5-2011

Log-Data Visualization Tool for Analyzing and Improving Performance of Data De-Duplication Tool in Charm-II

Daniel Erickson
Utah State University

LOG-DATA VISUALIZATION TOOL FOR ANALYZING AND IMPROVING PERFORMANCE OF DATA DE-DUPLICATION TOOL IN CHARM-II

by

Daniel Erickson

A report submitted in partial fulfillment of
requirements for the degree

of

MASTER OF SCIENCE

in

Computer Science

Approved:

_____	_____
Dr. Stephen W. Clyde Major Professor	Dr. Vicki Allan Committee Member

_____
Dr. Scott Cannon
Committee Member

UTAH STATE UNIVERSITY
Logan, Utah

2011

ABSTRACT

Log-data Visualization Tool for Analyzing and Improving Performance of Data De-duplication Tool in CHARM-II

by

Daniel Erickson, Master of Science

Utah State University, 2011

Major Professor: Dr. Stephen W. Clyde
Department: Computer Science

A de-duplication tool used in CHARM-II, called the CHARM Matcher, produces log files that record why it decides two records are or are not a match. This data, if properly analyzed, could help CHARM developers improve the Matcher over time by tuning its configuration. However, the log data is complex and recorded chronologically in the log files instead of in a way that would aid analysis. Further, visually studying the raw log data is a laborious and difficult task. This report describes a tool that parses and organizes the raw log data, and then produces graphical reports that summarize key performance indicators. The performance indicators give CHARM developers exactly what they need to know to improve the Matcher's specificity and sensitivity [1] for any particular data source. A significant contribution of this report and prerequisite to creating a meaningful tool was the investigation into possible performance indicators and determination which would be best suited for the existing CHARM matcher. In

anticipation of further evolution of the CHARM matcher, the proposed tool is designed to be extensible, so additional indicators and reports could be added later, as the need arises.

(39 pages)

CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The Child Health Advanced Record Management (CHARM) system includes a component, called the *Matcher*, that is responsible for determining whether two or more person records represent the same individual [2]. Given a subject record from a particular data source, the Matcher accomplishes this task in two steps. First, it uses *candidate rules* to select a relatively small set of potential matches from millions of records contained in a central person database -- a filtering process often referred to as *blocking* or *search-space reduction* [3]. One difference between CHARM's approach to candidate set selection and a traditional blocking algorithm, however, is that CHARM uses different sets of candidate rules, based on the subject record's data source. This opens the door for the CHARM matcher to better accommodate the idiosyncrasies of each data source, but requires the CHARM developers to create, tune, and maintain multiple candidate rule sets.

The Matcher's second step is a classification process, in which it decides whether each candidate record matches the subject record and how confident it is about that decision. To do this, the Matcher uses a set of clue-based classification rules organized into a decision tree. The clues that make up a rule represent various ways in which a candidate and a subject record can be compared. For example, one clue may be a birth-date comparison. Another clue may be an edit-distance comparison on first names. Still, another clue may be a name-bag comparison that matches any name (first, middle, or last) from one record against any name from the other. The results of a clue can provide

evidence that the two records match or that they do not match. However, some carry more weight than others, so a rule can combine their results using a weighted sum. A rule then compares the aggregate result against several different thresholds to classify the pair of records as a) a definite match, b) a possible match, c) a definite non-match, or d) an undeterminable pair. The results of this classification from one rule can trigger the execution of another classification rule or the re-execution of step 1 with a different candidate rule set. For example, a classification of a possible match might trigger the execution of another more detailed classification rule.

As with candidate rule sets, the Matcher can use a different set of classification rules for each data source. Again, this allows the Matcher to find differences in the availability and quality of *personal-identify information* (PII) [4]. It also allows the CHARM developers to tune the Matcher's *specificity and sensitivity* [1], and balance other performance factors. Unfortunately, deciding which clues to use, organizing them into rules, adjusting the clue weights, and organizing the rules into a decision tree is a complicated task that requires good insight into the nature of the data source's data and the ability to try many different variations.

Sadly, the CHARM Matcher currently lacks the ability to measure the effectiveness of specific aspects of a candidate set or classification rule sets. Information about impact of a particular candidate rule, classification rule, or clues on match decisions would give CHARM developers valid feedback for optimizing the matcher for a particular data source.

One way to gather the such information, without changing the existing Matcher, would be to analyze its log files. These log files contain a chronological record of Matchers activities, which for a single match request would include:

- Information about which candidate rule was used
- The size of the resulting set of potential matches
- Information about the first classification rule that was fired
- The result of each of that rule's clues
- The rule's classification
- If another rule was fired, information about that rule
- And so forth

In addition, the log file includes the elapse time for each match request. Unfortunately, in its raw format, the log data is difficult to understand, let alone analyze.

This report describes a tool, called the *Match Log Analyzer* (*MLA*) for analyzing the Matcher's performance relative to any data source, quickly and effectively. Its primary goal is to parse and organize log data from a set of log files, and then generate visual reports that summarize that data in meaningful ways. However, to satisfy this goal, it must also meet the following secondary goals:

1. Define a set of indicators or metrics that will give the CHARM developers meaningful abstractions on the Matcher performance. Specifically, the indicators need to bring to the forefront those factors that most affect the Matcher's specificity, sensitivity, match rate, and efficiency, while hiding uninteresting details that have not significantly affected its performance.

2. Access and parse the rule sets that the Matcher uses for given data source, so it can correctly process and interpret the log data corresponding to data from that source.
3. Allow the tool to be easily modifiable so it can adapt to future Matcher changes and to shifts in meaning or format of the log data.
4. Allow the tool to be extensible so CHARM developers can easily add new reports that might give insight into the Matcher's performance.

This project covers the design, implementation, and testing of the initial log-analyzing and reporting tool. It does not address the procedures or policies for using that information or for making improvements to the Matcher and rule definitions.

There are a few issues that add to the complexity of this project. First, the complexity of the log files is one of the biggest problems which need to be addressed. Tracing the path of a search query of more than a dozen criteria through a participating program's chains of candidate rules, classifier rules, and clues from just the log files is challenging.

Second, this project needs to handle changes to the configuration of the Matcher. In other words, changes to the Matcher's configuration should not require the MLA's code to be changed or rebuilt. It is expected that, over time, CHARM will grow in both scope and functionality. For example, additional information systems will start providing information to CHARM and using it to access other systems' data. Each new participating program will require their own set of matching rules. Also, as existing

participating programs evolve, their matching rules will undoubtly also have to evolve to take into account shifts in data structures and data semantics.

