DataZapper: A Tool for Generating Incomplete Datasets

Yingying Wen, Kevin B. Korb and Ann E. Nicholson
Bayesian Intelligence Pty Ltd,
2/21 The Parade, Clarinda, VIC 3169, Australia
ying100@yahoo.com,
{kevin.korb,ann.nicholson}@bayesian-intelligence.com

May 31, 2010

Abstract

Evaluating the relative performance of machine learners on incomplete data is important because almost all non-artificial data sets are incomplete. Machine learning evaluation, however, is often best done with artificial data, so here we introduce DataZapper, a tool for uncreating data. Given a dataset containing joint samples over variables, DataZapper will make a specified percentage of observed values disappear, replaced by an indication that the measurement failed. Since the causal mechanisms of measurement that result in failed measurements may depend in arbitrary ways upon the system under study, it is important to be able to produce incomplete data sets which allow for such arbitrary dependencies. DataZapper is the only tool that allows any kind of dependence, and any degree of dependence, in its generation of missing data. We illustrate its use in a machine learning experiment and offer it to the data mining and machine learning communities.

Keywords: Machine learning, incomplete data, data generation, data analysis, absent data, missing data, data mining, machine learning evaluation.

1 INTRODUCTION

Machine learning (ML) research aims at finding the most effective algorithms for constructing models from data. Therefore, machine learning researchers need to find the means for assessing the performance of different ML algorithms applied to common datasets representing varying domains and degrees of difficulty. Although much work in machine learning has concentrated upon data without noise, real-world data always have noise, with the most extreme form being simply the absence of a measured value. In consequence, interest has grown in finding new methods to cope with incomplete datasets and in assessing those methods (e.g., (1; 2; 3)).

Absence of data values is ubiquitous in part because there are many ways in which measurements can fail. We illustrate with the simple causal Bayesian network of Figure 1. We shall assume that joint observations of these variables come from sample surveys, but similar failures to measure can arise from any measurement technique. First, some missing values may arise simply from survey takers entirely overlooking a question, independently of what the question is about or the values of any variables. Second, the failure to measure particular variables may depend upon the values of other variables; for example, it may turn out that lawyers as a class are less inclined to reveal their incomes than people of other occupations. Third, the failure to measure may be sensitive *additionally* to the unmeasured value of the variable at issue; for example, it may be that it is primarily the *wealthy* lawyers who are reluctant to reveal their incomes. Following Rubin (4), it has become common to refer to these three mechanisms for values to be missed as, respectively, missing completely at random (MCAR), missing at random (MAR) and not missing at random (NMAR). These names are somewhat misleading, and we shall below present reasons for adopting a more descriptive nomenclature.

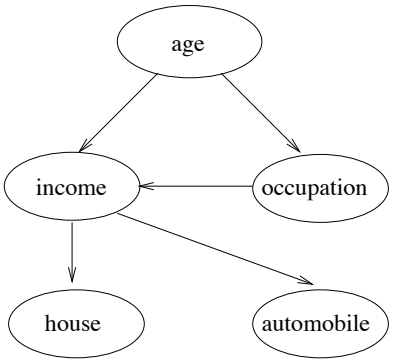

Figure 1: An example model.

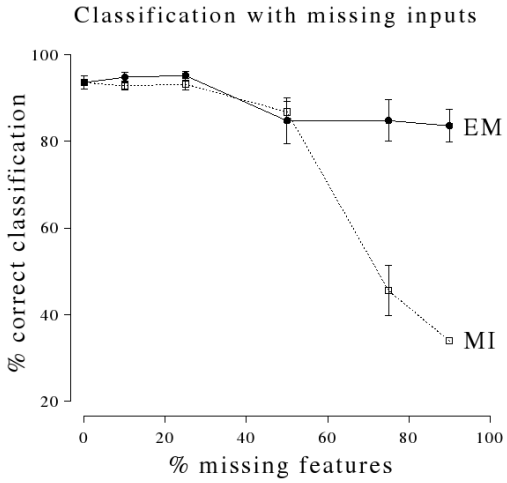

Figure 2: Example 1 of ML research on varying missing values. Classification of the Iris data from (5).

