Learning Random Numbers: A Matlab Anomaly

Petr Savicky[1], Marko Robnik-Šikonja[2]

[1] Institute of Computer Science,
Academy of Sciences of Czech Republic
Pod Vodarenskou Vezi 2, 182 07 Praha 8, Czech Republik
savicky@cs.cas.cz

[2] University of Ljubljana,
Faculty of Computer and Information Science,
Tržaška 25, 1001 Ljubljana, Slovenia
Marko.Robnik@fri.uni-lj.si

Abstract

We describe how dependencies between random numbers generated with some popular pseudorandom number generators can be detected using general purpose machine learning techniques. This is a novel approach, since usually, pseudo-random number generators are evaluated using tests specifically designed for this purpose. Such specific tests are more sensitive. Hence, detecting the dependence using machine learning methods implies that the dependence is indeed very strong. The most important example of a generator, where dependencies may easily be found using our approach, is Matlab's function rand if the method state is used. This method was the default in Matlab versions between 5 (1995) and 7.3 (2006b), i.e. for more than 10 years. In order to evaluate the strength of the dependence in it, we used the same machine learning tools to detect dependencies in some other random number generators, which are known to be bad or insufficient for large simulations: the infamous RANDU, ANSIC, the oldest generator in C library, Minimal Standard generator, suggested by Park and Miller in 1988, and the rand function in Microsoft C compiler.

Keywords: random numbers, machine learning, classification, attribute evaluation, regression

1 Introduction

Random numbers are part of many machine learning and data mining techniques, for example, we randomly select instances and features, test our algorithms with many variants of randomly generated data sets, etc. Many methods heavily rely on random

numbers, e.g., many ensemble methods, variants of Hidden Markov Models, random projections, etc. Some have the word random even in their name, for example Random Forests. The purpose of this paper is to bring the importance of using good random number generators into attention of machine learning and data mining communities. Users and developers are usually not concerned about the random numbers, when developing and using machine learning tools, they simply use the generator supplied within the favorite software package, library, or tool. We show in this paper, that this approach may be dangerous, even if the considered software package is such a widely used and renowned tool for technical computing as Matlab.

A pseudorandom number generator is an algorithm that is initialized with a seed and generates a deterministic sequence of numbers depending on the seed, which mimics a sequence of independent and identically distributed (i.i.d.) numbers chosen uniformly from $[0, 1]$. In practice, pseudo-random numbers are important for simulations and for certain algorithms, e.g., Monte Carlo methods. A good generator produces numbers that are not distinguishable from truly random numbers in a limited computation time, if the seed is not known. It is sometimes tolerated, that the numbers are distinguishable from truly random numbers using functions, which cannot appear as a part of the application domain, where the generator is used. This is, in particular, true for Mersenne Twister (Matsumoto and Nishimura, 1998), a popular recent generator, which may easily be predicted, if the arithmetic modulo 2 is used and several hundreds of last numbers are known.

The history of scientific computing is full of bad pseudorandom number generators, producing sequences with strong and easy to detect dependencies (Park and Miller, 1988). Possibly, the most infamous generator of this kind was RANDU, introduced in 1970 (more details on this generator can be be found in Wikipedia (`http://en.wikipedia.org/wiki/Pseudo-random_number_generator`). One of the reasons for the existence and awareness of the bad generators is the rapidly increasing power of the computers. A generator that is good for generating sequences of certain length may become inappropriate, when faster computers require longer sequences for larger simulations. Another reason is that the software developers sometimes underestimate the necessity to have a reliable generator and are satisfied with a sequence that looks chaotic enough. Moreover, the users of the generators frequently think that some dependencies should necessarily be tolerated, since they are using pseudorandom numbers. This is not true. It is consistent with the current state of knowledge that good generators do exist, such that no dependence is detectable in them in a tractable amount of time. See, for example (Goldreich, 1998; 2000).

As Press et al. (1988) say: "If all scientific papers whose results are in doubt because of bad rand()s[1] were to disappear from library shelves, there would be a gap on each shelf about as big as your fist". The tolerance of existing dependencies in generators may lead to severe consequences, since large simulations are sensitive even to small discrepancies from the uniform distribution on sequences. Therefore, if the results of the simulation are to be trusted, it is absolutely necessary, that no known dependency exists in the generator.

