashtable with bad words

if(debug) //MESSAGE: Good/bad HTs filled
    cout << "Filled good and bad hash tables...\n";

Probability prob(good_ht, bad_ht); //create a probability object given
// the good and bad hash tables

if(debug) //MESSAGE: Probability object
    cout << "Initialized probability object...\n";

//initialize variables to read parsed words
string word;
ifstream input;
input.open(filename.c_str()); //open the input stream
int item_count = 1;

while ((input >> word) && (item_count <= 1000)) { //while there are still words,
    prob.fill_var(word); //place the word in the prob object
    Word_data word_obj(word, prob.calc_prob());
    prob_ht.insert(word, word_obj); //insert the word in the hashtable
    item_count++;
}

if(debug) //MESSAGE: Print word count
    cout << "Word count: " << item_count << "\n";

if(item_count > 1000) { //Check if maximum vector size is exceeded
    cout << "Error: Vector limit of 1000 exceeded.\n";
    exit(1); //exit the program
}

input.close(); //close the input stream

if(debug) //MESSAGE: Filled probability HT
    cout << "Filled probability hash table...\n";

//create a vector of words in order by probability
tvector<Word_data> in_order = prob_ht.order_prob(filename);

if(debug) //MESSAGE: Filled probability HT
    cout << "Probabilities sorted in order...\n";

prob.extract_sig_prob(in_order); //extract significant probabilities
//to calculate the overall probability

if(debug) //MESSAGE: Extract significant probabilities
    cout << "Significant probabilities extracted...\n";

if(debug) {
    cout << "Final Prob: " << prob.calc_final_p() << "\n"; //calculate the final
prob and print
    cout << "Final Probability calculated.\n\n"; //MESSAGE: Final Probability
}

```

```
    else {
        fstream file_op("probOutput.dat",ios::out); //open a dat file for writing
        file_op << prob.calc_final_p(); //calculate the prob and write
to the file
        file_op.close(); //close the file
    }

}
else {
    cout << "Bayes_filter: Insufficient arguments\n"; //wrong number of command line
arguments entered
    cout << "Usage: Bayes_filter [file] (*|debug)\n\n"; //display usage
}
return 0;
}
```

B.2 Nu_hashtable class

The purpose of this class is to extend the functionality of a normal hash table.

```

/*****
Jonathan Bulava
Filtering Research
Dr. Peter DePasquale
Nu_hashtable.cpp
*****/

/*****
This class extends the functionality of a custom
hashtable class that is much simpler.
*****/

#include "Nu_hashtable.h"
#include <string>
#include "tvector.h"
#include "Word_data.h"
#include <fstream>
#include <iostream>
using namespace std;

/*****
Nu_hashtable constructor.
*****/
Nu_hashtable::Nu_hashtable()
: total_occ(0),
  web_num(0)
{
}

/*****
This function takes the word it was passed and, if it
does not already exist in the
hashtable, puts it
into a Word_data obj w/ count=1, and inserts into a
hashtable.
If the word already exists, it
increments the count# of the existing Word_data obj.
*****/
void Nu_hashtable::insert_w(string s) {

    Word_data word_obj(s);

    //for now, assume every word hashes uniquely
    if(wd_hash_table.contains(s)) {

        //set word_obj equal to the object removed at that key
        word_obj = wd_hash_table.remove(s);
        word_obj.increment_count(); //increase the count #
        wd_hash_table.put(s,word_obj); //insert the updated word_obj
    }
    else { //if the word is the first of
           //its kind, insert with count=1
        wd_hash_table.put(s,word_obj);
    }
}

```

```

}

/*****
  INSERTED IN CASE THIS FUNCTIONALITY
  IS NEEDED IN FUTURE DEVELOPMENT.

  This method opens up the specified file, and
  tokenizes and removes each word.
  If count = 1, the
  entire Word_data obj of that word is removed.
  Otherwise, the count number
  is decremented.
*****/
void Nu_hashtable::remove_w(string file) {

}

/*****
  This function tokenizes the words in the specified
  file and inserts into the hashtable.
  It calls insert_w() to do the actual insertion.
*****/
void Nu_hashtable::fill_word(string file) {

  string word;
  ifstream input;
  input.open(file.c_str());

  while (input >> word) {
    insert_w(word);
  }

  input.close();
  web_num++; //increment the number of webpages entered into this hashtable

}

/*****
  This function returns the number of webpages that
  were inserted into the hashtable
*****/
int Nu_hashtable::get_web_num() {
  return web_num;
}

/*****
  INSERTED IN CASE THIS FUNCTIONALITY
  IS NEEDED IN FUTURE DEVELOPMENT.