Log files have been used to analyze many aspects of IT systems. The authors of [5] identified several uses for the information retrieved from log files, such as debugging, operational profiling, finding anomalies, detecting security threats, and measuring performance. Their research produced a general method for abstracting log lines to log event types. The MLA will do similar work in order to process the Matcher log files. Similar to methods used in [6], the MLA will be parsing log files to create static visualizations for offline analysis. Also, another common problem of using log files for analysis, as pointed out by the authors of [7], is the huge amounts of data contained in log files that make them impossible to analyze in their raw format. The authors work in [7] is concerned with producing visualizations of log files that help with identifying anomalies as part of a security audit, which is very similar to the goals of the MLA, except that it is attempting to analyze performance instead.

Since MLA is specific to the CHARM Matcher, this report first provides in Chapter 2 an overview of the CHARM system in terms of its purpose and architecture. To help clarify the problem, Chapter 2 also provides some background on what methods currently exist to measure Matcher performance. Chapter 3 then discusses the functional and non-functional requirements for MLA, which lay the foundation for its design and implementation, which are explained in Chapter 4. Chapter 5 explains how MLA was tested and how its use has helped improve the Matcher's performance. Finally, Chapter 6 discusses what further work could be done to continue to enhance the MLA tool.

CHAPTER 2

CHARM OVERVIEW

The CHARM system was developed to create a virtual health-care profile for all children who have records in many of Utah's child health-care programs [2]. The CHARM Integrated Infrastructure (CHARM-II) is a distributed middleware system that allows these health-care programs to securely share data. Each health-care program maintains stewardship over their own data, and CHARM presents a virtual health-care profile that spans the data from each program. Therefore, much of the data is decentralized in potentially heterogeneous systems. Although most of the programs within the Utah Department of Health (UDOH) are the primary producers and consumers of these virtual health-care profiles, other entities outside UDOH may also benefit from accessing this data, as long as these entities follow UDOH's privacy, confidentiality, and security policies.

The goals of CHARM-II include providing access to authoritative data in near real-time, allowing integration with participating programs with minimal impact to their information systems and doing so with relative ease, allowing each program to maintain stewardship over their own data, and enforcing privacy, confidentiality, and security policies appropriate for health information [2].

Each participating program integrates with CHARM via a CHARM Agent, which provides the means for querying the CHARM server for data, as well as exporting data that the participating program shares. The CHARM server itself is responsible for executing distributed queries, guaranteeing security, and keeping audit trails.

The Utah Statewide Immunization Information System (USIIS), Women, Infants, and Children (WIC), Vital Statistics (VS), Early Hearing Detection and Intervention (EHDI), and Heel-Stick (metabolic) Screening are the some of the information systems deployed by UDOH and are participants in the CHARM system. Several more participating programs are being developed as well, and all of these programs would benefit from sharing data with each other, yet at the same time, they all need to maintain stewardship over their own data, and meet strict privacy, confidentiality, and security requirements. Because each of these systems has been developed independently, each solution varies greatly in system design, supporting software, data abstractions, data quality, and other factors. In order to provide the benefits of a common child health profile to these heterogeneous systems, the CHARM Matcher is configurable on a per-program basis. General configurations of the Matcher can be shared between programs when appropriate. Additions of new or changes to existing participating programs requires changes to the Matcher's configuration, and the performance measurements gathered by the MLA will help ensure that the Matcher functions correctly and within reasonable time constraints.

The Matcher uses a two-step matching algorithm [8]. The first step is called *blocking.* The Matcher's implementation uses a loose SQL query to generate the initial set of candidates to be processed by the second step, called *clustering* or *classification*. Finding the right balance for the blocking step is a key to identifying matching candidates in reasonable amount of time. If the blocking step is too tight, it may eliminate positive matches from the set of candidates, losing accuracy. If it is too loose, it could include too

many candidates for the clustering step to process, resulting in intolerably long queries or a larger number of false positives.

The clustering step does more in-depth processing of the candidates on a field-by-field basis. The Matcher uses classifier rules with weighted scores that add to or take away from the confidence level that a particular candidate is a positive match to the search query. The classifier rules to use and the weights to apply are determined manually by a domain expert, one who has a deep knowledge of the systems in question. The classifier rules and the weights used could also vary between participating programs, so the needs of each program should be understood.

The configuration for the Matcher is very complex. It takes into consideration each participating program, configuration of different classifiers for children and adults, classifiers for single- and multiple-birth children, as well as other factors that address the participants' needs. When configuring the Matcher, the CHARM developer is tasked with defining rules that execute in a reasonable amount of time, but with few false positive matches. The configuration must also consider which classifiers or clues are relevant, and which can be safely left out to improve matching speed without sacrificing accuracy.

The Center for Disease Control and Prevention (CDC) has developed a probabilistic record linkage program to link records for cancer registries, called Link Plus [9]. Link Plus is part of a larger suite of programs called Registry Plus. Link Plus was designed for use by cancer registries, but can also be used for linking any type of data using fixed width or delimited formats. One way to measure this program's performance would be to calculate the sensitivity and specificity of matches found using test data. The

tool provides the ability to export its results, which could then be used to calculate the values for the performance indicators. One study [10] manually checked the results of doing electronic data linkage using Link Plus, and determined that the results of accurate linkage can be useful in estimating the prevalence of HIV/AIDS.

Lan Hu, developer of the CHARM Matcher, created a tool, called the *Log Reader*, to gather statistics from the Matcher's log files. This tool parses log files produced by the Matcher and provides a summary of some statistics, such as average request time per query, the total number of queries, the number of candidates within each confidence threshold, and the percentage that each criterion is used in a query. Additional statistics are gathered for some candidate nodes, such as a further breakdown of matching criteria, as well as the number of candidates within each confidence threshold.

Although not part of the original work of building the Matcher, the Log Reader did provide insight into the performance of the Matcher. Many of the ideas for the MLA, such as some of the statistics gathered, came from the Log Reader. There are several problems with the Log Reader that make it difficult to use, maintain, and extend. This is due to duplicate code and hard-coded details about participating programs and candidate rules. For example, the code for parsing the log file formats is duplicated in many places and does not always gather the same information. This requires multiple passes when parsing the log files. Also, many of the candidate rule configurations are hard-coded. This becomes a problem when the rules configuration is updated, but the Log Reader code is not. The design of the MLA needs to address these issues. I discuss additional problems and design considerations in the next few chapters.

The MLA provides visual representations of the reports from the Log Reader, and introduces new reports as well. Additionally, the MLA is aware of the Matcher's rule configuration and is able to use that additional context to provide more informative reports. Effectively, the MLA is an extension of the Log Reader.

Due to the sensitive nature of the data involved, the MLA must also conform to the security policies required by the CHARM system. The log files produced by the Matcher do contain PII, so great care must be taken to ensure that any sensitive information is not leaked. This means that the MLA should only be run on properly secured hardware, and only by authorized personnel. The output of the MLA should not contain any PII, but should only contain statistics and reports that can freely be shared with any interested parties, without fear of leaking sensitive information.

