WikiTruthiness

www.wikitruthiness.com

Katherine Baker
David Koenig
Aaron Miller
Cullen Walsh

CSE 454, Advanced Internet and Web Services
December 17, 2010

1. Overview

WikiTruthiness is a novel web service which highlights contentious content in Wikipedia articles. WikiTruthiness has benefits for all levels of Wikipedia users:

- **Readers** can use WikiTruthiness to question existing information in an article.
- **Editors** can use WikiTruthiness to clean up existing articles.
- **Administrators** can use WikiTruthiness to streamline their regular article clean-up tasks.

2. Goals

The main goal of our project is to analyze the contention level of Wikipedia content on an article-by-article basis and display analyzed articles in such a fashion to clearly show which parts of an article are contentious and which are relatively stable. From our specification: "Wikipedia allows users the ability to get the full history of any article on the site, and we are interested in seeing what paragraphs (or possibly sentences) have seen the most reversions, edit wars, or other indicators that a certain piece of information is contentious." Secondary goals included satisfactory performance of the application (with requests handled within ten seconds in the worst case) and creating a sufficiently easy to use application.

3. System Design and Algorithmic Choices

The system follows the high-level flow diagram in Figure 1. The system modules split into front-end and back-end segments, with "middleware" covering areas of interaction between the two segments.

3.1. Front-End Modules. As shown in Figure 1, the user starts at the home page, which contains information about our project and a search box powered by Google's Custom Search API. Performing a search takes the user to the search results for their query – a modification of the Google results which prominently lists links to the WikiTruthiness results as well as links to the original Google results. Through the Google Custom Search API, we limit all results to Wikipedia articles.

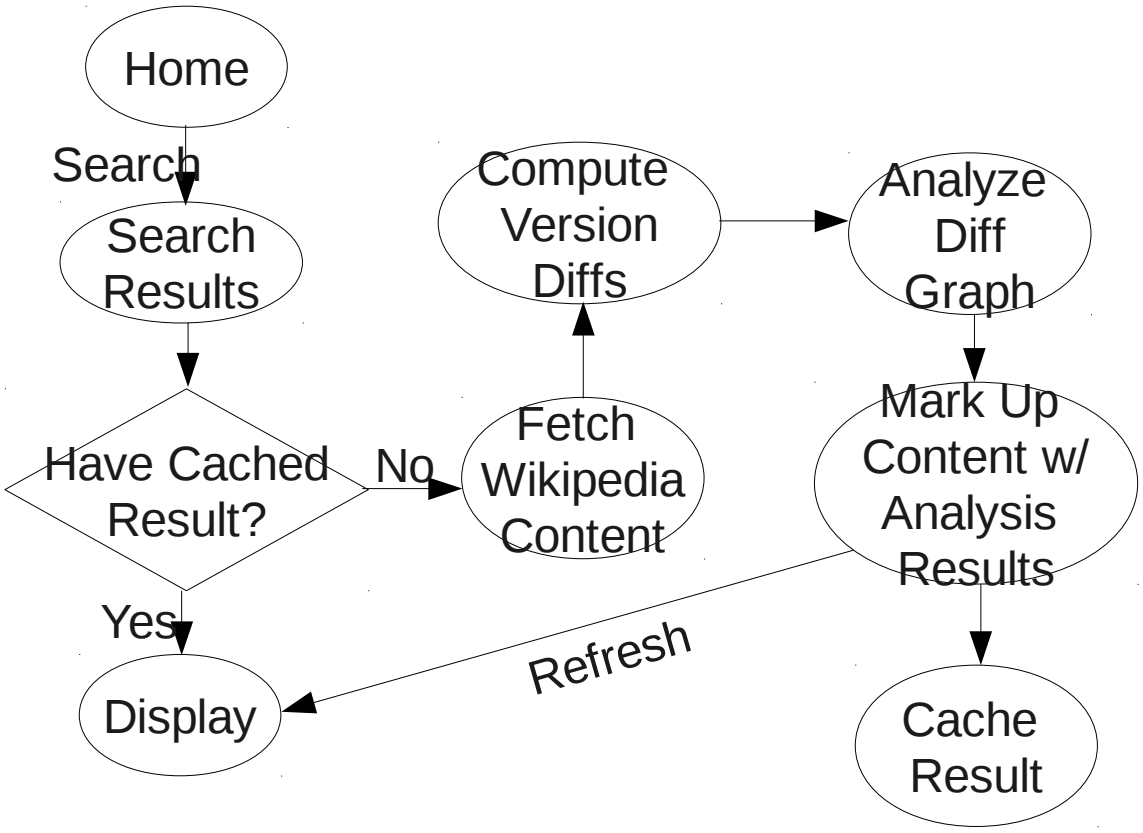

Figure 1. The flow diagram of WikiTruthiness from starting at the home page http://www.wikitruthiness.com and ending with display of the analyzed text.