  This function returns the total number of occurances
  of every word in the hashtable. In other words, it
  adds up the count # of every word. Since
  probability hashtables do not keep track of count
  #s, there is no point to use this method of them.
*****/
int Nu_hashtable::get_total_occ() {

}

```

```

/*****
  INSERTED IN CASE THIS FUNCTIONALITY
  IS NEEDED IN FUTURE DEVELOPMENT.

  This function sorts a hashtable into alphabetical
  order.  It returns the ordered hashtable as a
  Vector.
*****/
tvector<Word_data> Nu_hashtable::order_hash() {
}

/*****
  This function returns the hashtable as a numerically
  (by probability) ordered vector.
*****/
tvector<Word_data> Nu_hashtable::order_prob(string filename) {

  string word;
  ifstream input;
  input.open(filename.c_str());

  //get a vector of the website text without duplicates
  tvector<string> used_words(1000);
  int used_index = -1;

  //while there are still words, place them in a vector
  //without any duplicates
  int item_count = 1;

  while((input >> word) && (item_count <= 1000)) {
    bool duplicate = false;

    for(int i = 0; i <= used_index; i++) {
      if(used_words[i] == word) {
        duplicate = true;
      }
    }
    if(!duplicate) {
      used_index++;
      used_words[used_index] = word;
    }
    item_count++;
  }

  if(item_count > 1000) {
    cout << "Error: Vector limit of 1000 exceeded.\n";
    exit(1);
  }

  //place all the corresponding word_data objects into a
  //vector to be returned

  tvector<Word_data> wd_vector(used_index+1);
  Word_data wd;

  for(int i = 0; i <= used_index; i++) {

```

```

    wd = wd_hash_table.get(used_words[i]);
    wd_vector[i] = wd;
}

//sort the vector by probabilities

tvector<Word_data> wd_temp(used_index+1);
int size = 0;

for(int j = 0; j <= used_index; j++) {
    if(size > 0) {
        wd_temp[size] = wd_vector[size];
        int marker = size;

        while(wd_temp[marker].get_prob() < wd_temp[marker-1].get_prob()) {
            Word_data w = wd_temp[marker-1];
            wd_temp[marker-1] = wd_temp[marker];
            wd_temp[marker] = w;
            marker--;
        }

        if(marker == 0)
            cout << "0 marker reached\n";
            break;
    }
}
else {
    wd_temp[0] = wd_vector[0];
}
size++;
}
return wd_temp;
}

```

```

/*****
Prints out the vector it was passed. This function
is specifically for printing vectors of (word&count)
hashtables, b/c the print_word_data() function that
is called within only prints out words and counts.
If this function was used for a probability vector,
all count # would equal 0.
*****/
void Nu_hashtable::print_in_order(tvector<Word_data> v) {
    for(int i=0; i<v.size(); i++) {
        ((Word_data)v[i]).print_word_data();
    }
}

```

```

/*****
Prints out the Vector it was passed. This function
is specifically for printing vectors of (word&prob)
hashtables, b/c the print_word_data2() function that
is called within prints out only words and
probabilities.
*****/
void Nu_hashtable::print_in_order2(tvector<Word_data> v) {
    for(int i=0; i<v.size(); i++) {
        ((Word_data)v[i]).print_word_data2();
    }
}

```

```

}

/*****
  INSERTED IN CASE THIS FUNCTIONALITY
  IS NEEDED IN FUTURE DEVELOPMENT.

  Serializes the current hashtable into a file.  The
  string argument, name, is the name of the file that
  the hashtable is supposed to be serialized into.
*****/

void Nu_hashtable::serialize(string name) {

}

/*****
  This function is used to insert a word object in
  the hashtable with its string as the key.
*****/
void Nu_hashtable::insert(string key, Word_data wd) {
  wd_hash_table.put(key, wd);
}

/*****
  Returns the number of elements in the hashtable.
*****/
int Nu_hashtable::size() {
  return wd_hash_table.size();
}

/*****
  Returns true if the key is in the hashtable.
*****/
int Nu_hashtable::contains(string key) {
  return wd_hash_table.contains(key);
}

/*****
  Returns the word object mapped to the given key.
*****/
Word_data Nu_hashtable::get(string key) {
  return wd_hash_table.get(key);
}

```

B.3 Hash_table class

A custom hash table class created to be used only for Word_data objects.