CHAPTER 3

REQUIREMENTS

The first step was to gather requirements for the tool. I met with Lan Hu and Dr. Stephen Clyde to determine what would be necessary. Together we identified the inputs and outputs for the tool, including the Matcher rules file and the log files. We also identified some other configuration options, such as the ability to filter the logs by date or by participating program, and the ability to specify the location of the output file. We determined that to be consistent with the rest of the CHARM-II codebase, Java would be the language to use.

Several performance indicators were identified during requirements analysis. General request statistics concerning the execution time of a query indicate whether the system is responding quickly enough to requests. Statistics about request criteria show the frequency that each criterion was being used for searches. The overall matching rate that shows the number of requests that resulted in a positive match is necessary for calculating sensitivity and specificity measurements, using test data and queries with known positive matches. The matching rate for each individual candidate rule is also reported, which allows a more fine-grained sensitivity and specificity measurements to be calculated. These fine-grained measurements are also necessary in order to measure the performance of matcher rules built for a specific participating program. Of course, the MLA filter for participating programs can also be useful when analyzing a participating program's use of general matcher rules. Also, for each candidate rule, statistics about the

scores of each classifier used in the rule demonstrate well that the classifiers correctly discriminate candidates in the result set.

For most of the performance indicators above, it is useful to distinguish between child and adult searches. The Matcher itself defines separate rules for searching for adults and children.

The generated reports need to show the exact numbers needed in order to calculate the sensitivity and specificity measurements for when known test data and search criteria are used. From those statistic, bar and pie charts then show the relative counts and percentages of the data.

Initially, the preferred format for the reports was a Microsoft Excel workbook. The workbook is a template, made of multiple spreadsheets wherein several of the spreadsheets contain charts that reference the data in another spreadsheet. The MLA needs to produce a CSV file that conforms to the format that the data spreadsheet expects, such that the charts reference the correct data. The charts should update automatically when the MLA output is pasted into the data spreadsheet.

There are some advantages to using an Excel workbook to produce the reports. The charts are relatively easy to produce. Additional reports can be added by laying out the CSV file in such a way that the charts can reference them. The template can be updated at the same time new reports are added, and the programming involved is no more complex than modifying a text file. Producing charts programmatically would be much more complex, even with the help of a third-party library.

On the other hand, there are also distinct disadvantages to using Excel. The template sometimes needs to be modified when the Matcher's configuration changes, such as when adding or removing candidate nodes, classifiers, or clues. Also, using Excel requires the manual step of copying the template file, then copying and pasting the output into a worksheet.

There are alternatives to using Excel to generate the required charts. JFreeChart is a free, open-source Java library that can programmatically generate charts. It is capable of exporting charts in several different formats, including image files, PDF files, or even as Swing components, which would be very useful for a GUI version of the MLA.

Another alternative is the popular Business Intelligence and Reporting Tools (BIRT) reporting system. Based on Eclipse, BIRT provides more than just a library for generating charts. For example, it also provides an API for connecting to a data source that contains the data from which reports can be generated. The data source could be a database connection, a CSV file, or a completely custom format.

A third option is the JasperReports, an open source Java reporting library. JasperReports is similar to BIRT in that it provides more functionality than generating charts. In fact, it uses the JFreeChart library to render the charts that it includes in reports. Similar to BIRT, it provides an API for connecting to several common data sources, as well as allowing custom data sources to be created.

All three of these alternatives allow creating charts programmatically. Any one of these options could potentially reduce the number of manual steps required to generate and maintain reports. However, each tool has its own API and requires the maintainer of

the MLA to learn these technologies to produce reports. As I mentioned previously, the Excel charts are relatively simple to understand and update. Thus, I decided to use the Excel workbook because of its relative simplicity.

A sample Excel file was included with the requirements document that defined the expected format for the file, as well as the specific reports that were expected. The following are descriptions of the expected reports:

1. General - This report should contain general statistics, such as the total number of queries, the number of queries for adult records and child records, as well as the minimum, maximum, and average execution times for those queries.
2. Request Statistics - This report should show how often each criterion was used for the queries, separated by adult and child queries. For instance, out of the total number of queries for child records, the number of queries including the social security number.
3. Candidate Sets - This report should show the number of queries that found candidates within any of the defined candidate set thresholds.
4. Candidate Nodes - This report should show specific information about each candidate node, such as the number of requests that had candidate sets of a certain size, the number of requests that produced candidate sets within each threshold, and statistics about the scores of each classifier included in the candidate node. There also needs to be a separate Excel worksheet for each candidate node defined in the Matcher's configuration.

The information for these reports comes from two log files produced by the Matcher. The first log file, called the matching process log, records contains the following information:

1. Matching ID - A unique id that identifies one query.
2. Real Time - The date and time that the query was performed.
3. Client IP address and port - The IP address and port of the client requesting the query.
4. Matching Criteria ID - An identifier, assigned by the client, for the matching criteria sent .
5. Participating Program ID – An identifier for specifying the participating program, and therefore a set of candidate nodes, classifiers, and clues.
6. Candidate Node IDs - The set of candidate node ids that were used during the matching process.
7. Number of Candidates - The total number of candidates found by the set of candidate nodes.
8. Total Matching Process Time - The number of milliseconds it took to run the query.
9. Candidate set sizes - The number of high confidence matches, probable matches, maybe matches, and no matches candidate sets.
10. Matching Criteria - The criteria provided by the client that was used to perform the matching.

The second log file, called the rule detail log, contains more detailed information about each matching process. Multiple log entries in the rule detail log correspond to a single log entry in the matching process log. Each log entry in the rule detail log contains information for a specific candidate rule. Each log entry contains the following information:

1. Matching ID - This ID refers to the matching process log entry. This is not unique in this file, as there are multiple entries, one for each candidate rule used during the matching process.
2. Candidate Node ID - The ID of the candidate node. This references the candidate node in the matcher's rules configuration.
3. Candidate Rule Name - The name of the candidate rule.
4. Candidate Charm IDs - The charm IDs of all candidates added to a candidate set by this candidate node.
5. Number of candidates - Total number of candidates added to a candidate set by this candidate node. This should simply be a count of the previous field.
6. Candidate rule execution time - the number of milliseconds spent executing this candidate rule.
7. Classifier Node ID - The ID of a classifier node. There are possibly multiple entries, one for each classifier node contained in the definition of the candidate node. The log for each classifier node includes the following information:
    a. Clue name and score - Each clue produces a score that is totaled to determine which candidate set the potential matches fall into.

b. Candidate classification score - The total score produced by the classifier for the particular candidate.

c. Candidate classification time - The time in milliseconds to perform the classification.

8. Candidate set sizes and Charm IDs - The total number of candidates in each candidate set, as well as the Charm IDs of the candidates in those sets. There should be four sets: high confidence matches, probable matches, maybe matches, and no matches.

