ROOT: A Data Analysis and Data Mining Tool from CERN

Ravi Kumar ACAS, MAAA, and Arun Tripathi, Ph.D.

Abstract

This note briefly describes ROOT, which is a free and open-source data mining tool developed by CERN, the same lab where the World Wide Web (WWW) was invented. Development of ROOT was motivated by the necessity to address the challenges posed by the new generation High Energy Physics experiments, which are expected to produce and analyze thousands of terabytes of very complex data every year.

ROOT is an object-oriented data analysis framework, written in C++. It contains several tools designed for statistical data exploration, fitting, and reporting. In addition, ROOT comes with powerful high-quality graphics capabilities and interfaces, including an extensive and self-contained GUI development kit that can be used to develop easy to use customized interfaces for the end users. This note provides some simple examples of how ROOT can be used in an insurance environment.

INTRODUCTION

In this paper, we provide an introduction to some features of ROOT [1] by using it to simulate and analyze the simulated data. We also show some very basic, but necessary, first steps needed for one to become familiar with ROOT. Going through this process will hopefully give the reader a flavor of some of the analysis tasks that can be accomplished within ROOT. Also, hopefully this will provide the reader enough of a familiarity and hands-on experience with ROOT so that they can start using its more advanced features, customized to their own needs.

We want to emphasize that this is just a preview, intended for readers who might not be familiar with ROOT at all. The scope of various tasks that can be accomplished using ROOT is much more comprehensive. We will provide Web links and references at the end of this paper for the curious reader who wants to learn more about this tool.

ROOT is a free, open-source, object-oriented data analysis framework based on C++. This tool was developed at CERN [2], which is a particle physics lab located near Geneva, Switzerland. It is interesting to note that CERN is the same lab where the World Wide Web was born [3, 4].

Development of ROOT was motivated by the need to address the challenges posed by the experimental high-energy physics community, where scientists produce and analyze vast amounts of very complex data. For example, the ATLAS [5, 6] experiment at the Large

Hadron Collider (LHC) [7] at CERN will be generating over 1,000 terabytes of data per year. And this is just one of the experiments running at LHC.

ROOT is being used widely by several experiments in high-energy physics, astrophysics, etc. [8]. In terms of the cost of these research projects, and the people involved, the ROOT user community comprises a multibillion dollar "industry," with the labs and the users located pretty much across the whole planet.

WHY ROOT?

ROOT is a very appropriate tool for use by actuaries and other insurance analysts who do ad hoc data analysis and predictive modeling type work.

ROOT is a framework that is specifically designed for large scale data analysis. ROOT stores data in a very efficient way in a hierarchical object-oriented database. This database is machine independent and highly compressed. If one loads a 1 GB text file into a ROOT file, it will take up much less disk space than the original text file. ROOT also has tools to interact with data in a very efficient way. It has built in tools to do multi-dimensional histograms, curve fitting, modeling and simulation. All these tools are designed to handle large volumes of data.

Conversely, relational databases (databases where the data is organized as tables and rows) were originally designed for transactional systems and not for data analysis. Thus a relational database is very good for use in a policy administration system, which looks at one policy at a time, or claim administration system, which looks at one claim at a time. But, when one is interested in segmenting the data across all the policies or across all the claims, a relational solution falls apart. In order to make the relational solution work for large scale data analysis, we use the brute force method. A typical brute force method will involve adding considerable computing power, adding sophisticated I/O capabilities such as cache, etc., adding numerous indices to tables, creating additional summaries of the data (like OLAP cubes), and other similar techniques. If one loads a 1 GB text file into a relational database, it will take up multiple gigabytes to just store the data. When one further tweaks the database for performance with additional indices, pre-summaries and such, the original 1 GB data would have exploded to something very large. Most (if not all) of the commercial software for data analysis is built for accessing data from relational databases. These commercial tools cannot overcome the fundamental flaw in the way data is stored (tables and rows) except by using brute force.

Some data analysis tools are very memory intensive. Some data analysis tools are very I/O intensive. Some data analysis tools are both memory intensive and I/O intensive (like most commercial business intelligence tools operating on relational databases). In these systems, even if the data grows on a linear scale, the performance of the system degenerates on an exponential scale. Thus, these systems are not easily scalable, whereas ROOT stores and retrieves data in an optimal way that is conducive for data analysis. It avoids most memory issues and I/O performance issues by seamlessly buffering the data between memory and storage. One can thus get a very reasonable throughput from ROOT even from a small PC (all the analysis reported in this paper was done on a PC). A laptop with ROOT as a data analysis tool may be able to give a better performance than a powerful mainframe using one of the commercially available data analysis tools. ROOT can thus be a solution adopted by one person in an insurance company. Once proven, it can be easily extended to an entire team of data analysts or as a corporate wide solution. A ROOT solution is very highly scalable.

ROOT might be an appropriate solution even for smaller data sets. Typically, predictive modeling and ad hoc data analysis involve presenting the data in different graphical/tabular forms. These presentations are best done in a notebook device. This is one of the reasons why Excel is very popular among the actuaries. Using Excel, one can play with the data and once a story emerges from the data, it becomes easy to share the story with the rest of the team. This concept can be loosely termed as interactive computing. When one wants to do analysis on one column in an Excel spreadsheet, the entire spreadsheet must be read into memory. Like Excel, other technologies also suffer similar inefficiencies. When data is stored as tables and rows as in a relational database, subsets of the data cannot be accessed or modified in an efficient way without touching other parts of the data. The design of ROOT allows access to subsets of data without the need to touch the rest of the data. An entire ROOT file can be read sequentially if all the information must be processed. With no data explosion issues, a ROOT file can also be read randomly to process just a few attributes if that is what the analysis requires. ROOT is thus able to give us interactive computing capabilities where other solutions fail.

There are many other reasons why ROOT is an appropriate tool for predictive modeling. But efficiency in storing and accessing the data is where ROOT stands out from any other tool that is in the market today.

HOW TO GET ROOT

ROOT can be downloaded under GNU Lesser General Public License [9] from the ROOT download page [10]. Installation instructions are also provided there. A ROOT user's guide [11], complete class reference [12], tutorials [13], and useful how-to's [14] are also available online. A searchable ROOT user forum, called Root Talk [15], is a useful resource to find answers to several of the questions an average user might come up with.

Any references in this paper to the ROOT user's guide correspond to version 5.16, which is the current production version of ROOT as of the writing of this paper.

Throughout this paper, we will sometimes provide the CPU time taken for a given analysis. These times were measured on a PC running Windows XP, with a 1.73 GHz Intel Pentium M processor, and 1 GB of memory. Also, all these analyses were performed using CINT, the C interpreter provided by ROOT. See chapter 7 of the ROOT users guide to learn about CINT.

STARTING ROOT

If ROOT was installed correctly, a tree-shaped icon, shown in Figure 1, should automatically appear on your Windows desktop.

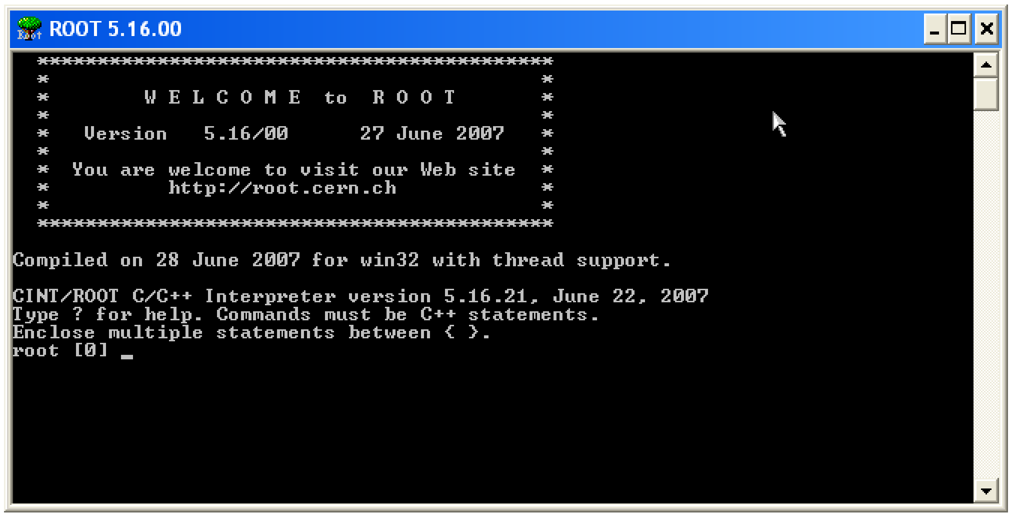
Figure 1: The ROOT shortcut icon on Windows desktop.

In order to start ROOT in Windows, just double-click on this icon. This will start the ROOT console, which is shown Figure 2.

In Unix/Linux environment, ROOT can be started by issuing the following command from the command line:

```
$ROOTSYS/bin/root
```

ROOTSYS is the environment variable pointing to the directory where ROOT was installed. If the directory containing the ROOT executable is already in the system path, then one just needs to type "root" from command line to start ROOT. Regardless of the operating system, the resulting ROOT console will appear the same (as shown in Figure 2).



```
  *******************************************
  *                                         *
  *        W E L C O M E  to  R O O T       *
  *                                         *
  *   Version   5.16/00      27 June 2007   *
  *                                         *
  *  You are welcome to visit our Web site  *
  *          http://root.cern.ch            *
  *                                         *
  *******************************************

Compiled on 28 June 2007 for win32 with thread support.

CINT/ROOT C/C++ Interpreter version 5.16.21, June 22, 2007
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
root [0] _
```

Figure 2: The ROOT console.

LOADING DATA INTO ROOT

ROOT provides TTree and TNtuple classes to store and access data efficiently. Chapter 12 of the ROOT user's guide provides a detailed discussion of ROOT Trees, why one should use them for storing data, and how to read data into a ROOT tree.

ROOT trees are designed specifically to store large volumes of data very efficiently, resulting in much smaller files on disk. Also, since a tree stores data in hierarchical branches, each branch can be read independently from any other branch. This can make for a very fast access to the data, since only the necessary information is read from disk, and not necessarily the whole file.

A very simple example of how to read data into a ROOT tree is given in appendix A. This example converts a space delimited file into a ROOT file, which can then be explored/manipulated further using ROOT.