When the user selects a result, the page module first looks at our data caches (a local instance of memcached as well as Amazon S3) to determine if the Wikipedia article has already been analyzed. If so, then that result is displayed; otherwise, the unanalyzed version of the Wikipedia page is displayed and a request to grab and analyze the Wikipedia article is appended to a back-end queue.

The front-end comes back into play after the user refreshes his request for the analyzed Wikipedia article. With the refresh request, the front-end grabs the analyzed graph with the marked up content (for description of the markup, see the next section) and displays the content via the Mediawiki

API. We utilized Ruby on Rails for all of the front-end work.

3.2. Back-End Modules. The application uses the back-end modules when a currently unanalyzed article is added to the task queue by inserting data about the article (timestamp of request and title) into the database. When a thread is available, the thread grabs the data about the next unanalyzed article from the database and parses the HTML code into nodes. Nodes are based on a size no greater than a sentence (they will break at end-of-sentence or upon encountering any HTML markup). Then the graph module computes the difference between the versions of the article on a node by node basis, based entirely on the last 30 revisions to the article. During this computation, the graph module also determines the nodes' "scores". A score is determined by what occurs to the node at each revision – the longer a node (and its neighbors) remain unchanged, the lower the node's score goes. Two different versions of the algorithm were tested, with the latter providing results indicating a more realistic, lower-level of contention in articles.

```
Each node's initial score = 1.0
For each revision:
        If node replaced:
                score *= 1.05;
        Else if node deleted:
                previous node's score *= 1.1;
        Else:  //the node stayed the same
                score *= 0.97;
```

Figure 2.a. Pseudocode of the initial scoring algorithm

```
For each revision:
        score * = Math.pow(1.03, count_node_unchanged);
        score *= Math.pow(0.92, count_node_readded);
```

Figure 2.b. Pseudocode of final scoring algorithm

From a score, the level of coloring of a node on the website can be determined. We encoded the coloring scheme in HTML tags including a span class distinction from trust level 1 to trust

level 10 (in what initially appears to be a backwards ranking – a trust10 is the highest level of contention and therefore is highlighted with the darkest hue of orange , and a trust1 is the lowest level with little to no contention and therefore no coloration—this was done so that we could use the same CSS styling as *WikiTrust*). The algorithm in Figure 3 determines the mapping from score to trust level. In this algorithm we perform smoothing on the scores to keep the trust level results within certain bounds to avoid all nodes being marked as trust10 and a page filled with high coloration. We discovered that due to the relatively small sample size of revisions (30) and the resulting limited movement of scores in the nodes, without score and trust level smoothing, all nodes appeared to be extremely recently added and therefore stuck with high trust level scores.

```
//place boundaries of min and max where most results should lie
trustLevel = (node score – min) / (max – min) * 10 + 1;
If ( trustLevel > 10) trustLevel = 10;
If  (trustLevel < 1) trustLevel = 1;
```

Figure 3. Pseudocode for translation from node score to trust level for adding coloring to article.

The back-end modules were initially written completely in Python. However, in a search to find a more efficient and accurate mapping from score to trust level, the score mapping module was rewritten in Java. The rewritten component yielded significant improvements in analysis execution speed – from several minutes for large articles to approximately 60 seconds – and reduced the number of nodes labeled as trust10 (the highest level, showing the most contention) to more realistic levels.

3.3. Interaction Between Front-End and Back-End. The front-end and back-end interacted through multiple pieces of technology. The front-end code made a direct request to the back-end code via HTTP to kick-start the back-end retrieval of the 30 most recent versions of the article and the subsequent analysis. We utilized a MySQL database to queue these requests and store those versions of requested Wikipedia articles. To speed up our processing, we cached our results in two locations which

both the front-end and back-end hit – (1) in a local instance of memcached and (2) Amazon Web Services' S3 key-value store to cache all of our results. Both sides were supported by AWS Elastic Compute Cloud with a small instance to run the front-end code and a high instance for analysis. We used Git to manage our source code.