```

/*****
    Jonathan Bulava
    Filtering Research
    Dr. Peter DePasquale
    Hash_table.cpp
*****/

/*****
    A custom hashtable class created for use only with
    Word_data objects.
*****/

#include "Hash_table.h"
#include "Word_data.h"
#include <string>

// #define READBUFLLEN (2048)
// static char buf[READBUFLLEN];
// static char buf2[READBUFLLEN];

int Hash_table::magic = 'N' | ('H'<<8) | ('S'<<16) | ('H'<<24);

/*****
    Hashtable constructor.
*****/
Hash_table::Hash_table()
{}

/*****
    Returns the Word_data object that is mapped to the
    given key.
*****/
Word_data Hash_table::get(string key) {
    if (key == "") {
        cout << "Hash_table::get(): null string key";
    }

    if (swmap.find(key) == swmap.end()) {
        Word_data empty;
        return empty;
    }
    else {
        return swmap[key];
    }
}

/*****
    Returns the first element in the hashtable.
*****/
string Hash_table::first_key() {
    map<string, Word_data>::iterator it = swmap.begin();

    if (it == swmap.end()) {
        return 0;
    }
    else {

```

```

    return it->first.c_str();
}
}

/*****
Returns the next element in the hashtable.
*****/
string Hash_table::next_key(string key)
{
    map<string,Word_data>::iterator it = swmap.find(key);
    if (it == swmap.end()) {
        return 0;
    } else {
        it++;
        if (it == swmap.end()) {
            return 0;
        } else {
            return it->first.c_str();
        }
    }
}

/*****
Places the Word_data object in the hashtable using
the given string key.
*****/
void Hash_table::put(string key, Word_data wd)
{
    if (key == "") {
        cout << "Hash_table::put(): null string key";
    }

    swmap[key] = wd;
}

/*****
Removes the element at the given key and the key
itself.
*****/
Word_data Hash_table::remove(string key)
{
    Word_data wd = swmap[key];
    swmap.erase(key);
    return wd;
}

/*****
Returns the number of elements in the hashtable.
*****/
int Hash_table::size() const {
    return swmap.size();
}

/*****
Returns true if the key is in the hashtable.
*****/
int Hash_table::contains(string key) {

```

```
return swmap.find(key) != swmap.end();  
}
```

B.4 Probability class

This class uses Bayes' Theorems to calculate the probability of a website containing inappropriate content.

```

/*****
    Jonathan Bulava
    Filtering Research
    Dr. Peter DePasquale
    Probability.cpp
*****/

/*****
    This class is used to calculate the probability of
    a website containing inappropriate content. It
    utilizes Bayes' Theorems to acheive this.
*****/

#include "Probability.h"
#include "Nu_hashtable.h"
#include "tvector.h"
#include "randgen.h"
#include <string>
#include <cmath>
using namespace std;

/*****
    This constructor makes copies of the current
    hashtables for its use.
*****/
Probability::Probability(Nu_hashtable g, Nu_hashtable b)
: count_g(0),
  count_b(0)

{
    good = g;
    bad = b;
}

/*****
    This function extracts the necessary info for each
    of the variables that are used to calculate a word's
    P(B|W) probability.
*****/
void Probability::fill_var(string w) {

    num_good = good.get_web_num(); // # webpages in good hashtable
    num_bad = bad.get_web_num(); // # webpages in bad hashtable
    good_total = good.size(); // # words in good hashtable
    bad_total = bad.size(); // # words in bad hashtable

    //if the good hashtable has the word at that key
    if(good.contains(w)) {
        count_g = ((Word_data)good.get(w)).get_count();
    }
    else {
        count_g = 0; //to make sure old values of count_g don't persist
    }
}

```

```

if(bad.contains(w)) {
    count_b = ((Word_data)bad.get(w)).get_count();
}
else {
    count_b = 0; //to make sure old values of count_b don't persist
}