CHAPTER 4

DESIGN AND IMPLEMENTATION

The primary goal of the MLA is to process the Matcher's log files and generate graphical reports. The review of what data was available in the log files constrained what sorts of performance indicators could be extracted and put into a useful visualization.

The first step in processing a log file is to open it and begin reading entries. Understanding of this step helped me identify two useful components: a *LogDatasource* that knows how to open files and a *LogEntry* that knows how to parse each entry in the log file. The *LogEntry* class is also important because it helps address another goal of the MLA, which is to make it easier to deal with changes in the log file format. The *LogEntry* class is the one place wherein parsing the log files takes place, unlike the Log Reader tool, which has multiple implementations for extracting partial entry information. Having a single object that extracts all the information once means the log files themselves only need to be read once.

Before a *LogEntry* continues through processing, it must pass through filters that could exclude the *LogEntry* from processing. One of the requirements that we identified was the ability to filter on participating programs and a date range. I introduce a *LogEntryFilter* interface to take on this responsibility. Two concrete implementations of *LogEntryFilter* exist, one to filter entries by the participating program and another to filter by date range. Additional filters can be implemented as needed. If a log entry is excluded, that entry is discarded and the next log entry begins processing.

After parsing a log entry, the MLA still needs to do something with it to generate a graphical report. All of the data for a report needs to be gathered before the report can be rendered. Therefore, I identify two additional classes to accomplish this: the *Report* class and the *ReportWriter* class. The responsibility of the *Report* class is to accept each *LogEntry* and extract the pertinent data for the report. Once the *Report* has completed its task, the *ReportWriter* is then responsible for generating a graphical representation of the report. In the case of the Excel spreadsheet, the *ReportWriter* generates a CSV file that can be pasted into the data spreadsheet of the workbook template. One advantage of separating the responsibilities of the *Report* class and the *ReportWriter* class is the ability to create additional *ReportWriters* that could generate a different graphical representation of the same report. The actual report is decoupled from its graphical representation. This design resembles the Reference Model pattern as described in [11], but lacks a controller class because there is no user interaction with the reports.

Some reports need more information than what is provided in the log files themselves. Specifically, the Matcher rules configuration file has information necessary for some reports in order to interpret the results in the log files correctly. I include a component that parses the xml and loads the configuration into model objects that are held in memory and made available to the reports that require the information.

As part of the design, I produced an activity diagram that shows the general algorithm used, identifying the major components and showing how they collaborate. The second diagram is a class diagram for the major components.

4.1 Algorithm

The algorithm was developed using an activity diagram, as shown in Figure 1. The algorithm is divided into two main sections. The first section deals with processing the log files. The log files are referred to as data sources. Each entry is passed to each of the reports, and the report is responsible for collecting relevant data. After all of the log

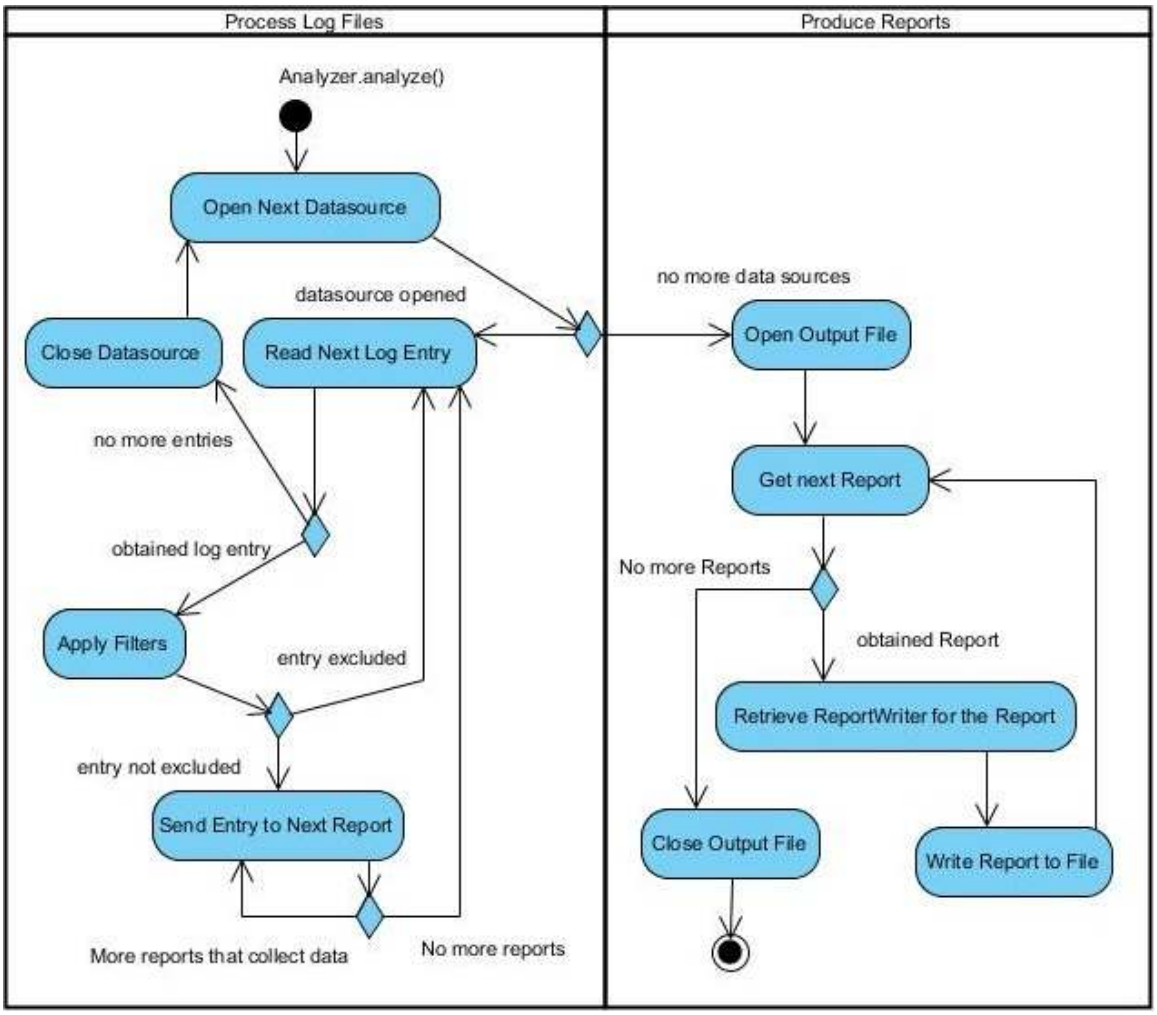

Figure 1. Activity diagram of the general algorithm.

entries have been read from all of the data sources, the algorithm moves on to the next section, where the reports are produced and written to a file.