Given the prevalence of incompleteness in real data, and its variety, it is important for ML researchers to investigate how their various algorithms perform given these different types of incomplete data, even if the actual mechanism for real data to go missing is unknown.

Of course, ML researchers *do* undertake these types of experiments with different missing data. For example, Ghahramani and Jordan (5) evaluated the performance of classification with missing data using Expectation-Maximization (EM) and mean imputation (IM) (see Figure 2). Gill et al. (6) examined the performance of artificial neural networks (ANNs) and support vector machines (SVMs) on MAR data. Another example is Richman et al. (7), who compared different methods of handling missing values and presented results in terms of mean absolute error (MAE) as shown in Figure 3. They used real data with some values removed randomly, that is, MCAR data.

However, it is difficult using only real data to compare the performance of algorithms for machine learning and methods for dealing with missing values, since the nature of the real system, including the mechanisms whereby data go missing, is at issue; it is difficult or impossible to determine which algorithm has produced a model closer to reality. For machine learning research, we want to test against artificial data generated from a known system with a known mechanism causing values to go missing. This provides more flexibility with the type of missing mechanisms, the type of datasets and the degrees of dependence. Moreover, performances can then be evaluated against the true model.

Here we present DataZapper, a versatile software tool for generating artificial datasets with missing values. DataZapper renders some values in a dataset absent according to specified conditions based upon any variable and any value within that dataset; these conditions can be tuned precisely for degrees of dependence, allowing for systematic experimentation. We shall make this

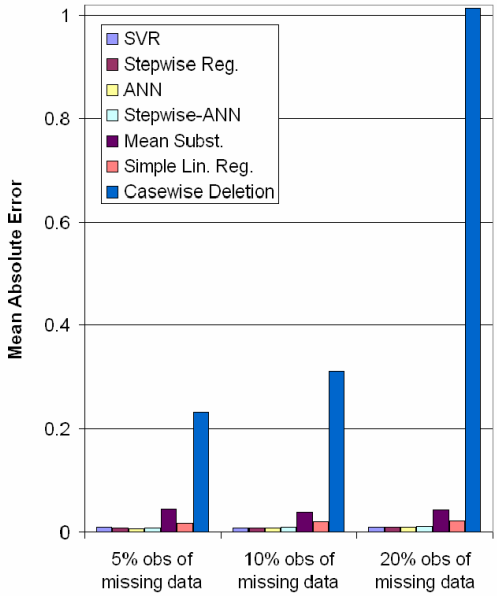

Figure 3: Example 2 of ML research on varying missing values. A bar chart illustrating the difference of variance between the original and imputed data sets from (7).

tool available to machine learning community via the Weka[1] machine learning platform. One of our motivations in producing this tool is to encourage the machine learning community to explore varieties of incompleteness beyond MCAR, which is the only kind assumed by many algorithms, such as the expectation maximization (EM) technique for replacing missing values in Weka. With a tool granting easy access to more realistic forms of incompleteness we expect more attention to them will be given.

The only previously reported tool we know of for generating incomplete data is that of Francois and Leray (8). They employ Bayesian networks (BNs) as a useful way to generate artificial data with missing values. Unfortunately, their tool is limited to MCAR and limited forms of MAR incompleteness, with no ability to produce NMAR data. As Francois and Leray point out, all of these forms of generating missing data can be useful for generic software testing, beyond machine learning research.

The structure of our paper is as follows. Section 2 describes the three absent data mechanisms and introduces our nomenclature for them. In Section 3 we present a BNF (Backus-Naur Form (9)) grammar for scripting DataZapper. In Section 4 we present the details of DataZapper, including data formats in Section 4.1 and an overview of how it works in Section 4.2. Section 5 illustrates DataZapper's use in an experimental setting.

