SANDIA REPORT

SAND2005-6007
Unlimited Release
Printed October 2005

Implementation of a Data Fusion Algorithm for RODS, a Real-time Outbreak and Disease Surveillance System

Douglas Brown
Genetha Anne Gray

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2005-6007
Unlimited Release
Printed October 2005

Implementation of a Data Fusion Algorithm for RODS, a Real-time Outbreak and Disease Surveillance System

Douglas Brown
Genetha Anne Gray
Computational Sciences and Mathematical Research
Sandia National Laboratories
P.O. Box 969, Mail Stop 9159
Livermore, CA 94551-0969

Abstract

Due to the nature of many infectious agents, such as anthrax, symptoms may either take several days to manifest or resemble those of less serious illnesses leading to misdiagnosis. Thus, bioterrorism attacks that include the release of such agents are particularly dangerous and potentially deadly. For this reason, a system is needed for the quick and correct identification of disease outbreaks. The Real-time Outbreak Disease Surveillance System (RODS), initially developed by Carnegie Mellon University and the University of Pittsburgh, was created to meet this need. The RODS software implements different classifiers for pertinent health surveillance data in order to determine whether or not an outbreak has occurred. In an effort to improve the capability of RODS at detecting outbreaks, we incorporate a data fusion method. Data fusion is used to improve the results of a single classification by combining the output of multiple classifiers. This paper documents the first stages of the development of a data fusion system that can combine the output of the classifiers included in RODS.

Acknowledgments

We would like to thank Keith Vanderveen for his invaluable assistance with the RODS software. We also thank David Gay and Ken Sale for their work in researching data fusion methods. We are grateful to the Computational & Information Sciences LDRD program for funding this work.

Contents

Figures

Abbreviations

CuSum	Cumulative Sum
DHS	Department of Homeland Security
DTRA	Defense Threat Reduction Agency
RLS	Recursive Least Squares
RODS	Real-time Outbreak Disease Surveillance
WAD	Wavelet-based Anomaly Detection
XML	eXtensible Markup Language

1. Introduction

A person infected by a life threatening bio-agent, such as anthrax, often exhibits the symptoms of a much less serious ailment (e.g. influenza), or shows no symptoms at all for several days [1]. Thus, infected individuals may receive treatment for other, minor illnesses and or may refrain from seeking treatment until more severe symptoms develop. Moreover, by the time a patient has been diagnosed as having been infected with anthrax, he is often near death or already dead. This is one reason that biological attacks are of particular concern; a terrorist release of anthrax or smallpox into the air could result in a high number of casualties simply because of diagnosis difficulties [2]. In order combat this problem and assist healthcare practitioners in making quick determinations when an outbreak of a disease has occurred, either due to a biological attack or some other means, the Real-time Outbreak and Disease Surveillance System (RODS) [3] was developed.

RODS is an open-source public health surveillance software package that is the result of a collaboration between the University of Pittsburgh and Carnegie Mellon University and funding from numerous federal agencies and state health departments. The software includes a variety of tools for the collection and analysis of disease surveillance data. The purpose of developing such software was to make improvements on current methods of outbreak detection such as human collection and surveillance of health data [1]. Such methods can be ineffective due to the time and effort needed to collect and analyze data. Often, it is too late to effectively treat victims by the time the necessary data collection and analysis has been completed. The RODS system was designed to consider existing data from disparate sources, such as absenteeism records from local schools and sales of over the counter drugs, in judging whether or not an epidemic outbreak has occurred in a particular geographic area [4]. Using detection algorithms such as the Recursive-Least-Squares adaptive filters [5] and the "What's Strange About Recent Events" heuristic [5], RODS analyzes different sets of data to try to determine if a biological attack has occurred. If any "abnormalities" are found in the given data, such as a recent spike in the number of bottles of cough syrup purchased, an alert is produced. These alerts are e-mailed to the system administrator who then decides what action, if any, to take.

Sandia has leveraged the RODS software package in its execution of the goals of the BioNet program (funded by DTRA and DHS), and the subsequent development of Bio-DAC, a decision analysis center for biological attacks. Our goal is to assist these efforts by implementing an automated method of heterogeneous data fusion within the RODS framework.