An example of a generator with intolerable dependency is the default behavior of

[1] rand() is a system supplied random number generator.

function rand in Matlab versions between 5 (1995) and 7.3 (2006b), see (Savicky, 2006). The main purpose of the current paper is to demonstrate the strength of the dependence in this generator by showing that the dependence may be detected not only by specialized tests, but also using general machine learning tools. Moreover, the dependence appears to be stronger than the dependencies in some other generators, which are known to be insufficient.

General purpose methods from machine learning are less sensitive to dependencies in random numbers than the specialized tests for this purpose and only relatively strong dependencies can be detected. However, the consequences are severe for machine learning and data mining results. Generators with this kind of dependencies cannot be used even for approximate calculations in machine learning. If a dependence may be detected even by general machine learning tools, then analytic results obtained using such generator are in doubt. The statistical estimates obtained from these results may be highly biased due to the dependencies in the used random numbers. In the current paper, we show that the above mentioned generator in Matlab belongs to this category of generators.

The level of dependency can be measured in terms of the length of the sequence that is needed to distinguish the output of the generator from the i.i.d. sequence of uniformly distributed numbers from $[0, 1]$ on a specific level of statistical significance. The (L'Ecuyer et al., 2002) is an example of an experimental study of random number generators from this viewpoint, where the severity of the dependencies is measured using specialized tests designed specifically for testing pseudorandom number generators. In particular, it is shown that linear congruential generators should not be used beyond the square root of their period (which for 2^{32} is $2^{16} = 65536$).

We tried different machine learning techniques to detect dependencies between random numbers: classification, attribute evaluation and regression. We tested the following random number generators

- the default behavior of the function rand in Matlab versions between 5 (1995) and 7.3 (2006b). Alternatively, this generator may be invoked by seeding rand with the command rand('state',seed). This way of invoking the generator is available also in the current releases of Matlab. We call this generator as method state of rand and abbreviate it rand_state,
- Minimal Standard generator (MINSTD) (Park and Miller, 1988),
- ANSIC generator, which is the generator from the oldest versions of C library,
- rand() function in Microsoft C compiler, which we call MS_rand.
- RANDU generator, which is the well-know bad generator introduced in 1970. Since even seeds may lead to shorter periods, we used only odd seeds in our experiments, unless otherwise stated explicitly.

Besides the first one, these generators are pure linear congruential generators. The spectral test of some of them may be found at `http://random.mat.sbg.ac.at`. These four generators are known to be insufficient for current simulation experiments,

although MINSTD is still sometimes used. We used these generators for comparison of the strength of the dependence in rand_state.

For our study, we needed also a reliable random number generator. We used MRG32k5a from (L'Ecuyer, 1999). For comparison, we used also WELL19937a, which is an improvement of Mersenne Twister with the same period, see (Panneton et al., 2006). The differences between MRG32k5a and WELL19937a were insignificant in all our experiments.

In Section 2 we define different learning problems used in our analysis. In Section 3 we describe the analytical methods, which we used. The experimental results are presented in Section 4. Section 5 contains conclusions and recommendations.

1.1 Related Work

There are many studies about pseudorandom numbers and their generation for non specialists e.g., (Park and Miller, 1988). Comprehensive information on pseudorandom number generation may be found e.g. in (Knuth, 1998; L'Ecuyer, 2006). A large collection of specialised test of random number generators is described in (L'Ecuyer and Simard, 2006), where the dependency in rand_state is also detected.

We are not aware of any work giving attention to random numbers in the context of machine learning. A recent study concerning insufficient quality of another widely used generator is (Wichman and Hill, 2006).

2 Learning Problems

We formulate a problem of detecting dependencies in pseudorandom number generators as a machine learning problem of predicting the next number. Our notion of an instance is a sequence of 32 consecutive random numbers Z_i, $i = 1,\ldots,32$ from [0,1] interval generated with the tested random number generator. The length 32 of the considered subsequences is chosen as a number, which is slightly larger than 28, which is necessary to detect the dependence in rand_state. We take the first 31 numbers $Z_1,\ldots,Z_{31}$ as independent variables (attributes) and Z_{32} as response variable (prediction value). We consider several variants of the problem, which differ in the way, how Z_{32} is used. Namely,

1. **Regression**. In this problem, we try to predict the response variable Z_{32} directly from the independent variables.