2 ABSENT DATA MECHANISMS

A dataset is a matrix in which rows represent the cases (joint samples) and columns represent variables measured for each case. Ideally, a dataset has all the cells filled—i.e., it is a complete data set. However, most real datasets have some values unobserved—i.e., they are incomplete.

As we mentioned, Rubin (4) introduced and named three types of missing data mechanisms. We shall now motivate the adoption of new names for these. First, we prefer to talk of "absent data" rather than "missing data", for the simple but sufficient reason that "absent" has a natural nominal form, "absence", while "missing" leads to the awkward neologism "missingness". More significantly, two of Rubin's labels are clearly inadequately descriptive of the mechanisms involved:

[1] http://www.cs.waikto.ac.nz/ml/weka/

<m-statement> :: = if <antecedent> then <consequent>
<antecedent> ::= <condition>*
<condition> ::= <variable> in <range>
<variable> ::= alpha alphanum*
<range> ::= [<value>, <value>]
<value> ::= alpha alphanum* | number | symbol
<consequent> ::= (<prob>) <variable>*
<prob> ::= number

Figure 4: BNF grammar for generating absent data

Missing completely at random (MCAR): as the absence of values is independent of all variable values, including the value for this particular cell, this label is actually appropriate. Therefore, we propose calling these cases *Absent Completely At Random (ACAR)*.

Missing at random (MAR): these missing cases have arbitrary dependencies upon the values of *other* variables. In consequence, they may not even be random at all, but functionally dependent upon the values of other variables in extreme cases. Hence, we prefer calling them *Absent under Other Dependence (AOD)*.[2]

Not missing at random (NMAR): The natural way of interpreting this phrase is by negating the second kind of "missingness", which would be entirely wrong. This case is simply a generalization of AOD, allowing the absence of data to depend also upon the actual value which has failed to be measured. Hence, we have *Absent under Self and Other Dependence (ASOD)*.[3]

We submit that the most common case in real data is the case most commonly ignored, ASOD, where the values going unmeasured depend both on the values of some other variables and the absent values themselves, as in wealthy lawyers hiding their wealth.

3 SCRIPTING DATAZAPPER

The specifications for how the data should go missing are made in a simple scripting language, whose BNF grammar is shown in Figure 4. These rules are applied to a dataset file to generate a new dataset file with some observed values replaced by a token indicating absence. The basic form of a sentence is that of an "if... then..." production rule. The antecedent describes the dependencies that absence has on variables and values in the system, while the consequent lists the variables that take absent values on these conditions and with what probability. If the antecedent is empty, then the absent data generation is unconditional—i.e., the data are ACAR in so far as this production rule is concerned. If the consequent is empty, then the absence mechanism is applied to *all* variables in the dataset. When the data are AOD or ASOD, the antecedent grammar rule specifies which variable(s) the absence depends upon and for what values or value ranges. The effects of the script rules are cumulative. The result is a language in which any strength of dependence upon any set of variables can be specified, and such dependencies may be combined arbitrarily. For example, "OR" can be represented by having two different conditions.

Figure 5 shows some examples of the absent data specifications, across the range of types, together with a corresponding English description. Note that example 6 is of a mixed type, producing AOD for variable D and ASOD for variable A.

[2] We described this as *absent under dependence (AUD)* in (10).

[3] We described this as *absent under self-dependence (AUSD)* in (10).

BNF:

1. if then (20)
2. if then (30) $A\ C$
3. if C in [?] then (40) E
4. if $Gender$ in $[F]$ Age in $[10, 20]$ then (40) $Income$
5. if $Gender$ in $[F]$ $Income$ in $[70000, 90000]$ then (40) $Income$
6. if A in $[A1]$ B in $[B1]$ then (60) $A\ D$

Explanation:

1. ACAR: every variable will have 20% of its values absent
2. ACAR: each of the variables "A" and "C" will have 30% of its values absent
3. AOD: variable "E" will have 40% of its values absent when variable "C" takes the value "?", namely variable "C" is already absent
4. AOD: variable "Income" will have 40% of its values absent when "Age" is between 10 and 20 (inclusive) and "Gender" is "F".
5. ASOD: variable "Income" will have 40% of its values absent when variable "Gender" has value "F" and "Income" is between "70000" and "90000"
6. ASOD: variable "A" and "D" will both have 60% of their values absent when variable "A" has value "A1" and "B" has value "B1"

Figure 5: Examples of absent data specification in the DataZapper script language (above) with the corresponding English descriptions (below).