}

/*****
This function calculates the P(W|B) part of Baye's
Theorem; i.e., it calculates the probability of a
word W appearing in the bad corpus. This means
divide the number of times the word appears by the
total number of words in the bad hashtable.
*****/
double Probability::PWB() {
    return count_b/bad_total;
}

/*****
This function calculates the P(B) part of Baye's
Theorem; i.e., it calculates the probability of Bad
webpages appearing in general. This means divide
the number of bad webpages inserted into the bad
hashtable by the total number of webpages inserted
in both good and bad hashtables.
*****/
double Probability::PB() {
    return num_bad/(num_bad+(num_good*2)); //bias ngood by multiplying by 2
}

/*****
This function calculates the P(W|G) part of Baye's
Theorem; i.e., it calculates the probability of a
word W being in the Good corpus. This is just
dividing the number of times a word appears in the
good hashtable by the total number of words in the
good hashtable.
*****/
double Probability::PWG() {
    return (count_g*2)/(good_total*2); //bias all good numbers by 2
}

/*****
This function calculates the P(G) part of Baye's
Theorem; i.e., it calculates the probability of a
webpage being good. This is done by dividing the
number of good webpages inserted into the good
hashtable by the total number of webpages inserted
into both hashtables.
*****/
double Probability::PG() {
    return (num_good*2)/(num_bad+(num_good*2)); //bias all good numbers by 2
}

```

```

/*****
  This function calculates the P(B|W) part of Baye's
  Theorem; i.e., it calculates the probability of a
  webpage being bad if it contains the word W.
*****/
double Probability::calc_prob() {
  //calculates the probability of the word being in spam
  word_prob = (PWB()*PB())/(PWB()*PB() + PWG()*PG());

  //if word is neither in good nor in bad corpus
  if((PWB() == 0) && (PWG() == 0)) {
    word_prob = 0.4;
  }

  //if word is only in bad or only in good, assign a probability
  // to keep 0's and 1's from saturating equation
  if(word_prob == 1) {
    word_prob = 0.99;
  }

  if(word_prob == 0) {
    word_prob = 0.01;
  }

  return word_prob;
}

/*****
  This function extracts 20 "significant"
  probabilities, i.e., probabilities farthest from
  neutral 0.5, and puts them into an array. It starts
  out by taking the first and last entries in the
  vector (the smallest and largest entries in the
  vector, respectively), and after comparing them,
  takes the one with the largest difference from 0.5.
  If neither entry is larger, then one of them is
  randomly chosen.
*****/
void Probability::extract_sig_prob(tvector<Word_data> v) {

  Word_data temp_f;          //temp holder for entry from front end
  Word_data temp_b;          //temp holder for entry from back end
  int counter_f = 0;         //counter for the front end of the vector
  int counter_b = v.size()-1; //counter starting at the back end of the vector

  if(v.size() < 40) {
    cout << "\nError:\n Website size too small to calculate probability\n";
    cout << " Only " << v.size() << " words found from site.\n\n";
    exit(1);
  }
  else {
    for(int g=0; g<20; g++) { //until 20 sig probabilities are found

      //take an entry from the front and the end of the vector and compare
      temp_f = v[counter_f];
      temp_b = v[counter_b];

      if(fabs(0.5 - temp_f.get_prob()) > fabs(temp_b.get_prob() - 0.5)) {

```

```

        //if the difference is greater for temp_f, temp_f is chosen
        sig_w[g] = temp_f;           //enter temp_f into array
        sig_p[g] = temp_f.get_prob(); //enter the probability into array
        counter_f++;                 //increment its counter to the next entry in
front
    }
    else if(fabs(0.5 - temp_f.get_prob()) < fabs(temp_b.get_prob() - 0.5)) {
        sig_w[g] = temp_b; //temp_b is farther from 0.5, it is chosen and added into
array
        sig_p[g] = temp_b.get_prob();
        counter_b--; //decrement its counter to the previous largest entry in vector
    }
    else if(fabs(0.5 - temp_f.get_prob()) == fabs(temp_b.get_prob() - 0.5)) {
        //if the differences are equal, choose by generating a random number.
        //if the number is less than 5, choose tempF, otherwise choose tempB
        //This is done because if only front or back entries were chosen in cases
        //where the differences are equal, it may throw the final verdict too
heavily
        //in one direction or the other

        if(gen.RandInt(1,10) < 5) { //if number is less than 5
            sig_w[g] = temp_f; //temp_f is chosen, enter into array
            sig_p[g] = temp_f.get_prob();
            counter_f++; //increment its counter to the next entry in front
        }
        else { //if number is greater than or equal to 5
            sig_w[g] = temp_b; //choose temp_b
            sig_p[g] = temp_b.get_prob();
            counter_b--; //decrement its counter to the previous largest entry in
vector
        }
    }
}
}
}
}
}

```