4. Sample Screenshots of Typical Usage Scenarios

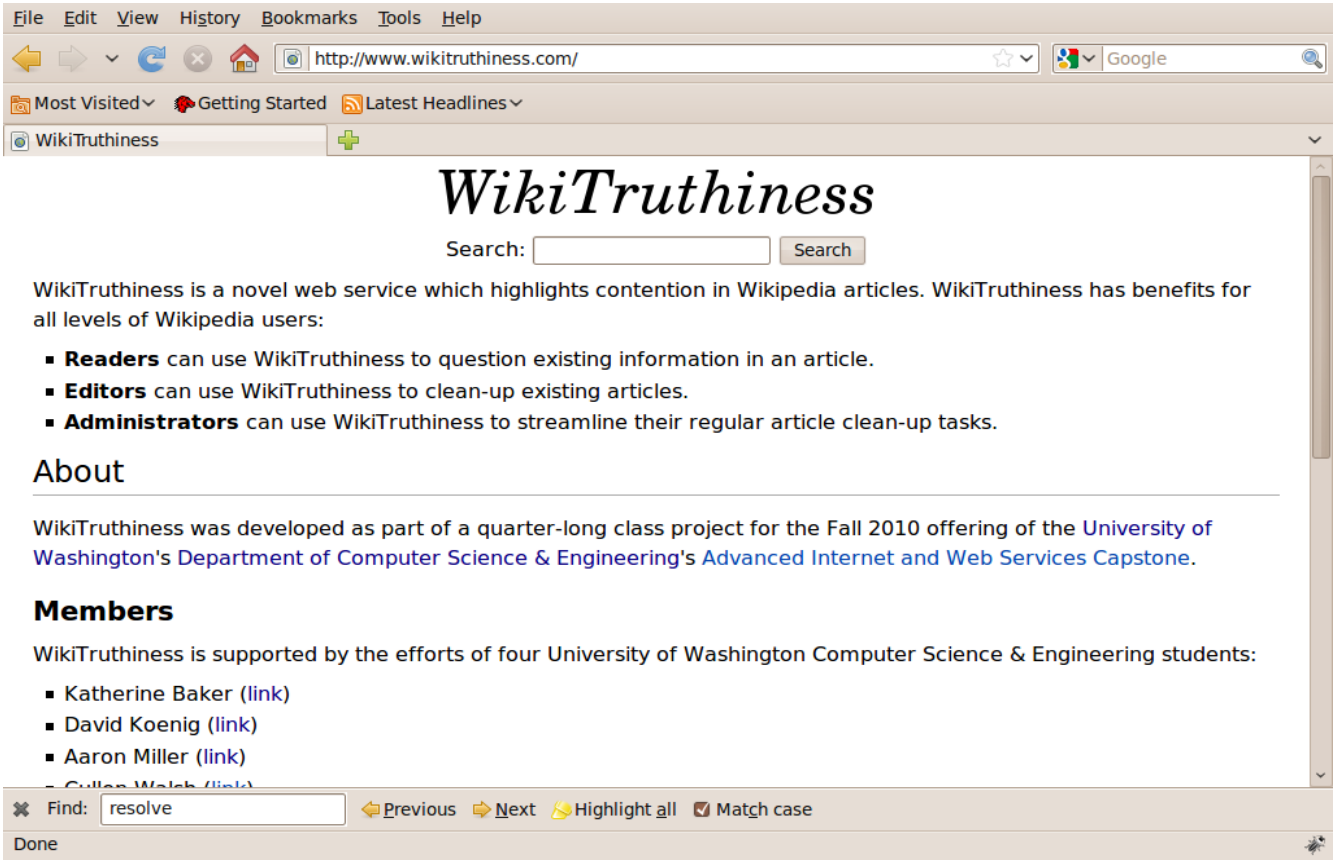

Figure 4. WikiTruthiness home page.

One of the goals for the project was to keep the site easy to use. On the home page, we provide a simple search box (like Google) and a description of our project about why our results would be interesting to different Wikipedia users.

Figure 5. The search results page.

After a user enters a search term (here, it is “slackware”), the specialized Google search is run and our results, along with the Google results for the search are displayed (in concurrence with the agreement to use the Google Search API). Again, on this page, the UI is basic and easy to understand.