2. Data Fusion

Although RODS can be an effective method of outbreak detection, it does not currently include an automated method of data fusion. Data fusion is based on the principle that combining data sources should give better inferences than using one

source alone. Moreover, data fusion is a form of ensemble classification and may improve on the effectiveness of RODS [6]. Classifier ensembles are used to improve the classification results of a single classifier [7]. Each of the underlying or base classifiers are trained individually and then used to classify one data set. Next, an ensemble classifier or fusion algorithm is used to combine the results of these base classifications in some way. The resulting combined output is often referred to as a decision. Most classifier ensembles combine the results of classifications of the same data sets using different base classification methods to arrive at a decision. In contrast, this work uses the ensemble classifier for disparate, but related, data sets. The data is obtained from different sources, and the fusion algorithm is used to provide a decision related to of the data collected in the RODS framework.

There are several existing techniques of data fusion. One of the most common is called voting [8, 9]. In voting, each classifier casts a "vote" with its classification of the data. Then, each vote may be counted equally, or the "expertise" of the classifier may be considered in order to make its vote worth more or less than other votes. The classification that receives the most votes is used as the decision of the ensemble. The simplest voting algorithm is unweighted majority voting [8]. Simply stated, the algorithm is as follows: Given n base classifications, a decision is accepted if at least k of these classifications agrees where

$$k = n/2 + 1, \text{ if } n \text{ is even}$$
$$k = (n+1)/2, \text{ if } n \text{ is odd.}$$

It should be noted that within the multiple classifier community, it is common practice to compare any new fusion algorithm to unweighted voting. Therefore, we implemented the unweighted voting algorithm as a comparison tool. It was also used as a simple means of identifying and trouble-shooting any software integration problems.

3. Implementation of the Unweighted Voting Algorithm

In order to incorporate a data fusion method in the RODS framework, the unweighted majority voting algorithm was implemented in Java. It consists of the following six classes:

- `Vote`
- `Classifier`
- `RODSClassifier`
- `DataFusion`
- `Voting`
- `UnweightedMajorityVoting`

Each is described in detail in this section.

The `Vote` class simply represents the output of a base classifier included in the ensemble. It consists of the name of the classifier that produced the output, the decision of this classifier, and the weight of the decision, if any. Weighting is not used

in the unweighted majority algorithm so the value of the decision weight is always one. However, this field was included to simplify future work with similar algorithms such as weighted majority voting. It can be easily changed.

The `Classifier` class is an abstract class that any base classifier to be used in the ensemble must extend. This class is used to greatly simplify the implementation of the unweighted majority algorithm through the use of polymorphism. The `Classifier` class was also implemented to increase the flexibility of the data fusion application. There is nothing in this class tying it directly to RODS, so any application could potentially use it. Since all classifiers associated with the application must extend from this class, new classifiers can be easily added to the system with out having to change any of the data fusion algorithm code. The `Classifier` class contains fields that store the classifier's name, usually just the name of the class that implements the classifying algorithm, and a weight that determines the weight that the classifier's decision should have with respect to the consensus of the ensemble. It also provides methods that allow the field values to be retrieved and changed and includes two abstract methods. The first, *getDecision*, retrieves the base classifier's decision. The second is the *run* method which executes the base classification algorithm.

The `RODSClassifier` is an abstract class which is a subclass of `Classifier`. This class represents any base classifier to be used with RODS and implements some functionality specific to RODS. This class contains a List field which is used to store alerts produced by the classifier and a method to retrieve this field. `RODSClassifier` also contains methods that allow uses to add alerts to the List field (*addAlert*) and implements the *getDecision* method declared as abstract in the `Classifier` class. This method checks the List field in order to see if any alerts have been produced by the classifier. If an alert has been produced, "**Yes**" is returned. Otherwise, "**No**" is returned. This data is returned in the form of a `Vote` object which contains the classifier's name, weight, and the String object which contains either "**Yes**" or "**No**." This method is used by the data fusion algorithm to determine the consensus of the algorithm.

The `DataFusion` class is an abstract class that represents any data fusion algorithm to be used with RODS. This class has a Map field that maps the name of a classifier to a reference to Classifier object associated with that name. This class contains a method that can be used to retrieve the Map field and an *addClassifier* method that allows users to add classifiers to the Map object. The *addClassifier* method throws an exception if more than one classifier with the same name is added. For this reason, it may be necessary to change the name field of the Classifier object to something besides its class name. This class also has an abstract method, *run*, that should be implemented in any subclass of `DataFusion`. This method is used to implement the actual data fusion algorithm used to combine the output of an ensemble of classifiers.