```

/*****

```

This function calculates the final probability of a page being bad. It uses Bayes' Theorem using the form:

$$\frac{a*b}{a*b + (1-a)(1-b)}$$

where a and b are individual probabilities.

```

*****/

```

```

double Probability::calc_final_p() {

```

```

    double num = sig_p[0];
    double denom = 1 - sig_p[0];

```

```

    //get the value of the numerator: a*b*c*d...etc
    for(int i=1; i<20; i++) {
        num = num * sig_p[i];
    }

```

```

    //get denominator value: num + denom

```

```

for(int b=1; b<20; b++) {
    denom = denom * (1-sig_p[b]);
}

denom = denom + num;
return num/denom;
}

/*****
This function prints out the 20 most significant
words. More specifically, it prints out the
Word_data objects that were entered into the array
sig_w. This method should be called only after
extract_sig_prob() is called.
*****/
void Probability::print_sig_prob() {

    Word_data temp;
    cout << "Most Significant Words:";

    //print out the items in the array sigW
    for(int g=0; g<20; g++) {
        cout << g << " " << sig_w[g].get_word() << '\t' << sig_w[g].get_prob();
    }
}

```

B.5 Word_data class

The function of this class to keep track of the words extracted from a webpage or the words determined to be of good or bad content. Each instance holds a word and its probability or frequency.

```

/*****
Jonathan Bulava
Filtering Research
Dr. Peter DePasquale
Word_data.cpp
*****/

/*****
The function of this class to keep track of the
words extracted from a webpage or the words
determined to be of good or bad content. Each
instance holds a word and its probability or
frequency.
*****/

#include "Word_data.h"
#include <string>
using namespace std;

/*****
This constructor initializes a word to nothing, and
its count to zero.
*****/
Word_data::Word_data()
: word(" "),
  count(0)
{}

/*****
This constructor initializes a word to word_in, and
its count to 1, since the only time a new Word_data
obj will be created is when a word that is not in
the hashtable already is parsed.
*****/
Word_data::Word_data(string word_in)
: word(word_in),
  count(1)
{}

/*****
This constructor initializes a word to word_in, and
prob to p.
*****/
Word_data::Word_data(string word_in, double p)
: word(word_in),
  prob(p)
{}

/*****
This function returns the word.
*****/
string Word_data::get_word() {
    return word;
}

```

```

}

/*****
  This function returns a word's count number.
*****/
double Word_data::get_count() {
  return count;
}

/*****
  This function returns a word's probability.
*****/
double Word_data::get_prob() {
  return prob;
}

/*****
  This function increments the count number by 1.
  This is done when a word has been parsed more than
  once into the same hashtable.
*****/
void Word_data::increment_count() {
  count++;
}

/*****
  This function decrements the count number by 1.
  This is done when a word existing in a hashtable
  has been removed.
*****/
void Word_data::decrement_count() {
  count--;
}

/*****
  This function prints out a word and its count
  number.
*****/
void Word_data::print_word_data() {
  cout << word << "\t\t" << count << endl;
}

/*****
  This function prints out a word and its
  probability.
*****/
void Word_data::print_word_data2() {
  cout << word << "\t\t" << prob << endl;
}

```

Appendix C

Here is the code for the PHP application used to create files containing a given website's textual information. The section of code used to parse the page was previously written.