4 TECHNICAL DETAILS

4.1 Data Format

DataZapper accepts two data formats: a default format and Weka's (11) data format—Attribute-Relation File Format (ARFF).[4]

The default format is the data format used by the BN learning software CaMML (12), Tetrad (13) and BNT (14). (We describe how we used DataZapper for the empirical comparison of some of these methods in Section 5.) An example of complete data in the default format is shown on the left side in Table 1. The first two lines are the number of variables and the number of observations, respectively. The next line lists the names of the variables in the dataset. Columns are separated by tab. Consider again Example 2 in Figure 5 above: "if then (30) $A\ C$", the corresponding corrupted data after applying dataZapper is given on the right side in Table 1, with the absent values represented by "?" in the default data format. (The token used to represent absence can be changed from this default using a runtime parameter.)

DataZapper supports the ARFF format in order to be compatible with the Weka machine learning platform, which has become a standard toolkit for ML studies (e.g. (11)). In Table 2 we reproduce the above example in an ARFF file. Note that an additional attribute for absent values must be indicated for those variables which are consequents of a DataZapper rule.

4.2 DataZapper Operation

DataZapper processes the absent data specifications one line at a time. In processing each script command, DataZapper first parses it, validating its syntax against the BNF grammar. It then makes some values in the complete data absent, using a uniform random variate in comparison with the specified probability. DataZapper then writes the resultant incomplete dataset to an intermediate

[4] http://www.cs.waikato.ac.nz/ml/weka/arff.html

Table 1: Examples of complete data and corrupted data in DataZapper's default format.

Complete	data				Corrupted	data			
5					5				
10000					10000				
E	A	B	C	D	E	A	B	C	D
E0	A1	B1	C0	D1	E0	A1	B1	?	D1
E1	A0	B0	C1	D1	E1	A0	B0	C1	D1
E0	A1	B0	C1	D0	E0	A1	B0	C1	D0
E1	A1	B1	C0	D0	E1	?	B1	?	D0
E1	A0	B1	C1	D1	E1	A0	B1	C1	D1
...					...				

Table 2: Examples of complete data and corrupted data in ARFF format.

Complete data		Corrupted data	
5		5	
10000		10000	
@RELATION input		@RELATION input	
@ATTRIBUTE E	{E0,E1}	@ATTRIBUTE E	{E0,E1}
@ATTRIBUTE A	{A0,A1}	@ATTRIBUTE A	{A0,A1,?}
@ATTRIBUTE B	{B0,B1}	@ATTRIBUTE B	{B0,B1}
@ATTRIBUTE C	{C0,C1}	@ATTRIBUTE C	{C0,C1,?}
@ATTRIBUTE D	{D0,D1}	@ATTRIBUTE D	{D0,D1}
@DATA		@DATA	
E0,A1,B1,C0,D1,input		E0,A1,B1,?,D1,input	
E1,A0,B0,C1,D1,input		E1,A0,B0,C1,D1,input	
E0,A1,B0,C1,D0,input		E0,A1,B0,C1,D0,input	
E1,A1,B1,C0,D0,input		E1,?,B1,?,D0,input	
E1,A0,B1,C1,D1,input		E1,A0,B1,C1,D1,input	
...		...	

file. DataZapper emulates parallelism by generating intermediate output files for each such line and, in the end, merging the intermediate files into a final output file. In the merging process absent values dominate; that is, a value ends up missing if it is missing in *any* intermediate file. DataZapper finishes by generating a data report on the final dataset, comparing the proportions of absent values with the original dataset.