There are two different approaches that I considered when implementing this algorithm in the *Analyzer* class. The first is the Template pattern as described in [12]. The algorithm would be implemented in a single method of the abstract *Analyzer* class, with calls to abstract template methods for each major step. A subclass would then be responsible for implementing those template methods.

The second approach is the one that I actually implemented. The *Analyzer* class requires concrete implementations of abstract factories, following the Abstract Factory pattern, which is also described in [12]. The result is very similar to using the Template pattern, but instead of template methods being called, factory methods are called instead. This implementation uses composition instead of direct inheritance to accomplish the same task. I describe the factories used after presenting the class diagram.

The class diagram, in Figure 2 below, identifies the classes that are used as part of the algorithm. The classes are identified by name there, and are the names that are used for the implementation.

For the Matcher Log Analyzer, the *LogDatasourceFactory* is configured with an implementation of a *LogSource* that loads both that matching process log file and the rule detail log file. When retrieving the next entry, the implementation checks to see if all the matching process log entries have been read before reading the rule detail log entries. The *ReportFactory* is configured with all of the defined reports in the system. This is the extension point for adding new reports to the tool. The new reports simply need to be

registered with the *ReportFactory*, and they will be given all of the log entries for processing. The *ReportExporterFactory* itself needs one *ReportExporter* for each of the *Reports* registered with the *ReportFactory*. The *ReportExporterFactory* was separated from the *ReportFactory* to allow different types of *ReportExporters* to export *Reports* into varying formats. The current implementation uses a CSV format, but other formats

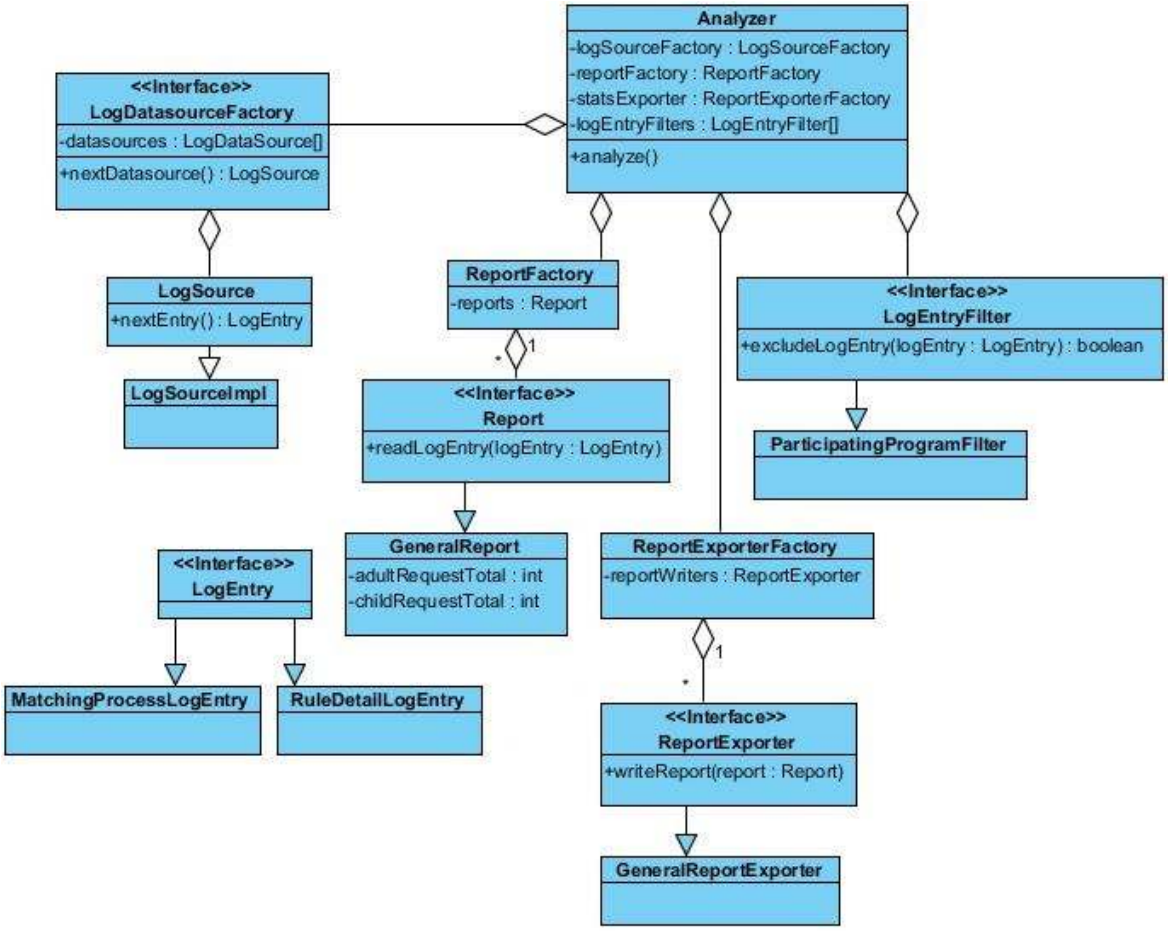

Figure 2. Class diagram.

could be added easily as the need arises. Finally, *LogEntryFilters* are defined. These should be configurable via a configuration file, with no coding necessary to add or remove these filters. These filters simply short-circuit the processing of a log entry, if the criteria for the filter have been met. So far, the only filters implemented involve specifying which participating programs are included in the report and specifying a date range for the log entries. Adding new filters is accomplished by implementing the *LogEntryFilter* interface and registering the filter with the *Analyzer*, based on external configuration.

4.2 Report Formats

For the *General Report*, a simple bar chart is used, as shown in Figure 3. The report need only tally up the number of requests from the matching process log, distinguishing requests for a child versus adult record.

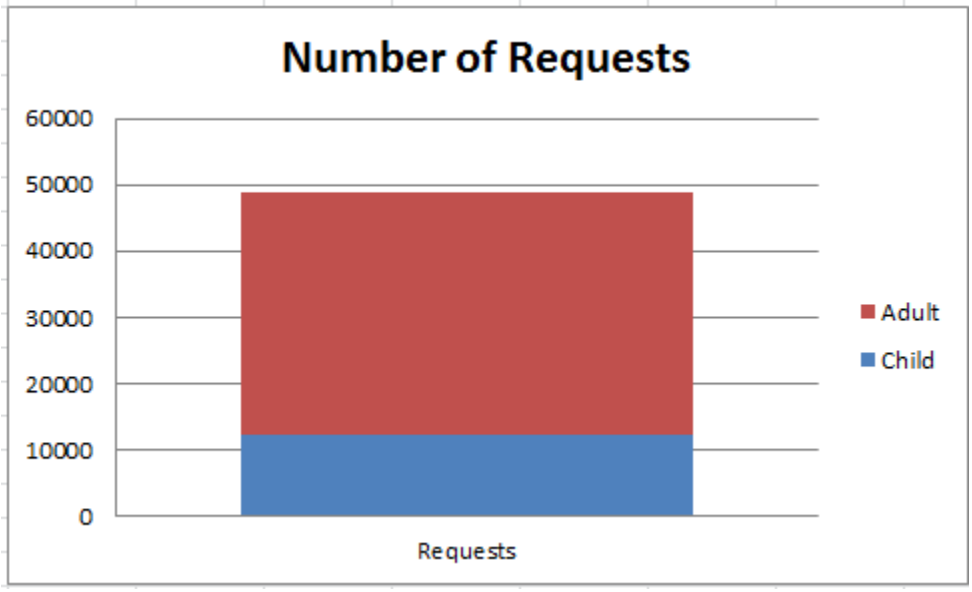

Figure 3. Number of requests.