The `Voting` class is a subclass of `DataFusion`. This class is an abstract class that represents any data fusion algorithm that uses voting to determine the decision of an ensemble of classifiers. This class contains two fields, a Map object which maps each

vote to the number of times it was cast and a List object which stores each vote cast. In addition to the two fields, the class also implements the abstract run method from DataFusion. The method is implemented to sequentially execute each classifier and store the decision of the classifier in the `Voting` class's Map field. The class also contains two abstract methods: *addVote*, which should implement how votes are to be added to the Map field of any subclasses of `Voting`, and *vote*, which should implement a `Voting` subclass's voting algorithm.

The `UnweightedMajorityVoting` class is a subclass of `Voting`. The class implements the unweighted majority voting algorithm that serves as a point of comparison with other algorithms in the data fusion application. Since this class extends from both `Voting` and `DataFusion`, it inherits the *run* and *addClassifier* methods, along with the methods to used to access the fields of those two classes. In addition, the class implements the two abstract methods from the `Voting` class: *addVote* and *vote*. This class's implementation of *addVote* simply adds a casted vote to both the List and Map objects inherited from the `Voting` class. This method first checks the Map object to see if the vote being added has already been cast. If it has, the count for that particular vote is incremented; otherwise, a new entry in the Map is created for the new vote, and its counts it set to 1. The *vote* method included in `UnweightedMajorityVoting` looks at the results of each classifier and passes them to the *addVote* method. The method then looks at the Map object and tries to determine what the consensus of the ensemble is by looking at the decision that has the highest number of votes. If this number is greater than or equal to $n/2+1$ for an even number of classifiers or $(n+1)/2$ for an odd number, the decision is returned. Otherwise, a String object containing the text "**No consensus reached**" is returned.

4. Current RODS Base Classifiers

The RODS source code contains implementations for three different classifying algorithms: *Recursive Least Square* (RLS), *Cumulative Sum* (CuSum) [5,10], and Wavelet-based Anomaly Detection (WAD) [10]. These three classes were altered slightly so that they extend the abstract Classifier class and can more easily be incorporated in a data fusion algorithm. Each classifying algorithm uses a different method of classifying data.

The RLS classifier uses the RLS adaptive filter to search for data anomalies. In the context of disease outbreak surveillance, this algorithm attempts to predict the numbers associated with a particular syndrome or illness based on the counts of previous days. If the count on any given day is greater than the 95% confidence interval of the classifier prediction, an alarm is fired. This method requires less historical data than most other classification methods. Figure 1 gives an example of the output generated by the RLS classifier.

In Figure 1, the x value is the normalized number of reported cases of a particular syndrome that have been counted on a particular day; the raw x value is the non-normalized count value. The y value is the number of counts predicted by the algorithm

```
Syndrome:Rash syndrome ID=4, count size=21.
INFO  :  07/12/2005 13:53:43     x=14908.0.
INFO  :  07/12/2005 13:53:43     y=23252.71132608319.
INFO  :  07/12/2005 13:53:43     raw x=14908.0.
INFO  :  07/12/2005 13:53:43     mean of last 30 days =25943.380952380954.
INFO  :  07/12/2005 13:53:43     TIMES OF THRESHOLD= 3.0.
INFO  :  07/12/2005 13:53:43     Threshold= 23201.74381487237.
INFO  :  07/12/2005 13:53:43     Prediction error= -8344.71132608319.
INFO  :  07/12/2005 13:53:43     Times of error STD= -1.0789763984120295.
```

FIGURE 1: The output of RLS.

for a particular day. This number is subtracted from the x value to find the "Prediction error." The "TIMES OF THRESHOLD" value is used to determine whether or not an alarm will be fired. This value is multiplied by the standard deviation for the count value to get the value for "Threshold". If the prediction error is greater than this threshold value, an alarm is fired. The "Times of error STD" value is the number of standard deviations away from the predicted value the actual counted value is, and the "mean of last 30 days" is the average count over the past 30 days for a particular syndrome.

The CuSum classifier uses the cumulative sum algorithm in an attempt to identify any anomalies. Unlike RLS which focuses on values from a single day, CuSum looks for gradual changes in the mean values of the counts for syndromes and illnesses. This is accomplished by looking at the amount that daily counts differ from their expected value over a period of time. These amounts are summed, and an alarm is fired if these amounts go above a particular value known as the threshold. Sample output generated by the CuSum classifier is shown in Figure 2.