We will now look at in more detail of the main functions. The steps with examples will be presented in the following sections.

Functions included in *main*():

```
parser()
corruptedDataGenerator()
mergeData()
dataReport()
```

function *main*(`script file, complete data`**)**

```
for each condition line in script file
    if parser(condition line, absentInfo[5]) > 0
        corruptedDataGenerator(complete data, corrupted data,
                               absentInfo, flag)
```

[5] An array which stores information about absence

```
      dataReport(complete data, corrupted data, &absentInfo, flag)
    endif
  endfor

  mergeData(corrupted data 1, corrupted data 2)
  dataReport(complete data, corrupted data, &absentInfo, flag)
end
```

4.2.1 The parser

The parser function checks the grammar of BNF. It returns 0 for lines which are not proper commands, such as comment line, and move on to the next command, 1 for data ACAR, 2 for data AOD, 3 for data ASOD, and 4 for mix of data AOD and ASOD. The program exits if there any error in the command.

Input: a line from script file represent a condition
address of $absentInfo$ passed from main function

Output: value 0 to 4

Functions included:
$tokenizer()$

```
function parser(commandLine, &absentInfo)
  if the first letter in a line is not a proper start of command
    return 0

  % Read tokens and save them in an array tokens[ ].
  tokenizer(commandLine, tokens)

  % If <condition> is empty, then the data is ACAR.
  if <condition> is empty
    if absent variable names are not given
      % The absence is for all variables
      return 1
    else
      if variable names are valid
        store the names
        return 1
      else
        print error message and exit
      endif
    endif
  endif

  % If <condition> is not empty, then the data is AOD, ASOD or mix
  % of AOD and ASOD.
  else
    for each dependent variable in <condition>
      if the name is valid
        if the value/range of the variable is valid
          store the names and value/range
        else print error message and exit
        endif
```

```
        else printf error message and exit
        endif
    endfor

    if the proportion of absence is valid
        store the proportion
    else print error message and exit
    endif

    % Check absent variable names
    for each absent variable
        if the name is valid
            save it
        else print error message and exit
        endif
    endfor

    % Check absent type and return different value.
    if absent type == AOD
        return 2
    elseif absent type == ASOD
        return 3
    % mix of AOD and ASOD
    else
        return 4
  endif
end
```

function *tokenizer*(commandLine, tokens[ ]). In general, it separates tokens by space. However, "[" and "]" are considered as tokens. Tokens between them are separated by ",". For example, the last BNF example in Section 3: "if A in $[A1]$ B in $[B1]$ then (60) A D". The outcome of function $tokenizer()$ is a list of tokens: A, in, [, A1,], B, in, [, $B1$,], then, 60, A, D.

4.2.2 The Corrupted Data Generator

This is the key processing step that renders some values in the input data absent. The proportion of the absence is applied to each selected target variable, evenly distributed over all the relevant observations for that variable – that is, those observations which satisfy the dependency condition.

Input: complete data
corrupted data filename
address of $absentInfo$
$flag$ (value $1-4$) from $parser()$

Output: corrupted data

function *corruptedDataGenerator*(complete data, corrupted data filename, $absentInfo$, $flag$)

```
  for each record in complete data
    get all the values of the variables
```

Table 3: Examples of two corrupted datasets.

corrupted	data 1				corrupted	data 2			
5					5				
10000					10000				
E	A	B	C	D	E	A	B	C	D
E0	A1	B1	?	D1	E0	?	B1	C0	?
E1	A0	B0	C1	D1	E1	A0	B0	C1	D1
E0	A1	B0	C1	D0	E0	A1	B0	C1	D0
E1	?	B1	?	D0	E1	?	B1	C0	?
E1	A0	B1	C1	D1	E1	A0	B1	C1	D1
...					...				

```
% For data ACAR.
if flag == 1
   for each value in the record
      if the value is for one of the absent variables
         generate a random value
         if random value < absent proportion
            change the value to ``?''
         else
            keep the value unchanged
         endif
      else
         keep the value unchanged
      endif
   endfor

% For data AOD, ASOD or mix.
else
   for each value in the record
      if the value is for one of the absent variables
         if the record satisfies the <condition>
            generate a random value
            if random value < absent proportion
               change the value to ``?''
            else
               keep the value unchanged
            endif
         else
            keep the value unchanged
         endif
      endif
   endfor
endif
output the record to corrupted data file
endfor
end
```