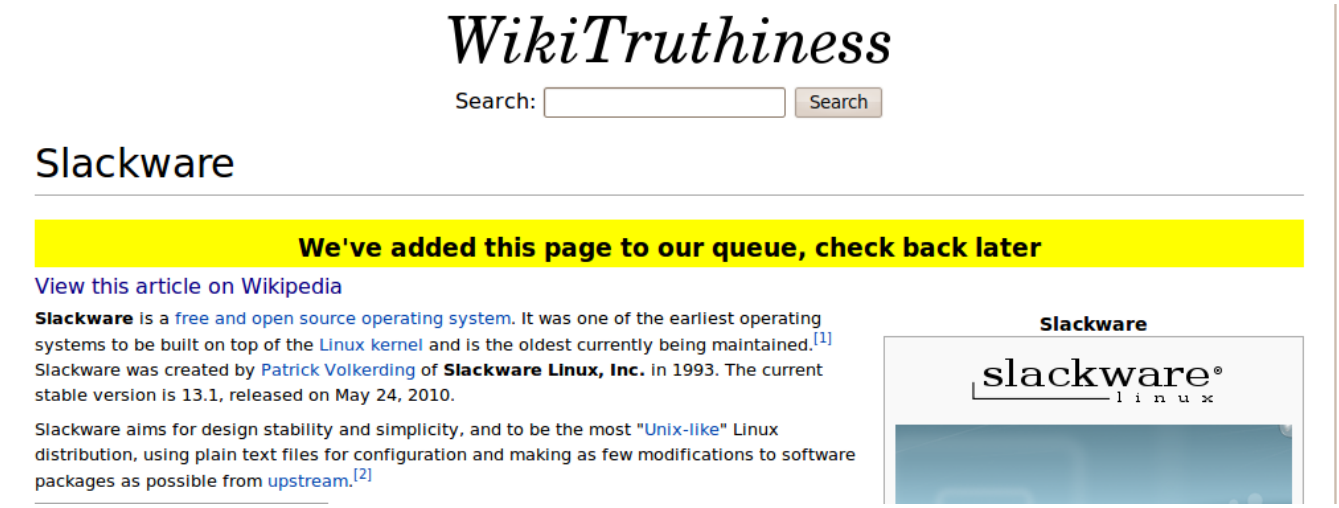

Figure 6. The Wikipedia result shown when the given article has not yet been analyzed.

Figure 7. The analyzed result.

5. Experiments and Results

Since our results are of a subjective nature, we decided to evaluate the "correctness" of our system against existing work for similar applications. *WikiTrust* (http://www.wikitrust.net), a product from The Institute for Scalable Scientific Data Management at the University of California, Santa Cruz, highlights untrustworthy words within a Wikipedia article. *WikiTrust* computes the trustworthiness of each word in an article by examining the previous six versions of the article and aggregating statistics about the differences between consecutive version pairs into tuples. *WikiTrust* then uses a trained data classifier to determine how untrustworthy (on a scale from 0 – 10) each word in the initially requested article version is.

We leverage *WikiTrust*'s Firefox Web browser extension for initiating requests to their service for a Wikipedia article and their web API for retrieving the HTML-formatted output of their service for an article. We extend the open-source HTMLParser Java framework for evaluating our results for a Wikipedia article versus those from *WikiTrust* for the same article. By counting the number of potentially untrustworthy words identified by *WikiTrust* which fall within our hypothesized blocks of contentious content we are able to compute the number of true positives, false positives and false negatives our system generates with respect to the results from *WikiTrust*. These metrics are defined as follows.

- **True Positives:** Count of our identified blocks of contention which contain at least one identified word from *WikiTrust*.
- **False Positives:** Count of our identified blocks of contention which do not contain any identified words from *WikiTrust*.
- **False Negatives:** Count of words identified by *WikiTrust* which do not fall within our identified blocks of contention.

Upon computing these metrics for an article, we are able to compute our precision and recall against *WikiTrust*'s results by applying the following formulae.

Precision = True Positives / (True Positives + False Positives)
Recall = True Positives / (True Positives + False Negatives)

By evaluating a well-mixed set of 33 Wikipedia articles, our system experienced an average precision of 20.25% against *WikiTrust* and an average recall of 68.93% against *WikiTrust*. More detailed results are summarized in the table in Figure 8.

