Outlier Identification and Imputation using Robust Regression Trees

Zhao Xinqiang and Ray Chambers
Department of Social Statistics
University of Southampton

28 March 2002

EUREDIT DELIVERABLES

D4.2.1: See Sections 1and 2
D4.2.2: See Appendices A, B

1. Overview of the Approach

1.1 The WAID Toolkit

WAID is software for building regression and classification trees. The original version of this software was intended for automatic imputation of missing data in censuses and surveys, and was developed as a C++ Windows application under the AUTIMP project (R.L. Chambers, J. Hoogland, S. Laaksonen, D.M. Mesa, J. Pannekoek, P. Piela, P. Tsai and T. De Waal, 2001, The AUTIMP-Project: Evaluation of Imputation Software. Report, Statistics Netherlands, Voorburg). Under the EUREDIT project a "toolkit" of Splus programs has been created that emulates and extends the capabilities of WAID. These programs work both under Splus and under R, a public domain statistical software package that is compatible with Splus. Availability of this toolkit means that it is now easy to create statistical applications that "build" on the WAID algorithm. Similar software products are CART from Salford Systems, the Splus tree() function and the CHAID module in SPSS. The code for the programs in the WAID toolkit is set out in Appendix 1.

The basic idea behind WAID (and other tree modelling software packages) is to sequentially divide the original data set into subgroups (or nodes) that are increasingly more homogeneous with respect to the values of a response variable. The splits themselves are defined in terms of the values of a set of categorical covariates. The WAID splitting algorithm is described in the next section. By definition, WAID is a nonparametric statistical procedure. It also has the capacity to implement outlier robust splitting based on M-estimation methodology. In this case outliers are downweighted when calculating the measure of within node heterogeneity (weighted residual sum of squares) used to decide whether a node should be split or not. The weights used for this purpose are themselves based on outlier robust influence functions.

1.2 Outlier Identification using WAID

The basic idea behind the WAID outlier identification strategy is straightforward. Each time WAID splits the data set to create two new nodes it creates a new set of robust weights for the units making up those nodes. These weights are scaled so that they sum to the number of observations in the node, with outliers receiving weights close to zero and inliers receiving weights around one. These weights reflect distance from a robust estimate of location for the values in the node. Consequently a value that is not immediately identifiable as an outlier within "larger" nodes is more likely to become identified as such as it is classified into smaller and smaller nodes. In effect, the weights associated with such units tend to move towards zero. The WAID outlier identification algorithm defines an outlier as an observation with an average weight over all node splits that is less than a specified threshold. This threshold is defined as the value at which the most observations that are "real" errors are identified as outliers and where the least number of error-free observations are classified as outliers. See Section 2 for details. The process itself can be defined in terms of the following three steps.

- Step 1: Build a robust tree using WAID.
- Step 2: Define a weight "threshold".

- Step 3: Identify outliers as those with average weights less than this threshold.

In practice of course we do not know which data values are errors and which are not. Consequently it is difficult to identify the optimal threshold. Where historical data are available on errors, the threshold can be defined in terms of this historical information. This is the approach taken with the ABI application reported on in section 3. An alternative approach is to randomly peturb the current data and to then define the optimal threshold in terms of maximal correct identification of peturbed values and minimal incorrect identification of non-peturbed values. Such a perturbation scheme could be created based on the residuals generated by the robust WAID tree. This is an area that needs further investigation.

1.3 Outlier and Missing Data Imputation using WAID

Once a set of outliers have been identified, the robust tree structure generated by the WAID algorithm can be used to impute replacement values for these units. These imputed values can be derived in many different ways. However, the lessons learned in the AUTIMP project indicate that two imputation methods in particular are suited for use with WAID trees. These are mean imputation and random donor imputation within terminal nodes. In the context of outlier contaminated data, these imputation methods need to be modified as follows:

- Use a robust weighted mean within a terminal node as the "mean" imputation.
- Identify a donor within a terminal node by randomly selecting a value from those cases within the node whose average weights are greater than a specified threshold.

Note that imputation for missing (rather than outlier) values proceeds in exactly the same way. That is, the record with a missing value is "dropped" down the WAID tree until it reaches a terminal node. It is then imputed using either the weighted mean for the node or via a donor value obtained from a non-outlier in the node.

2. The WAID Outlier Identification Algorithm

2.1 Introduction

In this section we describe the three main components of WAID outlier identification methodology. The first is the algorithm used to build a robust regression tree in WAID. The second is the procedure for identifying potential outliers based on a robust WAID tree. The last component describes the algorithm used to find the weight threshold.

WAID assumes a rectangular data set containing n observations, values $\{y_i\}$ of a scalar response variable Y and values $\{x_{1i}, ..., x_{pi}\}$ of p covariates $X_1, ..., X_p$. The values of Y are assumed to contain outliers. In contrast, the covariates $X_1, ..., X_p$ are all assumed to be categorical. No missing X-values are allowed in the current R version of WAID. For a scalar response variable WAID builds a regression tree. If the response variable is categorical, WAID builds a classification tree. The only difference between these two types of trees is the heterogeneity measure used to determine tree splitting behaviour. Since our focus is outlier identification, we are concerned with scalar response variables only and so we restrict consideration to WAID's regression tree algorithm.

2.2 The WAID Regression Tree Algorithm

The basic idea used in WAID (as well as other tree-based methods) is to split the original data set into smaller subsets or nodes in which the Y-values are more homogeneous. In WAID this is accomplished by sequential binary splitting. At each step in the splitting process, all nodes created up to that point are examined in order to identify the one with minimal homogeneity. An optimal binary split of this "parent" node is then carried out. This is based on identifying a set of values of one of the covariates X_1, ..., X_p such that a split of the parent node into one child node containing only cases possessing these values and another child node containing the remaining cases maximes (minimises) the homogeneity (heterogeneity) of these child nodes. The measure of heterogeneity used is the weighted sum of squares of residuals (WSSR) defined with respect to a robust measure of the location of the Y-values in the parent node. The splitting process continues until a suitable stopping criterion is met. At present this is when either (i) all candidate parent nodes are effectively homogeneous; (b) all candidate parent nodes are too small to split further; or (c) the maximum number of nodes is reached. There is no attempt to find an "optimal" tree. The set of nodes defining the final tree are typically referred to as the terminal nodes of the tree.

The two key algorithms used to determine a node split are the algorithm for calculating the robust weight (used in calculation of WSSR) and the algorithm for finding the optimal split. We describe them below.

2.2.1 Calculation of WSSR and Robust Weights

We denote the values of the response variable Y in node k by $y_1, \cdots, y_{n_k}$, where n_k is the number of observations in node k. The weighted sum of squared residuals within this node ($WSSR_k$) can be written as follows

$$WSSR_k = \sum_{i=1}^{n_k} w_i (y_i - \bar{y}_{wk})^2$$

where w_i is the weight attached to i^{th} case in node k and $\bar{y}_{wk}$ is the weighted mean of Y in node k,

$$\bar{y}_{wk} = \sum_{i=1}^{n_k} w_i y_i \Big/ \sum_{i=1}^{n_k} w_i \, .$$

The weight w_i is calculated as the ratio

$$w_i = \frac{\psi(y_i - \bar{y}_{wk})}{y_i - \bar{y}_{wk}}.$$

where ψ(x) denotes an appropriately chosen influence function. The Splus/R toolkit version of WAID computes these weights by first calling the robust regression function rlm() in the MASS robust statistics library (available for both R and Splus). This function returns weight values which are then rescaled within WAID to sum to the number n_k of cases within node k.

A standard (nonrobust) regression tree in WAID uses the Ordinary Least Squares (OLS) influence function, ψ(x) = x, in which case $w_i = 1$. Besides OLS, WAID currently has three robust weighting schemes, corresponding to the weighting options available for rlm(). These correspond to use of Huber's Min/Max, Tukey's Biweight and Hampel's Redescending influence functions . These are defined as follows:

Huber's Min/Max: $\psi(x) = \min(|x|, c)$

where c is a tuningconstant. With c = 1.345 this influence function achieves 95% efficiency when estimating the mean of a normal distribution.

Tukey's Biweight: $\psi(t) = t\left[1 - \frac{t^2}{R^2}\right]^2$

where R is a tuning constant. The value R = 4.685 gives 95% efficiency at the normal (Ripley, 1994).

Hampel's Redescending: $\psi(t) = \begin{cases} t & |t| \le a \\ a\,\mathrm{sgn}(t) & a < |t| \le b \\ a(c\,\mathrm{sgn}(t) - t)/(c - b) & b < |t| \le c \\ 0 & |t| > c \end{cases}$

where a b and c are tuning constants (Hampel, 1986). In passing we observe that it is a straightforward matter to add extra weight functions for use by rlm() and hence by WAID.

2.2.2 Optimal Splitting

Without loss of generality, we consider optimal splitting of the k^{th} parent node. Suppose there are n_k observations in this node and p covariates $X_1, ..., X_p$. The best split for each covariate X_j, j = 1, ..., p is defined by the subset of values of X_j that generates child nodes with a WSSR value smaller than that generated by any other split of the parent node based on Xj. The optimal split overall is then defined as the best (in terms of minimising the WSSR value generated by the resulting child nodes) among these covariate-specific optimal splits.

For convenience, we let $\mathbf{V}_j$ denote the set containing the m_j unique values of X_j for the cases in the parent node. The search procedure for the best split based on X_j is then as follows.

Step 1: Sort the m_j values in $\mathbf{V}_j$. WAID allows a covariate to be declared as monotone or non-monotone. If X_j is monotone then the values in $\mathbf{V}_j$ are sorted in ascending order. If X_j is non-monotone, then the values in $\mathbf{V}_j$ are sorted as $\{v_1, v_2, \cdots, v_{m_j}\}$ where $\bar{y}_{wk}(v_1) < \bar{y}_{wk}(v_2) < \cdots < \bar{y}_{wk}(v_{m_j})$ and

$$\bar{y}_{wk}(v_q) = \sum_{\substack{i \quad \text{node } k \\ x_i = v_q}} w_i y_i \Big/ \sum_{\substack{i \quad \text{node } k \\ x_i = v_q}} w_i \,.$$

Let L_{qj} denote the values in $\mathbf{V}_j$ that are to the "left" of v_q with R_{qj} denoting the corresponding set of values in $\mathbf{V}_j$ that are to the "right" of v_q. We can then split the parent node into two child nodes at this value v_q. This split corresponds to a left child node (Node_{Lqkj}) and right child node (Node_{Rqkj}) defined by

$$\text{Node}_{Lqkj} = \{i \quad \text{node } k, x_{ij} = v_s, 1 \le s \le q\}$$
$$\text{Node}_{Rqkj} = \{i \quad \text{node } k, x_{ij} = v_s, q < s \le m_j\}.$$

Step II: Calculate a WSSR value for each of the m_j - 1 possible splits of $\mathbf{V}_j$. For each $v_q \quad \mathbf{V}_j$ the WSSR values for the child nodes corresponding to a split at this value are

$$\text{WSSR}_{Lqkj} = \sum_{i \quad \text{Node}_{Lqkj}} w_i (y_i - \bar{y}_{Lwkj})^2$$
$$\text{WSSR}_{Rqkj} = \sum_{i \quad \text{Node}_{Rqkj}} w_i (y_i - \bar{y}_{Rwkj})^2$$

where $\bar{y}_{Lwkj} = \sum_{i \quad \text{Node}_{Lqkj}} w_i y_i \Big/ \sum_{i \quad \text{Node}_{Lqkj}} w_i$ and $\bar{y}_{Rwkj}$ is defined similarly. The within-group WSSR associated with a split at $v_q \quad \mathbf{V}_j$ is then

$$\text{WSSR}_{qkj} = \text{WSSR}_{Lqkj} + \text{WSSR}_{Rqkj}.$$

Step III: Find the best splitting value $v_q \quad \mathbf{V}_j$. This is the value v_q that generates the minimum value of WSSR_{qkj} among all values v_q in $\mathbf{V}_j$. The overall best split among the candidate best splits defined by each covariate is then the split that generates the minimum value of WSSR_{qkj} over both q and j.

2.3 Outlier identification using WAID

We can calculate an "average" weight for every case in the original data set by averaging its weight over all splits defining the final tree. If a case is not involved in a particular split (it is not in the node that is selected for splitting) its weight remains unchanged from the last split in which it was involved. Outlier identification using a robust WAID tree is then based on the fact that outliers are likely to have small weights in most of the nodes in which they appear in such a tree. Consequently they will also have small average weights across all node splits, and we therefore can identify them as outliers on the basis that their average weight lies below a defined threshold. Note that we do not distinguish between outliers that appear early on in the construction of the tree and then gradually become "less" outlier-like because of progressive refinement by the tree-building process and outliers that are "hidden" early on in the tree-building process and then become more and more outlying relative to their within-node comparators in later stages of the tree.

The task is therefore one of finding a threshold value such that "important" outliers are identified as having average weights less than this threshold. The set of identified outliers for a given threshold w can be denoted

$$\text{out}(w) = \{i,\ \bar{w}_i < w,\ i=1, ..., n\}$$

where $\bar{w}_i = \frac{1}{m}\sum_{k=1}^{m} w_i^{(k)}$ is the average weight of the i^{th} case, with $w_i^{(k)}$ denoting the weight of the i^{th} unit at the k^{th} split, and m is the total number of splits defining the tree.

The main problem with this approach therefore reduces to finding a threshold w that successfully identifies outliers without also mis-identifying non-outliers. The optimal threshold value w is therefore the one that maximises the proportion of "true" outliers identified and minimises the proportion of "false" outliers identified. This requires information about true outliers to be available. In practice this information will not be available. However, information is often available about known errors in the data, most of which are associated with outlying values. Consequently, we can choose the optimal value of w to optimise identification of these errors.

Put N_{errors} equal to the total number of true errors, and, for a given threshold w, put $N_{outliers}(w)$ equal to the total number of outliers identified by WAID on the basis of the specified threshold w, $n_{errors}(w)$ equal to the corresponding number of errors identified as outliers, and $n_{non\text{-}errors}(w)$ equal to the total number of non-errors identified as outliers. The proportion of error-generated outliers identified by WAID is then

$$R_1(w) = \frac{n_{errors}(w)}{N_{errors}}$$

while the proportion of non-errors identified as outliers using the threshold w is

$$R_2(w) = \frac{n_{non-errors}(w)}{N_{outliers}(w)} = 1 - \frac{n_{errors}(w)}{N_{outliers}(w)}.$$

We denote the optimal threshold value as w_{opt}, where

$$w_{opt} = \arg\max_{w} [R_1(w)(1 - R_2(w))].$$

This definition is a simple implementation of the idea that at the optimal threshold value WAID identifies most of the error-generated outliers as well as minimises the number of non-errors identified as outliers. In the next section we illustrate this approach using the 1997 ABI data.

3. An Evaluation Using ABI Data

3.1 The ABI data

There are five ABI datasets available for evaluation in EUREDIT. See abimeta.xls for a general description of these datasets. Three of them are derived from 1997 ABI data. The remaining two are derived from 1998 ABI data. The first of these 1997 datasets (sec197(true)) is contains true values. The second (sec197(y2)) is derived from sec197(true) and only contains missing values. The third dataset (sec197(y3)) contains both missing values and adding errors. Similarly, one of the two evaluation datasets for the 1998 survey is a dataset containing only missing values, while the other is a dataset with missing value and errors.