ROOT also provides interfaces using ODBC to relational databases such as ORACLE, MYSQL, etc.

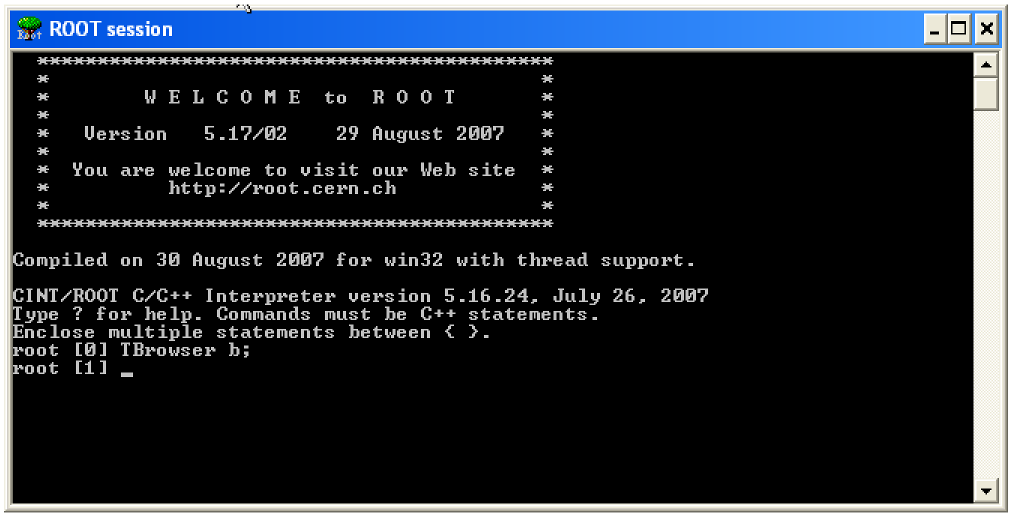
EXPLORING A ROOT FILE

The TTreeViewer class [16] of ROOT provides a GUI for convenient exploration of data, once it has been converted into a ROOT tree. In order to illustrate some of its functionality, we will use the ROOT file generated by the sample data load program mentioned in the previous section. Chapter 12 of the ROOT user's guide describes how to start a Tree Viewer.

First, one has to start a ROOT object browser (TBrowser class [17]) from the ROOT console:

root [] TBrowser b;

Figure 3 shows a screen shot of the ROOT console, with this command.



```
  ********************************************
  *                                          *
  *        W E L C O M E  to  R O O T        *
  *                                          *
  *   Version   5.17/02     29 August 2007   *
  *                                          *
  *  You are welcome to visit our Web site   *
  *          http://root.cern.ch             *
  *                                          *
  ********************************************

Compiled on 30 August 2007 for win32 with thread support.

CINT/ROOT C/C++ Interpreter version 5.16.24, July 26, 2007
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
root [0] TBrowser b;
root [1] _
```

Figure 3: A screen shot of the ROOT console, with the command to start the ROOT browser.

This will start the ROOT object browser, which looks like figure 4.

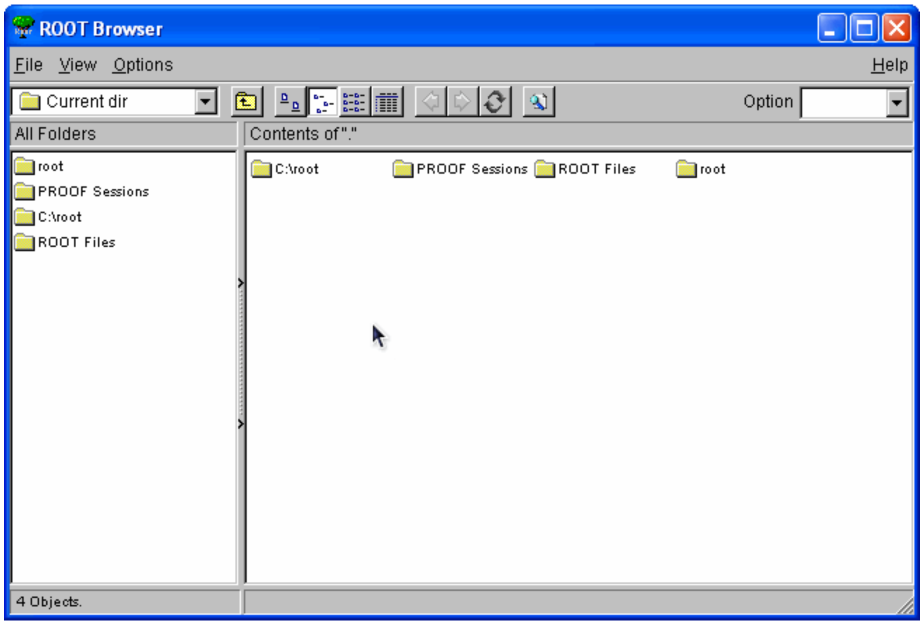


Figure 4: A screen shot of ROOT object browser.

Now, one can use this object browser to open a ROOT file by using **File->Open** menu. In this case, we will navigate to the ROOT file generated by the sample data load program of Appendix A. Using the File menu, open the root file called "SampleData.root". Figure 5 shows a screen shot of the file selection dialog, used to open the file.

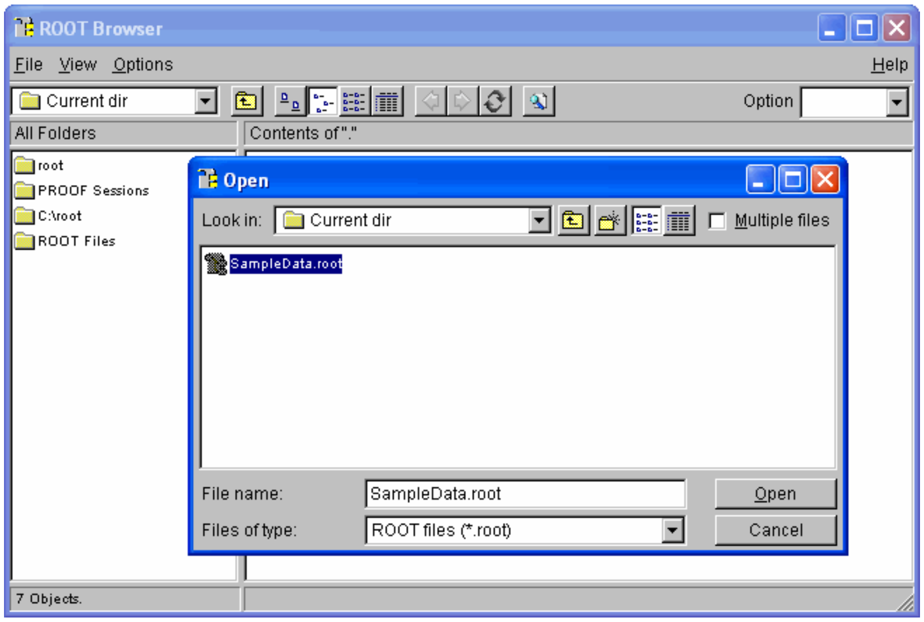


Figure 5: Screen shot of the ROOT object browser and file selection window, after navigating to the ROOT file generated by the sample data load program.

After selecting the appropriate ROOT file, click the "open" button in the file selection dialog. This will close the file-selection window, and the object browser will again appear like Figure 4. At this point, double-click the icon labeled "ROOT Files" in the right-hand panel of the object browser. After this action, the browser looks like Figure 6.

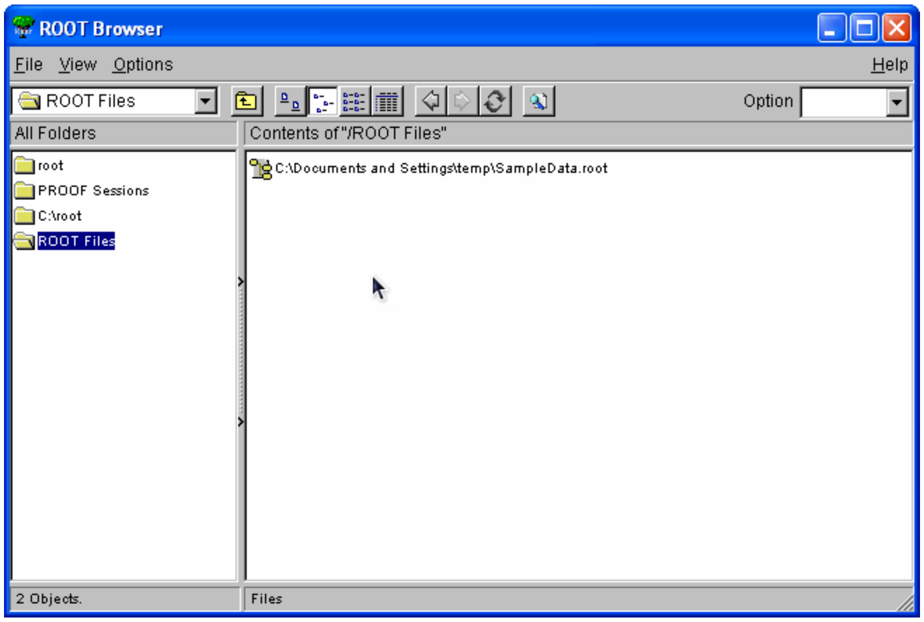


Figure 6: Appearance of the ROOT browser after double-clicking on the "ROOT Files" icon.

Notice that an icon representing the selected ROOT file appears in the right panel of the browser. The absolute path indicating the file name and location is also shown. Now, double-click on this ROOT file icon. The browser will now look like Figure 7.

Notice that a tree icon appears in the right panel of the browser. This is the tree that we created using the sample data load program. One can create several ROOT trees in a single ROOT file, but in this case, we have just one.

Now, right-click on the tree icon and a menu appears. From this menu, select "StartViewer". A new Tree Viewer window will appear. A screen shot of this window is shown in Figure 8.