For the CuSum classifier, the syndrome ID (in Figure 2, it is 4) is used to identify a particular syndrome within RODS, and in this case, the syndrome ID represents rashes. The count size is the number of days' worth of data being analyzed by this algorithm. The x value is the normalized number of cases of rashes found in a particular region, and the raw x value is the non-normalized number. The "cusum std dev" value represents the standard deviation of the count data. The "cusum threshold" is the value that the count must exceed in order for an alarm to be fired. The eligibility determines whether or not an alarm will be fired; a value 1.0 means that the data represents an abnormal condition where an alarm should be fired, and a value of 0.0 means that no alarm should be fired. The "mean of last 30 days" is the mean value of the raw x over the last 30 days of data. The information that follows "****** Prodrome=4 No. of days = 21" is an example of the type of data contained in a CuSum alarm. The alarm consists of the date the alarm

was generated, the syndrome ID, the count that led to the alarm's generation, and the threshold value. This information is sent to the system administrator in an e-mail.

```
INFO  :  08/02/2005 13:26:09 **** starting syndrome =4.
INFO  :  08/02/2005 13:26:09 in State CA.
Syndrome:Rash syndrome ID=4, count size=21.
INFO  :  08/02/2005 13:26:09     x=1.7976931348623157E308.
INFO  :  08/02/2005 13:26:09     cusum std dev. =1.4598833371035465E304.
INFO  :  08/02/2005 13:26:09     cusum threshold=64218.44360194223.
INFO  :  08/02/2005 13:26:09    eligibility= 1.0.
INFO  :  08/02/2005 13:26:09     raw x=14908.0.
INFO  :  08/02/2005 13:26:09     mean of last 30 days =25943.380952380954.
INFO  :  08/02/2005 13:26:09 .
       CUSUM detection for prodrome/total visit in CA.
INFO  :  08/02/2005 13:26:09 ****** Prodrome=4 No. of days=21.
INFO  :  08/02/2005 13:26:09 in State CA.
INFO  :  08/02/2005 13:26:09 Date: 2004-11-22.
INFO  :  08/02/2005 13:26:09 -----e-mail alert sent!-----.
              EpiAlert Report (by Cusum) in State CA.
.
Date: 2004-11-22 Minute: 240.
Prodrome:Rash in State CA .
Observed normalized count:100.000 (14908.0/14908.0) .
Threshold value: 64218.444 .
Last two months plot based on minute 240: nullRLSCharts?start=1097668800000&end=
```

FIGURE 2: The output of CuSum.

The WAD classifier uses wavelets to detect anomalies by subtracting long term trends from time series data. The values left over after subtracting long term trends are known as residuals. In the case of RODS, an example of a long term trend would be an increase in the number of bottles of cough syrup purchased due to an increase in the number of brands being sold. The algorithm then checks the residual values for the days in the time series to see if any of them exceeds the WAD threshold value; if the residual exceeds this value, an alarm is fired. An example of WAD output is displayed in Figure 3.

```
2005-08-08 14:22:51,031 INFO  [runWAD] **** starting syndrome =6.
2005-08-08 14:22:51,031 INFO  [runWAD] All --prodrome:6 mfst=6.
2005-08-08 14:22:51,031 INFO  [runWAD]  x(n)= 1.184263112E9.
2005-08-08 14:22:51,031 INFO  [runWAD]  LP_MEAN= 4.9250287181729l5E8.
2005-08-08 14:22:51,031 INFO  [runWAD]  residual= 6.917602401827085E8.
2005-08-08 14:22:51,046 INFO  [runWAD]  RSD_STD= 5.607144661454324E8.
2005-08-08 14:22:51,046 INFO  [runWAD]  TIMES_STD=1.2337121332683556.
.
2005-08-08 14:22:51,046 INFO  [runWAD]  err=6.917602401827085E8.
.
2005-08-08 14:22:51,046 INFO  [runWAD]  countThreshold*stdErr=2.8035723307271(
```

FIGURE 3: The output of WAD.

In Figure 3, the `syndrome` number is used to identify a particular syndrome within RODS. The `x(n)` value is the count for the number of recorded instances of a syndrome on a particular day. The `LP_MEAN` represents the expected value for a particular day. Both the `residual` and `err` values give the residual value for the previous day obtained by subtracting long term trends from data. The `RSD_STD` value gives the standard deviation of the residual values. The `TIMES_STD` value is the number of standard deviations between the residual value and the predicted value. The "`countThreshold*stdErr`" is the threshold that the residual value must exceed in order for an alert to be fired.