In this section we focus on the 1997 ABI dataset that contains both missing values and errors (sec197(y3)). We present results from an empirical study based on this dataset that evaluates the error and outlier identification and imputation performance of a WAID-based approach. The dataset has a total of 6099 records, with 30 variables. The descriptions of these variables (from abimeta.xls) are given in Table 1.

Our interest is in the variables corresponding to totals in this dataset. There are seven such variables in the ABI dataset, TURNOVER, EMPTOTC, PURTOT, TAXTOT, ASSACQ, ASSDISP and EMPLOY. Initially, however we focus on total business turnover (TURNOVER), which we select as our response variable. Similarly, we consider two covariates corresponding to the register size information available on the ABI dataset. These are TURNREG (a positive-valued continuous variable corresponding to the register value of total turnover for a unit) and EMPREG (a ordinal variable corresponding to the employee size band of a unit). Both these covariates, being register variables, have no missing values and no errors and therefore could be considered as "natural" covariates for explaining variation in ABI variables. There are 42 missing values and 241 "error" values (i.e. values different from the corresponding "true" values in sec197(true)) for TURNOVER in sec197(true). In what follows the 42 missing values are excluded, to be imputed after the tree is formed. As an aside we note that the version of WAID implemented in the toolkit cannot handle missing values in the covariates used to define a tree. Consequently records containing one or more missing values in the covariates need to be excluded from the dataset used to build a WAID tree.

Table 1. Variable descriptions

Variable name	Variable label
REF	Case reference number
CLASS	Anonymised industrial classification. The classification is hierarchical with the first digit indicating the higher level of classification. Numerically adjacent digits do not indicate similarity between classes/subclasses e.g. 3.1 is similar to 3.2 but not necessarily closer to 3.2 than 3.3.
WEIGHT	Design weight (N/n) for category (1st digit of CLASS) and employment size band (EMPREG)
TURNOVER	Total turnover
EMPWAG	Wages and salaries paid
EMPNI	Employers NI contributions
EMPNIOTH	Employers NI contributions and other employment costs
EMPENS	Contributions to pension funds
EMPRED	Redundancy and severance payments to employees
EMPTOTC	Total employment costs
PUREN	Purchases of energy, water and materials
PURENOTH	Purchases of energy and other goods for own consumption
PURCOTH	Purchases of other goods and materials for own consumption
PURESALE	Purchases of goods bought for resale
PURHIRE	Payments of hiring, leasing or renting
PURINS	Commercial insurance premiums paid
PURTRANS	Purchases of road transport services
PURTELE	Purchases of telecommunication services
PURCOMP	Purchases of computer and related services
PURADV	Purchases of advertising and marketing
PUROTHSE	Other services purchased
PUROTHAL	All other purchases of goods and services
PURTOT	Total purchases of goods and services
TAXRATES	Amounts paid for national non-domestic rates
TAXDUTY	Amounts paid for export duty
TAXOTHE	Other amounts paid for taxes and levies
TAXOTHD	Other amounts for taxes and levies excluding duty
TAXTOT	Total taxes paid
STOCKBEG	Value of stocks held at beginning of year
STOCKEND	Value of stocks held at end of year
ASSACQ	Total cost of all capital assets acquired
ASSDISP	Total proceeds from capital asset disposal
CAPWORK	Value of work of a capital nature
EMPLOY	Total number of employees
TURNREG	Register turnover
EMPREG	Employment size group from register: 1 = 0 to 9 employees, 2 = 10 to 19 employees, 3 = 20 to 49 employees, 4 = 50 to 99 employees, 5 = 100 to 249 employees, 6 = 250 or more employees
FORMTYPE	1 = long form, 2 = short form

3.2 Numerical Results

3.2.1 Tree Fitting

WAID assumes all covariates are categorical. Consequently, before building a WAID tree using TURNREG as a covariate, this variable was categorized into its 100 percentile classes. Furthermore, since the distribution of TURNOVER is highly skewed (see Figure 1) the tree was built using log(TURNOVER + 1) as the response variable. The addition of one to the observed TURNOVER value was to ensure that all non-missing cases, including the three businesses with zero values for TURNOVER, were used in building the tree.

Figure 2 shows the "nodemean tree" defined by WAID for non-missing values of log(TURNOVER + 1) from the error and missing contaminated dataset sec197(y3). This plot shows the robust means of the log-transformed response variable for the nodes defined at each split of the tree-fitting process. The tree was fitted using a maximum of 50 terminal nodes, with robust weights defined using Tukey's Biweight weighting scheme. No nodes containing less than 5 cases were allowed to be created in the fitting process. A maximum of 100 iterations were used in the iterative procedure rlm() that calculates the robust means shown in this plot. Also, both covariates TURNREG and EMPREG were declared as "monotone" in the tree-fitting process. In practice, this means that any splits defined using the categories of these variables must be in terms of contiguous categories, i.e. into subgroups defined by $X \leq x$ and $X > x$. Also, virtually all the splits in the tree were determined by TURNREG, with only the 17th split determined by EMPREG. This shows clearly on the nodemean plot.

For comparison, Figure 3 shows the corresponding WAID tree defined by the "true" values of TURNOVER in the file sec197(true). It is clear that the outliers and errors in the dataset sec197(y3) have little or no effect on the robust tree structure, as one would expect.

Figure 2: Robust nodemean tree for y3 values of log(TURNOVER+1)

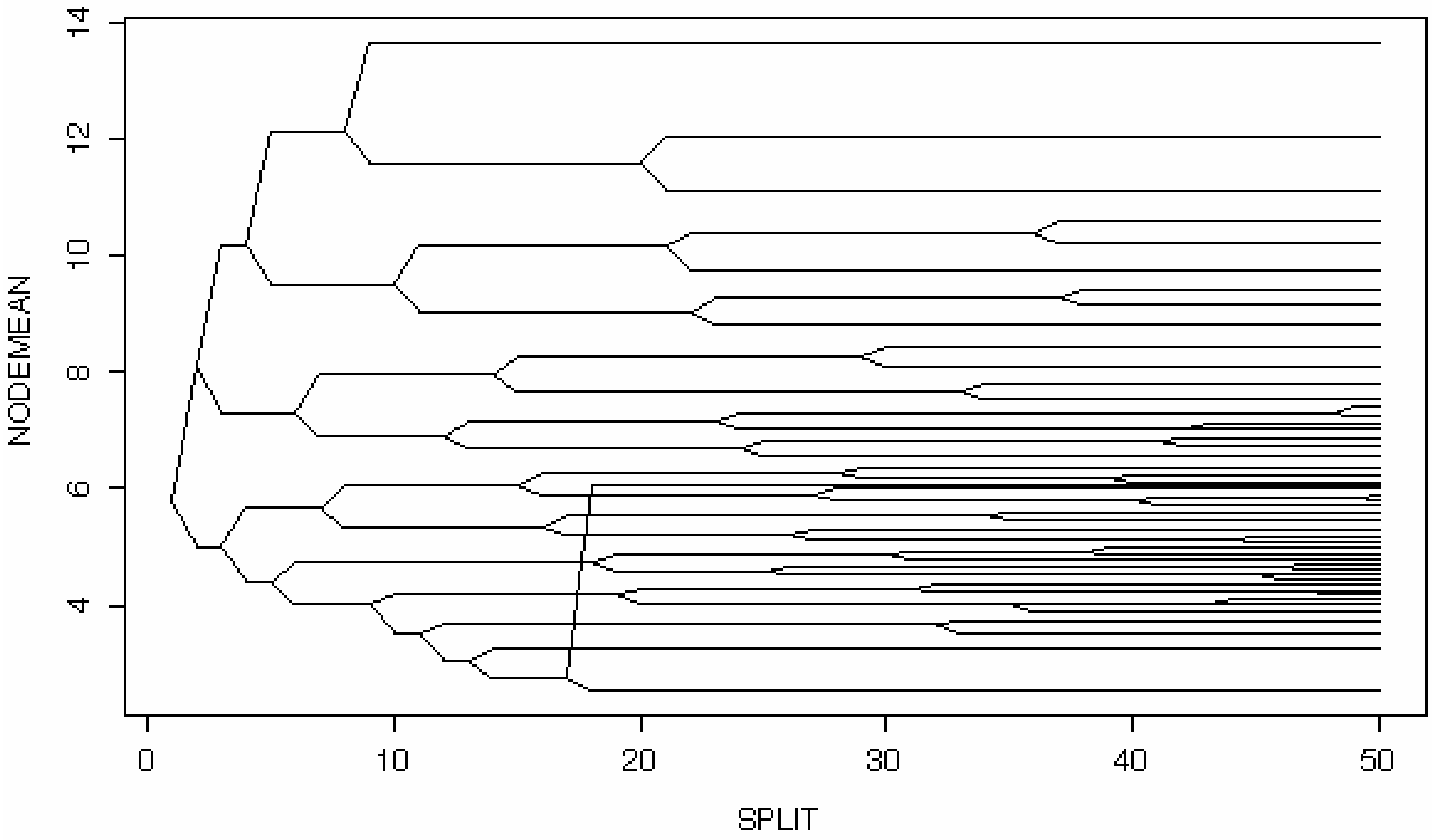
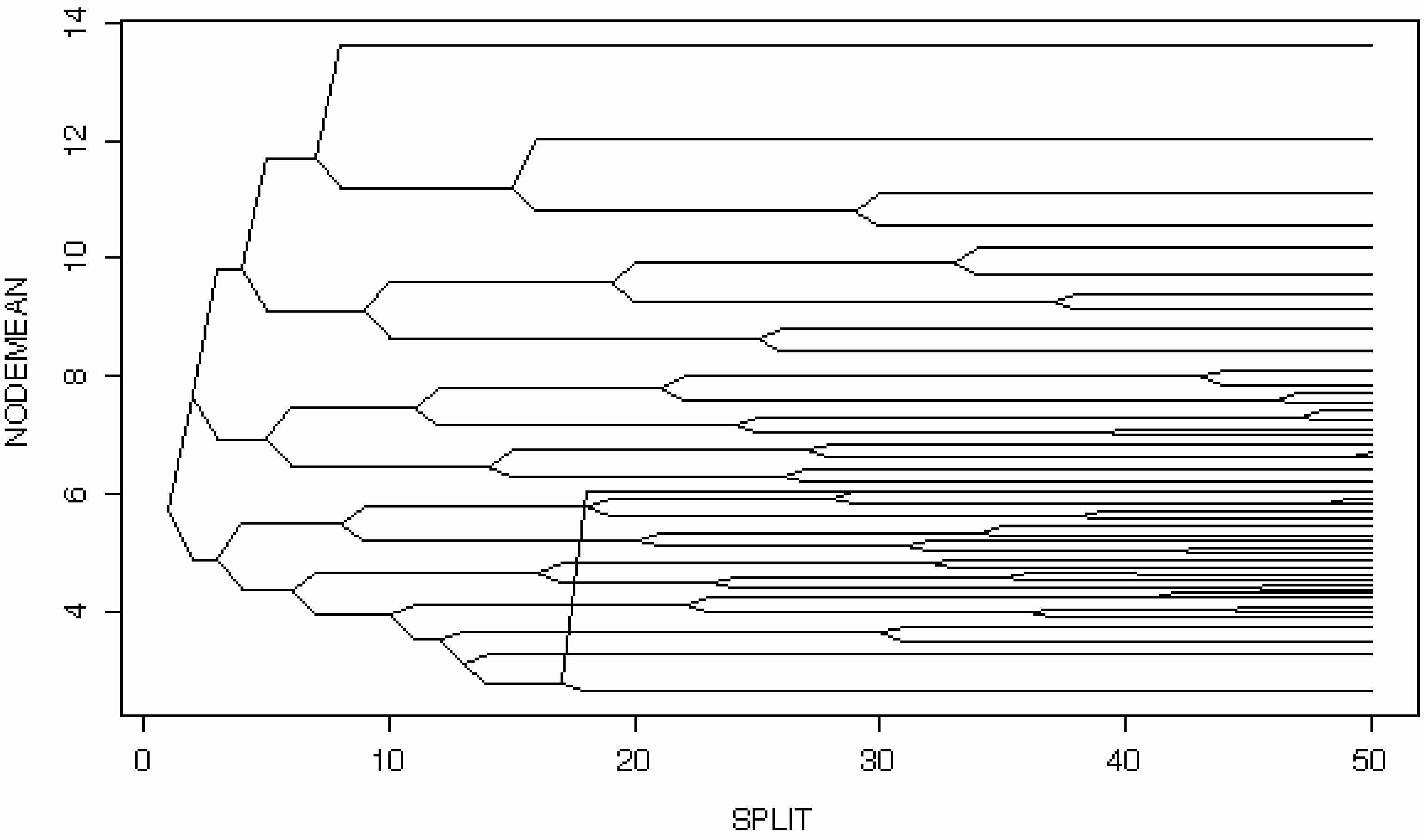


Figure 3: Robust nodemean tree for true values of log(TURNOVER+1)



3.2.2 Finding an Optimal Weight Threshold for TURNOVER

In Table 2 we illustrate the error detection performance of the robust regression tree displayed in Figure 2. For a range of values of a threshold w this table shows:

- The number N_{out} of outliers identified using that threshold
- The number N_{error} of errors identified using that threshold. There are a total of 241 errors in this dataset.
- The number N_{sig} of "significant" errors contained in these identified errors, where a "significant" error is one whose relative difference from its corresponding true value is greater than one. There are a total of 208 such significant errors in this dataset.
- The proportion R_1 of errors contained in the identified outliers
- The proportion R_{sig} of significant errors contained in the identified outliers.
- The proportion R_2 of "non-errors" identified as outliers at this value of w
- The value $R_1(1-R_2)$

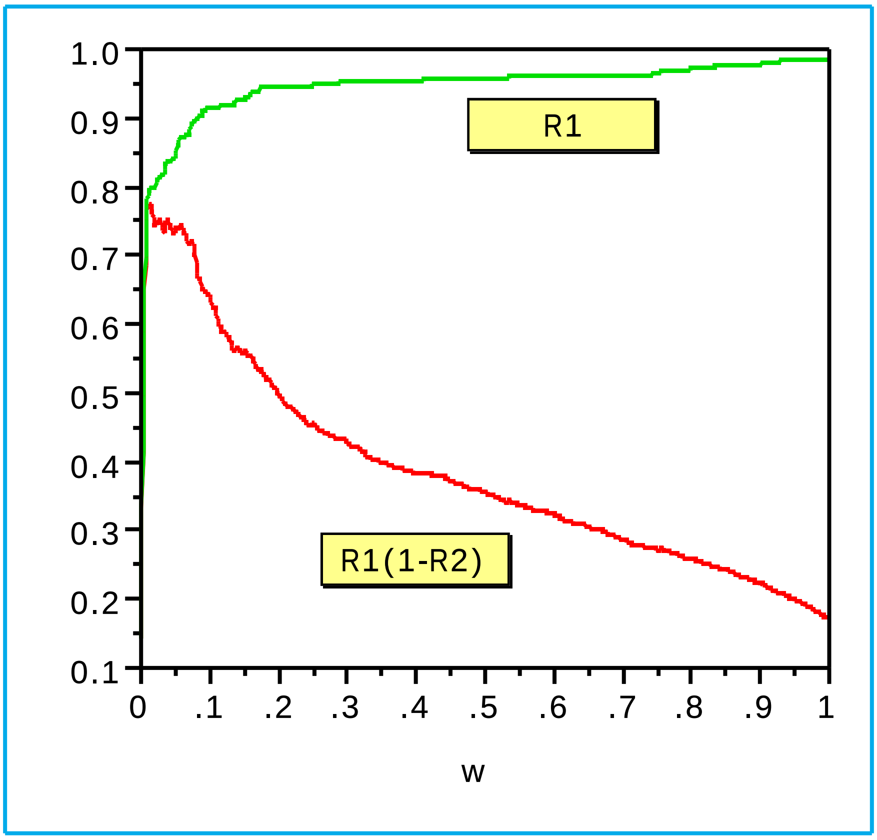
For TURNOVER we see that the optimal threshold (in terms of minimising $R_1(1-R_2)$ then lies between 0.015 and 0.016. These values are highlighted in Table 2. Observe that at this optimum over 80% (90%) of errors (significant errors) are detected. The 3% of identified outliers that are not errors are essentially "true outliers", which, in normal survey processing, would definitely need to be queried.