4.2.3 Merging Data Files

In this processing step, DataZapper merges multiple corrupted datasets with the same variables and the same number of observations. The datasets having a common source, the only differences

Table 4: Merged data from the examples in Table 3.

corrupted	data			
5				
10000				
E	A	B	C	D
E0	?	B1	?	?
E1	A0	B0	C1	D1
E0	A1	B0	C1	D0
E1	?	B1	?	?
E1	A0	B1	C1	D1
...				

Content of script file:
if then (20) E A
if (C) in [C0] D in [D1] then (20) D
if A in [A1] then (30) B E
Percent of absent values:

	A	B	C	D	E
Final:	20.00%	12.64%	0.00%	5.14%	30.35%
Original:	0.00%	0.00%	0.00%	0.00%	0.00%

Overall:
6813 values are absent, 13.63% of all values.
5201 cases contain absent values, 52.01% of 10,000 total cases.

Figure 6: Example of DataZapper's absent data report.

between them are those required by processing distinct script file commands. The merged data is a kind of union of the corrupted datasets, with the absence of a value in any cell forcing its absence in the final output. If there are many script commands being executed, or if the initial input file itself contained incomplete data, then the final dataset may contain less information (more absent values) than anticipated.

For example, consider again Examples 2 and 6 from Figure 5 for specifying absent data. Table 3 displays some examples from the two corrupted datasets respectively, while Table 4 shows the same examples in the final merged corrupted dataset.

4.2.4 Data report

DataZapper presents a statistical summary of the incompleteness of the final dataset. Figure 6 gives an example data report. This report can be used to fine tune the scripting rules in the event that the overall sparseness of the data is unexpectedly high, possibly due to the cumulative effect of multiple rules on some variables.

5 APPLICATION

We now describe an application of DataZapper in generating incomplete data for use in some of our machine learning research. The specific application is an empirical comparison of the performance of causal discovery algorithms in finding the causal Bayesian network (a kind of directed acyclic graph, or DAG) which has generated some observational data. The algorithms under test were K2 (15), GES (Greedy Equivalence Search) (16), and the PC algorithm from Tetrad (13). The first

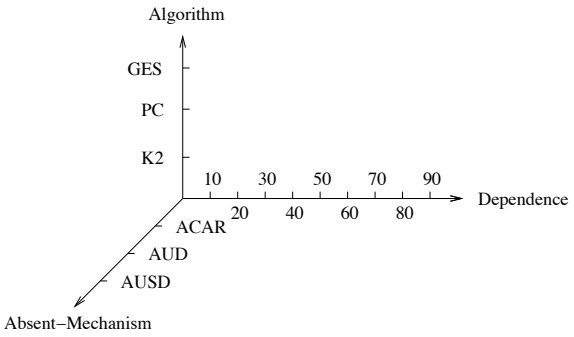

Figure 7: Three dimensional experimental design.