The *Request Statistics Report* is also a bar chart, showing the total number of requests that contain each of the search criteria fields. Examples of this report are shown in Figures 4 and 5.

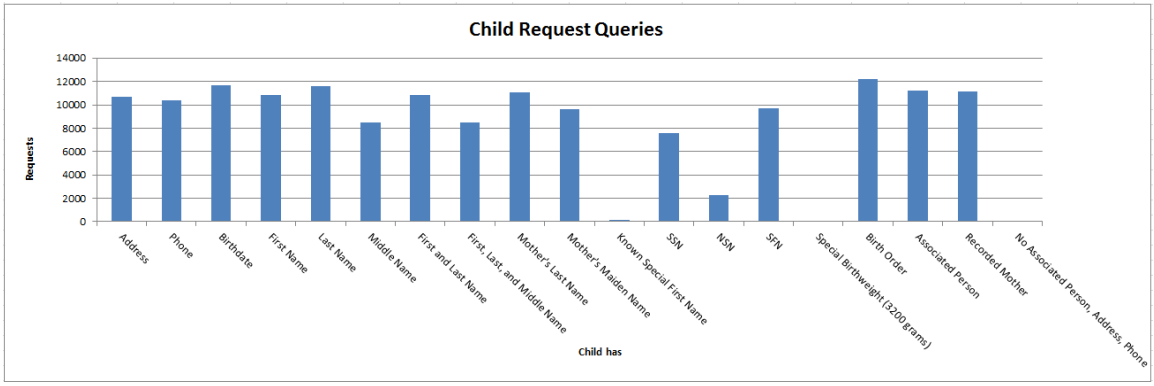

Figure 4. A count of each field in child request queries.

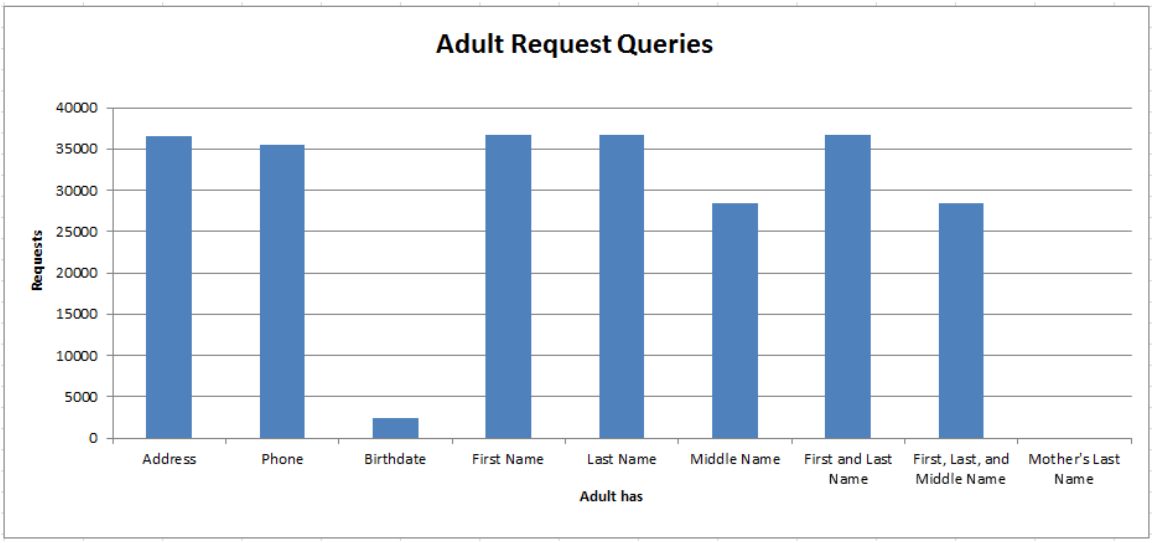

Figure 5. A count of each field in adult request queries.

The report on candidate sets uses pie charts to show how many requests produced candidate sets within each of the thresholds. Three pie charts are produced, one for child queries, one for adult queries, and one for all queries. An example is shown in Figure 6.

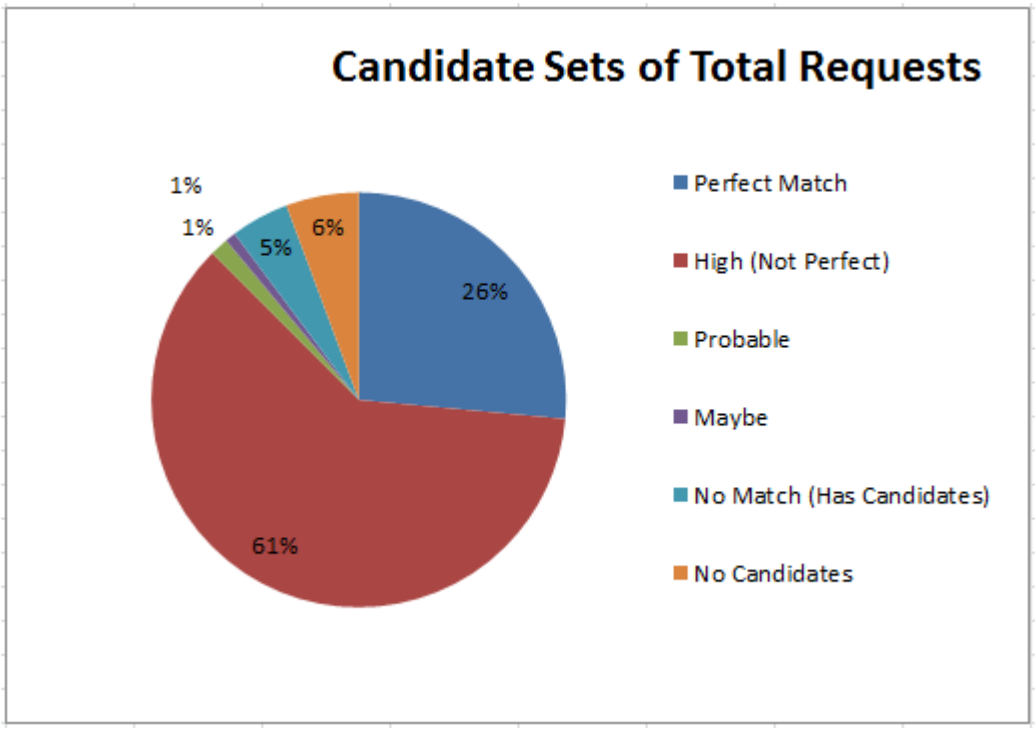

Figure 6. Break down of queries by the type of candidate set produced.