Table 2 Outlier and error detection performance for different threshold values

w	**N_{out}**	**N_{error}**	**N_{sig}**	**R_1**	**R_{sig}**	**R_2**	**$R_1(1-R_2)$**
0.001	35	35	35	0.145	0.168	0.000	0.145228
0.002	65	65	65	0.270	0.312	0.000	0.269709
0.003	83	82	82	0.340	0.394	0.012	0.336150
0.004	101	100	100	0.415	0.481	0.010	0.410829
0.005	121	120	120	0.498	0.577	0.008	0.493810
0.006	145	142	142	0.589	0.683	0.021	0.577021
0.007	163	160	159	0.664	0.764	0.018	0.651681
0.008	172	169	168	0.701	0.808	0.017	0.689014
0.009	186	182	180	0.755	0.865	0.022	0.738946
0.010	189	185	183	0.768	0.880	0.021	0.751389
0.011	193	189	187	0.784	0.899	0.021	0.767979
0.012	194	190	188	0.788	0.904	0.021	0.772126
0.013	197	192	190	0.797	0.913	0.025	0.776460
0.014	198	192	190	0.797	0.913	0.030	0.772539
0.015	**199**	**193**	**191**	**0.801**	**0.918**	**0.030**	**0.776684**
0.016	**199**	**193**	**191**	**0.801**	**0.918**	**0.030**	**0.776684**
0.017	202	193	191	0.801	0.918	0.045	0.765149
0.018	202	193	191	0.801	0.918	0.045	0.765149
0.019	203	193	191	0.801	0.918	0.049	0.761380
0.020	204	193	191	0.801	0.918	0.054	0.757648
0.021	207	193	191	0.801	0.918	0.068	0.746667
0.022	207	193	191	0.801	0.918	0.068	0.746667
0.023	207	193	191	0.801	0.918	0.068	0.746667
0.024	211	195	192	0.809	0.923	0.076	0.747773
0.025	212	196	193	0.813	0.928	0.075	0.751899
0.026	212	196	193	0.813	0.928	0.075	0.751899
0.027	212	196	193	0.813	0.928	0.075	0.751899
0.028	214	197	194	0.817	0.933	0.079	0.752492
0.029	214	197	194	0.817	0.933	0.079	0.752492
0.030	215	197	194	0.817	0.933	0.084	0.748992
0.035	221	198	194	0.822	0.933	0.104	0.736073
0.040	225	202	194	0.838	0.933	0.102	0.752494
0.045	228	202	194	0.838	0.933	0.114	0.742593
0.050	233	203	195	0.842	0.938	0.129	0.733870
0.055	242	208	199	0.863	0.957	0.140	0.741813
0.060	248	211	201	0.876	0.966	0.149	0.744897
0.065	252	211	201	0.876	0.966	0.163	0.733073
0.070	259	212	201	0.880	0.966	0.181	0.720037
0.075	265	215	204	0.892	0.981	0.189	0.723792
0.080	275	216	204	0.896	0.981	0.215	0.703976
0.085	292	217	205	0.900	0.986	0.257	0.669144
0.090	300	218	206	0.905	0.990	0.273	0.657317
0.095	309	220	206	0.913	0.990	0.288	0.649935
0.100	315	221	207	0.917	0.995	0.298	0.643364

In Figure 4 we show the behaviour of both R_1 and $R_1(1-R_2)$ as the threshold value w changes. Here we see that if a more conservative approach to error detection is required, then the optimal threshold would be set higher in order to detect more errors, even if this leads to more "correct" values being identified as outliers. Based on an inspection of Figure 4 a reasonable conservative optimal threshold for TURNOVER is then 0.08, at which (see Table 2), nearly 90% (98%) of all errors (significant errors) are detected. The cost, of course, is that this leads to the identification of a much larger number of correct TURNOVER values as outliers (59 compared with 6). However, it is clear from inspection of the TURNOVER data that these values are ones that we would want to query anyway.

Figure 4 Plot of $R_1(w)$ and $R_1(w)(1-R_2(w))$ for TURNOVER



3.2.3 Imputing for Missing, Error and Outlier values in TURNOVER

There are 42 records with missing TURNOVER in the dataset sec197(y3). Furthermore, if we set the outlier detection threshold at w = 0.02 in the robust WAID tree constructed for the non-missing values in this dataset then there are 204 identified outliers, out of which 193 are errors. All these values can be imputed by identifying their terminal nodes and then replacing them by TURNOVER values donated by randomly selected "inliers" in the node, or by the robust estimates of the corresponding node means for TURNOVER. These two options are denoted "Random" imputation and "Mean" imputation in what follows. Note that the Random imputation option requires specification of what constitutes an inlier in a terminal node. The default option for this is observations with average tree weight of one or greater, and this is the approach that was taken to obtain the results set out below.

Missing Data Imputation

Table 3 shows the Reference number , TURNREG class, EMREG class, true value of log(TURNOVER+1) and corresponding imputed value under both Random and Mean imputation for the 42 records in sec197(y3) with missing values for TURNOVER. In the original scale, the unweighted mean error of imputation for these cases is 114 for Random imputation and -16 for Mean imputation. The corresponding unweighted root mean square error values are 1409 for Random imputation and 1189 for Mean imputation.

Error Values Imputation

Table 4 shows Reference number , TURNREG class, EMREG class, true value of log(TURNOVER+1) and corresponding imputed value under both Random and Mean imputation for the 193 records with errors in TURNOVER in sec197(y3) that were identified as outliers using a threshold of w = 0.02. In the original scale, the unweighted mean error of imputation for these cases is 60670 for Random imputation and 54506 for Mean imputation. The corresponding unweighted root mean square error values are 881109 for Random imputation and 858946 for Mean imputation. Inspection of these records showed that one of them (Reference number 17816) is clearly an outlier as well as an error. When this record is excluded, the unweighted mean error of imputation drops sharply to -2703 for Random imputation and to -7209 for Mean imputation. The corresponding unweighted root mean square error values are then 39787 for Random imputation and 60006 for Mean imputation

"Correct" Outlier Imputation

Table 4 shows Reference number , TURNREG class, EMREG class, true value of log(TURNOVER+1) and corresponding imputed value under both Random and Mean imputation for the 11 records with correct values for TURNOVER in sec197(y3) that were identified as outliers using a threshold of w = 0.02. In the original scale, the unweighted mean error of imputation for these cases is 46518 for Random imputation and 46543 for Mean imputation. The corresponding unweighted root mean square error values are 89834 for Random imputation and 89882 for Mean imputation. Note that we expect to generate large mean errors and large root mean square errors for these imputations, since the comparisons in this case are with outlying true values of TURNOVER.

Table 3 Imputations for missing y3 values of log(TURNOVER+1)

REF	TURNREG CLASS	EMPREG CLASS	TRUE VALUE	RANDOM IMPUTE	MEAN IMPUTE
788	97	5	10.7011	10.6339	10.5759
4064	88	4	7.9983	8.1173	8.0892
5403	96	6	9.9889	10.0297	10.1933
8348	93	5	8.8343	9.3270	9.1466
9816	36	1	5.0814	5.0304	5.0955
9844	54	2	5.5175	5.8406	5.7337
9866	78	2	7.3232	6.9147	7.0157
10141	85	2	7.5735	7.8034	7.7851
10781	86	3	7.9424	7.9306	7.7851
11164	71	1	6.3886	6.4265	6.5741
11670	38	1	5.0239	5.0938	5.0955
12058	9	1	4.0254	3.7612	4.0141
12138	67	2	6.3154	6.2005	6.3690
12757	40	1	4.9972	5.2149	5.1875
12793	24	1	4.8040	4.5643	4.6375
12902	2	1	5.6630	2.5649	3.2615
12995	33	1	5.3845	5.0814	4.8907
14185	43	1	5.3613	5.2523	5.3080
14198	78	3	7.3957	7.1253	7.0157
14843	63	2	6.4184	6.0730	6.1132
14953	44	1	5.2311	5.4638	5.3080
15086	74	2	6.8855	6.7154	6.7356
15494	19	1	4.5747	4.4427	4.4676
15809	42	1	5.0876	5.3033	5.3080
15826	42	1	5.3327	5.3083	5.3080
16014	60	1	6.0753	6.1203	6.0088
16281	12	1	4.2627	4.1109	4.2034
16334	43	1	5.2311	5.4249	5.3080
16685	19	1	3.4965	4.3944	4.4676
16806	81	3	7.5569	7.1546	7.2567
16894	65	1	6.2025	6.2166	6.2344
17113	20	1	2.5649	4.3944	4.5513
17201	57	1	5.8021	5.8861	5.8261
17222	24	1	5.4931	4.7185	4.6375
17339	11	3	4.1744	4.2341	4.2034
17681	19	1	4.5109	4.2627	4.4676
17794	69	2	6.7405	6.3835	6.3690
18084	59	1	5.8608	5.9480	6.0088
18107	67	2	6.2860	6.2166	6.3690
18188	96	4	10.1914	9.9637	10.1933
18894	24	1	4.3041	4.4427	4.6375
18963	16	1	4.4773	4.2767	4.3603

Table 4 Imputations for identified errors in y3 values of log(TURNOVER+1)

REF	TURNREG CLASS	EMPREG CLASS	TRUE VALUE	RANDOM IMPUTE	MEAN IMPUTE
284	95	6	9.5599	9.5932	9.7412
303	98	6	10.9272	11.0953	11.1149
709	98	6	10.8215	10.8881	11.1149
775	96	5	9.9404	10.1301	10.1933
1315	96	6	10.3678	10.2942	10.1933
2497	90	4	8.4000	8.4029	8.4258
2902	87	3	8.3195	8.2022	8.0892
3011	80	4	7.4448	7.1709	7.2567
3097	100	6	12.7652	13.6733	13.6365
3489	96	6	10.1254	10.2583	10.1933
3572	71	2	6.4118	6.5610	6.5741
3676	84	4	7.5585	7.6971	7.5587
4052	98	6	10.7908	11.1701	11.1149
4370	100	6	16.3603	13.1454	13.6365
5803	94	5	9.2641	9.5485	9.4123
5937	91	5	8.8471	8.8051	8.8157
6155	76	4	6.9187	6.8967	6.8667
6326	97	6	10.5906	10.6339	10.5759
6629	79	3	7.1693	7.3784	7.1014
7009	99	6	11.9795	12.1266	12.0154
7022	100	6	12.7164	13.0603	13.6365
7125	97	6	10.3577	10.4862	10.5759
7912	69	3	6.3953	6.3986	6.3690
8593	100	6	12.8892	12.6550	13.6365
9651	30	1	4.8283	4.8828	4.8907
9714	53	1	5.6768	5.7652	5.5979
9754	48	1	5.4510	5.3613	5.4603
9826	32	1	5.8608	4.8903	4.8907
9863	71	5	6.9460	6.5917	6.5741
9938	46	1	4.8752	5.1475	5.3080
10215	17	1	4.5326	4.3041	4.3603
10420	44	1	5.3799	5.1705	5.3080
10532	91	4	8.5960	8.7751	8.8157
10559	36	1	5.1417	4.9127	5.0955
10635	22	1	4.5643	4.4067	4.5513
10763	85	4	7.6714	7.7820	7.7851
10829	93	4	9.2265	9.0508	9.1466
10893	62	2	5.7301	6.2324	6.1132
10903	50	2	5.5452	5.3566	5.5979
10992	41	1	5.0173	5.1874	5.1875
11089	90	4	5.3230	8.3772	8.4258
11119	12	1	4.2627	4.2485	4.2034
11143	93	5	8.8557	8.9274	9.1466
11174	29	1	4.2627	4.7005	4.7833

REF	TURNREG CLASS	EMPREG CLASS	TRUE VALUE	RANDOM IMPUTE	MEAN IMPUTE
11225	73	1	6.8438	6.9930	6.7356
11269	67	1	6.3630	6.2823	6.3690
11374	17	1	4.3944	4.2627	4.3603
11382	89	4	8.2340	8.3624	8.4258
11400	52	1	5.6971	5.5568	5.5979
11418	51	1	5.6384	5.5872	5.5979
11419	77	1	6.9903	7.1381	7.0157
11476	90	4	8.4532	8.3347	8.4258
11485	13	1	4.0604	4.3041	4.2559
11581	37	1	4.9488	5.0562	5.0955
11631	72	1	6.3456	6.5352	6.5741
11695	71	3	6.0822	6.5058	6.5741
11710	20	1	4.1897	4.5326	4.5513
11776	36	1	4.8828	5.2983	5.0955
11784	40	1	5.0562	5.3230	5.1875
11814	74	2	5.9269	6.4313	6.7356
11913	99	6	12.3400	11.9281	12.0154
11924	8	1	2.8904	3.9703	4.0141
11992	58	1	5.8551	5.8111	5.9028
12116	20	1	4.6250	4.6151	4.5513
12172	38	1	4.9345	5.1985	5.0955
12193	43	2	7.2591	5.3279	5.3080
12305	64	2	6.4102	6.3190	6.2344
12351	59	1	6.0913	6.0426	6.0088
12372	17	1	4.3820	4.4543	4.3603
12420	11	1	4.3944	4.3694	4.2034
12425	72	3	6.3699	6.6412	6.5741
12428	54	1	5.6904	5.5835	5.7337
12437	42	1	5.5255	5.2883	5.3080
12460	77	4	7.0707	7.1131	7.0157
12530	41	1	4.4308	5.2470	5.1875
12685	36	1	4.7362	5.2311	5.0955
12701	51	2	5.7104	5.4806	5.5979
12742	30	1	4.7622	4.7274	4.8907
12935	39	1	5.4723	5.3706	5.1875
12970	31	1	4.8828	4.6821	4.8907
12988	14	1	4.4067	4.3307	4.3603
13045	49	2	5.0434	5.5568	5.5979
13150	79	3	7.2542	7.1317	7.1014
13194	56	1	5.8111	5.7398	5.8261
13239	3	1	3.4340	3.5264	3.5262
13251	83	2	7.5126	7.5000	7.5587
13333	59	2	5.5491	5.8665	6.0088
13392	15	1	3.7377	4.3175	4.3603
13449	5	1	3.8067	3.9318	3.7430
13562	41	1	5.0304	5.1240	5.1875