The data used to test all three classifiers is syndromic data stored in an XML file. Figure 4 shows an example of some of the data.

```
<syndrome_civ>.
  <julianDay>13</julianDay>.
  <Gastro_Pct_ER>8.09899665551839</Gastro_Pct_ER>.
  <Constit_Pct_ER>1.87224080267559</Constit_Pct_ER>.
  <Respir_Pct_ER>8.65306577480491</Respir_Pct_ER>.
  <Rash_Pct_ER>1.67224080267559</Rash_Pct_ER>.
  <Hemorr_Pct_ER>0</Hemorr_Pct_ER>.
  <Botulin_Pct_ER>54</Botulin_Pct_ER>.
  <Neurol_Pct_ER>11.9171683389075</Neurol_Pct_ER>.
  <Pct_ER_Visit>1.430253991455l4E-02</Pct_ER_Visit>.
</syndrome_civ>.
```

FIGURE 4: An example of the syndromic data

The first element in Figure 4 (`julianDay`) represents the day of the year being observed. The next seven elements represent the types of health problems under surveillance: gastrointestinal, constitutional, respiratory, rash, hemorrhagic, botulinic, and neurological, respectively. The value stored in each of these elements is the number of cases of each syndrome discovered in a particular area on the day specified by the julianday element. By classifying the data obtained over several days, a determination can be made as to whether or not any of the counts should be cause for alarm. If any of the counts are abnormally high, the classifier will produce an alert and notify the administrators of the system of the problem so that they can take

appropriate actions. The final element in the syndromic data (`Pct_ER_Visit`) list represents the total percentage of the population hospitalized for any of these ailments.

In addition to syndromic data shown here, the three classifiers can also be applied used to data related to absenteeism in schools, emergency room visits, and the sales of over the counter drugs.

5. Example

Using the data fusion application is simple. In order to use it within RODS, first declare a subclass of `DataFusion` such as `UnweightedMajorityVoting`. Next, add the classifiers to be run as part of an ensemble with the `DataFusion` class's *addClassifier* method. Finally, run the ensemble using the *run* method included in the `DataFusion` class. The return value contains the decision of the ensemble. Figure 5 is an example of code that uses the data fusion application. It uses the `UnweightedMajorityVoting` class as its data fusion algorithm and creates an ensemble using the results of the RLS, WAD, and CuSum classifiers.

```
DataFusion fusion = new UnweightedMajorityVoting();.
/*setting geomode to zero lets RunRLS ignore zip and county*/.
fusion.addClassifier(RunRLS.class.getName(),new RunRLS(state, false, date, d
                                aggWindowSize, RunRLS.daysOfWeekAll, .
                                                    val, null, false, false, 0,
.
/*setting imode to zero lets WAD ignore zip and county*/.
fusion.addClassifier(WAD.class.getName(),new WAD("ED",state,intervalDays, ne
/*setting geomode to zero lets RunCusum ignore zip and county*/.
.
fusion.addClassifier(RunCusum.class.getName(),new RunCusum(state, false, dat
logger.info("results: " + fusion.run());.
logger.debug("finished running");.
```

FIGURE 5: Example of code that uses the data fusion application.

6. Conclusion

This application was written in a way that potentially allows it to be used anywhere that requires an ensemble of classifiers. Since the only class that contains any RODS specific code is `RODSClassifier`, anyone who wishes to incorporate data fusion may be able to use this application. However, an application specific subclass of Classifier may have to be created in order to implement any application specific functionality for the application where data fusion is being used.

References

[1] F. Tsui, J. U. Espino, V. M. Dato, P. H. Gestlande, J. Hutman, and M. M. Wagner, "Technical Description of RODS: A Real-time Public Health Surveillance System," *J Am Med Inform Assoc*, 399-408, 2003.

[2] D. L. Buckeridge, J. K. Graham, M. J. O'Connor, M. K. Choy, S. W. Tu, and M. A. Musen, "Knowledge-Based Bioterrorism Surveillance," *Proc AMIA Symp 2002*, 76-80, 2002.

[3] "RODS Laboratory, Real-time Outbreak and Disease Surveillance," Center for Biomedical Informatics, University of Pittsburgh, 2005, http://rods.health.pitt.edu.

[4] M. R. Sabhnani, D. B. Neill, A. W. Moore, F. Tsui, M. M. Wagner, and J. U. Espino, "Detecting Anomalous Patterns in Pharmacy Retail Data," KDD-2005 Workshop on Data Mining Methods for Anomaly Detection, Chicago, IL, August 2005, http://rods.health.pitt.edu/LIBRARY/2005%20kdd_workshop_otc_submitted.pdf.