2. **Feature evaluation**. We use feature evaluation algorithm to measure the importance of the features for prediction of Z_{32} as response variable in regressional problem.

3. **Thresholding**. This is a classification problem, where the response variable Z_{32} is transformed to a class variable by comparing to the threshold 0.5. Class 1 corresponds to $Z_{32} \geq 0.5$, class 0 to the opposite.

4. **Two values**. This is a classification problem, where only some of the generated instances are used. Class 0 corresponds to instances, where $|Z_{32} - 0.25| \leq 0.025$

and class 1 corresponds to instances, where $|Z_{32} - 0.75| \leq 0.025$. Instances not satisfying some of these two conditions are discarded.

In the two classification problems we considered the accuracy of random forests predictor and some other classifiers. The regression formulation was used for attribute evaluation using RReliefF algorithm (Robnik-Šikonja and Kononenko, 2003) and learning of regression models.

The data set is created by generating a sequence of consecutive numbers from the investigated generator and splitting them into disjoint subsequences of length 32 as discussed above. It is necessary to point out that in the presence of dependencies, the sample obtained in this way does not consist of independent realizations from the same distribution. The reason is that there are not only dependencies inside the considered subsequences, but also among the numbers in consecutive subsequences. Since the machine learning techniques, which we use, are not sensitive to the order of the instances in the sample, this effect is negligible.

3 Learning methods

For a good generator, the three learning problems described in Section 2 do not allow prediction of the response variable. Hence, when we construct a classifier using any method, its accuracy on a test set should not be distinguishable from accuracy of a random guess. We will see in Section 4 that for some of the investigated generators, the accuracy is better than random guess on a high level of statistical significance.

Alternatively, for a good generator, none of the predictors in the three learning problems has a measurable dependence on the response variable. Hence, if we apply any feature selection method, none of the features should receive a score different from the score of a completely irrelevant attribute. In Section 4, we show examples of generators, for which the scores of some of the features differ significantly from a score of a noisy attribute. This is another demonstration of the bad quality of the generators.

In the following subsections, we describe the machine learning tools used for classifier construction and feature evaluation in more detail. We have used machine learning algorithms in Weka (Witten and Frank, 1999), R Environment for Statistical Computing (R Development Core Team, 2006) and the Core learning system available from `http://lkm.fri.uni-lj.si/rmarko/software/`.

3.1 Constructing classifiers and regressors

For the two classification problems described in Section 2, we initially tried several classification methods from Weka (decision trees, boosting, random forests, SVM, neural networks, ripper). As several of them were competitive, for the reason of computational efficiency we decided to make more detailed experiments only with random forests implementation in R.

Random forests (RF) (Breiman, 2001) method is an ensemble method, which builds a large number of decision/regression trees using bootstrap sample of data and in each

tree node selects the best split from randomly selected subset of attributes. In this way we get fine-grained partitioning of the problem space which has good generalization due to combination of many base tree models.

We used similar approach also for regression problem (described in Section 2), and initially tried several methods from Weka and R (regression model trees, locally weighted regression, linear models, nearest neighbor, neural networks). Again several of them were competitive and we selected the computationally most efficient ones implemented in Core. These were regression trees using different models in the leaves: linear models, locally weighted regression, or nearest neighbor method with Gaussian kernel.

3.2 Attribute Evaluation

We try to determine which attributes the response variable depends on. There are two machine learning approaches to attribute evaluation. The first is based on the impurity functions (measuring purity of prediction value distribution after the split on selected attribute value). This approach assumes the conditional independence of the attributes upon the target variable and is therefore inappropriate for problems which possibly involve much feature interaction. The second kind of heuristics are context dependent using the distance and can correctly estimate the quality of attributes in the problems with strong dependencies between attributes. The most successful representatives are algorithms ReliefF in classification and RReliefF in regression (Robnik-Šikonja and Kononenko, 2003). A key idea of these algorithms is to estimate the quality of the attributes according to how well their values distinguish between instances that are near to each other. Values of a good attribute should separate instances with different prediction values and not separate instances with close prediction values. For a chosen sample of instances ReliefF and RReliefF find their nearest neighbors and update the quality estimates of the attributes depending how their values separate instances: they get a positive update for separating instances with different prediction values, and a negative update for separating instances with close prediction values. The way these algorithms work enable them to detect dependencies for which the nearest neighbor paradigm holds: close instances should have close prediction values. We have used RReliefF in our study using the default set of parameters.