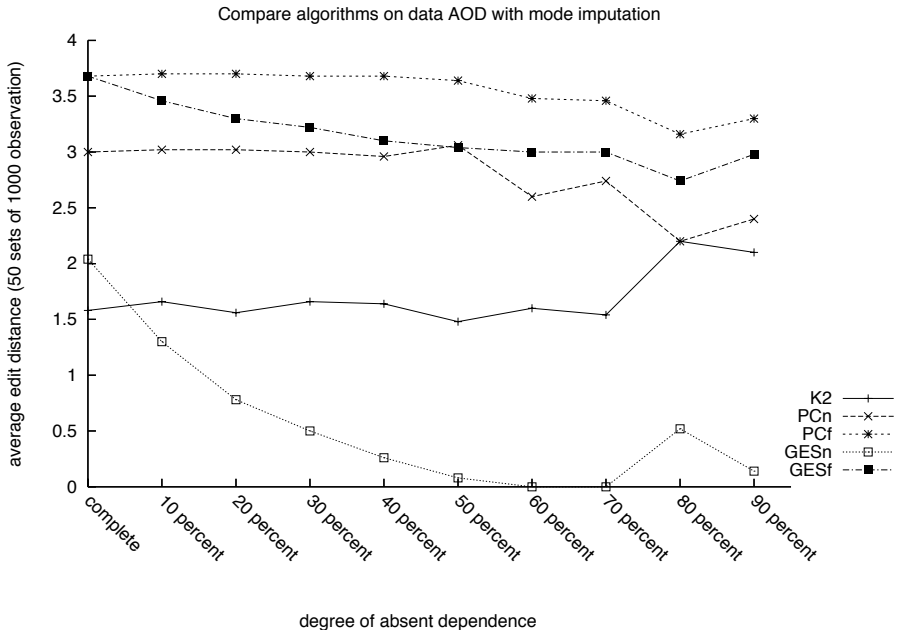

Figure 8: Example experimental results using DataZapper: comparison of 3 causal discovery algorithms, on data generate with AOD absence mechanism, varying the degree of data completeness.

algorithm, K2, returns a single DAG which fits the data best.[6] The other two algorithms return an equivalence class of DAGs (a pattern); that is, a set of DAGs which all have equal maximum-likelihood scores based upon any given set of observational data (18). In effect, these algorithms are asserting that all the DAGs within the pattern are equally likely to be the source of the observed data. In assessing such results, therefore, we use a pattern-to-DAG conversion algorithm (19) algorithm which returns two DAGs: that nearest to the original causal Bayesian network in structure (as measured in edit distance) and that farthest from the original network. This provides a range of performance for assessing such algorithms (assuming that the data are artificial, of course, since otherwise the original network is unknown).

The experiment we ran was a three dimensional evaluation: we varied the algorithm, the proportion of absence and the absent data mechanism as shown in Figure 7.

We used 50 sets of complete data generated from a known Bayesian network. We then applied DataZapper to produce 3×9 incomplete datasets for each complete dataset, given the three absence mechanisms and 9 steps of proportion of absence, as shown in Figure 7. We then designed comparison experiments for different combinations of these experimental parameters.

For example, one experiment involved selecting the absence mechanism and then comparing the performance of the causal discovery algorithms given varying proportion of absence. The results

[6] We have supplemented K2 by utilizing a Minimum Weighted Spanning Tree algorithm of (17) as a preprocessing step to produce the total ordering of variables that K2 demands.

of this particular experiment are shown in Figure 8. Here the evaluation measure we used is the edit distance of the learned BN to the true model—Figure 1, averaged over the 50 datasets. For the PC and the GES algorithms we report two results, one based on the DAG within the pattern returned that is *nearest* to the true model (PCn and GESn), another for the DAG within the pattern that is *farthest* from the true model (PCf and GESf). In this experiment we used one of the simplest methods for handling absent values, namely modal imputation (i.e., replacing each absence token with the modal value for that variable). Results are available for all ACAR, AOD and ASOD. Only AOD is used as an example. Figure 8 shows that under these circumstances the performances for PC and GES improve as the data quality improves, while K2 appears to be stuck. Overall, GESn shows the best performance.

6 CONCLUSION

DataZapper is a powerful and flexible tool for incomplete data generation, developed specifically for use in research comparing machine learning algorithms. DataZapper allows researchers to specify both the amount of absent data and the nature of the dependencies in generating the absent data, using simple conditional rules. Multiple conditions of absence can be described for each variable and for multiple variables, which will be applied cumulatively by DataZapper to the input dataset, which itself may be either complete or already corrupt. DataZapper is the only tool which can generate incomplete data for all types of absent data mechanisms (ACAR, AOD or ASOD) and with any degree of dependence. We offer it through Weka in the hopes that methods of coping with more interesting and difficult varieties of incomplete data may be investigated by the machine learning community.