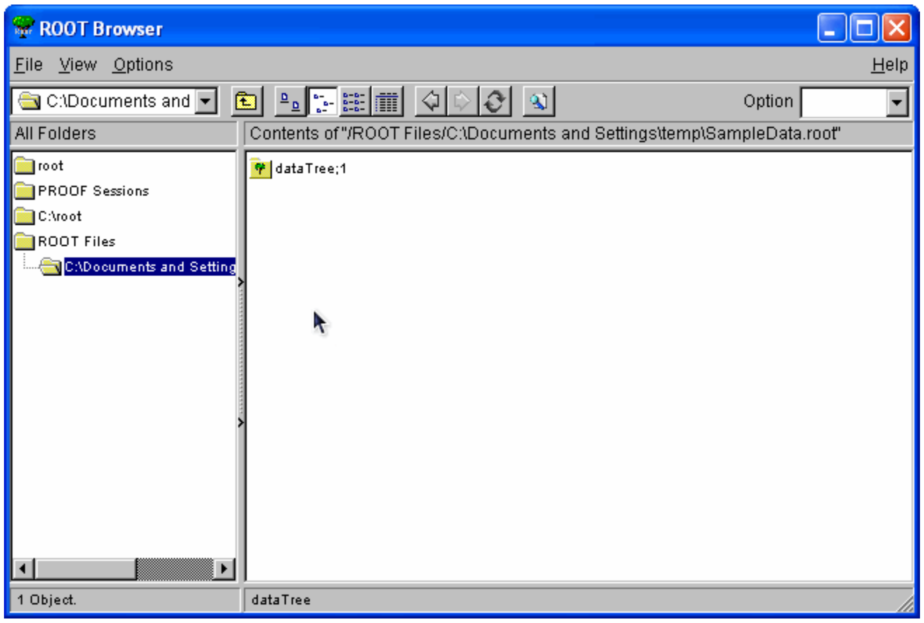


Figure 7: Appearance of the ROOT object browser, after double-clicking on ROOT file icon (shown in Figure 3).

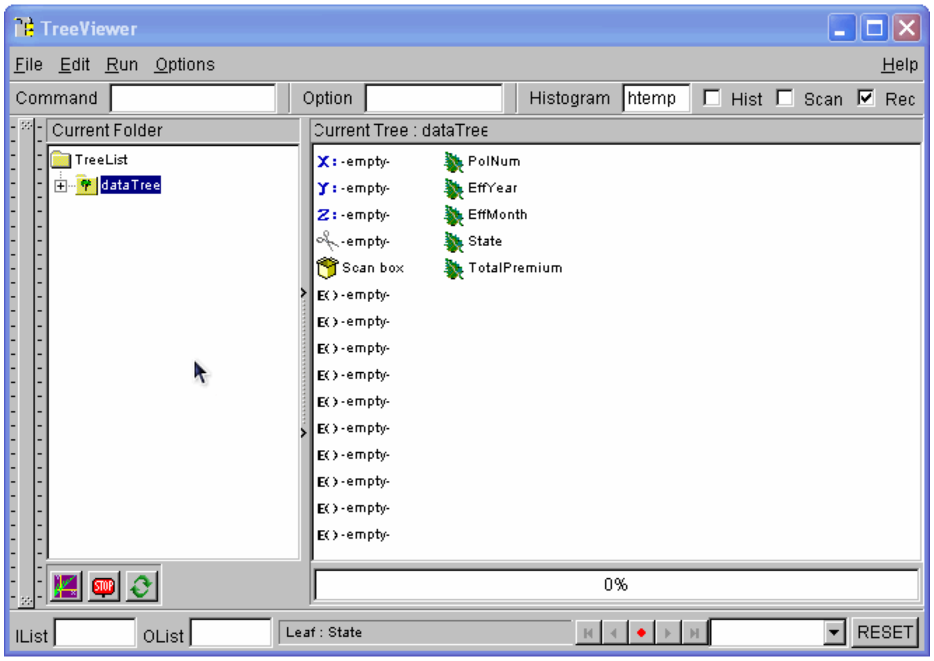


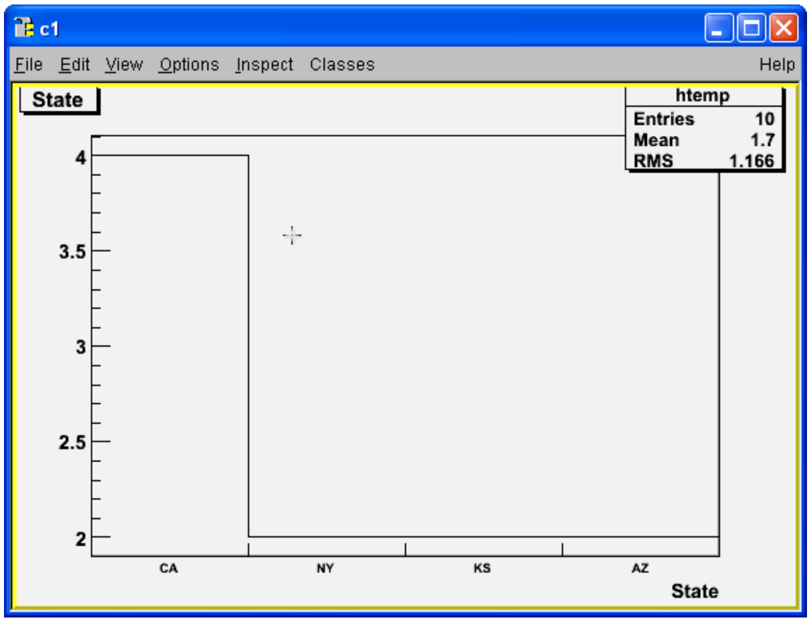
Figure 8: The ROOT tree viewer window, displaying the information contained in the sample data tree.

Notice in Figure 8 the leaf shaped icons in the right panel of the tree viewer. These are the leaves of the ROOT tree we created in the sample data load program. Next to each leaf is the name of the corresponding variable.

The ROOT tree viewer is a powerful data exploration tool, which can be used for one-, two-, and three-dimensional data exploration, fitting, etc. In this section, we will just see how to quickly generate one- and two-dimensional histograms from this simple data.

Suppose we want to see a histogram of all the states in our data set, which will show us the number of policies written in each state. In order to get this histogram, simply double-click the leaf labeled "State" in the tree viewer. A separate window (object of type TCanvas) appears with this histogram. A screen shot of this window is shown in Figure 9. We see that there are four policies in California, and two each in New York, Kansas, and Arizona; as expected from our input data.

The TCanvas object itself is a very complex object, which allows the user to interactively explore the data, and customize the visual appearance of the graphics that appears on the canvas. See chapter 9 of the user's guide for more details.



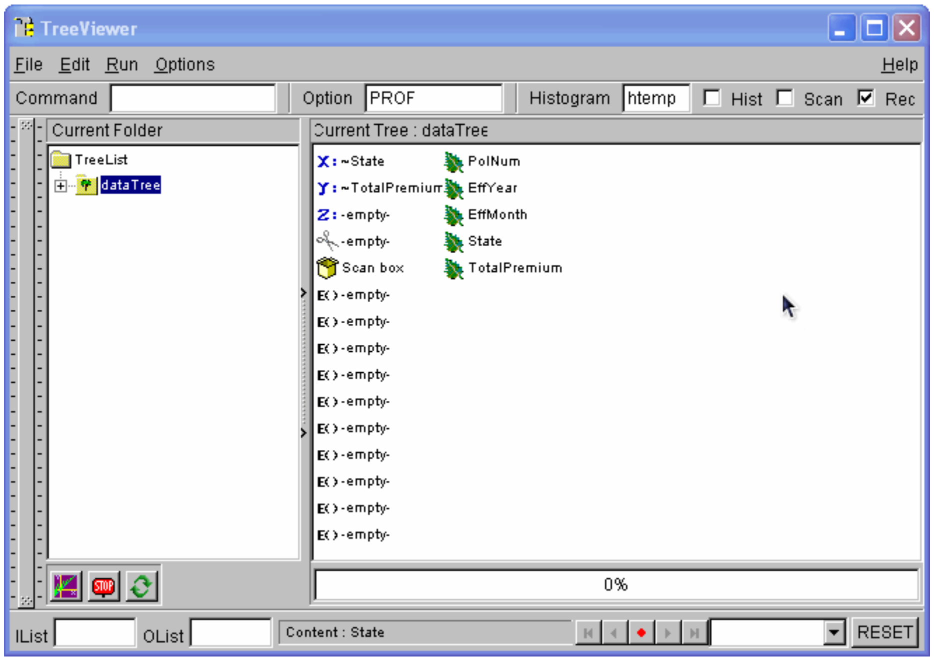
```
htemp
Entries      10
Mean        1.7
RMS       1.166
```

Figure 9: A histogram of all the states in the data set used by the sample data load program.

The histogram in Figure 9 can easily be saved to disk in various formats, e.g., gif, pdf, png, etc., by using **File->Save** (or Save As) menu in the menu bar at the top of the canvas.

The reader is encouraged to explore this interactive canvas, all the objects on it, and the associated context menus. Right-click on different parts of the canvas, e.g., the title box, statistics box, the histogram area, the lines of the histogram, the axes, etc., and explore the large amount of interactive functionality available in the context menus.

Now, suppose we are interested in finding out the average premium collected in each state. We can do that in the following manner. In the tree viewer (Figure 8), drag the leaf labeled "TotalPremium" to the icon (in the same panel) labeled Y (which is empty by default). Then, drag the leaf labeled "State" to the icon labeled X. This tells the tree viewer to plot TotalPremium (Y-axis) vs. State (X-axis). Next, using the Options menu at the top menu bar in the tree viewer, set the drawing option to Profile (**Options->2D Options->Profile**). This tells the tree viewer to plot a profile histogram, which plots the average Y value for each bin on the X-axis. After these steps, the tree viewer window should look like Figure 10.



Command | Option PROF | Histogram htemp | ☐ Hist ☐ Scan ☑ Rec

X: ~State
Y: ~TotalPremium

PolNum, EffYear, EffMonth, State, TotalPremium

Content : State

Figure 10: The appearance of tree viewer window, after preparing it to plot average total premium in each state.

In Figure 10, notice that the X icon says State next to it and the Y icon says TotalPremium next to it. Also notice that the Option is set to PROF in the small Option window, below the top menu bar. Now we are ready to produce the graph we want. Just click on the graphing icon near the bottom left corner of the panel, on the left of the icon labeled STOP. Again, a new canvas pops up, with the desired graph. A screen shot of this canvas is shown in Figure 11. Notice how the average premium, as well as RMS (root mean square) is plotted for each state. This allows for a visual exploration of the relationships between any two variables quickly. One can also perform fits to this relationship. The reader is referred to the ROOT user's guide to learn how to perform interactive fits to the data points on a canvas. For example, see chapter 5 of the ROOT user manual.