REF	TURNREG CLASS	EMPREG CLASS	TRUE VALUE	RANDOM IMPUTE	MEAN IMPUTE
13599	15	1	4.2485	4.4308	4.3603
13680	78	3	6.9976	6.9344	7.0157
13861	20	1	4.5643	4.6151	4.5513
13894	2	1	2.8904	2.7081	3.2615
13964	64	1	6.2166	6.1675	6.2344
14086	17	1	4.4886	4.3041	4.3603
14131	53	2	5.7203	5.6454	5.5979
14188	49	1	5.5607	5.6490	5.5979
14225	51	1	5.7398	5.5491	5.5979
14379	7	1	5.5759	3.8712	4.0141
14451	73	1	6.0981	6.6958	6.7356
14460	40	1	5.6021	5.1985	5.1875
14473	28	1	4.9972	4.9836	4.7833
14486	4	1	3.5835	3.9703	3.7430
14490	91	4	9.0101	8.7178	8.8157
14496	3	1	3.4657	3.4965	3.5262
14535	50	1	5.7071	5.5452	5.5979
14564	46	1	5.1180	5.2523	5.3080
14649	85	3	7.6123	7.7341	7.7851
14699	73	2	6.6214	6.7534	6.7356
14713	55	1	5.8319	6.0014	5.8261
14728	8	1	4.1431	4.0431	4.0141
14847	9	1	3.9512	3.9120	4.0141
14858	17	1	4.4188	4.3944	4.3603
14901	2	1	8.1259	3.2189	3.2615
14961	23	1	4.5951	4.5850	4.6375
15003	85	2	7.7437	7.7231	7.7851
15007	61	2	5.8636	6.1312	6.0088
15017	37	1	5.0434	5.0499	5.0955
15068	75	2	7.2284	6.9537	6.8667
15072	25	1	4.0073	4.6052	4.7053
15078	49	1	5.4337	5.6131	5.5979
15094	6	1	3.8501	4.1897	3.9152
15222	59	1	5.8081	5.9965	6.0088
15226	50	1	5.6384	5.4931	5.5979
15239	79	3	6.5525	7.2218	7.1014
15263	41	1	5.1818	5.2204	5.1875
15276	40	1	4.6151	5.1120	5.1875
15300	57	1	5.9940	5.8230	5.8261
15333	60	1	5.8777	6.1924	6.0088
15377	71	2	6.6567	6.5280	6.5741
15462	3	1	3.4965	3.5835	3.5262
15501	26	1	4.1897	4.7536	4.7053
15513	7	1	3.9703	4.2195	4.0141
15541	21	1	4.6250	4.5951	4.5513
15588	64	1	6.2364	6.1549	6.2344

REF	TURNREG CLASS	EMPREG CLASS	TRUE VALUE	RANDOM IMPUTE	MEAN IMPUTE
15594	54	1	5.9940	5.8721	5.7337
15674	95	5	9.5331	9.7092	9.7412
15749	69	2	6.6606	6.3172	6.3690
15972	13	1	4.2047	4.3944	4.2559
16011	34	1	5.0938	5.0039	5.0040
16028	56	1	5.5607	6.0450	5.8261
16057	37	1	4.7536	5.0938	5.0955
16087	2	1	4.0431	4.1109	3.2615
16134	59	1	6.0426	6.0403	6.0088
16181	18	1	4.4998	4.3820	4.4676
16200	68	1	5.2149	6.4313	6.3690
16239	47	1	5.2983	5.4249	5.4603
16278	17	1	4.4067	4.3567	4.3603
16292	11	1	4.3307	4.4773	4.2034
16400	62	1	4.6728	6.2538	6.1132
16479	66	1	6.2166	6.1612	6.2344
16526	3	1	2.9444	3.4340	3.5262
16531	41	2	5.2523	5.2627	5.1875
16564	58	1	6.0137	5.8665	5.9028
16678	48	2	5.4116	5.6276	5.4603
16853	73	2	6.6958	6.9622	6.7356
16952	49	1	5.5568	5.4848	5.5979
16959	54	1	5.6384	5.7071	5.7337
16985	13	1	4.4659	4.2047	4.2559
17055	11	1	4.1897	4.0943	4.2034
17104	31	1	4.8752	5.0304	4.8907
17148	25	1	4.5539	4.6821	4.7053
17210	77	2	6.8298	7.2027	7.0157
17317	29	1	4.7622	4.8828	4.7833
17397	37	2	4.9904	4.8752	5.0955
17528	2	1	3.5264	4.1109	3.2615
17559	15	1	4.4308	4.5109	4.3603
17816	11	1	10.9331	4.0073	4.2034
17875	62	1	6.0638	6.0981	6.1132
17920	13	1	4.2767	4.3567	4.2559
18015	1	1	4.1271	2.4849	2.5510
18102	67	2	6.2344	6.3630	6.3690
18113	76	3	6.1862	6.9187	6.8667
18175	80	3	7.2862	7.2682	7.2567
18182	1	1	4.6151	1.9459	2.5510
18203	72	2	6.4599	6.5889	6.5741
18334	10	1	3.7842	4.1109	4.0913
18340	57	1	5.9687	5.9713	5.8261
18428	36	1	3.9120	5.3279	5.0955
18434	32	1	4.8283	4.7958	4.8907
18469	2	1	3.8286	2.5649	3.2615

REF	TURNREG CLASS	EMPREG CLASS	TRUE VALUE	RANDOM IMPUTE	MEAN IMPUTE
18478	71	3	6.6503	6.7696	6.5741
18535	41	2	5.2575	5.1648	5.1875
18554	68	2	6.3886	6.2672	6.3690
18696	59	1	5.9738	5.9713	6.0088
18697	17	1	4.8203	4.2767	4.3603
18707	40	1	5.3706	5.2204	5.1875
18720	34	1	4.8978	5.0689	5.0040
18770	91	5	8.9141	8.7038	8.8157
18798	42	1	5.2523	5.1705	5.3080
18861	50	1	5.4848	5.6240	5.5979
18886	88	2	7.9700	8.2825	8.0892

Table 5 Imputations for identified "correct" outliers in y3 values of log(TURNOVER+1)

REF	TURNREG CLASS	EMPREG CLASS	TRUE VALUE	RANDOM IMPUTE	MEAN IMPUTE
3555	73	6	12.2534	6.6933	6.7356
3997	34	6	12.1357	4.9904	5.0040
8982	84	6	11.5517	7.8071	7.5587
9609	80	4	0.0000	7.0397	7.2567
10379	2	1	9.0091	3.2958	3.2615
14100	3	1	0.6931	3.4012	3.5262
15052	55	1	1.3863	5.9940	5.8261
15546	4	3	9.1144	3.7136	3.7430
16421	59	1	0.0000	5.9940	6.0088
17162	9	1	1.0986	3.7612	4.0141
18050	25	1	1.3863	4.8442	4.7053

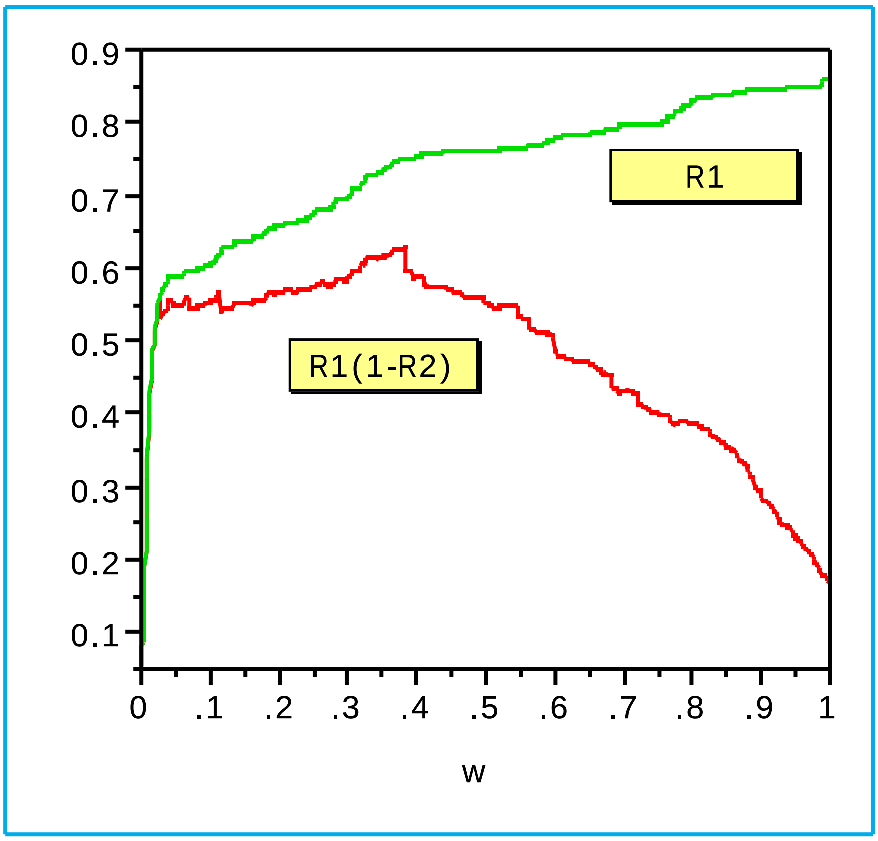
3.2.3 Finding the Optimal Weight Thresholds for other ABI Variables

The approach outlined in the previous subsection was applied to the other 1997 ABI total variables EMPTOTC, PURTOT, TAXTOT, ASSACQ, ASSDISP and EMPLOY. Below we discuss results for each of these variables in turn. In each case a robust WAID tree based on the register covariates TURNREG and EMPREG was fitted.

EMPTOTC

There are 41 missing values of EMPTOTC in the 1997 ABI dataset. Of the remaining 6058 values, 658 are zero. These values were not treated specially in fitting the robust WAID tree for this variable. Figure 5 shows the values of R_1 and $R_1(1\text{-}R_2)$ generated by this tree. We note that in this case the optimal and conservative choices for the optimal threshold w are the same, corresponding to w = 0.385. At this value, 75% of errors in EMPTOTC are detected and 16% of identified outliers are correct values.

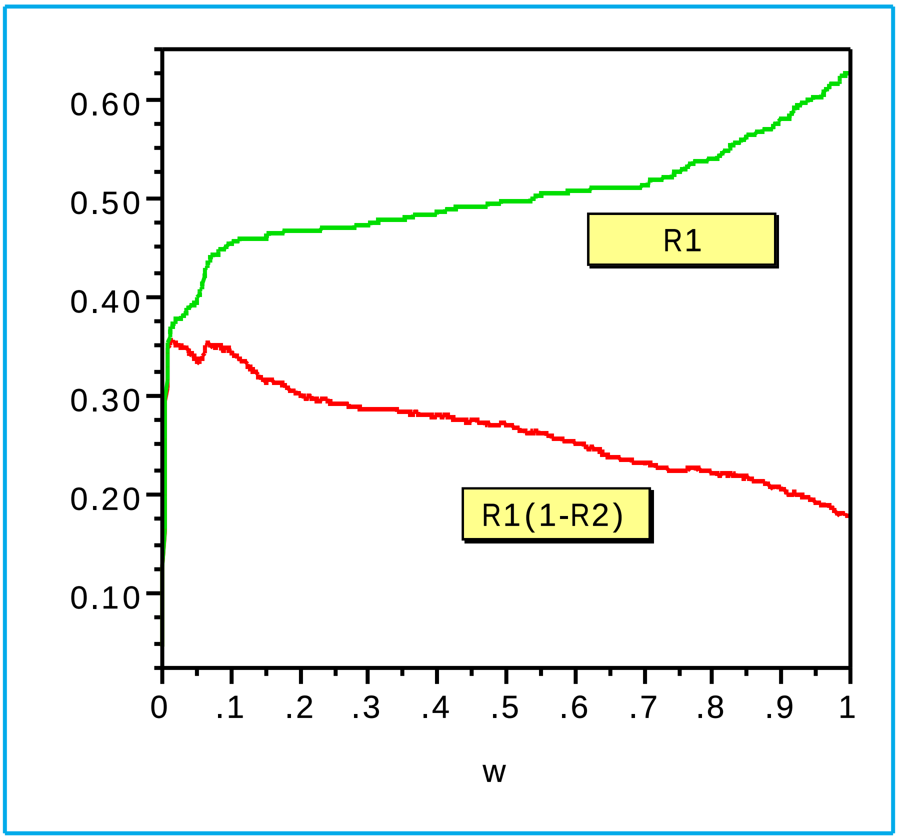
Figure 5 Plot of $R_1(w)$ and $R_1(w)(1\text{-}R_2(w))$ for EMPTOTC



PURTOT

There are 28 missing values of PURTOT in the 1997 ABI dataset. Of the remaining 6071 values, 5 are zero. These values were not treated specially in fitting the robust WAID tree for this variable. Figure 6 shows the values of R_1 and $R_1(1-R_2)$ generated by this tree. For this variable there is a clear distinction between the optimal and conservative values for the outlier detection threshold. The optimal threshold is w = 0.018, where 37% of errors in PURTOT are detected and 4% of identified outliers are correct values. The conservative threshold is w = 0.067, where 43% of errors are detected and 18% of identified outliers are correct values. This can be seen in Figure 5 where the increase in the rate of detection of errors slows down considerably after this conservative threshold value.