The last kind of report, the *Candidate Rule Report*, has many instances, one for each candidate rule. This report shows a bar chart for the number of candidate sets that produced a certain number of candidates. There are also bar charts that show the number of queries that received full or partial scores from each classifier. Figures 7, 8, and 9 show examples of these charts.

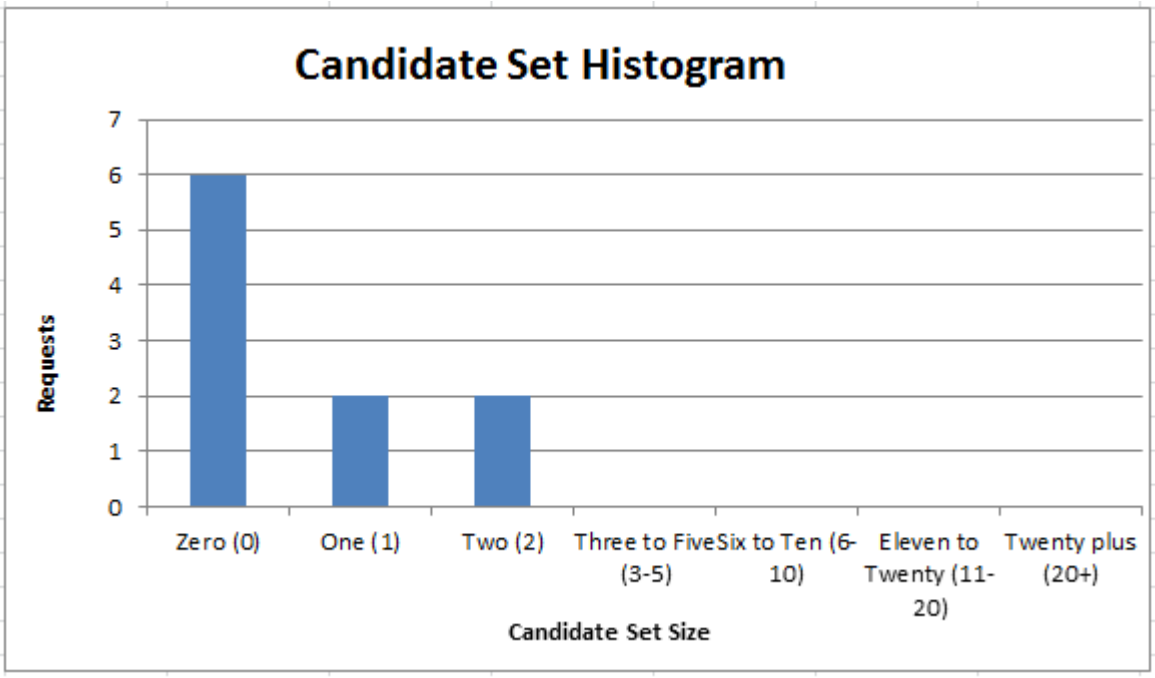

Figure 7. Histogram of candidate set sizes.

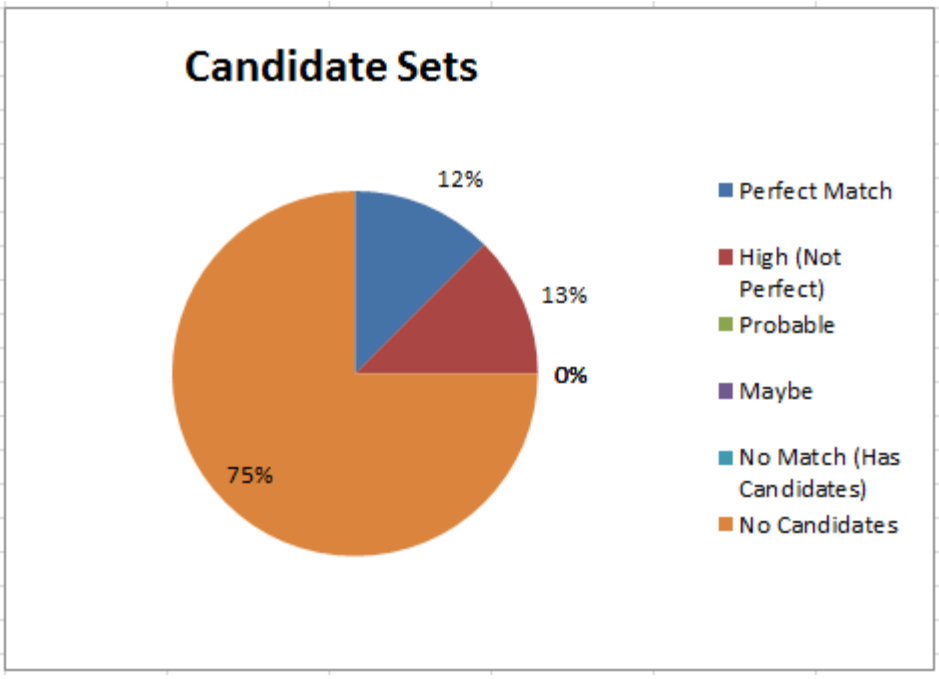

Figure 8. Percentages of requests within each candidate set threshold.

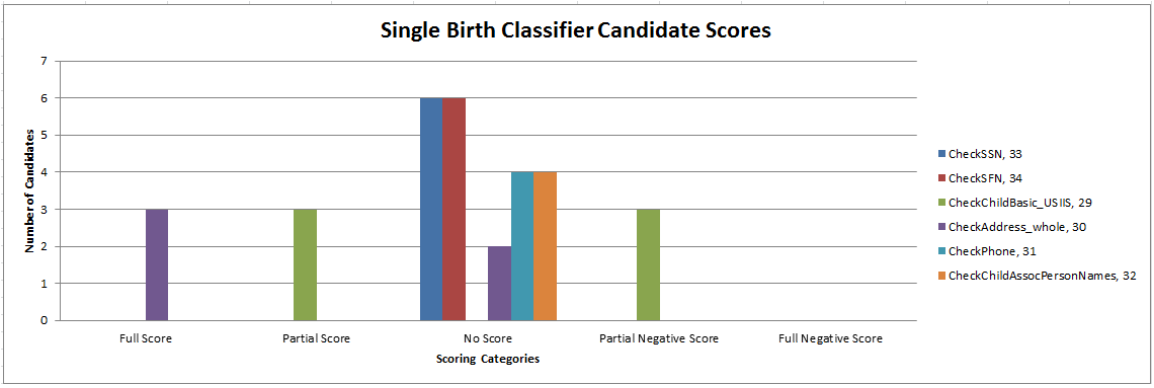

Figure 9. Candidate scores per classifier.