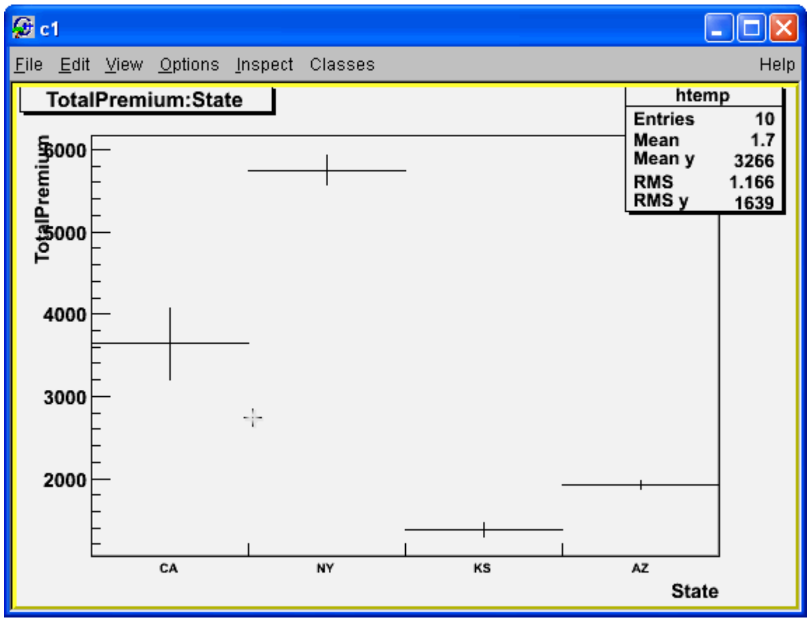


Figure 11: Graph of average premium vs. state.

Now, suppose we want to apply filters to our data. The tree viewer allows us to do that easily. In order to illustrate how to do this, we will create the profile histogram of average total premium vs. state, but this time, we only want to look at the states of California and New York.

There is a scissors shaped icon in the tree viewer, just below the "Z" icon (see Figure 10). This is used to apply "cuts" or selection criteria to the data. The cut can be any valid C++ expression, involving one or more of the variables stored in the tree. This expression must first be defined, using the "E()" icons in the tree viewer. All these expressions are empty by default.

In order to achieve our goal of looking at the average premium only in California and New York, double-click on the first "E()" icon in the tree viewer of Figure 10 (we will assume that the tree viewer is in the state shown in Figure 10; if not, follow the instructions above to bring it this state). A dialog appears that allows us to type in the selection criterion and also give it a name. Figure 12 shows the screen shot of the tree viewer after double-clicking on the "E()" icon.

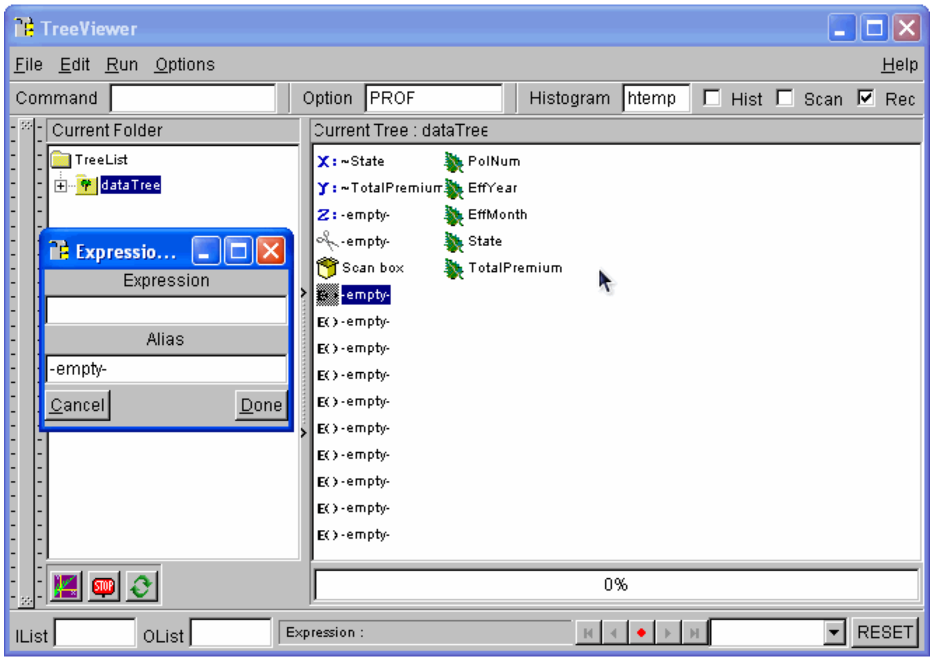


Figure 12: The tree viewer after double-clicking on the "E()" icon. Notice the dialog box at left-center of the tree viewer.

Now, we can type in our selection criterion (a valid C++ statement, involving variables in the tree) in this box, and also assign it a name (alias). We just want to keep California and New York. So in the Expression dialog, we will type (State = = \"CA\") | | (State = = \"NY\"). Note that the double quotes are preceded by backslash, since the expression itself is a string, and within this expression we are comparing with a string. Also keep in mind that both C++ and ROOT are case-sensitive.

In the alias box, we will give it the name "Cut1". After these steps, the screen shot of the tree viewer is shown in Figure 13.

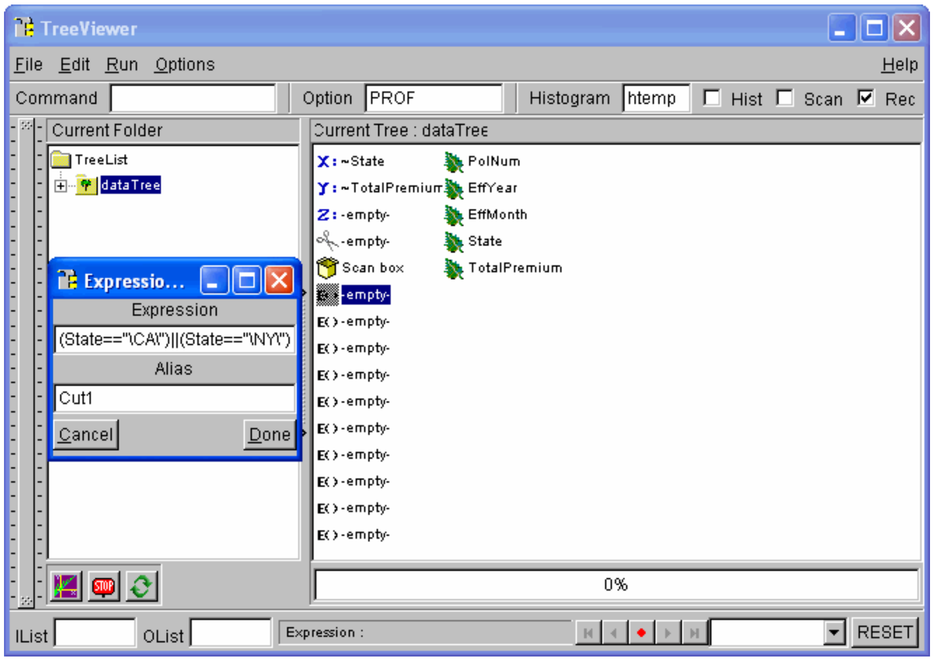
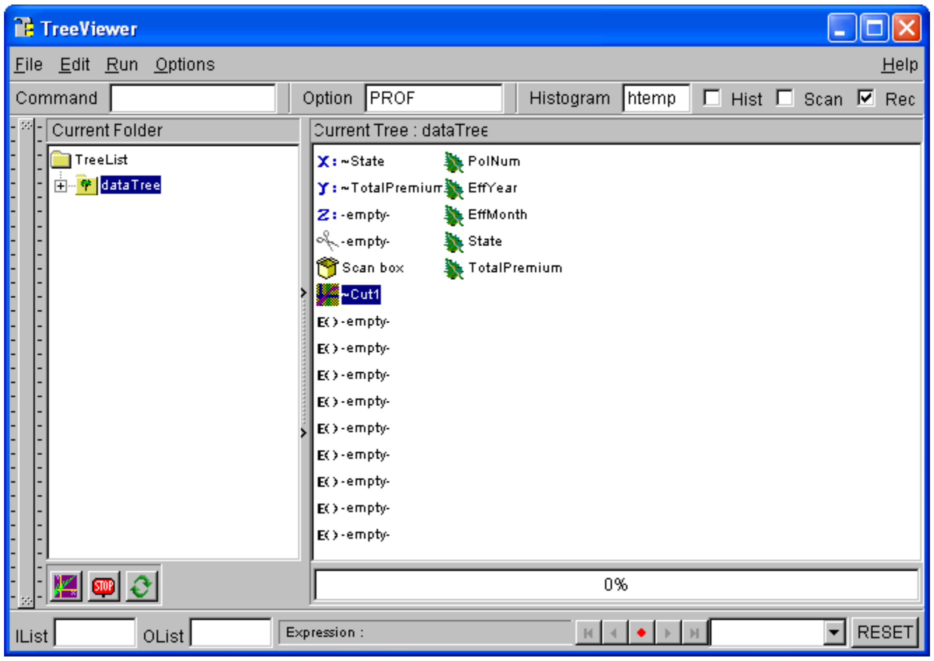


Figure 13: The tree viewer after typing in the selection criteria in the expression dialog.

Now, click the "done" button in the selection expression dialog. This will close the selection expression dialog. The screen shot of the tree viewer after this step is shown in Figure 14. Notice how the appearance of the selection expression icon we just edited has now changed. It no longer shows "E()", and it is no longer empty. Now it contains a valid selection criterion, which we can use to filter data. Notice how the scissors icon is still "empty"; meaning our selection criterion has not been activated yet.

In order to activate this filter, simply drag this selection expression icon (Cut1) to the scissors icon. After this step, the tree viewer should look like Figure 15.

Now we are ready to make the plot we need. Simply click on the drawing icon (near bottom left, just left to the "STOP" icon). The canvas will now show the updated plot, which is shown in Figure 16. Notice that only the states of California and New York appear on the X-axis now, according to our selection. One can apply any arbitrarily complex filter to the data in this manner.