[5] A. M. Moore, G. Cooper, R. Tsui, and M. Wagner, "Summary of Biosurveillance-relevant Statistical and Data Mining Technologies," RODS Technical Report, 2003.

[6] E. M. Voorhees, N. K. Gupta, and B. Johnson-Laird, "The Collection Fusion Problem," in D. K. Harman, ed., *The Third Text REtrieval Conference (TREC-3), Gaithersburg, MD, (in press)*, National Institute of Standards and Technology, 500-525, 2005.

[7] D. Optiz, R. Maclin, "Popular Ensemble Methods: An Empirical Study," *Journal of Artificial Intelligence Research*, 11:169-198, 1999.

[8] A. F. R. Rahman, H. Alam, and M. C. Fairhurst, "Multiple Classifier Combination for Character Recognition: Revisiting the Majority Voting System and its Variations," BCL Technologies, Inc. (USA) and the University of Kent at Canterbury, www.bcltechnologies.com/rd/literature/Fuad_DAS.PPT.

[9] B. Zenko, L. Todorovski, S. Dzeroski, "A Comparison of Stacking with MDTs to Bagging, Boosting, and Other Stacking Methods," Josef Stefan Institute, Slovenia, 2003, http://ai.ijs.si/bernard/mdts/pub03.html.

[10] S. L. Rizzo, V. V. Grigoryan, G. L. Wallstrom, W. R. Hogan, and M. M. Wagner, "Using Case Studies to Evaluate Syndromic Surveillance," Technical Report, April 2005; http://rods.health.pitt.edu/Technical%20Reports/2005%20Rizzo_tech_paper%20Using%20Case%20Studies.pdf.

Appendix A: Installing and Running RODS

A.1. Installing RODS

1. Download and install **Java 1.4** or higher. Create the environment variable `JAVA_HOME` and set it to the base directory of the Java installation (for example, `C:\Program Files\Java\jdk1.5.0_03`).
2. Add `JAVA_HOME\bin` to your `PATH` environment variable.
3. Download and install **J2EE 1.4**. Create the environment variable `J2EE_HOME` and set it to the base directory of the J2EE installation (for example `C:\Sun\AppServer`).
4. Add `J2EE_HOME\bin` to your `PATH` environment variable.
5. Download and install **Apache Ant version 1.6.5**. Create the environment variable `ANT_HOME` and set it to the base directory of the Ant installation (for example `C:\apache-ant-1.6.5`).
6. Add `ANT_HOME\bin` to your `PATH` environment variable.
7. Create the environment variable `RODS_HOME` and set it to the directory where you plan to copy all of the RODS code and data (for example `C:\RODS`).
8. Install **MaxDB** database using a Windows command window. The setup file, `sdbinstall.bat` is located in the directory `RODS/install/maxdb/.`
9. Create RODS database by executing the `createdb` batch file located at `RODS/install/maxdb/createdb.bat` from the command line.
10. Build RODS code. The ant `build.xml` script is located in `RODS/src`.

A.2. Running RODS

The `rodsdata.bat` should automatically rebuild RODS, start up JBoss and the HL7Server, and run the classifiers. However, if you wish to do this manually, execute the following steps.

1. If RODS has been changed since the last time it was run, execute `Ant`. Make sure that JBoss and the HL7Server have been shut down or else `Ant` will not be able to run. The `build.xml` file is in the `RODS/src directory`.
2. Wait for `Ant` to finish executing and then execute the `runrods.bat` batch file. This will start up JBoss and the HL7Server.
3. In order to run the classifiers, execute the `rodsdata1.bat` batch file. This will recompile `BackgroundLoader`, `BackgroundDataManager`, and `RodsData`, and will execute the `RodsData` class.

Distribution

1	MS0188	D. Chavez, LDRD Office, 1030
1	MS0370	Dave Gay, 1411
1	MS0370	Scott Mitchell,1411
1	MS9154	Keith Vanderveen, 8112
1	MS9159	Heidi Ammerlahn, 8962
10	MS9159	Genetha Anne Gray, 8962
1	MS9159	Pam Williams, 8962
1	MS9201	Marion Martin, 8114
1	MS9292	Ken Sale, 8321
1	MS9292	Malin Young, 8321
2	MS9018	Central Technical Files, 8945-1
2	MS0899	Technical Library, 4536

This page intentionally left blank.