4 Experimental results

4.1 Accuracy of Random Forests classifiers

The dependencies in some of the investigated generators are so strong that a prediction of the response in the "thresholding" and "two-values" problems may be done using the Random Forests predictor. Prediction accuracy depends on the size of the training sample. We made some preliminary tests in order to determine the sample size appropriate for the main experiment. Already the training sample of size 10^4 is sufficient to obtain a classifier with the probability of correct prediction about 0.6 for rand_state. This is sufficient to reject the null hypothesis that the generator produces i.i.d. num-

	median of accuracy		p-value of KS test	
generator	thresholding	two values	thresholding	two values
ANSIC	0.49961	0.50046	0.1706	0.4400
MINSTD	0.49952	0.5005	0.1561	0.0933
MRG32k5a	0.50034	0.49986	0.7578	0.132
MS_rand	0.50021	0.4979	0.8954	$2.2 \cdot 10^{-4}$
rand_state	0.63928	0.68625	$< 2.2 \cdot 10^{-16}$	$< 2.2 \cdot 10^{-16}$
RANDU	0.50234	0.51537	$6.3 \cdot 10^{-5}$	$< 2.2 \cdot 10^{-16}$
WELL19937a	0.50014	0.49997	0.5664	0.7808

Figure 1: Accuracy of the constructed classifiers for the two classification problems

bers from uniform distribution on $[0,1]$. Under the null hypothesis, the probability of correct prediction should be 0.5. However, for other generators, the obtained accuracy for this training sample size is not sufficient for a similar conclusion. Hence, the main experiment uses sample size $5 \cdot 10^4$, for which at least the dependence in RANDU is detected besides rand_state.

For each tested generator and both learning problems "thresholding" and "two-values", we constructed 20 samples of size $s = 5 \cdot 10^4$ for training and 20 samples of the same size for testing accuracy. Hence, for each generator, we obtain 20 estimates of the prediction accuracy for both used problems, which will be denoted $a_{i,j}$, where $i = 1, \ldots, 20$ is the index of the pair of training and testing sample and $j = 1,2$ is 1 for "thresholding" and 2 for "two-values" problem. Under the null hypothesis, the prediction behaves exactly as random guessing, since the predicted value and the true response are independent. Hence, the number of correct predictions C is a random variable chosen from the binomial distribution with $n = s$ trials and probability $p = 0.5$ of success in each trial. Since n is quite large, the distribution of the prediction accuracy C/n is approximately normal with mean 0.5 and variance $1/(4n)$. The following table contains for each tested generator and both learning problems indexed by $j = 1,2$ the median of the numbers $a_{i,j}$ for $i = 1, \ldots, 20$ and the p-value of Kolmogorov-Smirnov (KS) test of the hypothesis that the numbers $a_{i,j}$ are chosen from the normal distribution with the parameters described above.

Using the p-values in Figure 1, we can make the following conclusions. For ANSIC, MINSTD, the deviation of the accuracies from random guessing is not significant. For MS_rand, the difference is significant for "two values" on a moderate level. For RANDU and rand_state, the accuracy for both problems allow to reject the randomness of the generator on a strong level of statistical significance. The "two values" technique is more sensitive in detection of dependencies in the tested generators than the "thresholding".

4.2 Detecting dependencies using RReliefF

The most sensitive method for detecting dependencies in the investigated generators was RReliefF algorithm for feature evaluation. For this experiment, we used the re-

sample	generator g				
s	ANSIC	MINSTD	MS_rand	rand_state	RANDU
$5 \cdot 10^3$	$1.4 \cdot 10^{-1}$	$2.6 \cdot 10^{-1}$	$6.9 \cdot 10^{-1}$	$6.2 \cdot 10^{-14}$	$1.2 \cdot 10^{-2}$
$1 \cdot 10^4$	$2.3 \cdot 10^{-5}$	$9.9 \cdot 10^{-1}$	$1.4 \cdot 10^{-3}$	$6.2 \cdot 10^{-14}$	$6.2 \cdot 10^{-14}$
$2 \cdot 10^4$	$6.2 \cdot 10^{-14}$	$9.9 \cdot 10^{-1}$	$6.2 \cdot 10^{-14}$	$6.2 \cdot 10^{-14}$	$6.2 \cdot 10^{-14}$
$5 \cdot 10^4$	$6.2 \cdot 10^{-14}$	$1.0 \cdot 10^{-4}$	$6.2 \cdot 10^{-14}$	$6.2 \cdot 10^{-14}$	$6.2 \cdot 10^{-14}$
$1 \cdot 10^5$	$6.2 \cdot 10^{-14}$	$6.2 \cdot 10^{-14}$	$6.2 \cdot 10^{-14}$	$6.2 \cdot 10^{-14}$	$6.2 \cdot 10^{-14}$