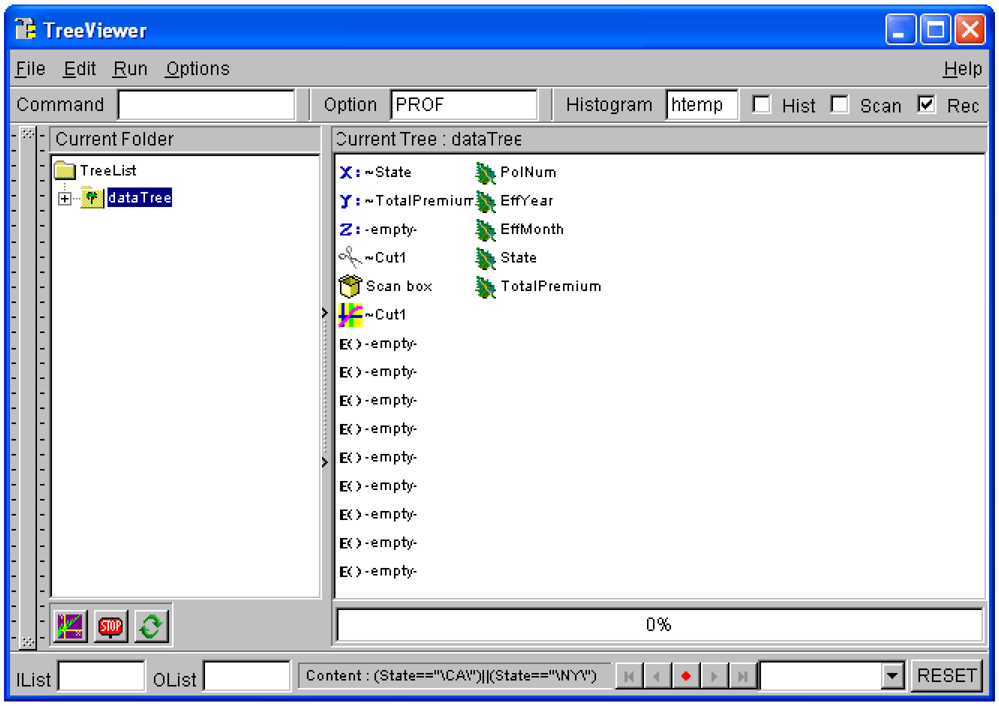
Option PROF
Histogram htemp
☐ Hist ☐ Scan ☑ Rec

X: ~State
Y: ~TotalPremium
Z: -empty-
-empty-
~Cut1
E() -empty-
E() -empty-
E() -empty-
E() -empty-
E() -empty-
E() -empty-
E() -empty-
E() -empty-
E() -empty-

0%

Expression :

Figure 14: Appearance of the tree viewer after editing the first selection (cut) expression.



Option PROF
Histogram htemp
☐ Hist ☐ Scan ☑ Rec

X: ~State
Y: ~TotalPremium
Z: -empty-
~Cut1
~Cut1
E() -empty-
E() -empty-
E() -empty-
E() -empty-
E() -empty-
E() -empty-
E() -empty-
E() -empty-
E() -empty-

0%

Content : (State=="\CA\")||(State=="\NY\")

Figure 15: Appearance of the tree viewer after activating the selection criterion (Cut1).

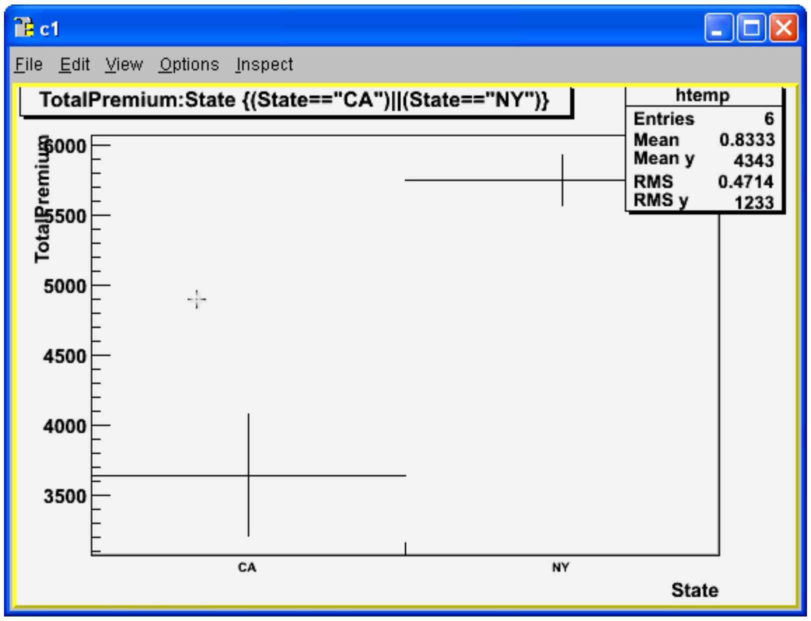


Figure 16: Plot of average total premium vs. State, after applying the state selection criterion.

Finally, all steps we have performed so far in the tree viewer can easily be saved as a ROOT macro (which is basically a C++ function, using ROOT classes), which can be run from the ROOT console to quickly reproduce these steps at a later time. The reader is referred to the ROOT user's guide to learn how to accomplish that.

DATA SIMULATION

Complex simulations are a significant part of research in a typical high-energy physics experiment, and ROOT provides tools to facilitate this process.

For the purpose of this paper, we prepared some simulated "insurance-like" data. The code used for this simulation is provided in appendix B. For this study, we chose to simulate 5 Million BOP policies. The loss ratio for each of these policies was generated based on five predictive variables:

- Age of business: Number of years the business has been around.
- Building Count: Number of buildings insured under the policy.
- Credit score: The commercial credit score of the business.
- Policy Age: Number of years the policy has been in force.
- Total Building Insurance: The total amount of building insurance on the policy.

First, for each observation, each of these variables was generated independently. Age of Business, Building Count, and Total Building Insurance were generated using Landau distribution. Credit Score was simulated using a flat distribution (on a scale of 1-100), and

Policy Age was made to follow a linear distribution, subject to the condition that policy age must be less than or equal to the building age.

In addition to these predictive variables, we also generated three random variables, called ran1, ran2 and ran3. These variables are uniformly distributed between 0-10. These variables don't have any predictive power, by design. But we will include them in our list of predictive variables while searching for and ranking the most predictive variable. The tools being used to search for the predictive variable should be able to identify these random variables as non-predictive. This works as a sanity check of the tool.

The loss ratio from the five predictive variables was generated using the following equation:

$$\text{LossRatio} = 0.5 - 0.00053*\text{AgeOfBusiness} + 0.0025*\text{BuildingCount} - 0.00057*\text{CreditScore} - 0.0227*\text{PolicyAge} + 0.0437*\text{TotalBuildingInsurance} \quad (1)$$

In addition, a random Gaussian noise is added to the LossRatio, with mean 0 and standard deviation of 0.04. Roughly speaking, this corresponds to smearing the true loss ratio by about 10%.

Figure 17 shows histograms of all the five predictive variables, and the loss ratio. We have chosen, on purpose, to simulate only the "lossy" policies.

We would like to point out one thing here. Since we simulated our data such that the relationship between the target variable (Loss Ratio) and the five predictive variables follows equation (1) above, we know exactly what results to expect from a correctly done regression analysis of this simulated data. The regression should give us back, within statistical uncertainties, the relationship defined in equation (1) above; otherwise something is wrong with the analysis.

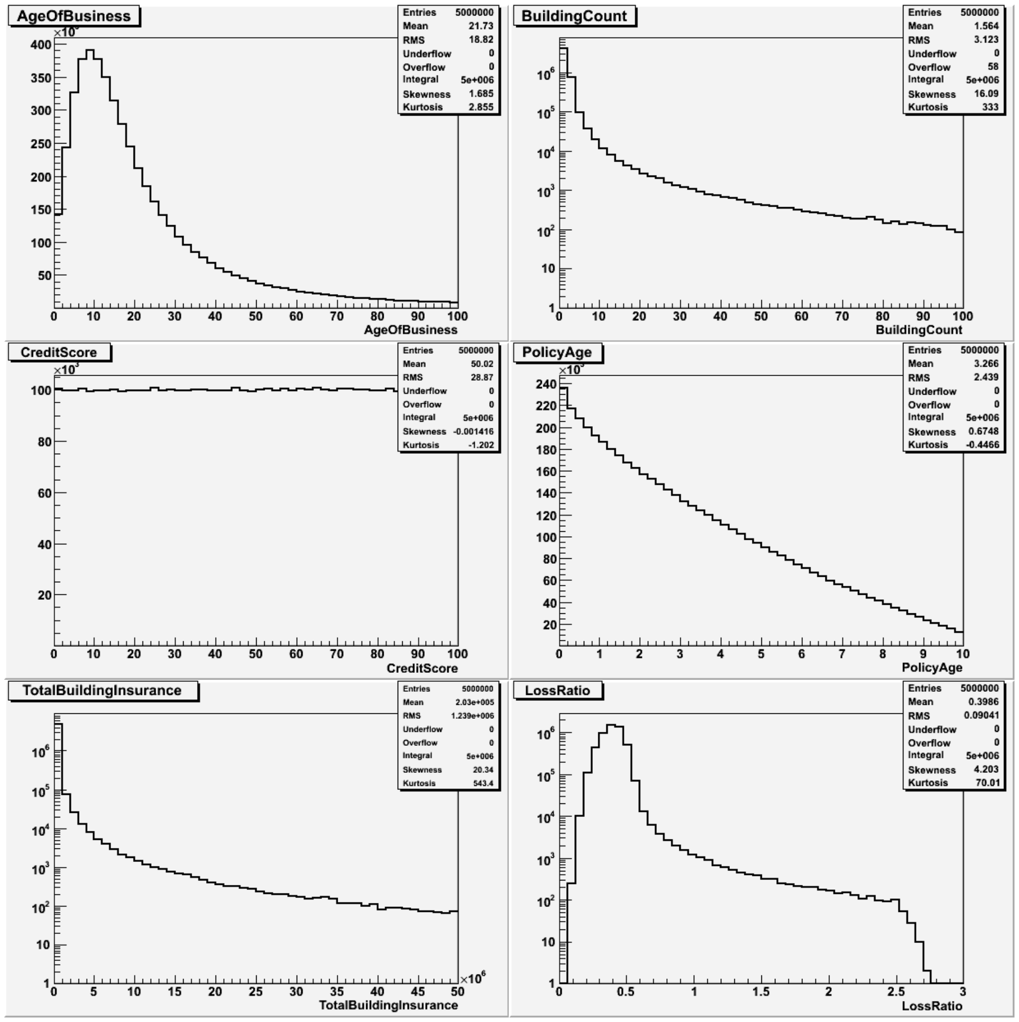


Figure 17: Raw histograms of the five simulated predictive variables and the loss ratio. The figure titles show which figure corresponds to which variable.