The implementation of this last report is also the first report that required the matcher's rule configurations. The candidate nodes are referenced in the log files by an integer identifier, so the node must be looked up. Classifiers are also referenced by an identifier. Clue scores are summed and compared with the maximum and minimum total scores to determine if full, partial, or no points were awarded.

Once the *Report* objects have collected what they need, they are passed to *ReportExporter* objects. Currently, the matcher log analyzer has CSV implementations of report exporters which produce CSV files that can be pasted into the predefined Excel spreadsheet. Further work can be done to implement new *Report Exporters*. Some ideas include producing charts programmatically and saving them as PDF files, or if a GUI is developed, charts could also be presented directly in the GUI.

Additional reports can be introduced by implementing *Report* and *ReportExporter*. New reports that use the CSV exporter may also need to update the template Excel spreadsheet to include a new sheet of charts.

CHAPTER 5

TESTING

To test the MLA, I used both automated unit and integration tests, as well as manual testing with real data. For the unit testing, I focused mostly on parsing files, including the log files as well as the rules configuration file. Unit tests are important because they help guarantee that any changes to the file formats will be handled correctly. New tests can be added for any additional data, and the old tests will ensure that the parsers are still collecting the correct data.

I also wrote an automated integration test that constructs the *Analyzer* class with all of its dependencies, configured to analyze the sample matching process and rule detail log files. This is an integration test because it exercises every component from end to end, unlike a unit test, which tests a single component in complete isolation from all other components. This test produces an output file that is ready to be pasted into the Excel template.

In addition to the automated unit and integration tests, I also performed some manual testing on real data obtained from the production server. Due to the sensitive nature of the data, I did not think it wise to bundle these log files with automated unit tests. Instead, I performed the manual tests and then removed the log files from my machine. I configured the MLA to analyze 20 requests from the Matcher log files, with a mix of search queries for adults and children. The small sample of requests, taken from the production log, was large enough to show that the MLA is able to process the files, but still small enough to reasonably verify its results by hand. To verify the *General*

Report, I totaled the requests for adults and children then compared them with the results produced by the MLA. Similarly, for the *Request Statistics Report*, for each request criteria field, I totaled the number of requests where that field was provided. For the *Candidate Sets Report*, I counted the number of requests that produced a candidate set within a certain confidence threshold. For the *Candidate Rule Report*, I also had to count the requests that resulted in a particular candidate rule producing a candidate set within a confidence threshold. I also had to calculate how many requests received a full or partial score for each classifier rule used by the candidate rule.

I performed other manual testing on the tool, such as making sure that every configuration option worked as intended, as well as reviewing the log files produced the by the MLA. The MLA logs errors or warnings about problems it encounters, such as malformed log entries.

The performance indicators identified for the initial version of the MLA are a good starting place, but may require refinement. Over time, the CHARM developer responsible for configuring the Matcher will be able to see which reports on performance indicators are useful, and which may require modification. Some of the performance indicators were selected based on the available data in the Matcher's log files. Additional views of the existing logs can be created, or if more data is necessary, more data can be logged for the MLA to process.

CHAPTER 6

SUMMARY AND FUTURE WORK

The MLA has been designed to produce visual reports of the Matchers performance from log files. Several extensible points in the architecture of the MLA have been defined to achieve several of the goals listed earlier. Each major piece of the MLA's algorithm encapsulates cohesive components, making it easier to modify the parsing of log files and the Matcher configuration. It is also easy to modify or create new reports and report visualizations. Each of the visualizations reports helps the maintainer of the Matcher understand the impact of configuration changes on the performance of the Matcher. However, there is room for improvement as well.

The manual steps involved in copying and pasting exported data into an Excel file could be eliminated with different implementations of the *ReportExporter*. The tool could be extended to use any of alternatives to Excel mentioned in Chapter 3.

The tool could be converted to monitor performance in real-time, as the log entries are written to the file. Many of the same components could be used, but the algorithm would probably need to be modified. One thing that would need to change is that as each log entry is read, it would also be passed to the report. When the report is updated, the report exporter would need to be notified of the update and redraw the charts. This would not work with the Excel template at all. The reports would have to be generated programmatically.

Another improvement would be to track statistics over time. Currently, to follow any trends, one must generate multiple reports and extract any information of interest.

The MLA could keep track of each use and produce addition reports that show trends in each report or even changes to the matcher rules file. From that kind of a report, it may be possible to identify whether a rule change has had a positive or negative effect on matcher performance.

REFERENCES

[1] Altman, G. and Bland, J. Diagnostic tests 1: sensitivity and specificity. *British Medical Journal,* 308 (1994), 1552.

[2] Clyde, S. *Executive Summary: Child-Health Advanced Record Management Integration Infrastructure, CHARM Project.* Utah Department of Health, Jan 2002.

[3] Elmagarmid, A., Ipeirotis, P., and Verykios, V. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering,* 19 (2007) 1-16.

[4] McCallister, E., Grance, T., and Scarfone, K. *Guide to Protecting the Confidentiality of Personally Identifiable Information (PII).* NIST Special Publication 800-122, Apr. 2010.

[5] Nagappan, M., Vouk, M.A. Abstracting log lines to log event types for mining software system logs. *Mining Software Repositories (MSR), 2010 7th IEEE Worker Conference* (2010) 114-117.

[6] Siirtola, H., Raiha, K.-J., Surakka, V., and Vanhala, T. Flexible Method for Producing Static Visualizations of Log Data. *Information Visualization, 2008. 12th International Conference* (2008) 127-132.

[7] Takada, T. and Koike, H. Tudumi: Information Visualization System for Monitoring and Auditing Computer Logs. *Information Visualisation, 2002. Sixth International Conference* (2002) 570-576.

[8] Hu, L. *A Deduplication System For Integrated Person-centric Information Systems.* Master's Thesis, Utah State University, 2007.

[9] U.S. Department of Health and Human Services, Centers for Disease Control and Prevention. National Center for Chronic Disease Prevention and Health Promotion. 2010. http://www.cdc.gov/cancer/npcr/. September 2011.

[10] Electronic Record Linkage to Identify Deaths Among Persons with AIDS - District of Columbia -2000—2005. *Morbidity and Mortality Weekly Report.* (2008) http://www.cdc.gov/mmwr/preview/mmwrhtml/mm5723a4.htm. September 2011.

[11] Heer, J. and Agrawala, M. Software design patterns for information visualization. *IEEE Transactions on Visualization and Computer Graphics* 12, 5 (2006) 853-860.

[12] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.