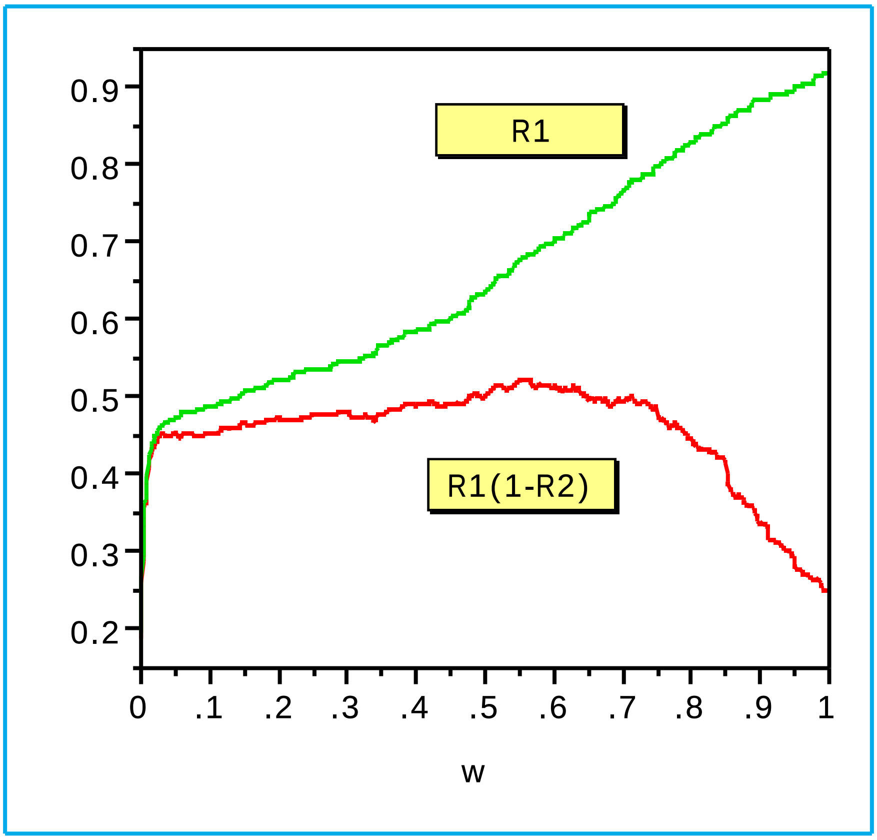
Figure 5 Plot of $R_1(w)$ and $R_1(w)(1-R_2(w))$ for PURTOT



TAXTOT

There are 45 missing values of TAXTOT in the 1997 ABI dataset. Of the remaining 6054 values, 435 are zero. These values were not treated specially in fitting the robust WAID tree for this variable. Figure 6 shows the values of R_1 and $R_1(1-R_2)$ generated by this tree. In this case there is a degree of uncertainty about the choice between the optimal and conservative choices for the weight threshold for outlier detection based on this tree. The optimal threshold is w = 0.559, where 68% of TAXTOT errors are detected, with 4% of identified outliers corresponding to true values. In contrast, at w = 0.750, 80% of errors are identified, with 39% of identified outliers corresponding to true values of TAXTOT.

Figure 6 Plot of $R_1(w)$ and $R_1(w)(1-R_2(w))$ for TAXTOT



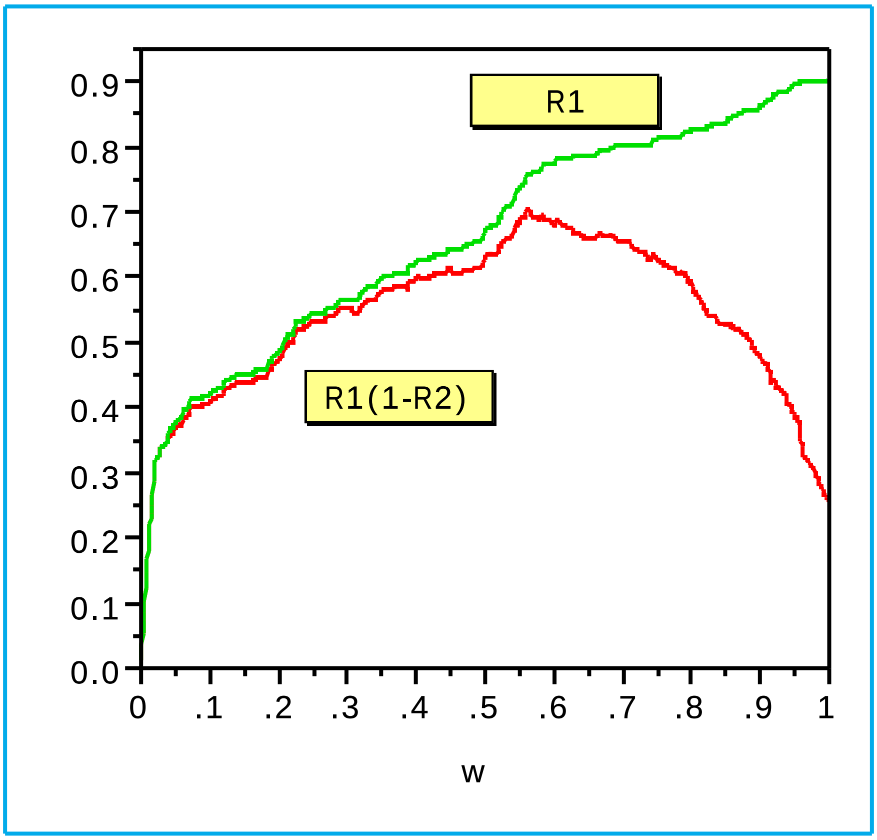
ASSACQ

There are 57 missing values of ASSACQ in the 1997 ABI dataset. In addition, there are 908 records with ASSACQ = -9. Both types of record were ignored when fitting the robust WAID tree for this variable. Of the remaining 5134 values, 2106 are zero. This high proportion of zero values is quite different from the preceding ABI total variables, On the basis that so many zero values cannot all be outliers, we therefore also excluded records with ASSACQ = 0 from the tree fitting process.

When zero values excluded, we obtain the plots for R_1 and $R_1(1-R_2)$ shown in Figure 7. Here the optimal threshold is w = 0.565, where 76% of non-zero errors are identified, with 7% of identified outliers corresponding to true values. We also see that there is no difference between the optimum and conservative thresholds in this case.

Since in practice some of the zero values in ASSACQ can be errors, it is necessary to have a strategy that can deal with these. However, since only four of the 248 errors in this variable corresponded to zero values in 1997, there seems little lost by ignoring these errors.

Figure 7 Plot of $R_1(w)$ and $R_1(w)(1-R_2(w))$ for positive values of ASSACQ

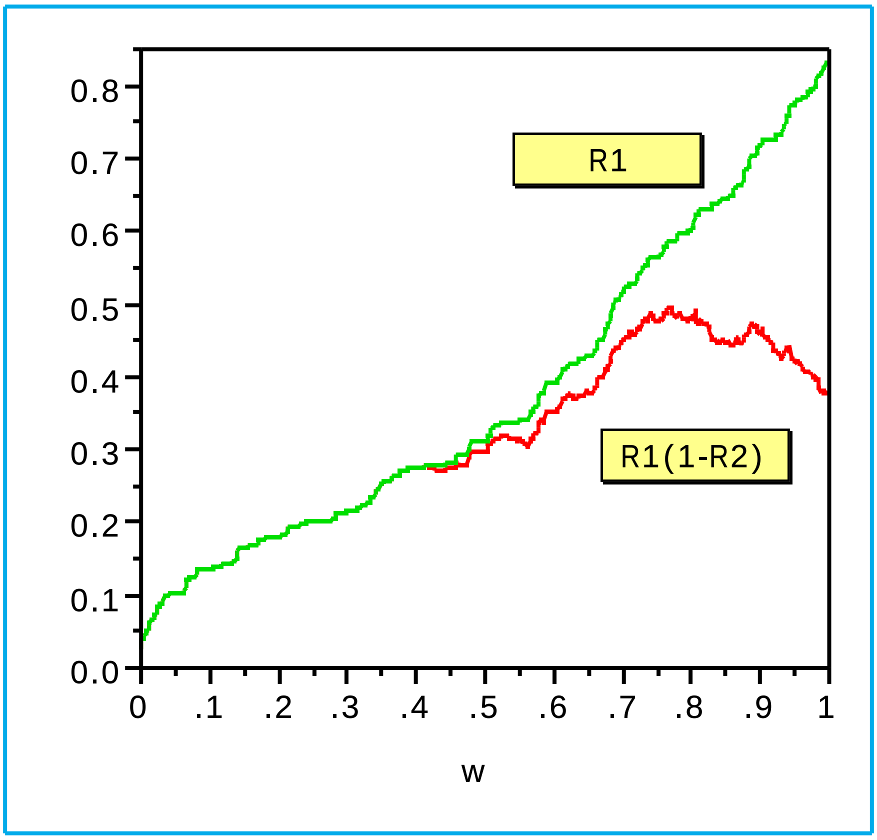


ASSDISP

There are 63 missing values of ASSDISP in the 1997 ABI dataset. In addition, there are 1389 records with ASSDISP = -9. Both types of record were ignored when fitting the robust WAID tree for this variable. Of the remaining 4647 values, 3208 are zero. This very large number of zero values made it impossible to fit a robust WAID tree that included them. The same argument as used with ASSACQ led to records with ASSDISP = 0 being excluded from the tree fitting process. Figure 8 shows the values of R_1 and $R_1(1-R_2)$ generated by these positive values of ASSDISP. Here the optimal and conservative thresholds are clearly identical, at w = 0.773. At this value 59% of errors are identified, with 16% of identified outliers corresponding to true values.

Note that, as with ASSACQ, the problem then becomes one of how to identify errors in the zero values of ASSDISP. However, since there are only 2 such cases in 1997, there does not seem to be too much lost by ignoring them.

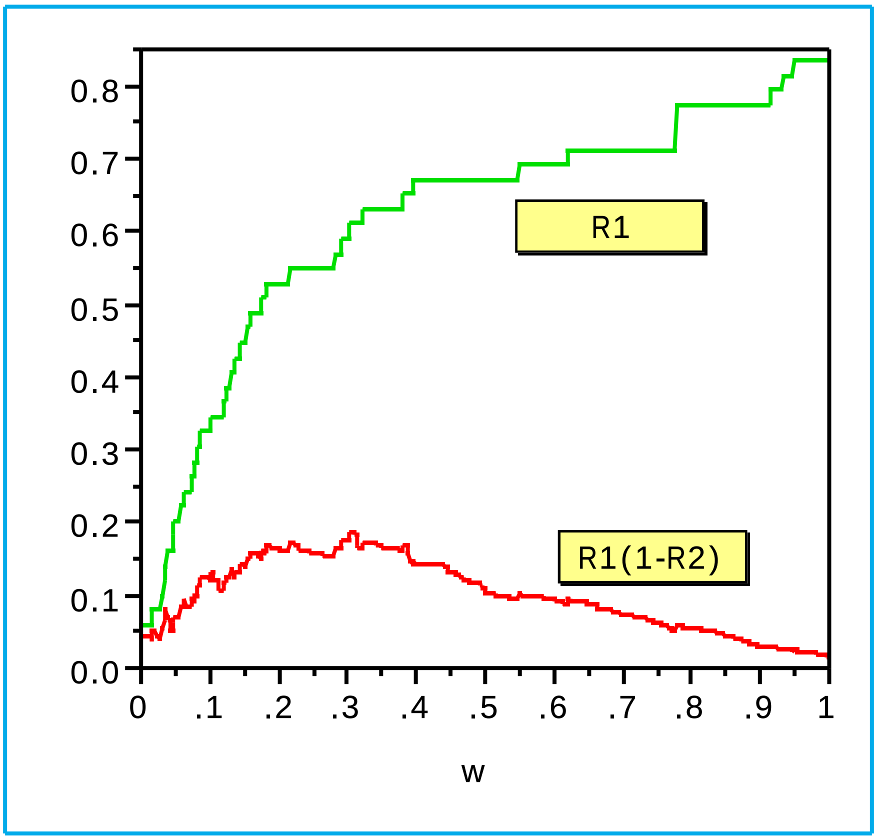
Figure 8 Plot of $R_1(w)$ and $R_1(w)(1-R_2(w))$ for positive values of ASSDISP



EMPLOY

This variable contained just 49 errors and 35 missing values in 1997. With missing values excluded, Figure 9 shows the values of R_1 and $R_1(1-R_2)$ generated by the robust tree defined by the remaining non-missing cases. Here again we see that optimal and conservative thresholds are identical, at w = 0.312. At this value 61% of errors are identified. However, then 69% of identified outliers also correspond to true values.

Figure 9 Plot of $R_1(w)$ and $R_1(w)(1-R_2(w))$ for EMPLOY



In theory a robust WAID tree for a particular ABI variable can be constructed using any or all of the other ABI variables, including the register variables TURNREG and EMPREG. It is therefore of interest to investigate the outlier and error detection performance of alternative robust WAID treees based on different covariate combinations Table 6 shows the values of outlier and error detection performance measures (the same as in Table 2) generated by these trees. Both single covariate trees and multiple covariate trees were investigated, with "A" defining the tree based only on the EMPLOY variable, "B" defining the tree based on all the other ABI total variables (except the analysis variable) and "C" defining the tree based on all the ABI total variables plus the two register variables TURNREG and EMPREG. Note that when other ABI total variables are used as covariates these variables are used with their observed (y3) values, with missing values coded to -1.

The results in Table 6 show that of the three types of trees A, B, C, those based on EMPLOY alone (A) is invariably the better performer, with outlier/error detectino performance similar to that of robust trees based on the register variables TURNREG and EMPREG. Furthermore, in almost all cases trees based on a "bigger" set of covariates perform worse (sometimes considerably worse) than trees based on a "smaller" set of covariates. This is an interesting result. However, since all trees were constrained to 50 terminal nodes, it may be due to the fact that more terminal nodes are required when constructing a tree based on multiple covariates. This is currently being investigated.

Table 6 Outlier identification performance of robust WAID trees based on different covariate combination. Here "A" denotes a tree built using EMPLOY as the only covariate, "B" denotes a tree built using the other total variables as the covariates, and "C" denotes a tree based on the register covariates TURNREG and EMPREG plus all the other total variables. Note that where another total variable is used as a covariate, it is categorized into its percentile groups, based on observed (y3) values and with missing values coded to -1. Note also that zero values were excluded when building the trees for ASSACQ and ASSDISP.

y	**TREE type**	**w**	**N_{out}**	**N_{error}**	**N_{sig}**	**R_1**	**R_{sig}**	**R_2**	**$R_1(1-R_2)$**
TURNOVER	**A**	**0.0307**	**196**	**190**	**187**	**0.7884**	**0.8990**	**0.0306**	**0.7642**
	B	0.0949	52	40	32	0.1660	0.1538	0.2308	0.1277
	C	0.1432	112	61	49	0.2531	0.2356	0.4554	0.1379
EMPTOTC	**A**	**0.3635**	**275**	**251**	**245**	**0.7560**	**0.9176**	**0.0873**	**0.6900**
	B	0.6179	227	108	94	0.3253	0.3521	0.5242	0.1548
	C	0.4606	139	83	73	0.2500	0.2734	0.4029	0.1493
PURTOT	**A**	**0.0487**	**245**	**234**	**231**	**0.3720**	**0.9094**	**0.0449**	**0.3553**
	B	0.2326	169	110	71	0.1749	0.2795	0.3491	0.1138
	C	0.2201	166	111	73	0.1765	0.2874	0.3313	0.1180
TAXTOT	**A**	**0.3588**	**300**	**272**	**271**	**0.5643**	**0.6145**	**0.0933**	**0.5116**
	B	0.7255	503	247	241	0.5124	0.5465	0.5089	0.2516
	C	0.7246	502	248	243	0.5145	0.5510	0.5060	0.2542
ASSACQ	**A**	**0.5590**	**202**	**186**	**186**	**0.7620**	**0.7980**	**0.0790**	**0.7019**
	B	0.4900	61	48	48	0.1970	0.2060	0.2130	0.1548
	C	0.5940	96	60	59	0.2460	0.2530	0.3750	0.1537
ASSDISP	**A**	**0.7674**	**149**	**136**	**136**	**0.6154**	**0.6326**	**0.0872**	**0.5617**
	B	0.7293	98	51	51	0.2308	0.2372	0.4796	0.1201
	C	0.7085	96	53	53	0.2398	0.2465	0.4479	0.1324
EMPLOY	B	0.0924	27	13	13	0.2653	0.4333	0.5185	0.1277
	C	**0.1148**	**34**	**17**	**17**	**0.3469**	**0.5667**	**0.5000**	**0.1735**

4. Further Research

The influence of tree size on outlier and error detection performance is being investigated using the 1997 ABI dataset. In addition, application of the WAID-based outlier detection and imputation approach to the Swiss EPE dataset is being carried out. Finally, the extension of the WAID algorithm to allow trees based on multivariate response variables is being implemented.