EXPLORATION OF SIMULATED DATA

Figure 18 shows several profile histograms, plotting LossRatio vs. the five predictive variables. These profile histograms show the mean value of the loss ratio, for a given value of the X-axis. This is a useful way to visualize the relationship between the target and predictive variables. In our simulated data set, each of the five predictive variables is correlated, by design, with the target variable. Consequently, we should see a non-flat pattern in all these profile histograms, which we do.

It may be useful to point out here the time taken to generate the 11 figures shown in Figures 17 and 18. Each of these figures was generated, as mentioned earlier, from the simulated data set of 5 Million policies. It took a total of 50 CPU seconds to generate these 11 figures.

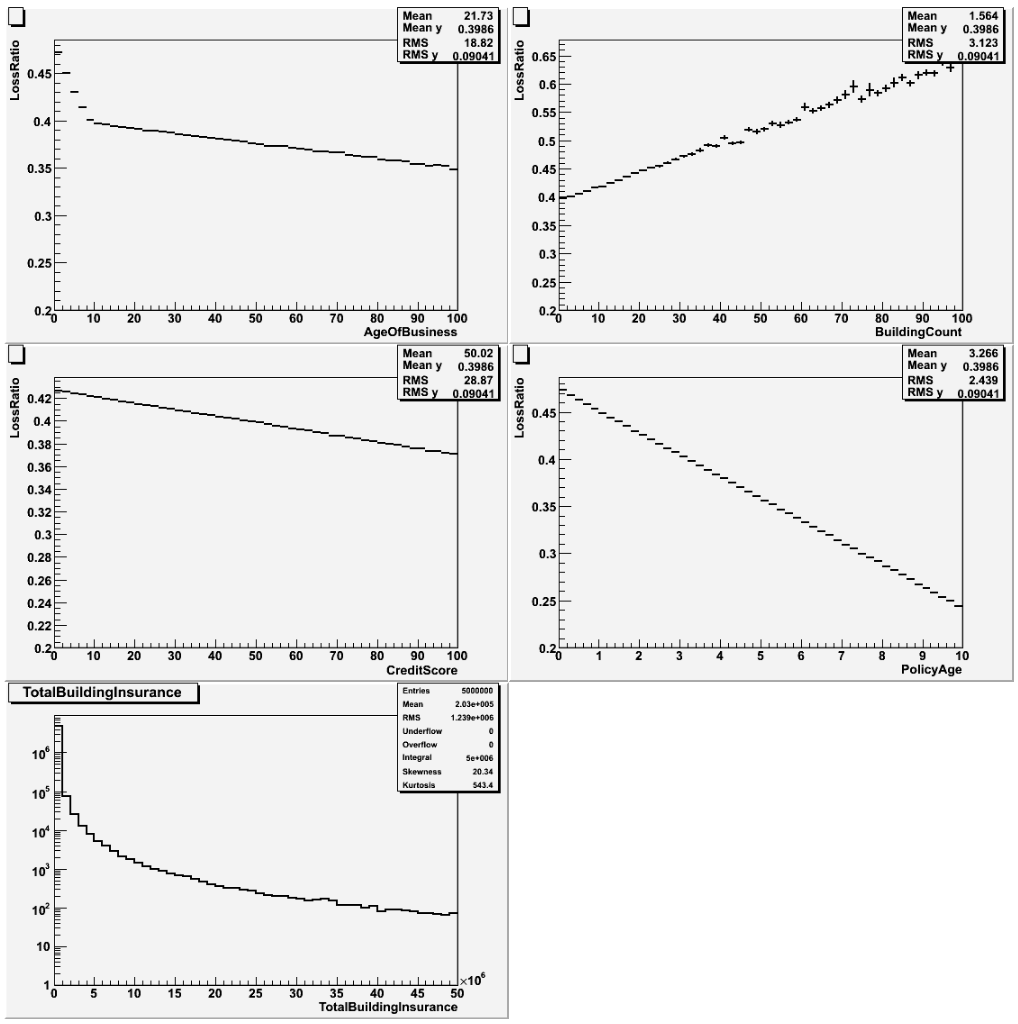


Figure 18: Profile histograms of LossRatio vs. various predictive variables. The Y-axis in each of these plots shows the average loss ratio for the corresponding value of the X-variable.

SEARCHING FOR PREDICTIVE VARIABLES

The first step in building a predictive model is to identify which variables to use as predictors. The TMVA [18, 19] package provides tools to facilitate this process. TMVA is an acronym for "Toolkit for Multivariate Data Analysis." It provides several classification algorithms, mainly designed to tackle the task of separating signal from background in complex high-energy physics data. However, it can be used in any environment where classification of data is needed. TMVA already comes bundled with ROOT.

Several of the algorithms in TMVA also provide the ranking of the predictive variables in terms of their discriminating power. These algorithms are: Likelihood, H-Matrix and

Fisher discriminants, Neural Network, Boosted Decision Trees, and RuleFit. The reader is referred to the TMVA user's guide [18, 19] for more details of TMVA and the available classification algorithms.

For our simulated data, we will use Fisher Discriminants, as an example, to find out the ranking of the discriminating power of the various variables. First, we need to determine the classification we are interested in. A natural classification of interest for us is based on loss ratio—we would like to use as predictors the variables that can effectively separate policies with low loss ratio from those with high loss ratio. Once we have identified the variables that are most effective in providing this kind of discrimination, then we can use them, for example, in multiple regression, to come up with the regression equation that will predict loss ratio from these predictive variables.

For our analysis, we define (arbitrarily) policies with loss ratio less than 0.4 as "good policies" (signal), and those with loss ratio greater than 0.4 as "bad policies" (background). Our goal is to find the ranking of predictive variables, based on how well they can distinguish between these two kinds of policies. Borrowing from high-energy physics terminology, we will use the words signal (loss ratio < 0.4) and background (loss ratio >= 0.4) from now on to represent these two classes of data.

We fed all the five predictive variables (shown in equation 1), and the three random variables (ran1, ran2, ran3) to the Fisher classifier. The random variables were included to test the effectiveness of the classifier—it should be able to identify these as the least predictive. The data sample used consisted of 100,000 observations of each type (signal and background, as defined above) for training and testing. So a total of 400,000 observations were used in this analysis.

Rank	Variable	Discriminating Power
1	PolicyAge	1.20E-01
2	CreditScore	1.51E-02
3	AgeOfBusiness	1.06E-02
4	TotalBuildingInsurance	4.56E-05
5	BuildingCount	2.24E-05
6	ran3	7.15E-06
7	ran2	8.95E-07
8	ran1	1.36E-07

Figure 19: The results of variable ranking using Fisher discriminant in TMVA.

Figure 19 shows the results of this analysis. We can see that the classifier has correctly identified ran1, ran2, and ran3 as least predictive. This is just a simple example of how

TMVA can be used to search for predictive variables. The meaning of rankings can be found in the TMVA user's guide.

From now on, we will just keep the five predictive variables for fitting and discard the three random variables.

FITTING

Once we have identified the predictive variables we want to use to make the prediction, we need to perform a fit to the data using these variables to come up with the optimal relationship between the target variable and the predictive variables. In our case, we are interested in obtaining the relationship between LossRatio and the five predictive variables shown in equation 1. Obviously, several techniques can be used to solve this problem, depending on the characteristics of the data at hand. ROOT provides several tools to achieve this, including least squares regression, method of maximum likelihood, neural networks, etc. In this section, we will show the results from using just a couple of these tools.

10.1 The TLinearFitter Class

We will use TLinearFitter [20] class in ROOT to fit LossRatio, with the five predictive variables mentioned above. We used the first 1 million observations from our simulated data set for the purpose of this fit. The total CPU time taken for this analysis was 11 seconds.

Figure 20 below shows the values of the parameters obtained by the fit, and also their true value. Since we simulated the relationship between the target variable, and the predictive variables, we know exactly what the correct parameter estimates should be. This allows us to do an *absolute end-to-end calibration/verification* of the analysis/regression chain.

	Parameter Estimate	**True Value**
Variable Name	**from the Fit**	**of the Parameter**
Intercept	0.50	0.5
AgeOfBusiness	-5.31E-4	-5.3E-4
BuildingCount	2.47E-3	2.5E-3
CreditScore	-5.70E-4	-5.7E-4
PolicyAge	-2.27E2	-2.27E-2
TotalBuildingInsurance	4.37E-08	4.37E-08

Figure 20: Results of the fit to the simulated insurance data. Also shown are the true values of the parameters.

In Figure 20, if we compare the parameter estimates from the fit with their true values, we see that the two agree quite well. This comparison gives us confidence that our analysis chain used here, from data preparation to fitting, is correct.

In order to test the performance of TLinearFitter on a much bigger data set, we used it to perform a fit on a data set consisting of 49 variables and 9 Million (9E6) observations. This analysis took 16.3 CPU Minutes, and 20.5 Real Minutes.

10.2 Non-parametric Fitting: Neural Network

In the previous section, we fit the simulated data using a linear relationship between the target variable (LossRatio) and the five predictive variables. This was appropriate, since the simulated data indeed follows such a linear relationship.

In several real-life situations, we don't know beforehand what the "true" functional form of the relationship between the target variable and the predictive variables is. In such situations, non-parametric fitting techniques might be more effective.

In this section, we will use ROOT's neural network package [21] to fit the data already simulated.

The network used in this analysis consisted of five input nodes (the five predictive variables used to simulate the data), two hidden layers with 8 and 5 nodes respectively, and a single output node (LossRatio). The network was trained on 2000 data points from the simulated data set, over 300 epochs. All the input and output nodes were normalized so that they take on values between 0 and 1.