	Precision	Recall
Worst	10.84%	52.43%
Average	20.25%	68.93%
Best	38.82%	79.37%

Figure 8. Table of precision and recall against *WikiTrust* results.

6. Surprises and What We Learned

We had several surprises and setbacks along the way from conception of the idea to final result. Devising and then refining the algorithm to do the analysis of the difference between the article versions and the translation of that analysis into a coloring scheme for the web page took significantly longer than we expected. Figuring out a workable level of granularity for analysis (paragraph v. sentence v. word) and then determining how to account for all of the changes from one version to the next (particularly in regards to deleted nodes) became a major challenge.

The importance of maintaining cache coherency across the local cache, AWS S3, and the MySQL database was another surprise. Without cache coherency, we ran the risk of overwriting data that was in the local cache prior to storing it in AWS S3. We dealt with this situation by checking every time an article is displayed that the local cache and AWS S3 contained the same material.

We experienced noticeable difficulty in comparing the results of our work with those from *WikiTrust*. *WikiTrust* employs a different method than ours for rendering their output into an HTML

format. After much debate, we settled on extracting the text content from the results from *WikiTrust* and our own results for an article and storing the starting and ending positions of highlighted words in both results for comparison. The delay in finalizing our results comparison algorithm and implementation significantly limited the number of articles we could evaluate our system with.

There is much variation in diff libraries as well. We had significant variances between diffs returned by difflib (part of the standard Python library) and equivalent Java libraries, and even between different runs of the Java library. (The Java library contains several difference algorithms and tries them all until it reaches a timeout, then picks the best result.) One could even conceive of a pathologically stupid difference algorithm that just says “delete all the items in the old sequence and replace them with the items in the new sequence”—technically it would return a correct result, but our algorithm would not return anywhere near the correct result.

In the end, our bottleneck ended up being retrieving revisions from Wikipedia. Each revision requires a separate API request to their servers, which we rate limited to 1 request per second (as permitted via their robots.txt file). If we were able to improve this retrieval speed, or work with a local file cache, we would greatly improve the latency of our analysis

With the many challenges encountered during the development process, we learned numerous things, including:

- Mixing technologies and having them interface smoothly is difficult. We figured out how to communicate with the S3 cache from both Ruby on Rails and Python code and how to communicate with the MySQL database from both Java and Python code. In hindsight, S3 and MySQL were both excellent choices of infrastructure due to their being very well supported by all platforms used in the project.
- The importance of choosing a development language both for performance issues and for code

readability and debugging concerns. We found that the Python code ran slower than Java code computing the same version analysis and ran into problems trying to debug the Python code because the other group members did not have enough familiarity with Python.

7. Ideas For Future Work

With additional time, several improvements to the current project are feasible. First, by grabbing additional versions of the article for the analysis we could provide a longer term view of the contention level of the article. For an improved user experience in regards to performance, we could prefetch and analyze articles linked from the currently viewed article. Even if the user did not end up selecting a link, having the data already fetched and analyzed provides better performance for subsequent users. Also, the further refinements to the contention algorithm are appropriate.

8. Conclusions

As a group, we found the project interesting and challenging. Utilizing S3 and EC2 instances at no cost was a great benefit, as it improved the performance of our end product. We are pleased with the results, although regret not having additional time to improve upon the contention determining algorithms.

9. Appendix

9.1. Mapping Group Members to Work

Katherine Baker	Comparative Evaluation against WikiTrust
David Koenig	Code for back-end
Aaron Miller	Comparative Evaluation against WikiTrust
Cullen Walsh	Code for front-end

All group members worked on coding, evaluation, and devising/improving upon the algorithm to analyze differences between versions of Wikipedia articles; the table above refers to primary focus. All members also worked collaboratively on the presentation and this report. There were no

problematic dynamics in our group.

9.2. Externally-written Code Utilized

- We used Python's difflib as the back-end algorithm. This is part of the Python standard library.
- Portions of the Java standard library, including Sockets and difference algorithms.
- Google Custom Search provides the search results on our site.
- Evaluation was was greatly aided by the HTMLParser Java framework.

9.3. Instructions on How to Start and Use Project

Our code branches are highly interdependent, and likely will require configuration (IP address adjustments, MySQL database setup, Python 2.6, Java 6, Ruby 1.8.7, Rails 3, etc) not specifically listed here. We are hosting a version of the program at http://www.wikitruthiness.com/; please visit this site to use the project. Also see the README file.