Appendix A The WAID Toolkit

This appendix contains a complete listing of the Splus/R code for the different functions that make up the WAID Toolkit. Appendix B illustrates the use of this Toolkit to carry out an edit and imputation analysis for the TURNOVER variable in the sec197(y3) ABI dataset.

```
RTWaid.basic <-
function(y,x,xlabels,mono,delta=0.00001,minn=5,maxnogroups=20,robust=TRUE,method="M",psi=psi.bisquare,c=4.685,maxit=10,verbose=0) {
# This function fits a robust regression tree
# y = response variable data values (numeric, interval scaled)
# x = predictor variable values (numeric, categorical), in matrix form (rows = observations, columns = variables)
# each distinct value in x is taken as defining a category
# there is a limit of 10 categories for any non-monotone x-variable (otherwise GenerateAllSamples function needs modification)
# xlabels = vector of column numbers (or column names) for x-variables
# mono = logical vector identifying which of the x-variables is monotone (TRUE) or non-monotone (FALSE)
# delta = minimum heterogeneity (relative to input y) required before a node can be split
# minn = minimum size of node that can be created
# maxnogroups = maximum number of terminal nodes that can be created
# robust = logical flag (TRUE = use robust WSS to split, FALSE = use unweighted WSS to split)
# method, psi, c, maxit = parameters used to control robust fit (type help(rlm.default) for details).
# verbose = control value for printing out of results
        if(robust==TRUE) library(MASS)
        y <- c(y)
        if(!is.matrix(x)) x <- matrix(x,ncol=1)
        N <- length(y)
        P <- maxnogroups*2+1
        SplitHistory <- vector("list",length=maxnogroups)
        ObsNodeNo <- vector(mode="integer",length=N)
        ObsWt <- vector(mode="double",length=N)
        NodeHetero <- vector(mode="double",length=P)
        NodeSplit <- vector(mode="integer",length=P)
        NodeSize <- vector(mode="integer",length=P)
        NodeLoc <- vector(mode="integer",length=P)
# Initialise variables
        ParentNodeNo <- 1
        NumberNodes <- 1
        NodeSize[1] <- N
        NodeSize[-1] <- 0
        ObsNodeNo[ ] <- 1
        ObsWt[ ] <- 1
        Finish <- FALSE
        TermHistory <- 1
        NodeHistory <- rep(1,N)
        WtHistory <- NULL
# GO
        if(verbose>0) cat("Starting Regression Tree Analysis\n")
        for (nstep in 1:maxnogroups) {
                if (Finish == TRUE) break
                NewNode <- NumberNodes-1
                if (nstep == 1) {
                        NewNode <- 1
                }
                for (j in NewNode:NumberNodes) {
                        NodeSize[j] <- sum(ObsNodeNo==j)
                        Fit <- WeightedSumSquares(x=y[ObsNodeNo==j],robust=robust,method=method,psi=psi,c=c,maxit=maxit)
                        NodeHetero[j] <- Fit$wss
                        NodeLoc[j] <- Fit$loc
                        ObsWt[ObsNodeNo==j] <- Fit$w
# Test if node is allowed to be split
                        if (nstep <= maxnogroups) {
# Check that node has at least twice the minimum number of cases allowed in a node
                                if (NodeSize[j] >= (2*minn)) {
                                        NodeSplit[j] <- 1
                                }
# If not write message
                                if(NodeSplit[j] == 0) {
                                        if(verbose>2) cat("Node ",j," cannot be split, since it is too small\n")
                                }
# Check NodeHetero fraction for node
                                if (zdiv(NodeHetero[j],NodeHetero[1]) <= delta) {
                                        NodeSplit[j] <- 0
# NodeHetero fraction is too small, so write message
                                        if(verbose > 2) cat("Node ",j," cannot be split, since it is too homogeneous\n")
                                }
# Print statistics if node cannot be NodeSplit
                                if (NodeSplit[j] == 0) {
                                        if(verbose > 2) {
                                                cat("Node           N             Heterogeneity\n")
                                                cat(j,"       ",NodeSize[j],"     ",NodeHetero[j],"\n")
                                        }
                                }
# End SplitCriteria
```

```
            }
        }
        candidates <- unique(ObsNodeNo)
        repeat {
            Finish <- TRUE
            for (j in candidates) {
                if (NodeSplit[j] == 1) {
                    Finish <- FALSE
                }
            }
            if(Finish == TRUE) {
                if(verbose > 1) cat("There are no parent nodes left to split\n");
                break
            }
# Parent group statistics
            if(verbose > 1) cat("Candidate nodes are as follows - \n")
            if(verbose > 1) cat("Node         N           Heterogeneity\n")
            for (k in candidates) {
                if (NodeSplit[k] == 1) {
                    if(verbose > 1) cat(k,"      ",NodeSize[k],"          ",NodeHetero[k],"\n")
                }
            }
# After first split find splittable group with maximum NodeHetero
            if(NumberNodes > 1) {
                WSSM <- 0.0
                for (k in candidates) {
                    if (NodeSplit[k] == 1) {
                        if (NodeHetero[k] >= WSSM) {
                            ParentNodeNo <- k
                            WSSM <- NodeHetero[k]
                        }
                    }
                }
            }
            if(verbose > 0) cat("Step ",nstep," New parent node = ",ParentNodeNo,"\n")
# Find best split of new parent group based on each X-variable
            xsplit <- vector("list",length=ncol(x))
            if(verbose > 2) cat("Predictor      Best Split Heterogeneity          Left Codes        Right Codes\n")
            for(k in 1:ncol(x)) {
                ksplit <-
BestRTSplit(y=y[ObsNodeNo==ParentNodeNo],x=x[ObsNodeNo==ParentNodeNo,k],mono=mono[k],w=ObsWt[ObsNodeNo==ParentNodeNo],min=minn,ws
s=NodeHetero[ParentNodeNo],loc=NodeLoc[ParentNodeNo])
                xsplit[[k]] <- ksplit
                if(verbose > 2) {
                    if(xsplit[[k]]$split == FALSE) {
                        cat("Cannot split on predictor ",xlabels[k],"\n")
                    }
                    else {
                        cat(xlabels[k],"        ",xsplit[[k]]$wss,"            ",xsplit[[k]]$left,"            ",xsplit[[k]]$right,"\n")
                    }
                }
            }
# Find the overall best split, if one exists
            splitok <- FALSE
            for(k in 1:ncol(x)) {
                if(xsplit[[k]]$split == TRUE) {
                    splitok <- TRUE
                    best.xsplit <- xsplit[[k]]
                    best.x <- k
                    break
                }
            }
# Where no predictor can be used to split the parent node, flag this parent and try next parent
            if(splitok == FALSE) {
                NodeSplit[ParentNodeNo] <- 0;
                if(verbose > 2) cat("Group ",ParentNodeNo," cannot be split. Decrease in heterogeneity or child node size is always too
small\n")
            }
            else {
                if((ncol(x) > 1) & (best.x < ncol(x))) {
                    for(k in (best.x+1):ncol(x)) {
                        if((xsplit[[k]]$wss < best.xsplit$wss) & (xsplit[[k]]$split == TRUE)) {
                            best.xsplit <- xsplit[[k]]
                            best.x <- k
                        }
                    }
                }
                if(verbose > 1) cat("Best split is on predictor ",xlabels[best.x],"\n")
            }
            if(NodeSplit[ParentNodeNo] == 1) {
```

```
				for(k in best.xsplit$left) {
					ObsNodeNo[(ObsNodeNo==ParentNodeNo)&(x[,best.x]==k)] <- NumberNodes+1
				}
				for(k in best.xsplit$right) {
					ObsNodeNo[(ObsNodeNo==ParentNodeNo)&(x[,best.x]==k)] <- NumberNodes+2
				}
				NumberNodes <- NumberNodes+2
				break
			}
		}
		TermHistory <- TermHistoryUpdate(x=TermHistory,p=ParentNodeNo,c1=NumberNodes-1,c2=NumberNodes)
		NodeHistory <- cbind(NodeHistory,ObsNodeNo)
		WtHistory <- cbind(WtHistory,ObsWt)
		SplitHistory[[nstep]] <- list(parent=ParentNodeNo,child.left=NumberNodes-1,child.right=NumberNodes,predictor=xlabels[best.x],codes.left=best.xsplit$left,codes.right=best.xsplit$right)
	}
	if (Finish != TRUE) {
# End of splitting since maximum number of terminal nodes has being reached
		if(verbose > 0) cat("End of run since number of terminal nodes >= ", maxnogroups,"\n")
	}
	NumberTerminals <- nrow(TermHistory)
	TerminalHistory <- vector("list",length=NumberTerminals)
	for(i in 1:NumberTerminals) {
		TerminalHistory[[i]] <- unique(TermHistory[i,])
	}
# Output results
	list(nodes = NodeHistory, splits = SplitHistory, terminals = TerminalHistory, weights = WtHistory)
}

CTWaid.basic <- function(y,x,xlabels,mono,delta=0.00001,minn=5,maxnogroups=20,gini=TRUE,verbose=0) {
# this function fits a classification tree
# y = response variable data values (numeric, categorical)
# x = predictor variable values (numeric, categorical), in matrix form (rows = observations, columns = variables)
# each distinct value in y (and x) is taken as defining a category
# there is a limit of 10 categories for any non-monotone x-variable (otherwise GenerateAllSamples function needs modification)
# xlabels = vector of column numbers (or column names) for x-variables
# mono = logical vector identifying which of the x-variables is monotone (TRUE) or non-monotone (FALSE)
# delta = minimum heterogeneity (relative to input y) required before a node can be split
# minn = minimum size of node that can be created
# maxnogroups = maximum number of terminal nodes that can be created
# gini = logical flag (TRUE = use Gini heterogeneity index, FALSE = use Kullback-Leibler heterogeneity index)
# verbose = control value for printing out of results
    y <- c(y)
    if(!is.matrix(x)) x <- matrix(x,ncol=1)
    N <- length(y)
    P <- maxnogroups*2+1
    SplitHistory <- vector("list",length=maxnogroups)
    ObsNodeNo <- vector(mode="integer",length=N)
    NodeHetero <- vector(mode="double",length=P)
    NodeSplit <- vector(mode="integer",length=P)
    NodeSize <- vector(mode="integer",length=P)
# Initialise variables
    ParentNodeNo <- 1
    NumberNodes <- 1
    NodeSize[1] <- N
    NodeSize[-1] <- 0
    ObsNodeNo[ ] <- 1
    Finish <- FALSE
    TermHistory <- 1
    NodeHistory <- rep(1,N)
# GO
    if(verbose>0) cat("Starting Classification Tree Analysis\n")
    for (nstep in 1:maxnogroups) {
        if (Finish == TRUE) break
        NewNode <- NumberNodes-1
        if (nstep == 1) {
            NewNode <- 1
        }
        for (j in NewNode:NumberNodes) {
            NodeSize[j] <- sum(ObsNodeNo==j)
            if(gini == TRUE) {
                NodeHetero[j] <- GiniCoefficient(y[ObsNodeNo==j])$gini
            }
            else {
                NodeHetero[j] <- GiniCoefficient(y[ObsNodeNo==j])$info
            }
# Test if node is allowed to be NodeSplit
            if (nstep <= maxnogroups) {
# Check that node has at least twice the minimum number of cases allowed in a node
                if (NodeSize[j] >= (2*minn)) {
                    NodeSplit[j] <- 1
                }
# If not write message
                if(NodeSplit[j] == 0) {
                    if(verbose>2) cat("Node ",j," cannot be split, since it is too small\n")
                }
# Check NodeHetero fraction for node
                if (zdiv(NodeHetero[j],NodeHetero[1]) <= delta) {
                    NodeSplit[j] <- 0
# NodeHetero fraction is too small, so write message
                    if(verbose > 2) cat("Node ",j," cannot be split, since it is too homogeneous\n")
                }
# Print statistics if node cannot be NodeSplit
                if (NodeSplit[j] == 0) {
                    if(verbose > 2) {
                        cat("Node             N              Heterogeneity\n")
                        cat(j,"         ",NodeSize[j],"        ",NodeHetero[j],"\n")
                    }
                }
# End SplitCriteria
            }
        }
        candidates <- unique(ObsNodeNo)
        repeat {
            Finish <- TRUE
            for (j in candidates) {
                if (NodeSplit[j] == 1) {
                    Finish <- FALSE
                }
            }
            if(Finish == TRUE) {
                if(verbose > 1) cat("There are no parent nodes left to split\n");
                break
            }
# Parent group statistics
            if(verbose > 1) cat("Candidate nodes are as follows - \n")
            if(verbose > 1) cat("Node             N              Heterogeneity\n")
            for (k in candidates) {
                if (NodeSplit[k] == 1) {
                    if(verbose > 1) cat(k,"         ",NodeSize[k],"               ",NodeHetero[k],"\n")
                }
            }
# After first split find splittable group with maximum NodeHetero
```

```
            if(NumberNodes > 1) {
                TGINIM <- 0.0
                for (k in candidates) {
                    if (NodeSplit[k] == 1) {
                        if (NodeHetero[k] >= TGINIM) {
                            ParentNodeNo <- k
                            TGINIM <- NodeHetero[k]
                        }
                    }
                }
            }
            if(verbose > 0) cat("Step ",nstep," New parent node = ",ParentNodeNo,"\n")
# Find best split of new parent group based on each X-variable
            xsplit <- vector("list",length=ncol(x))
            if(verbose > 2) cat("Predictor        Best Split Heterogeneity        Left Codes        Right Codes\n")
            for(k in 1:ncol(x)) {
                ksplit <- BestCTSplit(y=y[ObsNodeNo==ParentNodeNo],x=x[ObsNodeNo==ParentNodeNo,k],mono=mono[k],min=minn,gini=gini,gini.all=NodeHetero[ParentNodeNo])
                xsplit[[k]] <- ksplit
                if(verbose > 2) {
                    if(xsplit[[k]]$split == FALSE) {
                        cat("Cannot split on predictor ",xlabels[k],"\n")
                    }
                    else {
                        cat(xlabels[k],"        ",xsplit[[k]]$gini,"        ",xsplit[[k]]$left,"        ",xsplit[[k]]$right,"\n")
                    }
                }
            }
# Find the overall best split, if one exists
            splitok <- FALSE
            for(k in 1:ncol(x)) {
                if(xsplit[[k]]$split == TRUE) {
                    splitok <- TRUE
                    best.xsplit <- xsplit[[k]]
                    best.x <- k
                    break
                }
            }
# Where no predictor can be used to split the parent node, flag this parent and try next parent
            if(splitok == FALSE) {
                NodeSplit[ParentNodeNo] <- 0;
                if(verbose > 2) cat("Group ",ParentNodeNo," cannot be split. Decrease in heterogeneity or child node size is always too
small\n")
            }
            else {
                if((ncol(x) > 1) & (best.x < ncol(x))) {
                    for(k in (best.x+1):ncol(x)) {
                        if((xsplit[[k]]$gini < best.xsplit$gini) & (xsplit[[k]]$split == TRUE)) {
                            best.xsplit <- xsplit[[k]]
                            best.x <- k
                        }
                    }
                }
                if(verbose > 1) cat("Best split is on predictor ",xlabels[best.x],"\n")
            }
            if(NodeSplit[ParentNodeNo] == 1) {
                for(k in best.xsplit$left) {
                    ObsNodeNo[(ObsNodeNo==ParentNodeNo)&(x[,best.x]==k)] <- NumberNodes+1