Figure 21 shows the structure of the network, with the thickness of the connecting lines (synapses) being proportional to the corresponding weights.

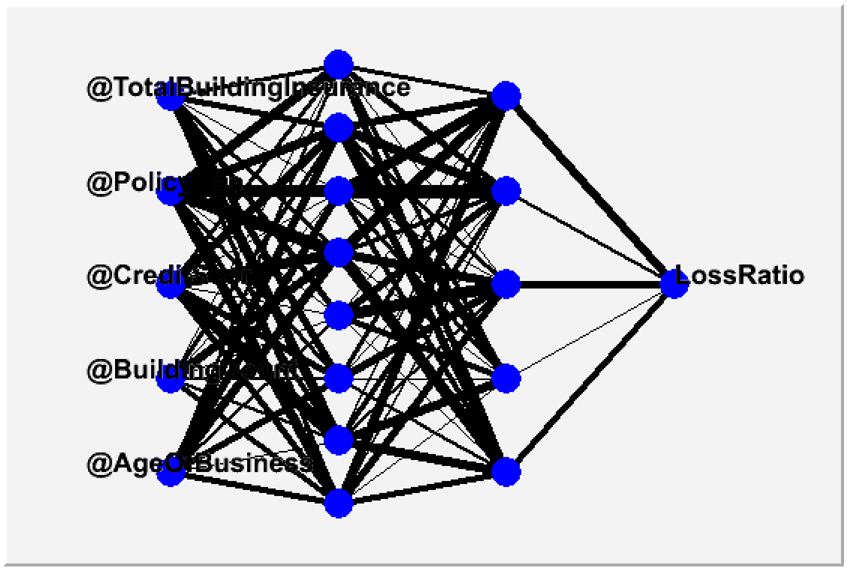


Figure 21: The structure of the neural network obtained by training it on the simulated data set. The thickness of lines is proportional to the weight for the corresponding connection.

Figure 22 shows a profile of the residuals for different values of the output (LossRatio) for the independent test data set.

The optimized neural network can be automatically exported to a C++ class, to be used elsewhere, e.g., for implementation.

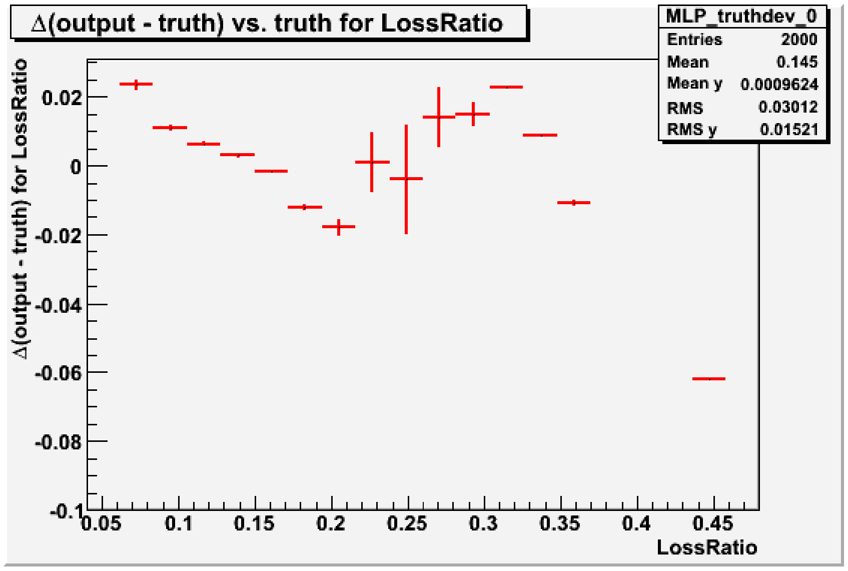


Figure 22: Residual profile for various values of the neural network (NN) output (LossRatio) on the test data set. The Y-axis is the mean value of the NN output minus the true LossRatio, for a given value of the true LossRatio. The true loss ratio has been normalized, so that it takes on values between 0 and 1.

10.3 Other statistical analysis packages in ROOT

The previous two examples use just two of the several statistical analysis classes available in ROOT. Here we list some other classes that might be of interest to anyone interested in statistical analysis of data in ROOT.

- The TMinuit [22] and TFumili [23] minimization packages can be used to do either least-squares or maximum likelihood fitting. Also see chapter 5 of the user's guide.
- The TPrincipal [24] class can be used for principal components analysis.
- The TMultiDimFit [25] is another non-parametric fitting package, which can be used to approximate the relationship between target and predictive variables in terms of simple basis functions.
- The TMath [26] class contains several useful mathematical functions. See chapter 13 of the user's guide.
- In addition to some of the algorithms mentioned earlier in this note, the TMVA package [18, 19] provides several other algorithms for data classification.
- The one-dimensional, two-dimensional, three-dimensional, and Profile histogram classes provide a convenient and interactive way to visualize, explore and fit data. See chapter 3 of the user's guide for more detail.
- The linear algebra package. See chapter 14 of the user's guide.

11. SOME OTHER USEFUL FEATURES OF ROOT

Previous sections demonstrated, using simple simulated data, some simple application of ROOT. In this section, we will list some more tools available in ROOT that might be relevant to us.

- Animation facilities, which can again be used to generate animations of some of the analysis chain, to be used in presentations.
- Efficient random number generators, with large periodicity = 2^{19937}-1. Also, one can easily generate random numbers either following any analytical distribution, or following any empirical distribution provided by the data.
- A complete, self-contained GUI toolkit, including a GUI builder, which can be used to develop customized GUIs for specific tasks. See chapter 25 of the user's guide.
- ROOT comes with CINT, the C/C++-interpreter, which allows one to script an analysis in C++, and quickly execute it on the command line, without having to compile. This provides a quick way to prototype an analysis, which can later be compiled for better performance. See chapter 7 of the user's guide for more details.
- All the libraries available in standard C++, of course, can easily be used and integrated with a ROOT analysis.
- Interfaces to Ruby and Python scripting languages. See chapter 19 of the user's guide for details.
- Parallel processing. See chapter 24 of the user's guide for details.
- Networking. See chapter 22 of the user's guide.
- Automatic HTML documentation generation. See chapter 27 of the user's guide for details.

- Three-dimensional graphics package.

12. SUMMARY AND CONCLUSIONS

In this note, we have briefly introduced ROOT via some simple examples. Hopefully this will give the reader a feel of how to get started with using ROOT in insurance environment. We have also provided links below that a curious reader can follow to get a more detailed and advanced understanding of this tool. The user's guide, online tutorials, and the how-to's pages provide a wealth of information and several working examples that one can leverage to get started with any kind of analysis.

13. REFERENCES AND LINKS:

[1] Rene Brun and Fons Rademakers, *ROOT – An Object Oriented Data Analysis Framework*, Proceedings of AIHENP'96 Workshop, Lausanne, Sep 1996, Nucl. Inst. & Meth. in Phys. Rev. A 389 (1997) 81-86. See also http://root.cern.ch/
[2] http://public.web.cern.ch/Public/Welcome.html
[3] http://info.cern.ch/
[4] http://public.web.cern.ch/Public/Content/Chapters/AboutCERN/Achievements/Achievements-en.html
[5] http://atlasexperiment.org/
[6] http://en.wikipedia.org/wiki/ATLAS_experiment
[7] http://lhc.web.cern.ch/lhc/
[8] http://root.cern.ch/root/ExApplications.html
[9] http://root.cern.ch/root/License.html
[10] http://root.cern.ch/twiki/bin/view/ROOT/Download
[11] http://root.cern.ch/root/doc/RootDoc.html
[12] http://root.cern.ch/root/Reference.html
[13] http://root.cern.ch/root/Tutorials.html
[14] http://root.cern.ch/root/HowTo.html
[15] http://root.cern.ch/phpBB2/
[16] http://root.cern.ch/root/html516/TTreeViewer.html
[17] http://root.cern.ch/root/html516/TBrowser.html
[18] A. Hocker et al., *TMVA – Toolkit for Multivariate Data Analysis,* http://arxiv.org/abs/physics/0703039. Also available at http://tmva.sourceforge.net/docu/TMVAUsersGuide.pdf
[19] http://tmva.sourceforge.net/
[20] http://root.cern.ch/root/html514/TLinearFitter.html
[21] http://root.cern.ch/root/htmldoc/TMultiLayerPerceptron.html
[22] http://root.cern.ch/root/html516/TMinuit.html
[23] http://root.cern.ch/root/html516/TFumili.html
[24] http://root.cern.ch/root/html516/TPrincipal.html
[25] http://root.cern.ch/root/html514/TMultiDimFit.html
[26] http://root.cern.ch/root/html516/TMath.html

Appendix A

SAMPLE DATA LOAD PROGRAM

Below is a simple data load program, which reads a tab-delimited text file, and stores them in a ROOT tree. This tree is then written to an output ROOT file.

The contents of the file to be read are produced below. This is a simple, space-delimited file, which contains 10 records. Each record consists of a Policy Number, Policy Effective Year and Month, State, and the Total Premium collected. The purpose of this exercise is to give a simple example of how one can read data into ROOT. Note that any line starting with "#" is ignored by the ReadFile method, which we will use to read the sample data file below. Therefore, the first line in this sample data set is just for our convenience to tell us what the various fields are.

#PolNum	EffYear	EffMonth	State	TotalPremium
123456789	2006	10	CA	5000
123456790	2007	01	NY	6000
123456791	2005	12	KS	1500
123456792	2007	08	CA	3500
123456793	2007	05	AZ	2000
123456794	2006	11	CA	3500
123456795	2006	04	NY	5500
123456796	2007	02	AZ	1850
123456797	2006	12	CA	2560
123456798	2007	03	KS	1250

The Program to Read the Data

This program reads in the above file, and stores its contents into a ROOT tree. This tree is then written to disk in a ROOT file. This ROOT file then can be use for data visualization, exploration, and analysis.

This program is a ROOT macro, which means it can be executed from the ROOT window. It will not run as a stand-alone program, since it does not contain the "main()" code block. In order to see how to generate stand-alone, compiled ROOT code, refer to the ROOT user's guide.