References

[1] Onisko, A., Druzdzel, M.J., Wasyluk, H.: An experimental comparison of methods for handling incomplete data in learning parameters of bayesian networks. In: Proceedings of the IIS'2002 Symposium on Intelligent Information Systems, Physica-Verlag (2002) 351–360

[2] Twala, B., Cartwright, M., Shepperd, M.J.: Comparison of various methods for handling incomplete data in software engineering databases. In: 2005 International Symposium on Empirical Software Engineering, Noosa Heads, Australia (November 2005) 105–114

[3] Twala, B.E.T.H., Jones, M.C., Hand, D.J.: Good methods for coping with missing data in decision trees. Pattern Recogn. Lett. **29**(7) (2008) 950–956

[4] Rubin, D.B.: Inference and missing data. Biometrika **63**(3) (Dec. 1976) 581–592

[5] Ghahramani, Z., Jordan, M.I.: Learning from incomplete data. Technical Report AIM-1509, Artificial Intelligence laboraory and Center for Biological and Computational Learning, Department of Brain and Cognitive Sciences, Massachusetts Institute of Technology (1994)

[6] Gill, M.K., Asefa, T., Kaheil, Y., McKee, M.: Effect of missing data on performance of learning algorithms for hydrologic predictions: Implications to an imputation technique. Water Resources Research **43**(W07416,) (2007)

[7] Richman, M.B., Trafalis, T.B., Adrianto, I.: Multiple imputation through machine learning algorithms. In: Artificial Intelligence and Climate Applications (Joint between 5th Conference on Applications of Artificial Intelligence in the Environmental Sciences and 19th Conference on Climate Variability and Change). (January 2007)

[8] Francois, O., Leray, P.: Generation of incomplete test-data using bayesian networks. In: Proceedings of International Joint Conference on Neural Networks, Orlando, Florida, USA (2007) 12–17

[9] Backus, J., Naur, P.: Revised report on the algorithmic language algol 60. Communications of the ACM **3**(5) (May 1960) 299–314

[10] Wen, Y., Korb, K.B., Nicholson, A.E.: Datazapper: Generating incomplete datasets. In: Proceedings of the First International Conference on Agents and Artificial Intelligence (ICAART 2009). (2009) 69–76

[11] Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. 2 edn. Morgan Kaufmann, San Francisco, CA, USA (2005)

[12] Wallace, C., Korb, K.B., Dai, H.: Causal discovery via MML. In: Proceedings of the Thirteenth International Conference on Machine Learning, Morgan Kaufmann (1996) 516–524

[13] Spirtes, P., Glymour, C., Scheines, R.: Causation, Prediction, and Search. 2 edn. Cambridge, MA:MIT Press (2000)

[14] Leray, P., Francois, O.: BNT structure learning package: documentation and experiment s. Technical Report Laboratoire PSI - INSA Rouen-FRE CNRS 2645, Universit et INSA de Rouen (2004)

[15] Cooper, G.F., Herskovits, E.: A Bayesian method for constructing Bayesian belief networks from databases. In: In Proceedings of the Conference on Uncertainty in AI, San Mateo, CA: Morgan Kaufmann (1991) 86–94

[16] Meek, C.: Graphical Models: Selecting Causal and Statistical Models. PhD thesis, Carnegie Mellon University (1997)

[17] Chow, C., Liu, C.: Approximating discrete probability distributions with dependence trees. IEEE Transactions on Information Theory **14** (1968) 462–467

[18] Chickering, D.M.: A tranformational characterization of equivalent Bayesian network structures. In Besnard, P., Hanks, S., eds.: UAI95, San Francisco (1995) 87–98

[19] Wen, Y., Korb, K.B.: A heuristic algorithm for pattern-to-dag conversion. In: Proceedings of IASTED International Conference on Artificial Intelligence and Applications. (2007) 428–433