                }
                for(k in best.xsplit$right) {
                    ObsNodeNo[(ObsNodeNo==ParentNodeNo)&(x[,best.x]==k)] <- NumberNodes+2
                }
                NumberNodes <- NumberNodes+2
                break
            }
        }
        TermHistory <- TermHistoryUpdate(x=TermHistory,p=ParentNodeNo,c1=NumberNodes-1,c2=NumberNodes)
        NodeHistory <- cbind(NodeHistory,ObsNodeNo)
        SplitHistory[[nstep]] <- list(parent=ParentNodeNo,child.left=NumberNodes-
1,child.right=NumberNodes,predictor=xlabels[best.x],codes.left=best.xsplit$left,codes.right=best.xsplit$right)
    }
    if (Finish != TRUE) {
# End of splitting since maximum number of terminal nodes has being reached
        if(verbose > 0) cat("End of run since number of terminal nodes >= ", maxnogroups,"\n")
    }
    NumberTerminals <- nrow(TermHistory)
    TerminalHistory <- vector("list",length=NumberTerminals)
    for(i in 1:NumberTerminals) {
        TerminalHistory[[i]] <- unique(TermHistory[i,])
    }
# Output results
```

```
        list(nodes = NodeHistory, splits = SplitHistory, terminals = TerminalHistory)
}

WeightedSumSquares <- function(x,robust=TRUE,method="M",psi=psi.bisquare,c=4.685,maxit=10) {
# this function computes within node heterogeneity for regression trees
        n  <- length(c(x))
        if(robust==TRUE) {
                meanfit <- rlm.default(x=rep(1,n),y=c(x),method=method,psi=psi,c=c,maxit=maxit)
                w <- meanfit$w
                w <- w*zdiv(n,sum(w))
                loc <- zdiv(sum(x*w),n)
        }
        else {
                w <- rep(1,n)
                loc <- mean(x)
        }
        wss <- sum(w*x^2)-n*loc^2
        list(loc = loc, wss = wss, w = w)
}

GiniCoefficient <- function(x) {
# this function computes within node heterogeneity for classification trees
        n <- length(x)
        values <- unique(x)
        p <- rep(0, length(values))
        gini <- 0
        info <- 0
        for(i in 1:length(values)) {
                freq <- sum(x==values[i])
                p[i] <- freq/n
                gini <- gini + (p[i]*(1-p[i]))
                info <- info - (p[i]*log(p[i]))
        }
        gini <- n*gini
        info <- n*info
        list(p = p, gini = gini, info=info)
}

BestRTSplit <- function(y,x,v=unique(x),mono=FALSE,w=rep(1,length(y)),min=5,wss=0,loc=0) {
# this regression tree function determines the optimal split of a node based on the values of a covariate
        splittable <- FALSE
```

```
	best.split <- list(left=v,right=NULL)
	best.split.wss <- 0
	if(length(v) > 1) {
		if(mono==TRUE) {
			v.sort <- sort(v)
		}
		else {
			v.ybar <- rep(0,length(v))
			for(i in 1:length(v)) {
				v.ybar[i] <- zdiv(sum(y[x==v[i]]*w[x==v[i]]),sum(w[x==v[i]]))
			}
			v.sort <- v[order(v.ybar)]
		}
		left <- v.sort
		right <- NULL
		best.split <- list(left=left,right=right)
		best.split.wss <- wss
		for(i in 1:(length(v)-1)) {
			left <- v.sort[1:i]
			y.left <- Subset(y=y,x=x,v=left)
			w.left <- Subset(y=w,x=x,v=left)
			n.left <- sum(w.left)
			ybar.left <- zdiv(sum(y.left*w.left),n.left)
			right <- v.sort[-(1:i)]
			y.right <- Subset(y=y,x=x,v=right)
			w.right <- Subset(y=w,x=x,v=right)
			n.right <- sum(w.right)
			ybar.right <- zdiv(sum(y.right*w.right),n.right)
			if((length(y.left) >= min) & (length(y.right) >= min)) {
				split.wss <- wss-(n.left*(ybar.left-loc)^2)-(n.right*(ybar.right-loc)^2)
				if(split.wss < best.split.wss) {
					best.split <- list(left=left,right=right)
					best.split.wss <- split.wss
					splittable <- TRUE
				}
			}
		}
	}
	list(left=best.split$left,right=best.split$right,wss=best.split.wss,split=splittable)
}
```

```
BestCTSplit <- function(y,x,v=unique(x),mono=FALSE,min=5,gini=TRUE,gini.all=0) {
# this classification tree function determines the optimal split of a node based on the values of a covariate
        splittable <- FALSE
        best.split <- list(left=v,right=NULL)
        best.split.gini <- 0
        if(length(v) > 1) {
                if(mono==TRUE) {
                        v.sort <- sort(v)
                        left <- v.sort
                        right <- NULL
                        gini.left <- gini.all
                        gini.right <- 0
                        best.split <- list(left=left,right=right)
                        best.split.gini <- gini.left+gini.right
                        for(i in 1:(length(v)-1)) {
                                left <- v.sort[1:i]
                                y.left <- Subset(y=y,x=x,v=left)
                                right <- v.sort[-(1:i)]
                                y.right <- Subset(y=y,x=x,v=right)
                                if((length(y.left) >= min) & (length(y.right) >= min)) {
                                        if(gini==TRUE) {
                                                gini.left <- GiniCoefficient(y.left)$gini
                                                gini.right <- GiniCoefficient(y.right)$gini
                                        }
                                        else {
                                                gini.left <- GiniCoefficient(y.left)$info
                                                gini.right <- GiniCoefficient(y.right)$info
                                        }
                                        split.gini <- gini.left+gini.right
                                        if(split.gini < best.split.gini) {
                                                best.split <- list(left=left,right=right)
                                                best.split.gini <- split.gini
                                                splittable <- TRUE
                                        }
                                }
                        }
                }
                else {
                        split <- GenerateAllBinarySplits(v=v)
                        left <- v
                        right <- NULL
```

```
            gini.left <- gini.all
            gini.right <- 0
            best.split <- list(left=left,right=right)
            best.split.gini <- gini.left+gini.right
            for(i in 1:length(split)) {
                left <- split[[i]]
                y.left <- Subset(y=y,x=x,v=left)
                right <- Delete(s=left,v=v)
                y.right <- Subset(y=y,x=x,v=right)
                if((length(y.left) >= min) & (length(y.right) >= min)) {
                    if(gini==TRUE) {
                        gini.left <- GiniCoefficient(y.left)$gini
                        gini.right <- GiniCoefficient(y.right)$gini
                    }
                    else {
                        gini.left <- GiniCoefficient(y.left)$info
                        gini.right <- GiniCoefficient(y.right)$info
                    }
                    split.gini <- gini.left+gini.right
                    if(split.gini < best.split.gini) {
                        best.split <- list(left=left,right=right)
                        best.split.gini <- split.gini
                        splittable <- TRUE
                    }
                }
            }
        }
    }
    list(left=best.split$left,right=best.split$right,gini=best.split.gini,split=splittable)
}

GenerateAllBinarySplits <- function(v) {
# this classification tree function generates all candidate splits of a non-monotone covariate
# note that there is a limit of 10 categories for the covariate at present
    maxsize <- floor(length(v)/2)
    number <- rep(0, maxsize)
    for(k in 1:maxsize) number[k] <- choose(length(v),k)
    n <- sum(number)
    splits <- vector("list",length=n)
    count <- 0
    for(k in 1:maxsize) {
        groups <- GenerateAllSamples(v,k)
        for(j in 1:length(groups)) {
            count <- count+1
            splits[count] <- list(groups[[j]])
        }
    }
    splits
}

GenerateAllSamples <- function(x,n) {
# function called by GenerateAllBinarySplits
    if(n > 5) stop("Sample size > 5")
    N <- length(x)
    sample.matrix <- matrix(0,nrow=choose(N,n),ncol=n)
    count <- 0
    for(i1 in 1:N) {
        if(n==1) {
            count <- count+1
            sample.matrix[count,] <- x[i1]
        }
        else {
            if(i1<N) {
            for(i2 in (i1+1):N) {
                if(n==2) {
                    count <- count+1
                    sample.matrix[count,] <- c(x[i1],x[i2])
                }
                else {
                    if(i2<N) {
                    for(i3 in (i2+1):N) {
                        if(n==3) {
                            count <- count+1
                            sample.matrix[count,] <- c(x[i1],x[i2],x[i3])
                        }
                        else {
                            if(i3<N) {
                            for(i4 in (i3+1):N) {
                                if(n==4) {
                                    count <- count+1
                                    sample.matrix[count,] <- c(x[i1],x[i2],x[i3],x[i4])
                                }
                                else {
                                    if(i4<N) {
                                    for(i5 in (i4+1):N) {
                                        count <- count+1
                                        sample.matrix[count,] <- c(x[i1],x[i2],x[i3],x[i4],x[i5])
                                    }
                                    }
                                }
                            }
                            }
                        }
                    }
                    }
                }
            }
            }
        }
    }
    out <- vector("list",length=nrow(sample.matrix))
    for(i in 1:length(out)) out[i] <- list(sample.matrix[i,])
    out
}

Delete <- function(s,v) {
# utility function
    unique(c(s,v))[-(1:length(s))]
}

Subset <- function(y,x,v) {
# utility function
    out <- NULL
    for(i in 1:length(v)) out <- c(out,y[x==v[i]])
    out
}

zdiv <- function(x,y){
# utility function
    if(y==0.0||is.na(x)||is.na(y))
        0.0
    else
        x/y
```

```
}

TermHistoryUpdate <- function(x,p,c1,c2) {
# function that creates the "history" of the tree splitting process
        if(!is.matrix(x)) x <- matrix(x[1],nrow=1,ncol=1)
        n <- ncol(x)
        if(n == 1) {
                x1 <- NULL
                x2 <- x
        }
        else {
                x1 <- x[x[,n] != p,]
                if(!is.matrix(x1)) x1 <- matrix(x1,nrow=1)
                x2 <- x[x[,n] == p,]
        }
        out <- rbind(x1,x2,x2)
        out <- cbind(out,c(x1[,n],c1,c2))
        out
}

TerminalDefinition <- function(t,SplitHistory,TerminalHistory) {
# function that returns the sequence of splits that lead to a particular node
        for(i in 1:length(TerminalHistory)) {
                count <- length(TerminalHistory[[i]])
                if(TerminalHistory[[i]][count]==t) {
                        trace <- TerminalHistory[[i]]
                        definition <- vector("list",length=(length(trace)-1))
                        for(j in 2:length(trace)) {
                                k <- trace[j]
                                split <- ceiling((k-1)/2)
                                right <- TRUE
                                if((k-1)/2 < split) {
                                        right <- FALSE
                                }
                                if(right == FALSE) {
                                        definition[[j-1]] <-
list(nodes=c(SplitHistory[[split]]$parent,SplitHistory[[split]]$child.left),x=SplitHistory[[split]]$predictor,codes=SplitHistory[[split]]$codes.left)
                                }
                                else {
                                        definition[[j-1]] <-
list(nodes=c(SplitHistory[[split]]$parent,SplitHistory[[split]]$child.right),x=SplitHistory[[split]]$predictor,codes=SplitHistory[[split]]$codes.right)
```

```
				}
			}
		break
		}
	}
	definition
}

TerminalFinder <- function(x,xlabels,SplitHistory) {
# function that returns the terminal node in a WAID tree given a set of values for the covariates that define the tree
	node <- 1
	for(i in 1:length(SplitHistory)) {
		if(SplitHistory[[i]]$parent==node) {
			xval <- x[xlabels==SplitHistory[[i]]$predictor]
			if(is.element(xval,SplitHistory[[i]]$codes.left)) {
				node <- SplitHistory[[i]]$child.left
			}
			else {
				node <- SplitHistory[[i]]$child.right
			}
		}
	}
	node
}

NodeMeans <- function(y,nodes,weights) {
# function that calculates the weighted mean values for all nodes in a WAID regression tree
	nodematrix <- nodes
	nsplits <- ncol(weights)
	if(ncol(nodes) > nsplits) nodematrix <- nodes[,1:nsplits]
	if(ncol(nodes) < nsplits) stop("incompatible node and weight matrix")
	nodemeans <- nodematrix
	for(i in 1:nsplits) {
		nodeset <- unique(nodematrix[,i])
		for(j in nodeset) {
			sumwj <- sum(weights[nodematrix[,i]==j,i])
			meanj <- sum(weights[nodematrix[,i]==j,i]*y[nodematrix[,i]==j])/sumwj
			nodemeans[nodematrix[,i]==j,i] <- meanj
		}
	}
	nodemeans
```

```
}

NodeMeanPlot <- function(nodes,means,splits=1:ncol(means),points=TRUE,title="NODEMEAN TREE") {
# function that plots the nodemean tree corresponding to a WAID regression tree
        if(ncol(nodes) < max(splits)) stop("incompatible nodes matrix")
        if(ncol(means) < max(splits)) stop("incompatible means matrix")
        nodematrix <- nodes[,sort(splits)]
        meanmatrix <- means[,sort(splits)]
        plot(x=c(min(splits),max(splits)),y=c(min(meanmatrix),max(meanmatrix)),type="n",xlab="SPLIT",ylab="NODEMEAN")
        if(points) {
                for(i in 1:ncol(nodematrix)) {
                        y <- unique(meanmatrix[,i])
                        m <- length(y)
                        points(x=rep(splits[i],m),y=y)
                }
        }
        else {
                terminals <- unique(nodematrix[,ncol(nodematrix)])
                for(i in sort(terminals)) {
                        nodeindices <- (1:nrow(nodematrix))[nodematrix[, ncol(nodematrix)]==i]
                        lines(x=splits,y=meanmatrix[nodeindices[1],])
                }
        }
        title(title)
}

TreeImputer <- function(ximp,xlabels,yvalues,treesplits,treenodes,treeweights,minwt=1,random=F) {
# function that returns imputed values from terminal nodes associated with input values of covariates
        x <- ximp
        weights <- treeweights
        nodes <- treenodes
        splits <- treesplits
        if(ncol(nodes)!=ncol(weights)) {
                if(ncol(nodes)==(ncol(weights)+1)) {
                        nodes <- nodes[,1:(ncol(nodes)-1)]
                        splits <- splits[1:(length(splits)-1)]
                }
                else stop("node matrix and weight matrix dimensions incompatible")
        }
        if(!is.matrix(x)) x <- matrix(x,nrow=1)
        imputes <- rep(0, nrow(x))
```

```
	for(i in 1:nrow(x)) {
		terminal <- TerminalFinder(x=x[i,],xlabels=xlabels,SplitHistory=splits)
		ydoners <- yvalues[nodes[,ncol(nodes)]==terminal]
		wdoners <- weights[nodes[,ncol(nodes)]==terminal,ncol(weights)]
		mean <- sum(ydoners*wdoners)/sum(wdoners)
		imputes[i] <- mean
		if(random) imputes[i] <- sample(ydoners[wdoners>=minwt],size=1)
	}
	cbind(imputes,x)
}

CatMap <- function(x,bounds,groups) {
# function that maps covariate values to a given set of groups
	if(length(bounds)!=(length(groups)-1)) stop("Incorrect number of groups")
	m <- length(groups)
	xcat <- rep(0, length(x))
	for(i in 1:length(x)) {
		if(x[i] <= bounds[1]) xcat[i] <- groups[1]
		if(x[i] > bounds[m-1]) xcat[i] <- groups[m]
		for(j in 2:(m-1)) {
			xcat[i] <- j
			if((x[i]>bounds[j-1]) & (x[i]<=bounds[j])) break
		}
	}
	xcat
}
```