Assuming this program is saved under the name "C:/Documents and Settings/temp/ ReadSampleData.cpp" on disk, the user can run it by typing the following command from the ROOT window:

.x C:/Documents and Settings/temp/ReadSampleData.cpp

This command will execute the program, which will result in the input file being read, and saved as a ROOT tree on disk. The program is reproduced below.

```
// Reads a simple data file
#include <TFile.h>
#include <TTree.h>
#include <string>
#include <iostream>
using namespace std;
void ReadSampleData() {
  // Location of the directory for input and output files
  string dirName = "C:/Documents and Settings/temp/" ;
  // Name of the input and output files
  string inFileName = dirName + "SampleData.txt" ;
  string outFileName = dirName + "SampleData.root" ;
  // Create a tree to store the data
  TTree *tree = new TTree("dataTree","Sample Data");
  // Open the output file to write to
  TFile *fout = new TFile(outFileName.c_str(),"RECREATE");
  fout->cd();
  // Read in the data from  the text file
  int nentries = tree->ReadFile(inFileName.c_str(),
"PolNum/I:EffYear/I:EffMonth/I:State/C:TotalPremium/D");
```

```
    cout << "Read " << nentries << " entries from the input file " << inFileName << endl;

    // Write the ROOT tree to output file

    tree->Write();

    fout->Close();

    // Cleanup

    delete tree;

    delete fout;

  }
```

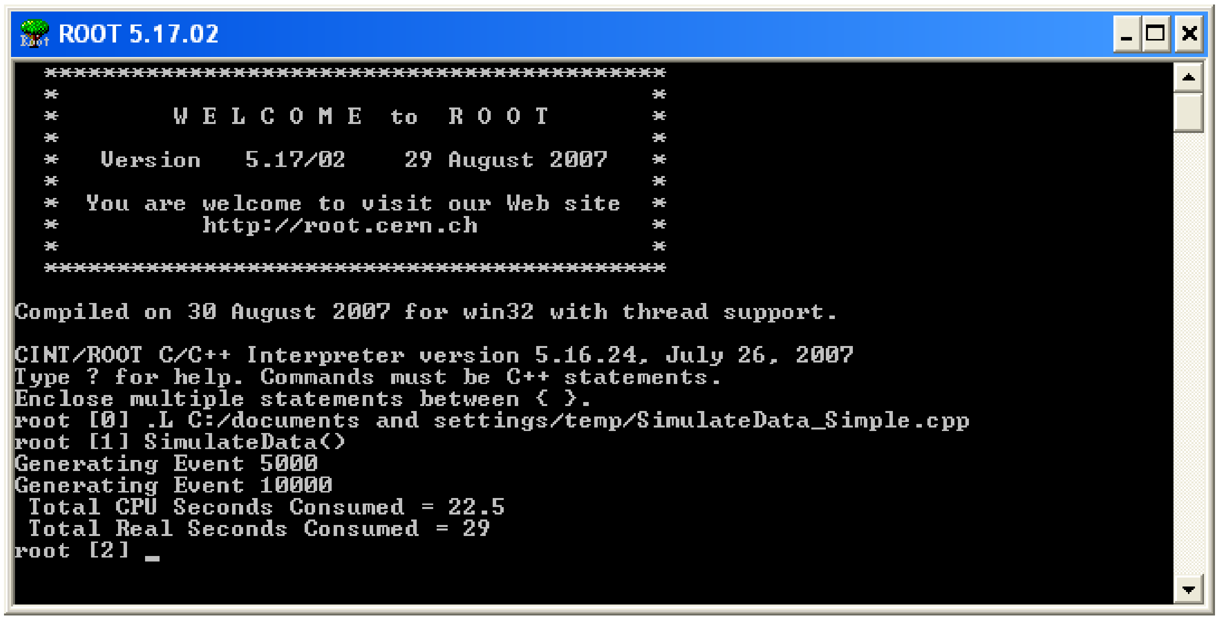
Appendix B

In this appendix, we reproduce the ROOT macro we used to generate the simulated data used in this paper. TNtuple class (http://root.cern.ch/root/html516/TNtuple.html) is used to store the simulated data. TNtuple class inherits from TTree class, and is useful when one is only storing numeric information. In order to run this macro, follow these steps:

1. Copy the following program (see below), and save it on your computer.
2. Change the value of variable Nloop (highlighted in bold face) to the number of observations you want to simulate.
3. Change the value of the variable DirName (highlighted in bold face) to the name of folder where you saved this program.
4. From ROOT console, type:
   .L Progam-File-Name (where Program-File-Name is the name of the file, including complete folder path, in which you saved this program).
5. Next, still in the ROOT console, type:
   SimulateData()

This will generate the chosen number of simulated observations. The resulting ROOT file (SimulatedData.root) will be written in the folder specified by DirName.

Figure B1 below shows the screen shot of the ROOT console after executing the above steps. In this case, the name of the file containing the simulation code (reproduced below) was called SimulateData_Simple.cpp; and it was located in the folder C:/Documents and Settings/temp.



```
  *******************************************
  *                                         *
  *        W E L C O M E  to  R O O T       *
  *                                         *
  *   Version   5.17/02    29 August 2007   *
  *                                         *
  *  You are welcome to visit our Web site  *
  *          http://root.cern.ch            *
  *                                         *
  *******************************************

Compiled on 30 August 2007 for win32 with thread support.

CINT/ROOT C/C++ Interpreter version 5.16.24, July 26, 2007
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
root [0] .L C:/documents and settings/temp/SimulateData_Simple.cpp
root [1] SimulateData()
Generating Event 5000
Generating Event 10000
 Total CPU Seconds Consumed = 22.5
 Total Real Seconds Consumed = 29
root [2] _
```

Figure B1: Screen shot of the ROOT console after executing the macro to generate 10,000 simulated observations.

////////////////////// **Beginning of the Data Simulation Code**////////////////////////////////////

```
// This is a ROOT macro to generate simulated data for a simple model.
// This model relates the loss ratio to five independent variables.

#include "TRandom2.h"
#include "TStopwatch.h"
#include "TFile.h"
#include "TNtuple.h"

#include <string>

using namespace std;

// How many events do we want to generate
Int_t Nloop = 10 000  ;

// Directory where we will be writing to or reading from
string DirName = "C:/Documents and Settings/temp/";


// Generate a random number from a Landau distribution
Double_t GetRandomFromLandau(Double_t mean, Double_t sigma, Double_t
xmin, Double_t xmax) {
  Double_t x = -999.0;
  while (( x < xmin) || (x > xmax)) {
      x = gRandom->Landau(mean, sigma);
  }
  return x;
}
```

```
// Generate a random number from a Uniform distribution
Double_t GetRandomFromUniform(Double_t xmin, Double_t xmax) {
  Double_t x = -999.0;
  while (( x < xmin) || (x > xmax)) {
      x = xmin + (gRandom->Rndm() * (xmax - xmin));
  }
  return x;
}


// Generate a random number based on any formula
Double_t GetRandomFromFormula(const char* formula, Double_t xmin,
Double_t xmax) {
  TF1 *f1 = new TF1("f1",formula, xmin, xmax);
 Double_t xx = -999.0;
  while ((xx < xmin) || (xx > xmax)) {
    xx = f1->GetRandom();
  }

  delete f1;
  return xx;
}



void SimulateData() {

  // Instantiate and start a stop watch

  TStopwatch StopWatch;

  StopWatch.Reset();
  StopWatch.Start();

  // First, open a file to output the data set
  string OutFileName = DirName + "SimulatedData.root";
  TFile *fout = new TFile(OutFileName.c_str(),"RECREATE");

  if (!fout) {
    cout << " Error opening input file" << endl;
      return;
  }

  // Next, book an ntuple
  TNtuple *nt = new TNtuple("nt","
","AgeOfBusiness:BuildingCount:CreditScore:PolicyAge:TotalBuildingInsur
ance:PolicyNumber:LossRatio:Premium:Loss:EventNum:ran1:ran2:ran3");

  // Create an array to store the simulated numbers
  Float_t VarList[13];

  // Now start generating the variables. Loop desired number of times.
  int counter = 0;
  for (Int_t i = 0; i < Nloop; i++) {
    counter++;
    if (counter%5000 == 0) cout << "Generating Event " << counter <<
endl;

    VarList[0] = (Float_t) GetRandomFromLandau(10.0, 5.0, 0, 100) ;
//AgeOfBusiness is based on Landau distribution
    VarList[1] = (Float_t) GetRandomFromLandau(0.4, 0.1, 0, 100) ;
//BuildingCount is based on Landau distribution
    VarList[2] = (Float_t) GetRandomFromUniform(0, 100);
//CreditScore is based on Uniform distribution
    VarList[3] = (Float_t) GetRandomFromFormula("85.57-7.7948*x", 0, 10
); //PolicyAge is based on a linear distribution

    VarList[4] = (Float_t) GetRandomFromLandau(-3.3736e4, 5.4577e2 , 0,
50000000); //TotalBuildingInsurance is based on Landau distribution
    VarList[5] = counter;  //PolicyNumber
    VarList[6] = (Float_t) 0.5 + VarList[0]*(-0.0053)*0.1 +
VarList[1]*0.025*0.1 + VarList[2]*(-0.0057)*0.1
                             + VarList[3]*(-0.0227) +
VarList[4]*(0.0437)*(1.0e-6) +  gRandom->Gaus(0.0, 0.04);  //LossRatio
is a linear combination of other variables plus an error term
    VarList[7] = (Float_t) gRandom->Landau(5000, 500); //Premium
    VarList[8] = (Float_t) VarList[7] * VarList[6]; //Loss
    VarList[9] = (Float_t) counter; //EventNum

    VarList[10] = (Float_t) 10.0*(gRandom->Rndm());  //ran1 is just a
random number
    VarList[11] = (Float_t) 10.0*(gRandom->Rndm());  //ran2 is just
another random number
    VarList[12] = (Float_t) 10.0*(gRandom->Rndm());  //ran3 is just
another random number

    nt->Fill(VarList);
  } // End of loop over Nloop

  // Write the ntuple to output file
  fout->cd();
  nt->Write();
  fout->Close();

  // Stop the stop watch, and report the time taken to run this macro

  StopWatch.Stop();

  cout << " Total CPU Seconds Consumed = " << StopWatch.CpuTime() <<
endl;
  cout << " Total Real Seconds Consumed = " << StopWatch.RealTime() <<
endl;

  // Clean up
  delete fout;
}
```

////////////////////////// **END of the Data Simulation Code** ///////////////////////////////////////