Figure 2: The p-values of the KS-test comparing RReliefF scores for different sample sizes and generators.

gression formulation of the problem. RReliefF algorithm determines a score for each predictor in the data, which estimates the level of the dependence between the predictor and the response variable. For bad generators, some of these scores are larger in absolute value than what may be expected for i.i.d. data from the uniform distribution, since there are dependencies between the predictors and the response variable. Several samples were constructed for each generator and the scores were calculated for each sample. In order to interpret the results, we used statistical hypothesis testing, where the null hypothesis is that the tested sample consists of i.i.d. numbers chosen from the uniform distribution on $[0,1]$. Since the exact distribution of the scores under the null hypothesis is not known, we compared the scores from a tested generator g with scores obtained under the same conditions from a reliable generator (MRG32k5a), which we denote g_0.

Sensitivity of RReliefF algorithm increases with the number of used instances, i.e. the sample size s. We used 5 different sizes s shown in the first column of the table in Figure 2 in order to show that the dependencies in different generators require different sample size to be detected. This allows comparison of the strength of the detected dependencies in different generators. For each used sample size s and each tested generator g, we constructed 24 samples of size s indexed by $i = 1, \ldots, 24$. For sample i, we obtained the score $v_{i,j}$ for each predictor $j = 1, \ldots, 31$. For simplicity, the vector of scores for sample i was transformed into a single statistics $d_i = \max_j v_{i,j} - \min_j v_{i,j}$. As a result, we obtained the numbers d_i, $i = 1, \ldots, 24$ for the generator g. Let d'_i, $i = 1, \ldots, 24$ be the numbers obtained using the same procedure for generator g_0. Under the null hypothesis on g and g_0, we have two i.i.d. samples of size 24 chosen from the same unknown distribution. In order to test this hypothesis, we used two-sided KS-test. The p-values for each s and g are presented in the table in Figure 2.

The table shows that the dependence in rand_state is detected for all tested sample sizes including the smallest one. Note that the p-value $6.2 \cdot 10^{-14}$ corresponds to the situation, when the ranges of the values in the two tested samples are disjoint. The dependence in RANDU is also detected for all sample sizes s. For the remaining generators, this is not true. Detecting the dependence in ANSIC and MS_rand requires at least 10^4 instances and in MINSTD at least $5 \cdot 10^4$ instances.

generator	RMSE (stdev)	p-value of KS test
ANSIC	1.12 (0.008)	0.77
MINSTD	1.12 (0.010)	0.77
MRG32k5a	1.12 (0.009)	1.0
MS_rand	1.12 (0.007)	0.77
rand_state	0.88 (0.016)	$4.7 \cdot 10^{-9}$
RANDU	1.12 (0.011)	0.86
RANDUall	1.03 (0.127)	$4.2 \cdot 10^{-5}$
WELL19937a	1.12 (0.012)	0.50

Figure 3: The results of regression trees with linear models in the leaves for different generators.

4.3 Learning regression models

We used 10^4 instances for training and the same number of instances for testing. As models we used regression trees with linear models in the leaves. Initially we tried also some other models in the leaves (locally weighted regression and nearest neighbor method with Gaussian kernel), as well as some other learning algorithms (random forests, neural networks), but as the results are quite similar, we report results only for the fastest method, i.e., regression trees with linear models in the leaves. We report the results in the Table 3. The RMSE column contains the average relative mean squared error (RMSE) and its standard deviation. RMSE is a mean squared error of the predictor divided by the error of the predictor which always predicts the mean response value of the training set. The numbers given are averages over over 20 runs. For learning to be successful RMSE should be less than 1. The p-value column gives the p-value of KS test of the hypothesis that the RMSE results for given generator are not different to RMSE of MRG32k5a.