```
<html>
<head>
<title>Untitled Document</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1">
</head>
<body>

<?
    ini_set('display_errors', 1);
    error_reporting(E_ALL & ~E_NOTICE);
    include("htmlparser.inc");

    if (isset ($_POST['submit'])) {
        $timestamp1 = time();
        $filename = $_POST['name'];
        $url = $_POST['url'];
        $lines_string = "";

        $handle = fopen($url,"r");

        while (!feof($handle)) {
            $buffer = fgets($handle, 256);
            $lines_string = $lines_string . $buffer;
        }
        fclose($handle);

        $wordArray = array();
        $inScript = false;
        $inStyle = false;

        $webpageText = array();
        $parser = new HtmlParser($lines_string);

        while ($parser->parse()) {
            if(strcasecmp($parser->iNodeName,"script")==0){//determine if inside or outside
a script tag
                if($parser->iNodeType==1){
                    $inScript = true;
                }

                if($parser->iNodeType==2){
```

```

        $inScript=false;
    }
}

if(strcasecmp($parser->iNodeName,"style")==0){//determines if inside or outside
a style tag

    if($parser->iNodeType==1){
        $inStyle = true;
    }
    if($parser->iNodeType==2){

        $inStyle = false;
    }
}

if ($parser->iNodeType == NODE_TYPE_TEXT && !$inScript && !$inStyle){//if text
is identified not in style or script tags

    $sValue = trim($parser->iNodeValue);

    $wordArray = explode(" ", $sValue);//break up text into array of words

    for($counter=0;$counter<count($wordArray);$counter++){

        $wordArray[$counter] =          =          ereg_replace('[^a-zA-Z]|\nbsp;','',$wordArray[$counter]);//get rid of non-alphabetic characters

        if($wordArray[$counter]!="" && strlen($wordArray[$counter])>2) {//eliminate
null string and strings less than 3 letters

            $webpageText[] = strtolower($wordArray[$counter]);//take out caps for
uniformity
        }
    }
}

if($parser->iNodeType == NODE_TYPE_ELEMENT){

    if(strcasecmp($parser->iNodeName,"META")==0){//take out meta tag information

        if(array_key_exists("content",$parser->iNodeAttributes))
{

            $val = $parser->iNodeAttributes["content"];
            $sValue = trim($val);

```

```

$wordArray = explode(" ", $sValue);

for($counter=0;$counter<count($wordArray);$counter++){

    $wordArray[$counter] = ereg_replace('[^a-zA-Z]|nbsp', '', $wordArray[$counter]);

    if($wordArray[$counter]!=" " && strlen($wordArray[$counter])>2) {
        $webpageText[] = strtolower($wordArray[$counter]);
    }
}

}

}

}

if($parser->iNodeType == NODE_TYPE_ELEMENT){ //gets alt tag information from
img, input, and area tags
    if(strcasecmp($parser->iNodeName, "img")==0 || strcasecmp($parser-
>iNodeName, "input")==0 || strcasecmp($parser->iNodeName, "area")==0) {

        if(array_key_exists("alt", $parser->iNodeAttributes)){
            $sval = $parser->iNodeAttributes["alt"];
            $sValue = trim($sval);
            $wordArray = explode(" ", $sValue);

            for($counter=0;$counter<count($wordArray);$counter++){
                $wordArray[$counter] = ereg_replace('[^a-zA-Z]|nbsp', '', $wordArray[$counter]);
                if($wordArray[$counter]!=" " && strlen($wordArray[$counter])>2) {
                    $webpageText[] = strtolower($wordArray[$counter]);
                }
            }
        }
    }
}

}

}

$timestamp2 = time();
$totaltime = $timestamp2 - $timestamp1;

$filename = $filename . ".txt";

if($fp = fopen($filename, 'w')) {
    for($i = 0; $i < sizeof($webpageText); $i++) {
        fwrite($fp, $webpageText[$i] . "\n");
    }
}

```

```
    }
    fclose($fp);
    print "<font color=\"#FF0000\">Parse Successful<br>";
    print "Time: " . $totaltime . " seconds</font><br><br>";
    $array = file($filename);
    print "Number of tokens: " . sizeof($array) . "<br><br>";
    print "<a href=\"\" . $filename . "\">View " . $filename . "</a><br><br><hr>";

}
else {
    print "<font color=\"#FF0000\">Error</font>";
}
}

?>
<h1>HTML Parser</h1>

<form method="POST" action="phpparser.php">
URL:<input type="text" name="url" value="http://" size="100"><br><br>
Filename:<input type="text" name="name" size="20"><br><br>
<input type="submit" name="submit" value="Submit Form">
<input type="reset" value="Reset Form">
</form>

</body>
</html>
```