Appendix B Example WAID Outlier Identification and Imputation Analysis

Read in values of total variables for sec197(true+y3) data

```
abidata <- matrix(scan("totals197(merged).dat",na.strings = "."),ncol=38,byrow=T)
```

Create individual data vectors that exclude missing values of TURNOVER (these have abidata[,8]==3)

Identifiers

```
abidata.ref3 <- abidata[(abidata[,8]==1)|(abidata[,8]==2),1]
```

Values of TURNREG

```
abidata.turnreg3 <- abidata[(abidata[,8]==1)|(abidata[,8]==2),2]
```

Values of EMPREG

```
abidata.empreg3 <- abidata[(abidata[,8]==1)|(abidata[,8]==2),3]
```

Values of true TURNOVER

```
abidata.tturnover3 <- abidata[(abidata[,8]==1)|(abidata[,8]==2),4]
```

Values of observed (with error) TURNOVER

```
abidata.oturnover3 <- abidata[(abidata[,8]==1)|(abidata[,8]==2),7]
```

Values of true(1)/error(2) flag for TURNOVER

```
abidata.turnover.error3 <- abidata[(abidata[,8]==1)|(abidata[,8]==2),8]
```

Calculate percentiles of TURNREG

```
abidata.turnreg3.tile <- quantile(abidata.turnreg3,probs=(1:99)/100)
```

Create classification variable based on TURNREG percentile groups

```
abidata.turnreg3.class <- cut(abidata.turnreg3,breaks=c(-1,abidata.turnreg3.tile,max(abidata.turnreg3)+1),labels = FALSE)
```

Calculate robust WAID tree for TURNOVER using TURNREG and EMPREG as covariates

```
abi.turnover3.waid.wls <- RTWaid.basic(y=log(abidata.oturnover3+1),x=cbind(abidata.turnreg3.class,abidata.empreg3),xlabels=c("TURNREG","EMPREG"),mono=c(TRUE,TRUE),minn=5,maxnogroups=50,robust=TRUE,maxit=100)
```

Calculate robust mean values for all nodes in the tree

```
abi.turnover3.waid.wtmeans <- NodeMeans(y=log(abidata.oturnover3+1),nodes=abi.turnover3.waid.wls$nodes,weights=abi.turnover3.waid.wls$weights)
```

Create a nodemean plot for the tree (see Figure 2)

```
NodeMeanPlot(nodes=abi.turnover3.waid.wls$nodes,means=abi.turnover3.waid.wtmeans,points=FALSE,title="Figure 2: Robust nodemean tree for y3 values of log(TURNOVER+1)")
```

Calculate average weights for each record defining the tree

```
abi.turnover3.waid.avwt <- apply(abi.turnover3.waid.wls$weights,1,mean)
```

Grid search for an optimal threshold weight identifying outliers/errors (This code generates the values in Table 2)

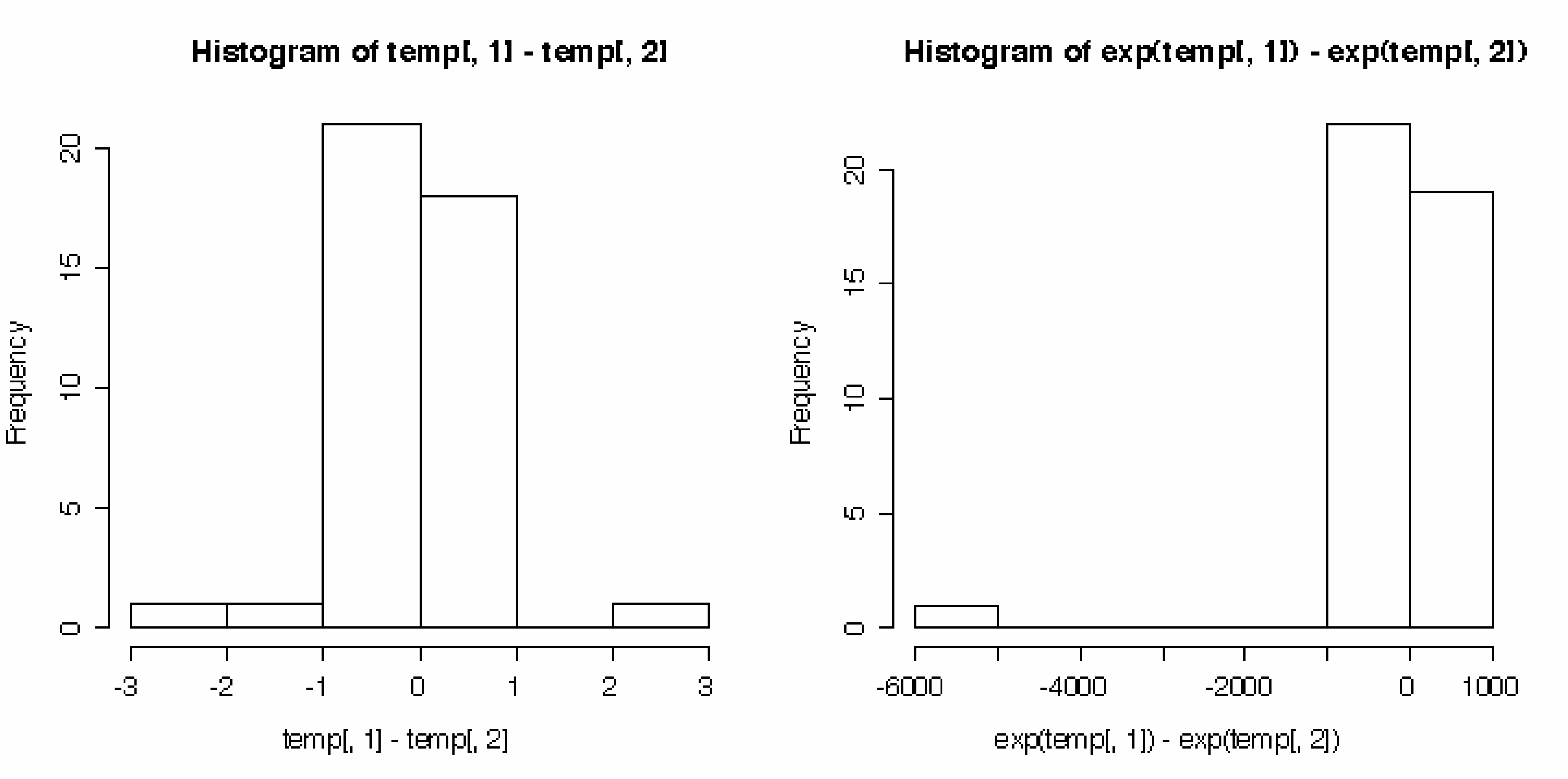
```
abidata.turnover3.rdiff <- abs(abidata.oturnover3-abidata.tturnover3)/abidata.tturnover3
abidata.turnover3.rdiff[abidata.tturnover3==0] <- 0
TE <- sum(abidata.turnover.error3==2)
TBE <- sum(abidata.turnover3.rdiff>1)
for(i in 1:100) {
        w <- i/1000
        N <- sum(abi.turnover3.waid.avwt<w)
        M <- sum(abi.turnover3.waid.avwt<w & abidata.turnover.error3==2)
        Mbig <- sum(abi.turnover3.waid.avwt<w & abidata.turnover3.rdiff>1)
        R1 <- M/TE
        R1big <- Mbig/TBE
        R2 <- 1-M/N
        C <- R1*(1-R2)
        print(c(w,N,M,Mbig,R1,R1big,R2,C))
}
```

Create random imputations for missing values of TURNOVER. Note that donors are selected fom cases with average weight > 1

```
temp <- cbind(log(abidata[abidata[,8]==3,4]+1),TreeImputer(ximp=cbind(CatMap(x=abidata[abidata[,8]==3,2],bounds=abidata.turnreg3.tile,groups=1:100),abidata[abidata[,8]==3,3]),xlabels=c("TURNREG","EMPREG"),yvalues=log(abidata.oturnover3+1),treesplits=abi.turnover3.waid.wls$splits,treenodes=abi.turnover3.waid.wls$nodes,treeweights=abi.turnover3.waid.wls$weights,random=T))
```

Plot distribution of imputation errors (log scale and raw scale)

```
hist(temp[,1]-temp[,2])
hist(exp(temp[,1])-exp(temp[,2]))
```

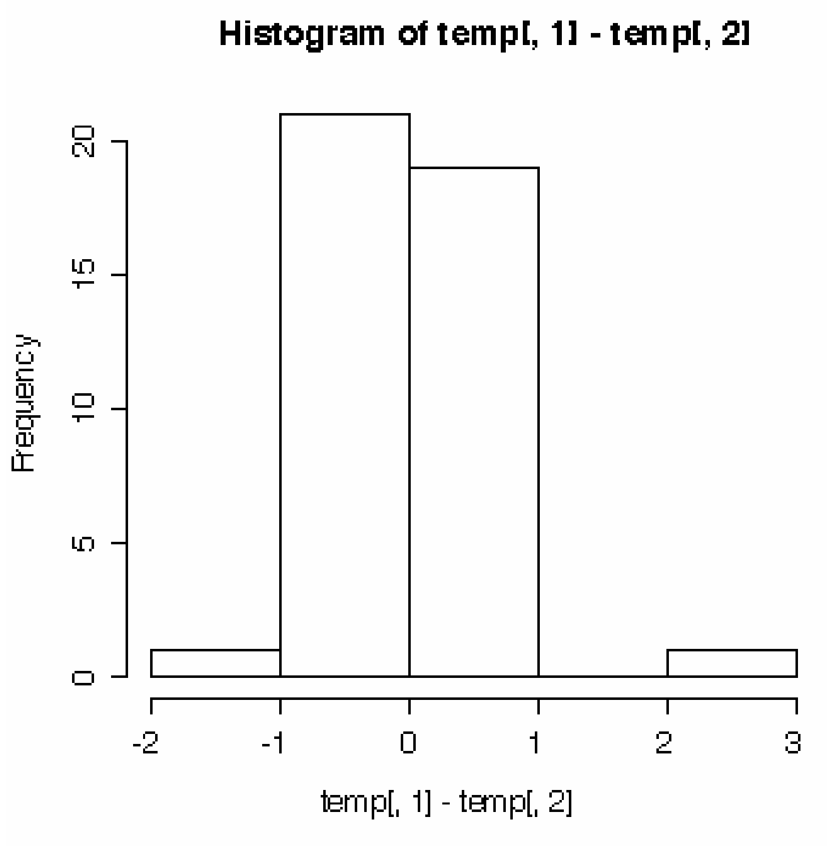


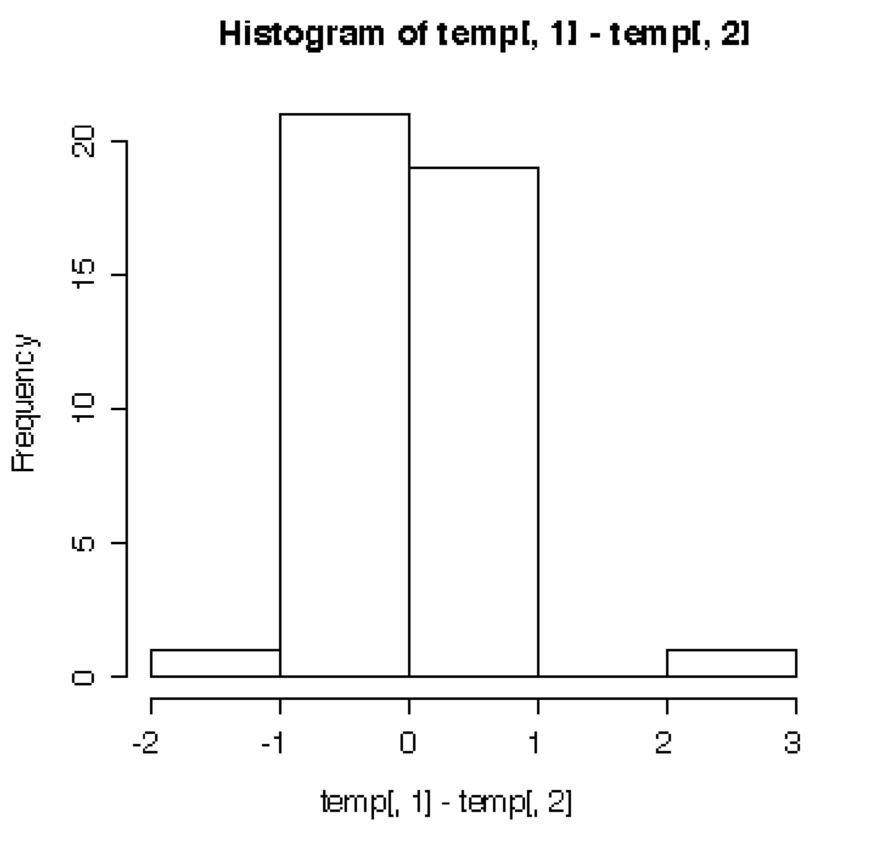
Create mean imputations for missing values of TURNOVER

```
temp <-
cbind(log(abidata[abidata[,8]==3,4]+1),TreeImputer(ximp=cbind(CatMap(x=abidata[abidata[,8]==3,2],bounds=abidata.turnreg3.tile,groups=1:100),abidata[
abidata[,8]==3,3]),xlabels=c("TURNREG","EMPREG"),yvalues=log(abidata.oturnover3+1),treesplits=abi.turnover3.waid.wls$splits,treenodes=abi.turnover
3.waid.wls$nodes,treeweights=abi.turnover3.waid.wls$weights,random=F))
```

Plot distribution of imputation errors (log scale and raw scale)

```
hist(temp[,1]-temp[,2])
hist(exp(temp[,1])-exp(temp[,2]))
```


Histogram of temp[, 1] - temp[, 2]
Frequency
temp[, 1] - temp[, 2]


Histogram of temp[, 1] - temp[, 2]
Frequency
temp[, 1] - temp[, 2]