As seen from the results the dependence in rand_state was clearly detected and RMSE of rand_state is consistently below 1. In other generators, the dependence was not detected, except of RANDUall, which denotes RANDU generator with random seed from both odd and even integers. Even seeds lead to shorter lengths of the period and, hence, lead to sequences that are easier to predict. Since the results are based on different period lengths, the results for RANDUall have larger standard deviation than from a other generators. If we use only odd seeds, RANDU behaves quite well in this test.

5 Conclusion

The paper demonstrates that the dependence in the default function rand in Matlab versions between 5 (1995) and 7.3 (2006b) may easily be detected by standard machine learning tools. This finding is a severe warning against relying on the results obtained using this generator. Experiments with other generators known to be bad reveal that only the RANDU generator has a dependence, which is detectable at comparable level

using the same tools. Dependencies in ANSIC, MINSTD and MS_rand are harder to detect (we need more instances for learning). Still we warn against their use in machine learning and data mining. It turned out that the most sensitive tool for detection of dependencies was RReliefF algorithm commonly used in feature selection.

Learning regression models (Section 4.3) detected the dependence only in method state of Matlab's rand. Probably, increasing the number of used instances allows to detect the dependence also in other tested generators, but this was not important to show our main point.

A possible direction for further research is to change the definition of classification tasks. Instead of deriving the class from one of the generated values, we may generate instances from two different generators, one reliable and the other suspicious, and label instances from each generator with different class values. According to our experiments, it is more difficult to detect dependencies this way, however, in case of success, the rejection of randomness of the suspicious generator is even more convincing.

Another interesting task would be to repeat some experiments described in machine learning and data mining publications where bad pseudorandom number generators were used. If the results using a good generator are indeed different, this would be an alarming result for the research community.

Acknowledgements

The first author was supported by Academy of Sciences of the Czech Republic under the grant number 1ET100300517 (Information Society) and by Institutional Research Plan AVOZ10300504. The second author was supported by Slovene Ministry of Higher Education, Science and Technology through the research programme P2-0209.

References

L. Breiman. Random forests. *Machine Learning Journal*, 45:5–32, 2001.

O. Goldreich. Lecture notes on pseudorandomness - part I (polynomial-time generators). http://www.wisdom.weizmann.ac.il/ oded/c-indist.html, 2000.

O. Goldreich. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness (Algorithms and Combinatorics, Vol 17).* Springer-Verlag, 1998.

D. E. Knuth. *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, 3rd ed.* Addison-Wesley, Reading, Massachusetts, 1998.

L'Ecuyer and R. Simard. TestU01: A C Library for Empirical Testing of Random Number Generators. *to appear in ACM Transactions on Mathematical Software*, 2006.

P. L'Ecuyer. Random number generation. In S. G. Henderson and B. L. Nelson, editors, *Handbooks in Operations Research and Management Science: Simulation*. Elsevier, 2006.

P. L'Ecuyer. Good parameter sets for combined multiple recursive random number generators. *Operations Research*, 47(1):159–164, 1999.

P. L'Ecuyer, R. Simard, and S. Wegenkittl. Sparse serial tests of uniformity for random number generators. *SIAM Journal on Scientific Computing*, 24(2):652–668, 2002.

M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. *ACM Trans. on Modeling and Computer Simulation*, 8(1):3–30, 1998.

F. Panneton, P. L'Ecuyer, and M. Matsumoto. Improved long-period generators based on linear recurrences modulo 2. *ACM Transactions on Mathematical Software*, 32 (1):1–16, 2006.

S. K. Park and K. W. Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, I2(10):1192–1201, 1988.

W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical recipes in C*. Cambridge University Press, 1988.

R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2006. URL `http://www.R-project.org`.

M. Robnik-Šikonja and I. Kononenko. Theoretical and empirical analysis of ReliefF and RReliefF. *Machine Learning Journal*, 53:23–69, 2003.

P. Savicky. A strong nonrandom pattern in Matlab default random number generator, 2006. URL `http://www.cs.cas.cz/~savicky/papers/rand2006.pdf`.

B. Wichman and I. Hill. Generating good pseudo-random numbers. *Computational Statistics and Data Analysis*, 51, 2006.

I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, San Francisco, 1999.